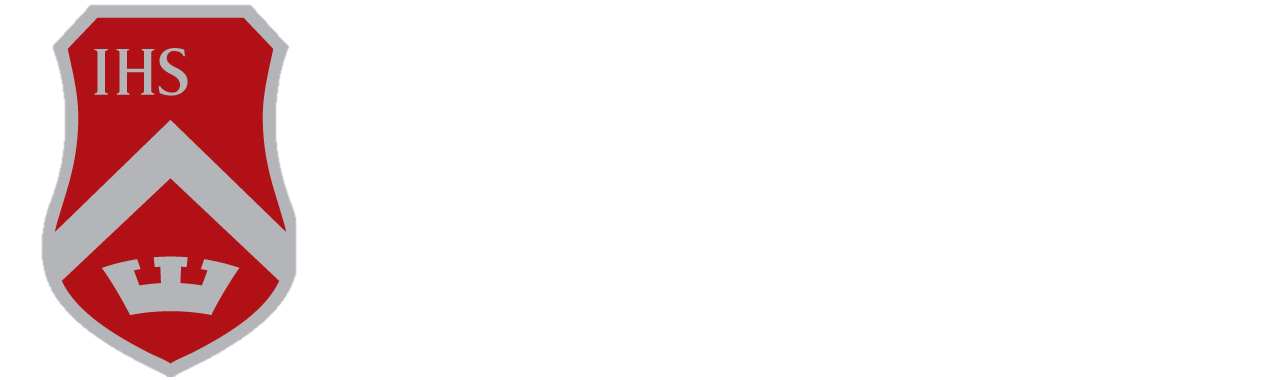
**2020 Fall- Operating System**

**Pintos Project 1**

**User Program (1)**

****

|  |  |
| --- | --- |
| 담당 교수 : | 김영재 교수 |
| 학번 : | 20160366 |
| 이름 : | 김지수 |
|  |  |

목차

1. **개발 목표**
2. **개발 범위 및 내용**
   1. **개발 범위**
      1. Argument Passing
      2. User Memory Access
      3. System Calls
   2. **개발 내용**
3. Argument Passing
4. User Memory Access
5. System Calls
6. **추진 일정 및 개발 방법**
   1. **추진 일정**
   2. **개발 방법**
7. **연구 결과**
   1. **Flow Chart**
      1. Argument Passing
      2. User Memory Access
      3. System Calls
   2. **제작 내용**
8. Argument Passing
9. User Memory Access
10. System Calls
11. Additional System calls
    1. **시험 및 평가 내용**

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

**1. 개발 목표**

현재 핀토스에는 User Program을 작동할 수 있도록 도와주는 기본적인 OS의 기능들이 구현되어 있지 않다. User Program을 작동시키는 명령어가 입력된 경우, User Stack에 argument를 parsing하여 쌓은 후 프로세스를 생성하고 실행할 수 있어야 한다. 하지만 현재에는 User Stack에 인자가 쌓이도록 구현되지 않았으며, User 프로세스를 생성한 후 자식 프로세스가 종료되도록 대기하는 것 또한 구현되지 않았다. 이번 프로젝트를 통해서 핀토스에서 응용프로그램의 실행을 돕는 기본적인 OS의 기능, 커맨드의 인자를 유저 스택을 통해 넘기는 Argument Passing, User Program에서 kernel에 서비스를 요청한 경우 그 address space가 valid한지 검사하는 User Memory Access, 프로세스를 중단, 실행, 대기하는 System Calls을 처리하기 위한 System Call Handler와 추가적인 System Call들을 구현할 것이다.

**2. 개발 범위 및 내용**

1. **개발 범위**
2. **Argument Passing**

커맨드 라인에서 프로세스 이름을 확인하여 그 이름으로 스레드를 만들고, 커맨드 라인을 Parsing하여 인자를 확인 하고, 인자를 스택에 push.

1. **User Memory Access**

user program이 요청한 virtual address가 valid한지 확인하고 invalid한 경우 프로세스 종료.

1. **System Calls**

0x30번으로 호출된 interrupt에 대해, interrupt handler가 system call number를 확인하여 알맞은 system call 함수를 호출하여 execute, exit, wait, read, write이 실행된다. 또한 추가적으로 Fibonacci, max\_of\_four\_int의 system call도 구현됨.

1. **개발 내용**
2. **Argument Passing**

응용 프로그램을 실행하게 되면 run\_task()가 호출되며 process\_execute()에서는 argv로 전달받은 file\_name을 parsing하여 그 이름을 가지고 thread를 만든다. 만들어진 thread는 ready queue에 추가되고, tid가 return되어 스레드를 만든 해당 프로세스는 process\_wait()을 통해 tid를 가진 자식 프로세스가 종료될 때까지 기다린다. 만들어진 thread의 실행 함수로 지정된 start\_process()가 실행되면서 load()를 통해 프로그램을 메모리에 적재하고, 그 과정에서 stack 공간을 setup하여 file\_name에서 parsing한 다른 argument들을 stack에 right-to-left순으로 쌓는다. 메모리에 적재를 성공하면 user program이 시작된다.

file\_name에서 parsing한 다른 argument들이 stack에 쌓일 때는 **80x86 calling convention**을 따르며, argument는 총 128Byte이내로 들어오는 것으로 한정한다. 80x86 convention이란, argv로 전달받은 file\_name을 공백 단위로 parsing하여 right-to-left순서로 null-pointer sentinel을 포함하여 stack에 push한 후, word-align을 통해 4byte단위를 맞춘다. 그리고는 첫번째로 push된 argument부터 그 주소를 또 순서대로 push한다. 마지막으로는 argv의 시작 주소, 총 argument의 개수(argc)와 (dummy) return address를 차례로 push한다. PHYS\_BASE를 기준으로 인자 크기만큼 downward로 스택이 쌓이며, 마지막에는 caller의 next PC값이 저장된다(Callee의 return value는 eax 레지스터에 저장됨). 다음의 표는 0xc0000000를 PHYS\_BASE로 가질 때 ‘/bin/ls -l foo bar’을 parsing하여 stack에 쌓은 결과의 예시이다.

