User Program 과제 보고서

운영체제 (ELE-3021 12781)

교수명 : 원유집

조교명 : 오준택

학생명 :

서창범 2014004648, 서문은지 2016025487

제출일자 : 2018 년 04 월 18 일

User Program

- 커맨드 라인 파싱

1. 과제 설명

pintos는 프로그램과 인자를 구분하지 못하는 구조이다.

예) ls -a 를 pintos는 'ls -a'를 하나의 프로그램명으로 인식한다.

그래서 이번과제를 통하여 프로그램 이름과 인자를 구분하여 스택에 저장하고, 인자를 응용프로그램에게 전달하는 기능을 구현한다.

2. 함수 수정 : 문자열을 파싱하고 user 스택에 인자 값이 저장 되도록 수정한다.

(1) process_excute()

strtok_r() 함수를 통해 인자로 들어온 file_name을 띄어쓰기로 구분하여 strtok_r() 함수를 한 번 실행하여 0번째 token으로 분리된 프로그램 이름을 thread_create의 첫번째 인자(=스레드 이름)로 넣어준 후, 해당 프로그램의 이름으로 스레드(프로세스)를 생성한다.

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

  strtok_r(file_name, " ", &save_ptr);


  /* Create a new thread to execute FILE_NAME. */
        //question token[0] addres??
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

(2) start_process()

위와 동일하게 strtok_r() 함수를 이용하여, file_name 문자열을 파싱하였다. 그리고 load()함수의 인자로 넣어, 디스크에서 file_name (예: echo)에 해당하는 실행파일을 찾아서 메모리에 적재한다.

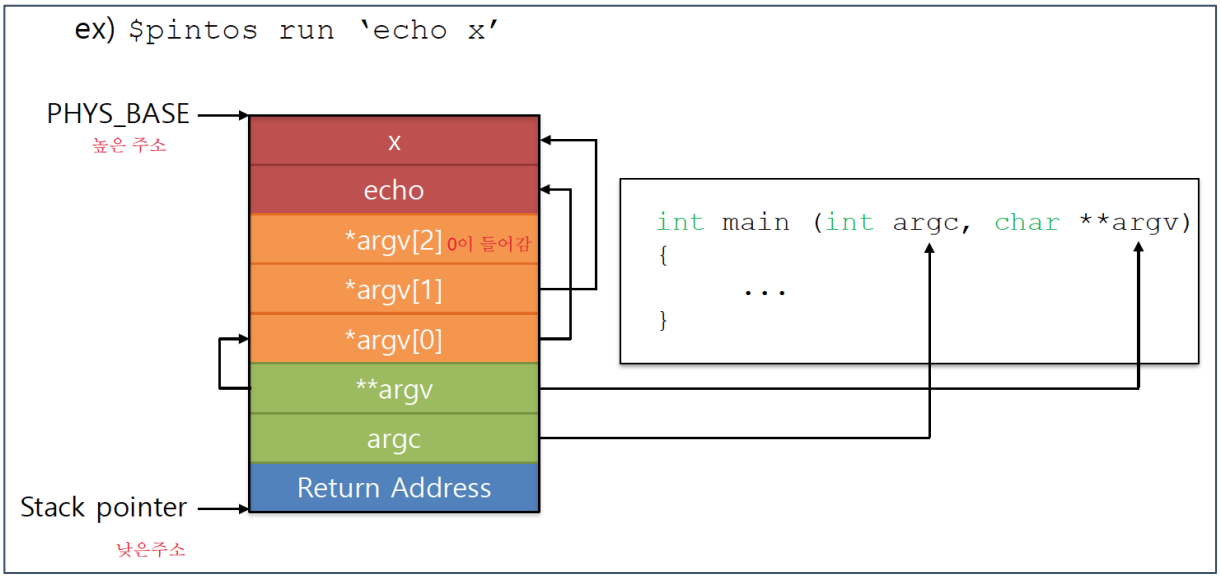
```
success = load(file_name, &if_.eip, &if_.esp);

token_count++;
while(strtok_r(NULL, " ", &save_ptr))
{
  token_count++;
}

parse = ((char**)malloc((sizeof(char*))*token_count));
```

