운영 체제 과제 보고서

project #1~5

과목명: 운영 체제

담당 교수: 원유집 교수님

담당 조교: 오준택 조교님

제출 일자: 2018/ 04/ 17

2014004411 김시완

2016025714 이동윤

Github address:

https://github.com/gimsiwan/Operating_system.git

목차

1. Command Line Parsing

과제 목표:

커맨드 라인 파싱 기능 구현

과제 설명:

프로그램 이름과 인자를 구분하여 스택에 저장, 인자를 응용 프로그램에 전달하는 기능을 구현

a. process_execute 함수

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  char *token, *save_ptr;


  fn_copy = palloc_get_page (0);// alloc the memory to the variable , it will be using on copy the file_name
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);

  token = palloc_get_page(0);
  strlcpy(token, file_name, PGSIZE);
  token = strtok_r(token," ",&save_ptr);//using the strtok_r function to parsing the file_name

  tid = thread_create (token, PRI_DEFAULT, start_process, fn_copy);// create a thread with the parsing name

  palloc_free_page (token);

  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);

  return tid;
}
```

기존 process_execute 함수에서는 인자로 받아오는 file_name(argv 전체)을 parsing 하지 않은 채로 모두 thread_create 함수의 첫 번째 인자로 넘겨주었다.

이에 따라 file_name 전체를 이름으로 하는 프로세스가 실행 되었고, 띄어쓰기를 기준으로 첫번째 argument 만을 프로세스의 이름으로 실행시키기 위해 기존의 함수를 수정하였다. 예를 들어, 'echo x y z'를 실행 시 기존 함수에서는 'echo x y

z'이름을 가진 프로세스가 실행, 이를 'echo'의 이름을 가지고 x, y, z 를 인자로 가지는 스레드가 실행되게 변경.

```
  token = palloc_get_page(0);
  strlcpy(token, file_name, PGSIZE);
  token = strtok_r(token, " ", &save_ptr);//using the strtok_r function to parsing the file_name

  tid = thread_create (token, PRI_DEFAULT, start_process, fn_copy);// create a thread with the parsing name

  palloc_free_page (token);
```

palloc_get_page 함수를 이용하여 메모리 할당을 해준 token 인자에 file_name 을 복사 해준 다음, strtok_r 함수를 이용하여 delimiter 인 " "를 기준으로 첫 번째 문자열을 parsing 한 후 token 에 다시 저장해준다.

이는, 우리가 실행시킬 스레드의 이름이고, thread_create 함수의 첫 번째 인자에다 이를 넣어주어 해당 이름을 가진 thread 를 실행시켜주었다. 마지막으로 사용한 token 의 메모리를 해제해주므로 문자열 파싱 기능을 구현하였다.

b. start_process 함수

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;
  char *token, *save_ptr;
  char **parse;
  int i = 0;
  int count = 0;

  parse = palloc_get_page(0);

  for(token=strtok_r(file_name," ",&save_ptr); token!=NULL; token=strtok_r(NULL," ",&save_ptr)){
    parse[count] = palloc_get_page(0);
    strlcpy(parse[count],token,strlen(token)+1);
    count += 1;
  }
  // parsing the file_name with " " delimiter and store it on the variable named 'parse'
  // parse[0] is the real file_name, and from parse[1] is the argument of the function
  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 = load (parse[0], &if_.eip, &if_.esp); // load the file to memory with parsing name

  palloc_free_page (file_name);

  if (!success){
    thread_current()->loaded = -1;
    sema_up(&thread_current()->load_sema);
    thread_exit();
  }
  thread_current()->loaded = 2;
  sema_up(&thread_current()->load_sema);
  argument_stack(parse, count, &if_.esp); //store the argument on user stack
  // hex_dump((uintptr_t)if_.esp, if_.esp, PHYS_BASE - if_.esp, true);

  for(i=0;i<count;i++)
    palloc_free_page(parse[i]);
  palloc_free_page(parse);

  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

start_process 의 함수에서도 마찬가지로, 기존 함수에서는 프로그램을 메모리에 탑재해주는 load 함수에서 parsing 이 되지 않은 file_name 이 그대로 첫 번째 인자로 들어가게 된다.

```
  parse = palloc_get_page(0);

  for(token=strtok_r(file_name," ",&save_ptr); token!=NULL; token=strtok_r(NULL," ",&save_ptr)){
    parse[count] = palloc_get_page(0);
    strlcpy(parse[count],token,strlen(token)+1);
    count += 1;
  }
```

본 함수에서는, load 함수의 첫 번째 인자에 parsing 된 이름이 들어갈 수 있도록 strtok_r 함수를 이용하여 process_execute 함수와 같이 file_name 을 parsing 해 주었다. 다만, argument_stack 함수를 통하여 유저 스택에 프로그램 이름과, 인자들을 모두 저장해주기 위해 file_name 이 끝날 때까지 for 문을 돌려 parsing 을 해 주었으며, 이를 parse 이름을 가진 메모리 공간에 저장해 주었다. (parse[0]: 프로그램 이름, parse[1]….parse[n]: 인자 값)