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Name | Data | Type |
| 0xbffffffc | argv[3][…] | ‘bar\0’ | char[4] |
| 0xbffffff8 | argv[2][…] | ‘foo\0’ | char[4] |
| 0xbffffff5 | argv[1][…] | ‘-l\0’ | char[3] |
| 0xbfffffed | argv[0][…] | ‘/bin/ls\0’ | char[8] |
| 0xbfffffec | word-align | 0 | unit8\_t |
| 0xbfffffe8 | argv[4] | 0 | char \* |
| 0xbfffffe4 | argv[3] | 0xbffffffc | char \* |
| 0xbfffffe0 | argv[2] | 0xbffffff8 | char \* |
| 0xbfffffdc | argv[1] | 0xbffffff5 | char \* |
| 0xbfffffd8 | argv[0] | 0xbfffffed | char \* |
| 0xbfffffd4 | argv | 0xbfffffd8 | char \*\* |
| 0xbfffffd0 | argc | 4 | int |
| 0xbfffffcc | return address | 0 | void (\*) () |

1. **User Memory Access**

user program이 요청한 virtual address가 kernel영역을 침범하거나 Null 포인터 거나, 해당 스레드의 pagedir에 mapping되어 있지 않은 주소인 경우 invalid하다고 말한다. User Program이 system call을 통해 kernel을 호출하면, kernel은 서비스를 제공하기에 앞서 요청 받은 user virtual address의 validity를 확인한 후, invalid한 경우에는 프로세스를 종료 시키고, 그렇지 않은 경우에는 해당 system call을 처리하도록 할 수 있다. 따라서 system call handler에서 발생하는 모든 주소를 접근 전에 확인하는 용도로 사용한다.

1. **System Calls**

운영체제는 kernel의 코드영역을 user program영역과 분리하여 kernel 영역으로의 접근에 제한을 두고, 또는 다른 program의 영역을 서로 침범하지 못하도록 하여 보안성을 높이고 효율적인 자원관리를 가능하게 한다. 그렇기 때문에 kernel이 관할하는 서비스를 user program이 이용하기 위해서 system call이라는 인터페이스를 제공하여, 서비스를 요청하는 경우 kernel mode에서 서비스를 수행하고, 다시 user mode로 복귀하는 구조이다.

pintos에서 user program이 어떤 system call을 부르면 lib/user/syscall.c의 해당 함수가 호출되고, 함수에서 받아진 인자의 개수에 따른 system call 매크로를 통해 user stack에 인자를 쌓은 후, interrupt vector table의 0x30번에 있는 system call handler가 호출된다. system call handler에서 stack에 쌓인 system call number와 argument들을 확인하여 kernel service를 수행하게 된다. 이번 프로젝트에서 구현할 system call의 종류는 크게 8가지로 다음과 같다.

* **void sys\_halt( )**: pintos를 종료 시키는 시스템 콜.
* **void sys\_exit(int status)**: 현재 실행 중인 프로세스를 종료 시키는 시스템 콜. 종료 시 *“[프로세스 이름]: exit([status])”* 를 출력해야 한다.
* **pid\_t sys\_exec(const \* cmd\_line)**: 자식 프로세스를 생성하고 실행시키는 시스템 콜. cmd\_line은 자식 프로세스에서 실행할 명령어이다. 부모 프로세스는 자식 프로세스의 프로그램이 메모리에 적재하기를 기다렸다가, 자식 프로세스 적재의 상태에 따라 실패하면 -1, 성공하면 자식 프로세스의 pid를 반환한다.
* **int sys\_wait(pid\_t pid)**: process\_wait()을 호출하여, 자식프로세스가 종료될 때까지 기다리고, 올바르게 종료됐는지 확인하는 시스템 콜.
* **static int sys\_read(int fd, void \*buffer, unsigned size)**: fd가 0일 때 키보드의 데이터를 읽어 buffer에 저장하는 시스템 콜. size는 읽을 데이터의 크기이며, 성공 시 읽은 바이트 수를 반환하지만 실패하면 -1을 반환한다.
* **static int sys\_write(int fd, void \*buffer, unsigned size)**: fd가 1일 때 buffer에 저장된 데이터를 화면에 출력하는 시스템 콜. size는 출력할 데이터의 크기이며, 성공 시 출력한 바이트 수를 반환하지만 실패하면 -1을 반환한다.
* **int sys\_fibonacci(int n)**: n번째의 피보나치 수를 계산하여 반환한다.
* **int sys\_four\_of\_max(int a, int b, int c, int d**): 들어온 4개의 숫자 중 가장 큰 값을 반환한다.

exit, exec와 wait을 제대로 작동 시키기 위해서는 부모 프로세스와 자식 프로세스의 관계를 구현하고, 부모 프로세스가 자식 프로세스의 종료를 기다릴 수 있도록 만들어야 한다. 현재의 pintos에서는 run\_task()로 user process가 생성되면, thread가 성공적으로 만들어졌다 하더라도 무조건 부모 프로세스가 종료되도록 만들어져 있다. 부모, 자식 관계를 구현하고, 부모 프로세스는 자식 프로세스의 실행과 종료를 기다리고, 그 status를 받아 종료되어야 한다.

**3. 추진 일정 및 개발 방법**

* + 1. **일정**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~ 10.13 | 10.14 | 10.15 | 10.16 | 10.17 | 10.18 | 10.19 |
| Pintos 메뉴얼 읽기 | Argument Passing 코드 초안 작성 | | | User Memory 코드 초안 작성 | | System Call |
| 10.20 | 10.21 | 10.22 | 10.23 | 10.24 | 10.25 | 10.26 |
| System Call 코드 초안 작성 | | | | | | |
| 10.27 | 10.28 | 10.29 | 10.30 | 10.31 | 11.1 | 11.2 ~ 3 |
| System Call | | 보고서 작성 | | | 디버깅 및 보고서 완성 | |

* + 1. **개발 방법**

1. **Argument Passing**

(1) **src/userprog/process.c/tid\_t process\_execute (const char \*file\_name)**에 Termination message 출력을 위해 프로세스 이름을 알아야 하므로, file\_name으로 넘겨받은 command를 parsing하여 thread\_create()에 사용할 수 있는 스레드 이름으로 전달한다.