while문을 통해 띄어쓰기를 기준으로 파싱된 토큰들의 갯수를 세어 token_count에 argc를 저장하고, parse라는 포인터 변수에 argv 배열을 저장할 메모리공간 시작 주소를 동적 할당한다. strlcpy (file_name, file_name_, PGSIZE); 를 통해 분리된 file_name을 다시 원래상태로 복구한 후, argv를 저장하기 위해 parse배열에 file_name을 다시 tokenize하여 token들의 주소 값을 저장한다. 그리고 argument_stack함수의 인자로 parse, token_count, esp의 주소를 넣어주어, argument_stack함수를 통해 user스택에 파싱된 token들을 저장하여 argv와 argc가 유저 프로그램에게 전달 되도록 한다.

```
for(token = strtok_r(file_name, " ", &save_ptr), i = 0; token != NULL; token
= strtok_r(NULL, " ", &save_ptr), i++)
{
  parse[i] = token;
}

argument_stack(parse, token_count, &if_.esp);
```

(3) void argument_stack(char **parse, int count, void **esp)

- parse: 프로그램이름과 인자가 저장되어 있는 메모리 공간, argv
- count: 인자의 개수, argc
- esp: 스택의 꼭대기를 가리키는 주소
- argument_stack() : user 스택에 argv와 argc를 저장하여 인자를 전달하는 함수



이 그림과 같이 user 스택에 token들을 저장하는 함수를 구현 하였다.

먼저, 각 token들을 가리키는 주소를 담는 arg_ptr[count]라는 포인터 배열을 만들었다. 이 배열을 만든 이유는 미리 token들의 주소 값을 저장해 놓아, argv[i]에 저장할 때 쉽게 하기 위함이다. 그리고 parse[i]에 저장되어 있는 token들(프로그램 이름 및 인자(문자열)을 스택에 저장한다.

```
void argument_stack(char **parse ,int count , void **esp)
{
  int i, align_size;
  void *arg_ptr[count];

  for (i = count -1; i>=0; i--)
  {
    *esp -= strlen(parse[i])+1;
    arg_ptr[i] = *esp;
    strlcpy(*esp, parse[i], strlen(parse[i])+1);
  }
```

파싱된 token들을 저장한 후, word-align(padding)을 해준다.
0x3(0011)과 esp랑 AND연산한 값만큼 패딩을 해주면 esp주소가 32bit단위로 addressing 가능해져서 memory I/O효율이 유리해지게 된다. 메모리 캐싱이 메모리를 읽을 때 4바이트 단위로 일어나기 때문이다.

```
align_size = (((int)*esp)&0x00000003);
*esp -= align_size; //(뒤의 2자리를 00으로만듬)
memset(*esp, 0x00, align_size);
```

패딩 후, argv[count] = 0 을 저장한 후(argv 마지막 + 1 자리), 앞에서 만든 arg_ptr[i]를 이용해 프로그램 이름 및 인자를 가리키는 주소를 저장한다. 그리고, 차례대로 argv, argc, fake address (NULL)을 저장한다.

```
*esp -= sizeof(char *);
*(char **)*esp = 0;

for(i = count-1; i >= 0; i--)
{
  *esp -= sizeof(char *);
  *(char **)*esp = arg_ptr[i];
}

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

*esp -= sizeof(int);
*(int *)*esp = count;

*esp -= sizeof(void *);
*(void **)*esp = 0;

}
```

3. 결과

1번 과제의 필요한 기능을 구현하고 나서

hex_dump((uintptr_t)if_.esp, if_.esp, (size_t)(PHYS_BASE-if_.esp), true); 를 이용하여 프로그램의 스택 메모리를 출력해보았다. 다음은 pintos -v – run 'echo x'의 결과값이다.