```
  argument_stack(parse, count, &if_.esp);
  hex_dump((uintptr_t)if_.esp, if_.esp, PHYS_BASE - if_.esp, true);
```

위에서 parsing 을 한 결과 값을 넣어둔 parse 인자를 argument_stack 함수를 통해 user stack 에 저장해주었다. (argument_stack 함수의 코드 설명은 밑에 되어있다.)

또한 hex_dump 함수를 통해 스택 메모리에 들어간 값들을 보았고, 해당 기능이 잘 구현이 되었는지 이를 통해 확인할 수 있었다.

c. argument_stack 함수

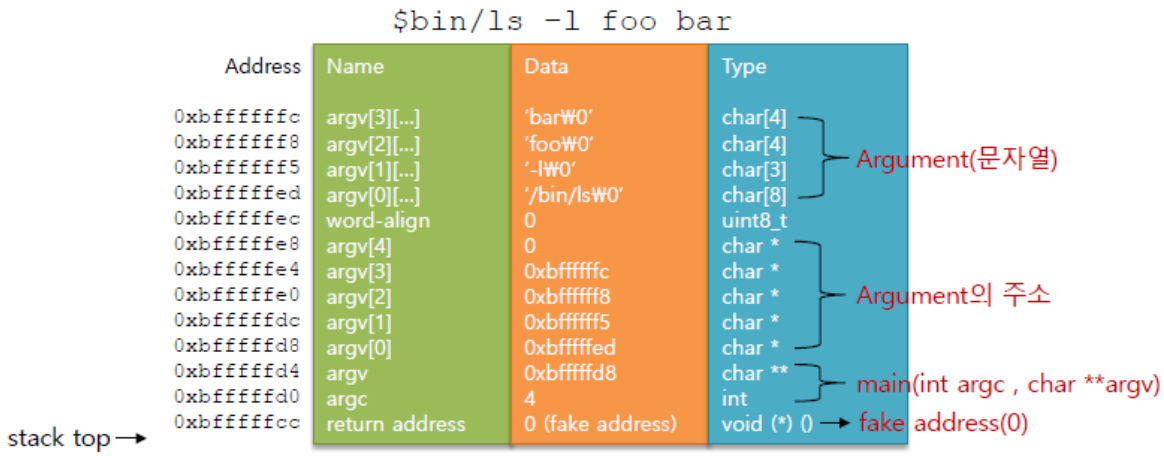
```
void argument_stack(char **parse, int count, void **esp){

  int i, word_align, amount_stack, pointer;
  int len = 0;
  int return_addr = 0x00000000;
  char *argv_addr;

  for (i = 0; i < count; i++)
    len += strlen(parse[i]);

  word_align = sizeof(int) - ((len + count) % sizeof(int));

  amount_stack = len + count + word_align + ((count + 1) * 4) + 12;

  *esp = *esp - amount_stack;

  memcpy(*esp, &return_addr, 4);
  memcpy(*esp + 4, &count, 4);

  argv_addr = *esp + 12;
  memcpy(*esp + 8, &argv_addr, 4);

  pointer = 12 + (count*4);

  memset(*esp + pointer, 0x0, 4);
  pointer += 4;
  memset(*esp + pointer, 0x0, word_align);

  for (i = 0; i < count; i++) {
    memcpy(*esp + word_align + pointer, parse[i], strlen(parse[i]));
    argv_addr = *esp+word_align+pointer;
    memcpy(*esp+12+(4*i), &argv_addr, 4);
    pointer += strlen(parse[i])+1;
  }

}
```

유저 스택에 프로그램 이름과, 인자 값들을 저장해주기 위해 argument_stack 함수를 구현하였다. start_process 함수에서 parsing 한 문자열을 저장한 parse 를 인자로 받았으며, 이 밖에 argument 의 개수를 나타내는 count, stack pointer 를 가리키는 주소 값인 esp 를 인자로 받았다.

stack 에는 아래 그림과 같이 return address, argc, argv... 순서로 데이터들이 저장 되어야 한다. 함수를 보면, 문자열의 길이, word_align 의 크기, 문자열의 개수, return address, argc, argv 의 메모리 크기를 모두 다 더해준 값을 amount_stack 변수로 지정해 주었고, 이 값만큼 esp 에서 빼주어 데이터가

저장되는 시작 주소를 찾아주었다. memcpy 함수를 통해 아래서부터 순서대로 return address, argc, argv... 값들을 넣어주었다.

여기서 인자로 받은 argument 의 주소를 모두 넣어준 다음, 끝에 null 의 값을 갖는 하나의 argument 를 더 넣어주었으며, 그 다음 padding 을 해주기 위해 0 의 값을 갖는 word_align 을 stack 에 저장해 주었다. 마지막으로, 실제 argument 에 들어가는 문자열들을 순서대로 저장해주었다.



실행 결과