(2) **src/userprog/process.c/bool load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp)**에서 filesys\_open (file\_name)을 하기 전에 file\_name을 parsing하여 오픈할 파일명으로 전달한다. setup\_stack() 이후에 만들어진 공간에 80x86 calling convention에 따라 argument을 push한다.

1. **User Memory Access**

(1) **threads/thread.c/uint32\_t\* thread\_get\_pagedir(struct thread \*thrd)**을 구현하여 현재 실행 중인 스레드의 pagedir를 반환하고, 이 반환되는 값을 (2)의 validity검사에 활용되는 pagedir\_get\_page()함수에 인자로 전달한다.

(2) **src/userprog/syscall.c/bool check\_validity\_useraddr(const void \*uaddr)**을 구현하여, user-provided pointer의 참조 범위가 kernel영역을 침범하지 않는지, null포인터는 아닌지, mapping되지 않은 user virtual address는 아닌지 확인한다. 이 함수를 구현하여 user program이 system call을 호출하게 될 때마다 validity를 확인하고, invalid할 경우 프로세스를 종료 시킨다.

1. **System Calls**

(1) **userprog/process.h/struct process**를 새롭게 정의하여 PCB(Process Control Block)을 구현한다. 이 PCB에는 user 프로세스에 대한 정보(file\_name, pid, parent 등 부모자식 관계, exit status등)와 적재, 종료 waiting을 위한 semaphore와 해당 여부를 저장하는 flag정보가 들어가 있다.

(2) **threads/thread.h/struct thread**에 PCB와 children list를 #ifdef userprog와 #endif 사이에 추가하여 run\_task()에서 process\_execute()가 호출됐을 때 initial kernel thread가 child process를 가질 수 있어야 한다.

(3) **threads/thread.c**의 **static void init\_thread(struct thread \*t, const char \*name, int priority)**에 코드를 추가해 새롭게 추가된 구조체 field 또한 초기화 시킨다. 같은 디렉토리에 추가되는 함수로는: **void thread\_set\_pcb(struct process \*new\_pcb)**를 구현해 현재 스레드의 pcb를 새로운 pcb로 업데이트 하고, **struct process \*thread\_get\_pcb(void)**을구현해 현재 스레드의 pcb를 반환하도록 한다. **struct list\* thread\_get\_children(void)를** 구현해 현재 스레드의 children 리스트 포인터를 반환하도록 한다.

(4) **userprog/process.c/tid\_t process\_execute(const char \*file\_name)**에 struct process \* pcb를 선언하여 file\_name을 fn\_copy가 아닌 pcb에 저장하여 새로운 프로세스를 만들 때(start\_process)사용한다. 또한 struct process의 정보를 초기화하여 이를 바탕으로 새로운 스레드를 thread\_create하고 sema\_down으로 스레드의 로딩을 기다리며, 현재 실행 중인 스레드의 자식 리스트에 추가한다.

**static void start\_process(void \*pcb\_)**에 struct process \* pcb를 선언하여 넘겨 받은 pcb를 스레드에 세팅하고, 로딩이 성공한 후 pcb를 통해 부모에게 로딩 성공을 알리고 프로그램을 실행한다.

**int process\_wait(tid\_t child\_tid)**에 자식 프로세스가 종료될 때까지 sema\_down을 통해 기다리도록 구현하고, 종료되면 자식 프로세스를 child list에서 제거한 후 exit status를 반환한다.

**void process\_exit(void)**에서 자식 스레드들을 remove하고 page directory를 해제하며 sema\_up을 통해 process\_wait를 호출한 부모 스레드에 종료 상태 정보를 전달한다.

**struct process\* process\_get\_child(pid\_t pid)**를 구현하여 해당 프로세스 id를 가진 자식 프로세스가 맞으면 pcb를 반환하고 자식 프로세스가 아니라면 NULL을 반환한다. **void process\_remove\_child(struct process \*child)**을 구현해서 넘겨받은 pcb의 자식 스레드가 존재하며, 상태가 종료된 게 맞다 면, 자식 리스트에서 제거하고 페이지 할당을 해제한다.

(5) **userprog/syscall.c**에는 user program의 system call 호출 시 system call handler가 적절한 system call 번호에 따라 호출하는 함수들을 정의한다. **void sys\_halt( )**을 구현하여, 호출 시 pintos를 종료 시키도록 한다. **void sys\_exit(int status)**을 구현하여, 호출 시 status를 출력하며 해당 스레드를 종료 시킨다. **static tid\_t sys\_exec(const \*cmd\_line)을** 구현하여, 호출 시 cmd\_line 프로그램을 실행하고 tid를 반환한다. **static int sys\_wait(pid\_t pid)**은 wait기능을 구현하기 위해 process\_wait을 호출한다. **static int sys\_read(int fd, void \*buffer, unsigned size)**와 **static int sys\_write(int fd, void \*buffer, unsigned size)**를 구현하여, 표준 입출력에 대한 system call을 위한 함수들도 정의한다.

**static void syscall\_handler(struct intr\_frame \*f)**에 system call 번호를 스택에서 pop시켜 switch문을 통해 적절한 번호의 (바로 위에 구현한다고 적은) sys\_함수를 호출하도록 구현한다.

1. **Additional System Calls**

(1) **lib/user/syscall.c**에 **#define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3)**를 새로 define하여 인자가 4개 들어오는 경우의 system call 매크로를 만든다. 또한 **int fibonacci(int n)**와 **int max\_of\_four\_int(int a, int b, int c, int d)** 함수에 각각 syscall1과 syscall4를 호출하여 SYS\_FIBONACCI, SYS\_MAXFOUR의 system call number에 대해 커널 서비스를 요청하도록 한다.

(2) **userprog/syscall.c**에 **int sys\_fibonacci(int n)**와 **int sys\_max\_of\_four(int a, int b, int c, int d)**를 구현하여 해당 system call이 호출됐을 때 각각 n번째 피보나치 수와 네 가지 수 중에 가장 큰 수를 반환하는 함수들을 정의한다.