1번 과제(커맨드 라인 파싱 기능 구현) 가 완료된 것을 확인하였다.

```
Boot complete.
Executing 'echo x':
Execution of 'echo' complete.
bffffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bfffffff0  fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.........echo.x.|
system call!
```

User Program
- 시스템 콜

1. 과제설명

pintos는 시스템 콜 핸들러가 구현되어 있지 않아 시스템 콜이 처리되지 않는다. pintos의 시스템 콜 메커니즘을 이해하고 시스템 콜 핸들러를 구현한다.
시스템 콜(halt, exit, create, remove)를 구현하고 시스템 콜 핸들러를 통해 호출한다.

<pintos에서의 시스템 콜 호출 과정>

운영체제는 디스크, 메모리 같은 공유 리소스에 대한 신뢰 있는 관리와 제어를 하기 위해 kernel과 user mode에서 CPU의 privilege level을 다르게 하여 user mode에서 직접적으로 물리 메모리와 디스크 같은 공유 자원에 접근할 수 없도록 구분하고 있다. 하지만 user mode라 할지라도 OS가 직접 관리하고 있는 공유 리소스를 사용할 수 있게 요청할 수 있어야 하고, kernel에 있는 정보에 접근할 필요가 있다. 그래서 해당 프로세스에서 사전에 정의된 system call을 이용하여 제한적으로 kernel에서 privilege mode로 작동하는 함수를 호출하여 user mode에서 kernel에게 필요한 것을 요청할 수 있다.

pintos 에서는 아래와 같이 user mode 에서 호출 할 수 있는 system call API 들을 lib/user/syscall.c 에 정의하고 있다. 콘솔 화면에 문자를 출력하거나, 디스크에 있는 파일에 접근하거나, 물리 메모리에서 페이지를 할당 받는 등, kernel 에서 관리하는 공유 자원에 system call 을 통해 접근하게 되어있다.

```
96 bool
97 remove (const char *file)
98 {
99   return syscall1 (SYS_REMOVE, file);
100 }
101
102 int
103 open (const char *file)
104 {
105   return syscall1 (SYS_OPEN, file);
106 }
107
108 int
109 filesize (int fd)
110 {
111   return syscall1 (SYS_FILESIZE, fd);
112 }
113
114 int
115 read (int fd, void *buffer, unsigned size)
116 {
117   return syscall3 (SYS_READ, fd, buffer, size);
118 }
119
120 int
121 write (int fd, const void *buffer, unsigned size)
122 {
123   return syscall3 (SYS_WRITE, fd, buffer, size);
124 }
```

< lib/user/syscall.c에 정의된 user mode에서 호출할 수 있는 system call API들 >

가령, 콘솔 화면에 문자를 출력할 수 있는 printf(), msg() 함수 같은 경우 내부적으로 write() 함수로 작동하고, fopen()과 같이 디스크에서 파일을 열어 file descriptor를 만들어 주는 함수는 내부적으로 open() 시스템 콜을 사용한다.

위에 언급한 경우 외에도 프로세서 제어(process Control), 파일 조작(file manipulation), 장치 관리(Device Management), 정보 유지(Information maintenance), 통신(Communication) 등 과 관련된 다양한 상황에서 system call 호출을 필요로 한다.

하지만 위에서 코드는 user mode에서 작동하는 API이다. privilege mode로 kernel에서 동작하기 위해선 kernel mode로의 진입을 해야 하는데, kernel mode로 진입하기 위해 interrupt를 사용한다. 아래 소스코드에서는 pintos의 user process에서 시스템 콜을 호출하기 위해 interrupt를 발생시키는 매크로 함수를 보여준다.

```
47 /* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
48    ARG2, and returns the return value as an `int'. */
49 #define syscall3(NUMBER, ARG0, ARG1, ARG2)                      \
50         ({                                                      \
51           int retval;                                           \
52           asm volatile                                          \
53             ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "    \
54              "pushl %[number]; int $0x30; addl $16, %%esp"      \
55                : "=a" (retval)                                  \
56                : [number] "i" (NUMBER),                         \
57                  [arg0] "g" (ARG0),                             \
58                  [arg1] "g" (ARG1),                             \
59                  [arg2] "g" (ARG2)                              \
60                : "memory");                                     \
61           retval;                                               \
62         })
```