```
siwankim@siwankim-VirtualBox:~/pintos/src/userprog/build$ pintos -v -- run 'echo
 x'
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/siwankim/pintos/src/
utils/pintos line 935.
Constant subroutine SIGVTALRM redefined at /home/siwankim/pintos/src/utils/pinto
s line 927.
squish-pty bochs -q
========================================================================
                      Bochs x86 Emulator 2.6.2
              Built from SVN snapshot on May 26, 2013
                Compiled on Mar 11 2018 at 17:19:04
========================================================================
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyb
oard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.........
Kernel command line: run 'echo x'
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 147 sectors (73 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo x':
bffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bfffff0  fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.........echo.x.|
echo: exit(0)
Execution of 'echo x' complete.
Kernel PANIC at ../../lib/kernel/list.c:251 in list_remove(): assertion `is_inte
rior (elem)' failed.
Call stack: 0xc002857c 0xc0028be4 0xc0028d2a 0xc00228d9 0xc00212d9 0xc002095a.
The `backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
```

'echo x'가 아닌 'echo' 이름을 가진 프로세스가 실행되었음을 다음과 같이 확인할 수 있었다. 또한, 메모리에 들어간 값들을 위의 그림과 같이 확인하여, argument_stack 함수가 제대로 구현이 되었고, x 의 값을 인자로 받음을 확인 할 수 있었다.

2. System Call

과제 목표:

시스템 콜 핸들러 및 시스템 콜(halt, exit, create, remove) 구현

과제 설명:

시스템 콜(halt, exit, create, remove)를 구현하고 시스템 콜 핸들러를 통해 호출한다.

a. check_address 함수

```
void check_address(void *addr){

    if((void *)0x08048000 > addr || addr >= (void *)0xc0000000){
        exit(-1);
    }//check the address is user space
}
```

사용하려는 주소가 유저영역(0x8048000~0xc0000000)을 벗어나면 안되기 때문에, 해당 주소가 이 영역을 이탈을 하였는지 확인을 해주는 함수를 구현하였다. 만약 이탈할 경우 강제 종료 시킨다.

b. get_argumnet 함수

```
void get_argument(void *esp, int *arg, int count){

    int i = 0;
    int *ptr = NULL;

    check_address((void *)(esp+4));// check the start user stack address is user space
    check_address((void *)(esp+count*4));//check the end user stack address is user space

    for(i=0;i<count;i++){
        ptr = (int *)(esp+i*4+4);
        arg[i] = *ptr;// store the argument on arg
    }
}
```

argument_stack 함수를 통해 유저 스택에 존재하는 스택 프레임의 인자 들을 커널에 복사하도록 구현해주었다. esp 값은 system call number 를 가리키고, 다음 위치의 주소부터 인자 값들을 가리키므로 인자를 가리키는 시작 주소인

esp+4 부터 끝 주소인 esp + count*4 까지의 주소 위치가 유저 영역에 해당하는 위치인지 check_address 함수를 통하여 먼저 확인을 해준다.

그 다음, esp+4 부터 순서대로 주소에 들어 있는 값을 arg 인자에 저장을 해준다.

c. syscall_handler 함수

```
static void
syscall_handler (struct intr_frame *f)
{
  check_address((void*)f->esp);
  
  int syscall_nr = *(int *)f->esp;
  int arg[5];
  switch(syscall_nr){
    case SYS_HALT:
      halt();
      break;
    case SYS_EXIT:
      get_argument(f->esp,arg,1);
      exit(arg[0]);
      break;
    case SYS_CREATE:
      get_argument(f->esp,arg,2);
      check_address((void*)arg[0]);
      f->eax = create((const char*)arg[0],(unsigned)arg[1]);
      break;
    case SYS_REMOVE:
      get_argument(f->esp,arg,1);
      check_address((void*)arg[0]);
      f->eax = remove((const char*)arg[0]);
      break;
    case SYS_EXEC:
      get_argument(f->esp, arg, 1);
      check_address((void*)arg[0]);
      f->eax = exec((const char *)arg[0]);
      break;
    case SYS_WAIT:
      get_argument(f->esp, arg, 1);
      f->eax = wait(arg[0]);
      break;
  }
}
```

intr_frame 구조체에 저장되어 있는 esp 값을 통해 호출한 system call 을 파악한다. 유저 스택 포인터 주소와 시스템 콜 인자가 가리키는 주소가 유효주소인지 check_address 함수를 통해 확인하였으며, switch 함수를 통하여 파악한 system call number 을 통해 해당 시스템 콜의 서비스 루틴을 호출 하도록 구현하였다. 시스템 콜 함수의 리턴 값들을 인터럽트 프레임 구조체의 eax 인자에 저장되도록 구현해주었다.

d. halt/ exit/ create/ remove 함수

```
void halt(void){
    shutdown_power_off();
}

void exit(int status){

    struct thread *current_thread=thread_current();
    current_thread->exit_status = status;
    printf("%s: exit(%d)\n",current_thread->name,status);

    thread_exit();
}

bool create(const char *file, unsigned initial_size){
    return filesys_create(file, initial_size);
}

bool remove(const char *file){
    return filesys_remove(file);
}
```

halt 함수는 핀토스를 종료 시키는 시스템 콜로 <devices/shutdown.h>에 위치한 핀토스를 종료시키는 함수인 shutdown_power_off 를 호출함으로써 구현해주었다.

exit 함수는 스레드를 종료 시키는 시스템 콜로 현재 실행중인 스레드 구조체를 가져와 스레드의 이름과, exit status 를 출력해주었고, <threads/thread.h>에 위치한 thread_exit 함수를 이용하여 스레드를 종료 시켜주었다.

create 함수는 파일을 생성하는 시스템 콜로 <filesys/filesys.h>에 위치한 filesys_create 함수를 호출하여 구현해주었다.

remove 함수는 파일을 제거하는 시스템 콜로 <filesys/filesys.h>에 위치한 filesys_remove 함수를 호출하여 구현해주었다.

제공된 시스템 콜 관련 API 를 참조하여 쉽게 구현할 수 있었다.

실행 결과

1. halt system call 호출