(3) **example/additional.c**을 만들어system call을 호출하는 응용 프로그램 **int main(int argc, char \*\*argv)**을 추가한다. 총 4개의 인자를 받아서 첫번째 인자의 fibonacci 값과 4개 숫자 중 가장 큰 값을 차례로 출력한다.

**4. 연구 결과**

|  |  |
| --- | --- |
| 1. Argument Passing | 1. User Memory Access |
|  |  |
| 1. System Calls | |
|  | |

1. **Flow Chart**
2. **제작 내용**
3. **Argument Passing**

(1) src/userprog/process.c/tid\_t process\_execute (const char \*file\_name)

|  |
| --- |
| **tid\_t process\_execute (const char \*file\_name)**  {  **char \* fn\_copy, name**, \***save\_ptr;**  **tid\_t** **tid**;  ︙  /\* Create a new thread to execute FILE\_NAME. \*/  **name** = **strtok\_r**(**file\_name**, " ", &**save\_ptr**);  **tid** = **thread\_create** (**name**, **PRI\_DEFAULT**, **start\_process**, **fn\_copy**);  ︙  **return** **tid**;  } |

lib/string.c의 strtok\_r를 활용하여 file\_name 맨 첫 token을 parsing한 것을 thread의 이름으로 thread\_create()함수에 전달한다. 이렇게 전달된 thread\_create의 가장 첫 인자 name은 argc[0]에 위치한 실제 파일 이름이고, 마지막 인자 fn\_copy는 parsing전에 file\_name을 복제한 것으로 full command이다. thread\_create()에서 name은 init\_thread()함수의 스레드 초기화에 사용되며, fn\_copy로 전달된 full command는 실행함수인 start\_process의 인자가 되어 load로 전달된다.

(2-3) src/userprog/process.c/bool load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp) load()

|  |
| --- |
| **bool load (const char \*file\_name, void(\*\*eip)(void), void \*\*esp)**  **{**  **︙**  **struct file \*file = NULL;**  **int i;**  **int argc = 0;**  **char \*argv[32];**  **char \*word, \*save\_ptr;**  **︙**  /\* (2) Parse the file name. \*/  **word = strtok\_r(file\_name, " ", &save\_ptr);**  **while(word!=NULL)**  **{**  **argv[argc++] = word;**  **word = strtok\_r(NULL, " ", &save\_ptr);**  **}**  /\* Open executable file. \*/  **file = filesys\_open (argv[0]);**  **︙**  **}** |

start\_process가 실행되면 load()함수를 통해 file\_name의 프로그램을 메모리에 적재하게 된다. 이것이 성공하지 못하면 thread\_exit()함수가 호출되지만, 성공하면 user process가 실행되는 것이다.

load()에 전달된 file\_name은 process\_execute()함수에서 parsing되지 않은 full command로, 메모리에 적재할 실제 파일 이름을 찾아 open하기 위해서는 또 다시 parsing과정이 필요하다. 또한, 파일의 실제 이름 뒤에 따라오는 다른 인자들은 stack 생성 이후에 right-to-left로 push해야 하기 때문에, argv라는 배열을 이용해서 left-to-right순서로 parsing된 word들을 저장해 놓는 것이 필요하다. 따라서, filesys\_open의 호출 전에 strtok\_r()을 통해 공백 단위의 인자들을 모두 parsing하여 argv에 저장한다. 그리고 filesys\_open으로는 argv[0]에 있는 실제 파일이름을 전달한다.

|  |
| --- |
| **bool load (const char \*file\_name, void(\*\*eip)(void), void \*\*esp)**  **{**  **︙**  /\* Set up stack. \*/  **if (!setup\_stack (esp))**  **goto done;**    /\* (3). Push arguments into stack. \*/  **uintptr\_t addr[32];**    **for (i = argc - 1; i >= 0; i--)**  **{**  **\*esp -= strlen(argv[i]) + 1;**  **strlcpy(\*esp, argv[i], strlen(argv[i]) + 1);**  **addr[i] = (uintptr\_t)\*esp;**  **}**  **\*esp = (uintptr\_t)\*esp & ~0x3;** //word align  **\*esp -= sizeof(uintptr\_t);**  **\*(uintptr\_t\*)\*esp = (uintptr\_t)0;**  **for (i = argc - 1; i >= 0; i--)**  **{**  **\*esp -= sizeof(uintptr\_t);**  **\*(uintptr\_t \*)\*esp = addr[i];**  **}**  **\*esp -= sizeof(uintptr\_t);**  **\*(uintptr\_t \*)\*esp = (uintptr\_t)\*esp + sizeof(uintptr\_t);**  **\*esp -= sizeof(uintptr\_t);**  **\*(int \*)\*esp = argc;**  **\*esp -= sizeof(uintptr\_t);**  /\* Start address. \*/  **\*eip = (void (\*)(void))ehdr.e\_entry;**  **success = true;**  **︙**  **}** |

setup\_stack()함수가 호출되고 난 이후로는 이제 아까 parsing한 인자들을 push하기만 하면 된다. 하지만 80x86 calling convention에 따라 parsing한 인자들이 저장된 위치를 또 push할 필요가 있으므로, addr[]배열을 이용하여 push하면서 그 주소도 함께 트레킹 한다. parsing하며 argv배열에 담았던 인자의 개수 argc만큼 loop을 돌며, esp를 그 인자의 길이+1만큼(null-point sentinel포함)감소 시킨 후, 그 자리에 argv의 원소를 뒤에서부터 하나씩 strlcpy하여 right-to-left순서대로 esp포인터 위치에 넣는다. 인자를 하나씩 복사할 때마다 addr배열에는 그 복사된 위치를 저장한다.

인자를 모두 옮긴 후에 word-align을 위해 esp의 하위 2-bit를 00으로 맞추어 주고, 위와 같은 방식으로 이번에는 addr에 담긴 주소를 저장한 순서대로 스택에 쌓는다. 그러고 나서 argv의 시작주소를 push하고, argc를 push하고, esp를 한 칸 더 내림으로써 dummy return address도 push한다.