```
siwankim@siwankim-VirtualBox:~/pintos/src/userprog/build$ pintos run 'halt'
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/siwankim/pintos/src/
utils/pintos line 935.
Constant subroutine SIGVTALRM redefined at /home/siwankim/pintos/src/utils/pinto
s line 927.
squish-pty bochs -q
========================================================================
                       Bochs x86 Emulator 2.6.2
               Built from SVN snapshot on May 26, 2013
                 Compiled on Mar 11 2018 at 17:19:04
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyb
oard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading.........
Kernel command line: run halt
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 147 sectors (73 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'halt':
bffffe0                00 00 00 00-01 00 00 00 f0 ff ff bf |    ............|
bfffff0  fb ff ff bf 00 00 00 00-00 00 00 68 61 6c 74 00 |...........halt.|
Timer: 173 ticks
Thread: 0 idle ticks, 120 kernel ticks, 56 user ticks
hdb1 (filesys): 40 reads, 0 writes
Console: 727 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
========================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
========================================================================
```

2. exit system call 호출

```
siwankim@siwankim-VirtualBox:~/pintos/src/userprog/build$ pintos -q run 'exit 0'
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/siwankim/pintos/src/
utils/pintos line 935.
Constant subroutine SIGVTALRM redefined at /home/siwankim/pintos/src/utils/pinto
s line 927.
squish-pty bochs -q
========================================================================
                       Bochs x86 Emulator 2.6.2
               Built from SVN snapshot on May 26, 2013
                 Compiled on Mar 11 2018 at 17:19:04
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyb
oard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading.........
Kernel command line: -q run 'exit 0'
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 147 sectors (73 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'exit 0':
bffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bfffff0  fe ff ff bf 00 00 00 00-00 65 78 69 74 00 30 00 |.........exit.0.|
exit: exit(57)
Execution of 'exit 0' complete.
Timer: 203 ticks
Thread: 0 idle ticks, 148 kernel ticks, 57 user ticks
hdb1 (filesys): 41 reads, 0 writes
Console: 783 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
========================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
========================================================================
```

test 파일인 exit.c 파일을 열어보면

```
void
test_main (void)
{
  exit (57);
  fail ("should have called exit(57)");
}
```

과 같이 argument 를 숫자 57 을 받는다. 따라서 test 결과에 argument 가 0 이 아닌 57 이 들어가게 됨을 확인 할 수 있었다.

3. create system call 호출

```
siwankim@siwankim-VirtualBox:~/pintos/src/userprog$ pintos -q run 'create filete
st 30'
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/siwankim/pintos/src/
utils/pintos line 935.
Constant subroutine SIGVTALRM redefined at /home/siwankim/pintos/src/utils/pinto
s line 927.
squish-pty bochs -q
========================================================================
                    Bochs x86 Emulator 2.6.2
            Built from SVN snapshot on May 26, 2013
              Compiled on Mar 11 2018 at 17:19:04
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyb
oard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading.........
Kernel command line: -q run 'create filetest 30'
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 147 sectors (73 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'create filetest 30':
bffffd0  00 00 00 00 03 00 00 00-dc ff ff bf ed ff ff bf |................|
bffffe0  f4 ff ff bf fd ff ff bf-00 00 00 00 00 63 72 65 |.............cre|
bfffff0  61 74 65 00 66 69 6c 65-74 65 73 74 00 33 30 00 |ate.filetest.30.|
create: exit(0)
Execution of 'create filetest 30' complete.
Timer: 232 ticks
Thread: 0 idle ticks, 152 kernel ticks, 83 user ticks
hdb1 (filesys): 59 reads, 4 writes
Console: 897 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
========================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
========================================================================
```

마찬가지로 test 파일인 create-normal.c 파일을 열어보면

```
void
test_main (void)
{
  CHECK (create ("quux.dat", 0), "create quux.dat");
}
```

과 같이 구현 되어있다. 따라서 우리가 인자로 준 파일 이름과 사이즈가 아닌 "quux.dat" 이름의 사이즈 0 파일이 만들어진다.

3. Hierarchical Process Structure

과제 목표:

프로세스 간의 부모와 자식 관계를 구현하고, 부모가 자식 프로세스의 종료를 대기하는 기능 구현

과제 설명:

프로세스의 정보에 부모와 자식 필드를 추가하고, 이를 관리하는 함수를 제작한다.

a. thread 구조체

```
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. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
    struct thread *parent;
    struct list_elem child_elem;
    struct list child_list;
    int loaded;
    bool exited;
    bool wait;
    struct semaphore exit_sema;
    struct semaphore load_sema;
    int exit_status;

    struct file **fdt;                  /* pointer of file descripter table */
    int next_fd;                        /* number of next empty space in file descripter table */

    struct file *executing_file;
  };
```

thread 구조체에 부모 프로세스의 디스크립터를 나타내는 parent 구조체, 자식 리스트의 element 를 나타내는 child_elem 구조체, 자식 리스트 child_list 구조체, 프로세스의 메모리 탑재 유무를 나타내는 loaded 인자, 종료 유무를 나타내는 exited 인자, wait 상태 인지를 나타내는 wait 인자, wait 상태를 구현하기 위해 쓰이는 exit_sema, load_sema 두개의 semaphore, 마지막으로 exit 호출 시의 상태를 나타낼 exit_status 까지 추가해준다.

b. init_thread / thread_create 함수

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;
  list_push_back (&all_list, &t->allelem);

  list_init(&t->child_list);
}
```

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  ...........................
  ...........................
  ...........................

  t->parent = thread_current ();
  t->loaded = 0;
  t->exited = false;
  t->wait = false;
  sema_init(&t->exit_sema,0);
  sema_init(&t->load_sema,0);

  list_push_back (&thread_current()->child_list, &t->child_elem);
```

init_thread 와 thread_create 함수를 통해 추가한 thread 구조체의 인자들을 초기화 시켜준다. init_thread 함수에서는 list_init 함수를 통해 child_list 를 초기화 시켜주고, thread_create 함수에서는 그 외 나머지 인자들을 초기화 시켜준다.

c. get_child_process 함수

```
struct thread *get_child_process(int pid){

  struct thread *cur = thread_current();
  struct list_elem *elem = list_begin(&cur->child_list);

  for(;elem != list_end(&cur->child_list);elem = list_next(elem)){
    struct thread *child_thread = list_entry(elem, struct thread, child_elem);
    if(child_thread->tid == pid)
      return child_thread;
  }

  return NULL;
}
```

자식 리스트를 검색하여 해당 pid 에 맞는 프로세스 디스크립터가 있는지 검색하는 함수이다. list 로 구성 되어 있는 자식 스레드들 중 list_entry 를 사용하여 해당 pid 값의 스레드를 찾아주었고 만약 존재하지 않을 시에는 NULL 값을 return 하도록 구현 해 주었다.

d. remove_child_process 함수

```
void remove_child_process(struct thread *cp){
  list_remove(&cp->child_elem);
  palloc_free_page(cp);
}
```

list_remove 함수를 통해 해당 스레드를 자식 리스트에서 제거 해주는 함수를 구현하였다. 제거된 스레드는 palloc_free_page 함수를 통해 할당된 메모리 또한 해제 시켜 주었다.

e. exec 함수

```
tid_t exec(const char *cmd_line){

  struct thread *child;
  tid_t pid = process_execute(cmd_line);

  child = get_child_process(pid);
  if(!child) return -1;

  if(child->loaded == 0)
    sema_down(&child->load_sema);

  if(child->loaded == 1)
    return -1;
  return pid;
}
```

명령어를 통해 명령어에 해당하는 프로그램을 수행하는 자식 프로세스를 생성하고, 해당 프로세스가 메모리에 올라갈 때까지 semaphore 를 사용 하여 부모 프로세스를 대기 시켜준다. 만약 메모리에 탑재 되지 않았을 경우 -1 를 return 해주고, 정상적으로 탑재 되었을 경우에는 해당 프로세스의 pid 값을 return 해준다.

메모리에 정상 적으로 탑재 되었을 경우, 부모 프로세스의 실행을 재개 시켜 주기 위해 아래와 같이,

```
success = load (parse[0], &if_.eip, &if_.esp); // load the file to memory with parsing name

palloc_free_page (file_name);

if (!success){
    thread_current()->loaded = 1;
    sema_up(&thread_current()->load_sema);
    thread_exit();
}
thread_current()->loaded = 2;
sema_up(&thread_current()->load_sema);
argument_stack(parse, count, &if_.esp); //store the argument on user stack
```

load 함수를 통해 메모리에 탑재를 해준 후 sema_up 함수를 구현해주고, 스레드의 탑재 유무를 구조체 loaded 인자를 통해 바꾸어 준다.

```
        case SYS_EXEC:
            get_argument(f->esp, arg, 1);
            check_address((void *)arg[0]);
            f->eax = exec((const char *)arg[0]);
            break;
```

또한 위와 같이 syscall handler 에 exec 함수의 시스템 호출 기능을 추가 시켜준다.

f. process_wait 함수

```
int
process_wait (tid_t child_tid)
{
    int status;
    struct thread *child = get_child_process(child_tid);
    if(!child) return -1;
    if(child->wait) return -1;
    child->wait = true;
    if(!child->exited)
        sema_down(&child->exit_sema);
    status = child->exit_status;
    remove_child_process(child);

    return status;
}
```

자식 프로세스의 실행이 완료 될 때까지, 부모 프로세스의 실행을 대기 시켜주기 위해 process_wait 함수를 구현하였다. get_child_process 함수를 통해 pid 값을 통해 자식 프로세스를 찾아준 다음, 해당 프로세스의 종료의 대기를 기다리고 있는 process 가 있는지 한 번 확인해 주고, 만약 해당 프로세스가 종료되지 않은 상태인 경우 sema_down 함수를 이용하여 해당 프로세스가 종료 되기를 기다려준다. 자식 프로세스가 종료 되었을 경우 다시 부모 프로세스는 재개 될 것이고, 이 때 자식 프로세스를 remove_child_process 를 통해 삭제 시켜 주고 종료 상태를 return 값으로 반환한다.

자식 프로세스가 종료될 경우 다시 부모 프로세스가 실행되게 구현해 주어야 하므로