1. **User Memory Access**

(1) threads/thread.c/uint32\_t\* thread\_get\_pagedir(struct thread \*thrd)

|  |
| --- |
| **uint32\_t \* thread\_get\_pagedir(struct thread \* thrd)**  **{**  **return thrd->pagedir;**  **}** |

인자로 전달 받은 스레드의 pagedir을 반환한다.

(2) src/userprog/syscall.c/bool check\_validity\_useraddr(const void \*uaddr)

|  |
| --- |
| **bool check\_validity\_useraddr(const void \*uaddr)**  **{**  **struct thread \*cur\_thread = thread\_current();**  **if (uaddr == NULL || is\_kernel\_vaddr(uaddr)||!pagedir\_get\_page(thread\_get\_pagedir(cur\_thread), uaddr))**  **return false;**  **return true;**  **}** |

현재 전달받은 user virtual address 포인터가 1. NULL포인터이거나, 2. kernel 영역을 침범했거나, 3. pagedir에 mapping되어 있지 않은 주소를 참조하거나 한다면 false를 반환한다. if문에서 pagedir\_get\_page(userprog/pagedir.c에 있음)내에서 kernel 영역인 경우를 판단해 abort시키는 코드가 있으므로 is\_kernel\_vaddr(thread/vaddr.h에 있음)와 구현 순서를 주의한다. 이 함수의 반환 값이 false인 경우 system call handler에서 판단하여 process를 exit시킬 것이다.

1. **System Calls**

(1) userprog/process.h/struct process

|  |
| --- |
| **struct process**  **{**  **const char \* file\_name;** /\* Filename to execute. \*/  **pid\_t pid;**  **struct thread \*parent;** /\* Parent Process. \*/  **struct list\_elem child\_elem;** /\* List elements for children. \*/  **bool is\_loaded;** /\* whether process loading is failed. \*/  **bool is\_exited;** /\* True when the process is terminated. \*/  **struct semaphore sema\_exit;** /\* exit semaphore. \*/  **struct semaphore sema\_load;** /\* load semaphore. \*/  **int exit\_status;** /\* termination status when exit() called. \*/  **};** |

start\_process 함수에 file\_name 복사본을 넘기는 대신 struct process를 인자로 넘김으로써 구조체 속 file\_name을 사용할 수 있도록 한다. 더불어 child\_elem으로 자식 스레드들의 리스트 앞 뒤로 리스트를 연결 가능하게 하고, load와 exit에 대한 bool정보와 동기화할 수 있도록 semaphore 정보도 유지한다.

|  |
| --- |
| **struct thread**  **{**  **︙**  #ifdef USERPROG  **uint32\_t \* pagedir;** /\* Page directory. \*/  **struct process \*pcb**; /\* Process Control Block. \*/  **struct list children;** /\* List of children process. \*/  #endif  **︙**  **};** |

(2) threads/thread.h/struct thread

스레드 내에 struct process 포인터로 pcb와, init을 하게 되는 경우에는 process\_execute가 실행되지 않아 pcb가 생기지 않으므로 children 리스트의 head와 tail정보도 추가한다.

(3-1) threads/thread.c/static void init\_thread(struct thread \*t, const char \*name, int priority)

|  |
| --- |
| **struct void init\_thread(struct thread \*t, )**  **{**  **︙**  #ifdef USERPROG  **t->pcb = NULL;**  **list\_init(&t->children);**  #endif  **︙**  **};** |

thread를 초기화할 때, struct thread에 새롭게 추가한 pcb와 list\_init을 통해 children리스트도 초기화한다.

(3-2) threads/thread.c/void thread\_set\_pcb(struct process \*new\_pcb)

|  |
| --- |
| **struct process \* thread\_set\_pcb(struct process\* new\_pcb)**  **{**  **thread\_current()->pcb = new\_pcb;**  **}** |

인자로 넘어온 new\_pcb로 현재 실행 중인 스레드의 pcb를 업데이트 한다.

(3-3) threads/thread.c/struct process \*thread\_get\_pcb(void)

|  |
| --- |
| **struct process \* thread\_get\_pcb(void)**  **{**  **return thread\_current()->pcb;**  **}** |

현재 실행 중인 스레드의 pcb를 반환한다.

(3-4) threads/thread.c/struct list\* thread\_get\_children(void)

|  |
| --- |
| **struct list \* thread\_get\_children(void)**  **{**  **return &thread\_current()->children;**  **}** |

현재 실행 중인 스레드의 children리스트 위치를 반환한다.

(4-1) userprog/process.c/tid\_t process\_execute(const char \*file\_name)

|  |
| --- |
| **tid\_t process\_execute(const char \*file\_name)**  **{**  **char \*fn\_copy, \*fn\_copy2, \*name, \*save\_ptr;**  **tid\_t tid;**  **struct process \*pcb;**  /\* 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);**  **fn\_copy2 = palloc\_get\_page(0);**  **if (fn\_copy2 == NULL)**  **return TID\_ERROR;**  **strlcpy(fn\_copy2, file\_name, PGSIZE);**  /\* Initialize new thread's pcb. \*/  **pcb = palloc\_get\_page(0);**  **if (pcb == NULL)**  **return TID\_ERROR;**  **pcb->file\_name = fn\_copy;**  **pcb->parent = thread\_current();**  **sema\_init(&pcb->sema\_load, 0);**  **sema\_init(&pcb->sema\_exit, 0);**  **pcb->is\_loaded = false;**  **pcb->is\_exited = false;**  **pcb->exit\_status = -1;**  /\* Create a new thread to execute FILE\_NAME. \*/  **name = strtok\_r(file\_name, " ", &save\_ptr);**  **tid = thread\_create(name, PRI\_DEFAULT, start\_process, fn\_copy);**  **if (tid == TID\_ERROR)**  **{**  **palloc\_free\_page(fn\_copy);**  **palloc\_free\_page(pcb);**  **}**  **palloc\_free\_page(fn\_copy2);**  **sema\_down(&pcb->sema\_load);**  **if (pcb->pid != PID\_ERROR)**  **list\_push\_back(thread\_get\_children(), &pcb->child\_elem);**  **return tid;**  **}** |