```
void
thread_exit (void)
{
  ASSERT (!intr_context ());

#ifdef USERPROG
  process_exit ();
#endif

  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current()->exited = true;
  sema_up(&thread_current()->exit_sema);
  thread_current ()->status = THREAD_DYING;
  schedule ();
  NOT_REACHED ();
}
```

스레드가 종료 될 경우 호출 되는 thread_exit 함수에 sema_up 함수를 넣어준다. 또한, return 값으로 자식 프로세스의 종료 상태를 return 하기 위해서는

```
void exit(int status){

    struct thread *current_thread =thread_current();
    current_thread->exit_status = status;
    printf("%s: exit(%d)\n",current_thread->name,status);

    thread_exit();
}
```

exit 함수에서 인자 값인 status 를 스레드 구조체의 exit_stauts 인자에 저장 시켜준다. 마지막으로, process_wait 함수에서 remove_child_process 를 통해 자식 프로세스를 삭제 시켜주었으므로,

```
void
thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();

  ASSERT (intr_get_level () == INTR_OFF);

  /* Mark us as running. */
  cur->status = THREAD_RUNNING;

  /* Start new time slice. */
  thread_ticks = 0;

#ifdef USERPROG
  /* Activate the new address space. */
  process_activate ();
#endif

  /* If the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
    }
}
```

thread_schedule_tail 함수 안의 마지막 if 문에서 palloc_free_page(prev)라는 기존의 스레드 메모리를 해제하는 기능을 하는 명령어를 지워 주어 메모리 해제가 두 번 이루어지지 않도록 해준다.

g. wait 함수

```
int wait(tid_t tid){
    return process_wait(tid);
}
```

위에서 구현한 process_wait 함수를 통해 wait 시스템 콜을 만들어 주었다. 해당 시스템 콜을 사용하기 위해 마찬가지로 syscall handler 에 해당 시스템 콜이 호출 되었을 경우 처리 될 수 있도록

```
        case SYS_WAIT:
            get_argument(f->esp, arg, 1);
            f->eax = wait((tid_t)arg[0]);
            break;
```

과 같이 함수 구현을 해주었다.

4. File Descriptor

과제 목표:

파일 디스크립터 및 관련 시스템 콜 구현

과제 설명:

파일 입출력을 위해서는 파일 디스크립터의 구현 기능이 필요

a. thread 구조체

```
struct file **fdt;          /* pointer of file descripter table */
int next_fd;                /* number of next empty space in file descripter table */
```

스레드 구조체에 파일 디스크립 테이블을 나타내는 구조체 포인터 fdt 인자와 다음 비어 있는 테이블 번호를 가리키는 next_fd 인자를 추가 시켜주었다.

마찬가지로 새로 구성한 구조체 인자를 초기화 시켜 주기 위해 thread_create 함수에서

```
t->fdt = (struct file**)calloc(1,sizeof(struct file*)*MAX_FILE);
t->next_fd = 2; /* 0 is stdin, 1 is stdout */
```

과 같이 초기화를 시켜주었다. 최대 파일 개수는 MAX_FILE(256 개)로 설정해 주었으며, 파일 디스크립터 번호 0 번에는 표준 입력, 1 번에는 표준 출력이 있기 때문에, 초기 next_fd 는 2 로 설정해 주었다.

b. process_add_file 함수

```
int process_add_file(struct file *f){

	int ret;
	int i = 0;
	struct thread *t = thread_current();

	if(t->next_fd >= MAX_FILE)
		return -1;			// no space to store the file in file descripter table

	if(t->fdt[t->next_fd] != NULL)
		return -1;

	t->fdt[t->next_fd] = f;
	ret = t->next_fd;

	for(i = t->next_fd;i<MAX_FILE;i++){
		if(t->fdt[i] == NULL) break;
	}
	t->next_fd = i;

	return ret;
}
```

파일 디스크립터 테이블에 파일을 추가하는 함수로 인자로 받은 파일을 next_fd 위치의 테이블에 넣어주고 next_fd 의 값을 다음 비어 있는 next_fd 로 바꿔주도록 구현하였다.

c. process_get_file 함수

```
struct file *process_get_file(int fd){

	if(fd>=MAX_FILE) return NULL;

	return thread_current()->fdt[fd];
}
```

해당 번호의 파일 디스크립터 테이블에 저장되어 있는 파일을 return 하는 함수를 구현하였다.

d. process_close_file 함수

```
void process_close_file(int fd){

	if(fd>=MAX_FILE) return;

	if(thread_current()->fdt[fd] != NULL)
		file_close(thread_current()->fdt[fd]);
	thread_current()->fdt[fd] = NULL;
	if(thread_current()->next_fd>fd)
		thread_current()->next_fd = fd;
}
```

해당 번호의 파일 디스크립터 테이블에 저장 되어 있는 파일을 file_close 함수를 통해 닫아주고, 해당 파일 디스크립터 테이블은 NULL 값으로 초기화 시켜준다. 또한 해당 번호가 현재의 next_fd 보다 작을 경우 next_fd 의 값을 해당 번호로 바꾸어 주어, next_fd 값이 현재 비어 있는 테이블 번호 중 가장 낮은 번호가 되게끔 지정해준다.

e. process_exit 함수

```
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;
  int i;

  if(cur->executing_file){
    file_close(cur->executing_file);// if it has executing file, close the file(it allows the file writing from other process)
  }

  for(i = MAX_FILE-1; cur->next_fd>2;i--){//close the all open file
    process_close_file(i);
    if(i==2) i = MAX_FILE - 1;
  }

  free(cur->fdt);//and free the file descripter table memory

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }
}
```

process_exit 함수에서 프로세스가 종료 될 경우 현재 열려 있는 file 들을 모두 닫아주도록 구현 해준다.

while 문을 사용하여 next_fd 값이 2 보다 작거나 같은 상태가 될 때까지 계속 process_close_file 을 호출하여 모든 파일이 닫히도록 구현해 주었다. 마지막으로 모든 file 들이 닫히게 되면 처음 할당 해주었던 파일 디스크립터 테이블의 메모리를 해제 시켜준다.

f. open/ filesize/ read/ write/ seek/ tell/ close 함수

```
int open(const char *file){

    struct file *f = filesys_open(file);
    int fd;

    if(!f) return -1;
    fd = process_add_file(f);

    return fd;
}
```

open 시스템 콜은 filesys_open 을 이용하여 해당 이름의 파일을 열어 주는 함수이다. process_add_file 을 통해 파일을 파일 디스크립터 테이블에 추가하고, 저장한 위치인 파일 디스크립터 테이블 번호를 return 값으로 반환한다.

```
int filesize(int fd){

    struct file *f = process_get_file(fd);
    if(f == NULL) return -1;

    return file_length(f);
}
```

filesize 시스템 콜은 해당 파일 디스크립터 테이블 번호에 위치한 파일의 크기를 반환하는 함수로, process_get_file 함수를 통해 해당 파일을 가져온 다음, file_length 함수를 통해 filesize 를 가져와 return 하게끔 구현하였다.

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

    unsigned i;
    int byte;

    if(fd == STDOUT_FILENO)
        return -1;
    else if(fd == STDIN_FILENO){
        uint8_t *stdin_buffer = (uint8_t *)buffer;

        for(i=0;i<size;i++)
            stdin_buffer[i] = input_getc();
        return size;
    }

    lock_acquire(&filesys_lock);
    struct file *f = process_get_file(fd);

    if(!f){
        lock_release(&filesys_lock);
        return -1;
    }
    byte = file_read(f, buffer, size);
    lock_release(&filesys_lock);

    return byte;
}
```

read 시스템 호출은 해당 파일을 size 크기만큼 읽은 다음 buffer 에 저장하는 함수로 먼저 read 시 읽고 있는 파일에 대해 동시 접근이 이루어질 수 있으므로 filesys_lock 이라는 전역 변수를 하나 선언해주었다. 해당 lock 은

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");

  lock_init(&filesys_lock);
}
```

syscall_init 함수에서 lock_init 함수를 통해 초기화를 시켜주며, process_get_file 을 통해 파일을 가져온 다음, file_read 를 할 때까지 lock_acquire 함수와 lock_release 함수를 통해 lock 을 걸어준다.

만약 파일 디스크립터 번호가 STDOUT_FILENO(표준 출력, 1 번)일 경우 read 가 일어날 수 없으므로 -1 값을 return 해주고 STDIN_FILENO(표준 입력, 0 번)일 경우

input_getc 함수를 통해 키보드의 입력을 buffer 로 받아온다. return 값은 읽어온 파일의 크기를 반환해주게 구현하였다.

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

  int byte;

  if(fd == STDIN_FILENO)
    return -1;
  else if(fd == STDOUT_FILENO){
    putbuf((const char*)buffer, size);
    return size;
  }

  lock_acquire(&filesys_lock);
  struct file *f = process_get_file(fd);
  if(!f){
    lock_release(&filesys_lock);
    return -1;
  }

  byte = file_write(f, buffer, size);
  lock_release(&filesys_lock);

  return byte;
}
```

write 시스템 호출은 버퍼에 저장된 값을 size 만큼 파일에 써주는 함수로 read 와 마찬가지로 쓰고 있는 파일에 대해 동시 접근이 이루어질 수 있으므로 lock 을 사용하여 파일에 write 가 되고 있는 상태에서 다른 접근이 이루어지지 않도록 막아주었다. 만약 파일 디스크립터 번호가 STDIN_FILENO(표준 입력, 0 번)일 경우 write 가 이루어 질 수 없으므로 -1 값을 return 하게 구현하였으며, STDOUT_FILENO(표준 출력, 1 번)일 경우 putbuf 함수를 이용하여 문자열을 화면에 출력하게끔 구현해 주었다. 마찬가지로 return 값은 파일에 기록한 데이터의 크기이다.

```
void seek(int fd, unsigned position){

  struct file *f = process_get_file(fd);

  if(f){
    file_seek(f, position);
  }
}
```