process\_execute가 실행되면 전달된 file\_name의 user program을 실행하는 새로운 스레드를 시작 시킨다. struct process\* pcb를 하나 선언하여 현재 실행 중인 스레드를 부모로 하고, file\_name, semaphore 정보와 flag 등 user program의 pcb 정보를 초기화한다. 그리고 이 pcb와 file\_name을 parsing한 real file name을 인자로 넘겨 thread\_create 호출하여 전달된 정보로 새로운 스레드를 만들고, 현재 실행하던 스레드는 sema\_down을 통해 wait 상태에 들어가 새로운 스레드의 loading을 기다리게 된다. 그런 후 현재 스레드의 children 리스트에 새로 만든 스레드를 추가시킨다.

인자로 받은 file\_name을 그대로 strtok\_r로 parsing하게 되면 writing right violation이 발생할 수 있기 때문에 fn\_copy2로 복사본을 만들어 name을 parsing한다.

(4-2) userprog/process.c/static void start\_process(void \*pcb\_)

|  |
| --- |
| **static void start\_process(void \*pcb\_)**  **{**  **struct process \*pcb = pcb\_;**  **int argc = 0;**  **char \*argv[32];**  **char \*file\_name = pcb->file\_name;**  **struct intr\_frame if\_;**  **bool success;**  /\* Set current thread's pcb with pcb\_. \*/  **thread\_set\_pcb(pcb);**  /\* Initialize interrupt frame and load executable. \*/  **memset(&if\_, 0, sizeof if\_);**  **if\_.gs = if\_.fs = if\_.es = if\_.ds = if\_.ss = SEL\_UDSEG;**  **if\_.cs = SEL\_UCSEG;**  **if\_.eflags = FLAG\_IF | FLAG\_MBS;**  **success = pcb->is\_loaded = load(file\_name, &if\_.eip, &if\_.esp);**  **pcb->pid = success ? thread\_tid() : PID\_ERROR;**  **sema\_up(&pcb->sema\_load);**  /\* If load failed, quit. \*/  **palloc\_free\_page(file\_name);**  **if (!success)**  **sys\_exit(-1);**  **asm volatile("movl %0, %%esp; jmp intr\_exit"**  **:**  **: "g"(&if\_)**  **: "memory");**  **NOT\_REACHED();**  **}** |

thread\_create가 실행되면서 start\_process가 해당 만들어진 스레드를 로딩 및 실행시킨다. 전달받은 pcb 정보로 현재 만들어진 스레드의 pcb를 업데이트 하고, pcb 속 file name에 해당하는 실제 프로그램을 loading시킨다. loading의 결과는 pcb에 반영되며 sema\_up을 실행하여 부모에게 로딩이 성공했음을 알리고 프로그램을 실행한다.

(4-3) userprog/process.c/int process\_wait(tid\_t child\_tid)

|  |
| --- |
| **int process\_wait(tid\_t child\_tid)**  **{**  **struct process \*child = process\_get\_child(child\_tid);**  **int exit\_status;**  **if (!child)**  **return -1;**  **sema\_down(&child->sema\_exit);**  **exit\_status = child->exit\_status;**  **process\_remove\_child(child);**  **return exit\_status;**  **}** |

프로그램이 로딩된 후 process\_wait이 실행되는데, 여기서는 child\_tid를 받아 해당 tid를 가진 스레드의 실행 종료를 sema\_down을 통해 기다린다. tid가 invalid하거나 해당 부모 프로세스의 자식 스레드가 아닌 경우라면, 또는 커널에 의해 종료된 상태라면 -1을 반환한다. child 스레드가 성공적으로 exit한 후 process\_remove\_child를 통해 부모 스레드의 자식 리스트에서 제거하고, 자식의 exit status를 반환한다.

(4-4) userprog/process.c/void process\_exit(void)

|  |
| --- |
| **void process\_exit(void)**  **{**  **struct thread \*cur = thread\_current();**  **struct process \*pcb = thread\_get\_pcb();**  **struct list \*children = thread\_get\_children();**  **struct list\_elem \*e;**  **uint32\_t \*pd;**  **pcb->is\_exited = true;**  **for (e = list\_begin(children); e != list\_end(children); e = list\_next(e))**  **process\_remove\_child(list\_entry(e, struct process , child\_elem));**    **sema\_up(&pcb->sema\_exit);**  **pd = cur->pagedir;**  **if (pd != NULL)**  **{**  **cur->pagedir = NULL;**  **pagedir\_activate(NULL);**  **pagedir\_destroy(pd);**  **}**  **}** |

user program이 실행을 마치고 thread\_exit를 하면, process\_exit함수가 호출되면서 자식 스레드가 해당되는 부모 스레드의 자식 list의 원소들에 대해 for문을 돌며 exit된 자식 스레드들을 remove해준다. 또한 실행을 마친 현재의 스레드(자식 스레드)가 sema\_up을 통해 process\_wait에 자식 스레드의 exit 사실과 상태정보를 넘긴다. 그리고 부모 스레드의 page dir을 할당 해제한다. thread\_exit으로 돌아가서 process\_exit 이후의 action들을 마저 수행하여 부모가 완전히 exit하게 된다.

(4-5) userprog/process.c/struct process\* process\_get\_child(pid\_t pid)

|  |
| --- |
| **struct process \* process\_get\_child(pid\_t pid)**  **{**  **struct list \*children = thread\_get\_children();**  **struct list\_elem\* e;**  **for (e = list\_begin(children); e != list\_end(children); e = list\_next(e))**  **{**  **struct process \*pcb = list\_entry(e, struct process, child\_elem);**    **if (pcb->pid == pid)**  **return pcb;**  **}**  **return NULL;**  **}** |

해당 process id를 가진 자식 스레드를 찾아서 존재한다면 pcb를 반환하고, 해당 pid를 가진 자식 스레드가 존재하지 않는다면 NULL을 반환한다. 이 함수는 process\_wait에서 해당 child\_tid를 가진 child 스레드를 부모 스레드의 자식 리스트에서 찾기 위해 사용된다. 여기서 자식의 pid는 tid와 일맥 상통하다.