seek 시스템 콜은 파일의 offset 을 변경해주는 함수로 file_seek 함수를 통해 position 만큼 변경 되도록 구현하였다.

```
unsigned tell(int fd){

    struct file *f = process_get_file(fd);

    if(f){
        return file_tell(f);
    }
    return -1;
}
```

tell 시스템 콜은 파일 디스크립터 번호를 통해 해당 파일을 찾는 함수로서 위에서 구현한 process_get_file 함수를 이용하여 구현하였다.

```
void close(int fd){
    process_close_file(fd);
}
```

close 시스템 콜은 해당 번호의 파일 디스크립터 테이블에 위치한 파일을 종료시키는 함수로서 process_close_file 함수를 이요하여 구현하였다.

```
        case SYS_OPEN:
            get_argument(f->esp,arg,1);
            check_address((void*)arg[0]);
            f->eax = open((const char*)arg[0]);
            break;
        case SYS_FILESIZE:
            get_argument(f->esp,arg,1);
            f->eax = filesize(arg[0]);
            break;
        case SYS_READ:
            get_argument(f->esp,arg,3);
            check_address((void*)arg[1]);
            f->eax = read (arg[0],(void*)arg[1],(unsigned)arg[2]);
            break;
        case SYS_WRITE:
            get_argument(f->esp,arg,3);
            check_address((void*)arg[1]);
            f->eax = write(arg[0],(const void*)arg[1],(unsigned)arg[2]);
            break;
        case SYS_SEEK:
            get_argument(f->esp,arg,2);
            seek(arg[0], (unsigned)arg[1]);
            break;
        case SYS_TELL:
            get_argument(f->esp,arg,1);
            f->eax = tell(arg[0]);
            break;
        case SYS_CLOSE:
            get_argument(f->esp,arg,1);
            close(arg[0]);
            break;
```

마찬가지로 system call handler 에 위와 같은 함수를 추가함으로써 해당 시스템 콜이 호출 되었을 경우 해당 함수가 호출 되도록 구현을 해주었다.

※ userprog/exception.c 파일에서 page_fault 함수 안에 exit(-1)을 일시적으로 추가 시켜 주어 아직 구현이 되지 않아 잠시 동안 발생 되는 오류 메시지 출력을 막아주었다.

5. Denying Write to Executable

과제 목표:

실행 중인 사용자 프로세스의 프로그램 파일에 다른 프로세스가 데이터를 기록하는 것을 방지

과제 설명:

실행 중인 사용자 프로그램의 프로그램 파일이 다른 프로세스에 의해 변경되면, 프로그램이 원래 예상했던 데이터와 다르게 변경 된 데이터를 디스크에서 읽어 올 가능성이 있다. 프로그램 파일이 프로세스에 의해 접근 되고 있을 때에는 프로그램 파일이 변경 되는 것을 방지할 수 있도록 수정한다.

a. thread 구조체

```
struct file *executing_file;
```

thread 구조체에 현재 실행 중인 파일을 나타네는 executing_file 구조체 포인터를 추가 시켜준다.

b. load 함수

```
file = filesys_open (file_name);
if (file == NULL)
  {
    lock_release(&filesys_lock);
    printf ("load: %s: open failed\n", file_name);
    goto done;
  }
t->executing_file = file;
file_deny_write(file);
lock_release(&filesys_lock);

/* Read and verify executable header. */
if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
    || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
    || ehdr.e_type != 2
    || ehdr.e_machine != 3
    || ehdr.e_version != 1
    || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
    || ehdr.e_phnum > 1024)
  {
    printf ("load: %s: error loading executable\n", file_name);
    goto done;
  }
```

load 함수 안에서 실행 할 파일이 열릴 때 file_deny_write 함수를 호출 하여 파일의 데이터가 변경 되는 것을 예방해준다.

c. process_exit 함수

```
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;
  int i;

  if(cur->executing_file){
    file_close(cur->executing_file);
  }

  for(i = MAX_FILE-1; cur->next_fd>2; i--){
    process_close_file(i);
    if(i==2) i = MAX_FILE - 1;
  }
```

process_exit 함수에서 프로세스가 종료될 때 현재 스레드에서 실행 중인 파일이 존재하면 file_close 함수를 통해 해당 파일을 종료 시켜준다. file_close 함수에는 file_allow_write 함수가 존재하여 위에서 file_deny_write 함수를 통해 막아주었던 변경을 다시 변경 될 수 있도록 허락해준다.

6. 최종 결과

make check 호출 결과:

```
perl -I../.. ../../tests/filesys/base/syn-write.ck tests/filesys/base/syn-write tests/filesys/base/syn-write.result
pass tests/filesys/base/syn-write
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/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass 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
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
```

```
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
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
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
```

76 개의 test 모두 통과

make grade 호출 결과:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                  Pts Max  % Ttl  % Max
----------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality       108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness           88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                 1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                  30/ 30  30.0%/ 30.0%
----------------------------------------- --- --- ------ ------
Total                                             100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Total 100% test 통과