(4-6) userprog/process.c/void process\_remove\_child(struct process \*child)

|  |
| --- |
| **void process\_remove\_child(struct process \*child)**  **{**  **if (child && child->is\_exited)**  **{**  **list\_remove(&child->child\_elem);**  **palloc\_free\_page(child);**  **}**  **}** |

child로 받은 pcb가 존재하고, is\_exited상태가 참이라면(종료됐다면) 해당 자식 스레드를 리스트에서 제거하고 할당됐던 page공간을 할당 해제 한다. 이 함수는 process\_exit에서 부모 스레드가 exit하기 전에 자식 스레드 중 종료된 것들을 retrieve할 때와 process\_wait에서 자식 스레드가 종료되어 sema\_up된 후 해당 종료된 자식 스레드를 retrieve할 때 사용된다.

(5-1) src/userprog/syscall.c/static void sys\_halt( )

|  |
| --- |
| **static void sys\_halt( )**  **{**  **shutdown\_power\_off( );**  **}** |

user program에 의해 system call 중에 halt가 호출되면 system call handler가 sys\_halt()를 실행한다. devices/shutdown.c에 있는 shutdown\_power\_off()함수를 호출하여 pintos를 종료 시킨다.

(5-2) src/userprog/syscall.c/void sys\_exit(int status)

|  |
| --- |
| **void sys\_exit(int status)**  **{**  **struct process \*pcb = thread\_get\_pcb();**  **pcb->exit\_status = status;**  **printf("%s: exit(%d)\n", thread\_name(), status);**  **thread\_exit();**  **}** |

user program에 의해 system call 중에 exit이 호출되면 system call handler가 sys\_exit()을 실행시킨다. 전달 받은 status 상태 값과 함께 종료되는 스레드의 file name을 출력하고 thread\_exit으로 스레드를 종료 시킨다.

(5-3) src/userprog/syscall.c/static tid\_t sys\_exec(const \*cmd\_line)

|  |
| --- |
| **static pid\_t sys\_exec(const char\*cmd\_line)**  **{**  **pid\_t pid;**  **struct process \*child;**  **int i;**  **if (!check\_validity\_useraddr(cmd\_line))**  **sys\_exit(-1);**  **for (i = 0; \*(cmd\_line + i) != NULL; i++)**  **if (!check\_validity\_useraddr(cmd\_line + i + 1))**  **sys\_exit(-1);**  **pid = process\_execute(cmd\_line);**  **child = process\_get\_child(pid);**  **if (!child || !child->is\_loaded)**  **return PID\_ERROR;**  **return pid;**  **}** |

user program에 의해 system call 중에 exec이 호출되면 system call handler가 sys\_exec()을 실행시킨다. cmd\_line으로 전달받은 프로그램을 process\_execute로 실행시키고, 자식 스레드가 성공적으로 로딩됐다면 pid를 받아 반환한다.

(5-4) src/userprog/syscall.c/static int sys\_wait(pid\_t pid)

|  |
| --- |
| **static int sys\_wait(pid\_t pid)**  **{**  **return process\_wait(pid);**  **}** |

user program에 의해 system call 중에 wait이 호출되면 system call handler가 sys\_wait()을 실행시킨다. 이 함수는 process\_wait을 호출하기만 하면 된다.

(5-5) src/userprog/syscall.c/static int sys\_read(int fd, void \*buffer, unsigned size)

|  |
| --- |
| **static int sys\_read(int fd, void \*buffer, unsigned size)**  **{**  **int i;**  **if (!check\_validity\_useraddr(buffer))**  **sys\_exit(-1);**    **for (i = 0; \*(char \*)(buffer + i) != NULL; i++)**  **if (!check\_validity\_useraddr(buffer + i + 1))**  **sys\_exit(-1);**  **if (fd == 0)**  **{**  **unsigned i;**  **for (i = 0; i < size; i++)**  **\*(uint8\_t \*)(buffer + i) = input\_getc();**  **return size;**  **}**  **}** |

user program에 의해 system call 중에 read가 호출되면 system call handler가 sys\_read()을 실행시킨다. 이 함수는 이번 프로젝트에서는 fd == 0인 표준 입력에 한하여 실행되므로 input\_getc()함수를 이용하여 사용자가 입력하는 값을 size byte만큼 buffer에 전달하고 전달 size를 반환한다.

(5-6) src/userprog/syscall.c/static int sys\_write(int fd, void \*buffer, unsigned size)

|  |
| --- |
| **static int sys\_write(int fd, void \*buffer, unsigned size)**  **{**  **int i;**  **if (!check\_validity\_useraddr(buffer))**  **sys\_exit(-1);**    **for (i = 0; \*(char \*)(buffer + i) != NULL; i++)**  **if (!check\_validity\_useraddr(buffer + i + 1))**  **sys\_exit(-1);**  **if (fd == 1)**  **{**  **putbuf((const char \*)buffer, (size\_t)size);**  **return size;**  **}**  **}** |

user program에 의해 system call 중에 write가 호출되면 system call handler가 sys\_write()을 실행시킨다. 이 함수는 이번 프로젝트에서는 fd == 1인 표준 출력에 한해 실행하므로 putbuf으로 buffer의 값을 console화면에 출력한다.

(5-7) src/userprog/syscall.c/static void syscall\_handler(struct intr\_frame \*f)

|  |
| --- |
| **static void syscall\_handler(struct intr\_frame \*f)**  **{**  **void \*esp = f->esp;**  **if (!check\_validity\_useraddr(esp) || !check\_validity\_useraddr(esp + sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int sys\_num = \*(int \*)esp;**  **switch (sys\_num)**  **{**  **case SYS\_HALT:**  **{**  **sys\_halt();**  **}**  **case SYS\_EXIT:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int status = \*(int \*)(esp + sizeof(uintptr\_t));**  **sys\_exit(status);**  **}**  **case SYS\_EXEC:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **char \*cmd\_line = \*(char \*\*)(esp + sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_exec(cmd\_line);**  **break;**  **}**  **case SYS\_WAIT:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **pid\_t pid = \*(pid\_t \*)(esp + sizeof(uintptr\_t));**  **f-> eax= (uint32\_t)sys\_wait(pid);**  **break;**  **}**  **case SYS\_READ:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 4 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int fd = \*(int \*)(esp + sizeof(uintptr\_t));**  **char \*buffer = \*(char \*\*)(esp + 2 \* sizeof(uintptr\_t));**  **unsigned size = \*(unsigned \*)(esp + 3 \* sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_read(fd, buffer, size);**  **break;**  **}**  **case SYS\_WRITE:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 4 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int fd = \*(int \*)(esp + sizeof(uintptr\_t));**  **char \*buffer = \*(char \*\*)(esp + 2 \* sizeof(uintptr\_t));**  **unsigned size = \*(unsigned \*)(esp + 3 \* sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_write(fd, buffer, size);**  **break;**  **}**  **case SYS\_FIBONACCI:**  **{**  **if (!check\_validity\_useraddr(esp) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int n = \*(int \*)(esp + sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_fibonacci(n);**  **break;**  **}**  **case SYS\_MAXFOUR:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 5 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int a = \*(int \*)(esp + sizeof(uintptr\_t));**  **int b = \*(int \*)(esp + 2 \* sizeof(uintptr\_t));**  **int c = \*(int \*)(esp + 3 \* sizeof(uintptr\_t));**  **int d = \*(int \*)(esp + 4 \* sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_max\_of\_four(a, b, c, d);**  **break;**  **}**  **default:**  **sys\_exit(-1);**  **}**  **}** |

user program에 의해 system call이 호출되면 system call handler는 적절한 system call 번호를 스택에서 pop하여 해당 switch 문을 실행한다. user address를 기준으로 스택에서 필요한 esp에서 pop하는 모든 경우에 대해 check\_validity를 수행하여 invalid한 경우 system call을 호출한 user program을 종료 시킨다.

1. **Additional System calls**

(1-1) src/lib/user/syscall.c/#define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3)

|  |
| --- |
| **#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; \**  **})** |

user program이 인자를 4개 가지는 system call을 호출하는 경우 필요한 system call 매크로를 정의한다.

(1-2) src/lib/user/syscall.c/int fibonacci(int n)

|  |
| --- |
| **int fibonacci(int n){**  **return syscall1(SYS\_FIBONACCI, n);**  **NOT\_REACHED ();**  **}** |

user program에 의해 system call 중에 fibonacci가 호출되면 1개의 인자를 가지는 system call 매크로를 통해 system call handler에게 SYS\_FIBONACCI 번호의 함수를 실행하도록 한다.

(1-3) src/lib/user/syscall.c/int max\_of\_four\_int(int a, int b, int c, int d)

|  |
| --- |
| **int max\_of\_four\_int(int a, int b, int c, int d) {**  **return syscall4(SYS\_MAXFOUR, a, b, c, d);**  **NOT\_REACHED ();**  **}** |

user program에 의해 system call 중에 max\_of\_four\_int가 호출되면 4개의 인자를 가지는 system call 매크로를 통해 system call handler에게 SYS\_MAXFOUR 번호의 함수를 실행하도록 한다.

(2-1) userprog/syscall.c/int sys\_fibonacci(int n)

|  |
| --- |
| **int sys\_fibonacci(int n){**  **if (n <= 0)**  **return 0;**  **else if (n == 1)**  **return 1;**  **return sys\_fibonacci(n - 2) + sys\_fibonacci(n - 1);**  **}** |

user program에 의해 system call 중에 fibonacci가 호출되면 system call handler가 sys\_fibonacci()함수를 실행시켜 n번째 fibonacci 수를 반환한다.

(2-2) userprog/syscall.c/int sys\_max\_of\_four(int a, int b, int c, int d)

|  |
| --- |
| **#define max(a, b)? ((a) > (b)) ? (a) : (b))**  **int sys\_max\_four\_of(int n){**  **return max(max(a, b), max(c, d));**  **}** |

user program에 의해 system call 중에 max\_of\_four\_int가 호출되면 system call handler가 sys\_max\_of\_four()함수를 실행시켜, 새로 정의한 max 매크로를 이용해 4가지 숫자 중 가장 큰 값을 반환한다.

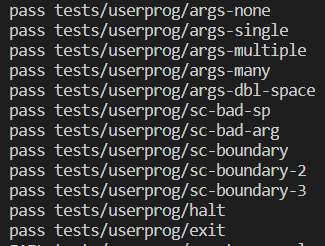
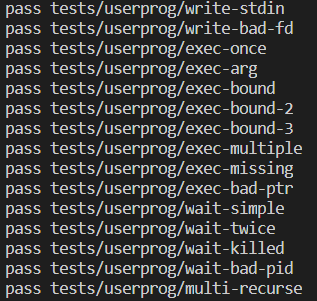
(3) example/additional.c/int main(int argc, char \*\*argv)

|  |
| --- |
| **int main (int argc, char \*\*argv)**  **{**  **int fibo = fibonacci(atoi(argv[1]));**  **int max4 = max\_of\_four\_int(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), atoi(argv[4]));**  **printf("%d %d\n", fibo, max4);**  **return EXIT\_SUCCESS;**  **}** |

user/user/syscall.c에 있는 fibonacci 함수를 호출하여 첫 번째 인자에 대해 피보나치 함수를 돌리고, 1~4번째 인자 중 가장 큰 값을 구하여 차례로 출력하는 어플리케이션 함수를 구현하였다.

1. **시험 및 평가 내용**

project1에서 요구하는 21가지 테스트를 모두 통과하였으며, 추가적으로 구현한 fibonacci와 max\_of\_four\_int를 따로 수행한 결과 또한 성공적임을 확인하였다.

***pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run "additional 10 20 62 40"*** 를 실행 한 경우 다음과 같이 결과가 출력됐다.