< user mode에서 시스템 콜을 호출하기 위해 interrupt를 발생시키는 루틴 (/lib/user/syscall.c) >

위 사진에서는 인자를 3개로 필요로 하는 write(), read()와 같은 system call API를 호출할 때 실행될 syscall3() 매크로 함수를 나타낸다. push arg2, push arg1, push arg0 에서 시스템 콜이 인자를 순서대로 넘겨받을 수 있게 calling convention에 맞춰 역순으로 user 스택에 push를 하고, push %[number] 에서 실행하고자 하는 system call number를 user 스택에 push한다. 그러고 나서 int 0x30을 통해 interrupt를 발생시켜(0x30은 system call interrupt number 이다.) privilege mode로 진입한다.

interrupt 가 발생하게 되면 cpu 하드웨어에 의해 eip, cs, eflags, esp, ss 가 kernel stack 에 push 된다. 그러고 나서 pintos threads/intr-stubs.S 에 정의되어 있는 intr_stubs 로 점프하여 ebp 를 kernel 스택에 백업하고 0 을 push 한 다음 system call interrupt 에 해당하는 interrupt number 를 kernel 스택에 push 한다.

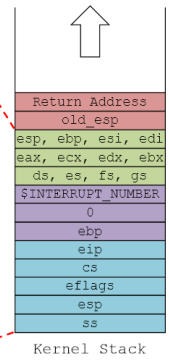
```
95 intr_stubs:
96
97 /* This implements steps 1 and 2, described above, in the common
98    case where we just push a 0 error code. */
99 #define zero                                    \
100     pushl %ebp;                             \
101     pushl $0
102
103 /* This implements steps 1 and 2, described above, in the case
104    where the CPU already pushed an error code. */
105 #define REAL                                    \
106         pushl (%esp);                           \
107         movl %ebp, 4(%esp)
108
109 /* Emits a stub for interrupt vector NUMBER.
110    TYPE is `zero', for the case where we push a 0 error code,
111    or `REAL', if the CPU pushes an error code for us. */
112 #define STUB(NUMBER, TYPE)                      \
113     .text;                                  \
114 .func intr##NUMBER##_stub;          \
115 intr##NUMBER##_stub:                            \
116     TYPE;                                   \
117     push $0x##NUMBER;                       \
118         jmp intr_entry;                         \
119 .endfunc;                       \
```

< threads/intr-stubs.S 에 정의된 intr_stubs >

그러고 나서 intr_stubs 는 intr_entry 로 점프하여 segment selector register 들과 여러 범용 register 들을 kernel stack 에 push 하여, 최종적으로 intr_frame 구조체를 kernel stack 에 구성한다. 이 intr_frame 구조체는 interrupt handler 가 모두 수행되고 나서 iret instruction 에 의해 privilege level 을 user mode 로 복귀할 때 user mode 에서의 context 를 복원하기 위해 저장된다.

```
18 .func intr_entry
19 intr_entry:
20     /* Save caller's regis
21     pushl %ds
22     pushl %es
23     pushl %fs
24     pushl %gs
25     pushal
26
27     /* Set up kernel envir
28     cld         /* String
29     mov $SEL_KDSEG, %eax
30     mov %eax, %ds
31     mov %eax, %es
32     leal 56(%esp), %ebp /*
33
34     /* Call interrupt hand
35     pushl %esp
36 .globl intr_handler
37     call intr_handler
38     addl $4, %esp
39 .endfunc
```

```
struct intr_frame
  {
    uint32_t edi;         /* Saved EDI. */
    uint32_t esi;         /* Saved ESI. */
    uint32_t ebp;         /* Saved EBP. */
    uint32_t esp_dummy;   /* Not used. */
    uint32_t ebx;         /* Saved EBX. */
    uint32_t edx;         /* Saved EDX. */
    uint32_t ecx;         /* Saved ECX. */
    uint32_t eax;         /* Saved EAX. */
    uint16_t gs, :16;     /* Saved GS segment register. */
    uint16_t fs, :16;     /* Saved FS segment register. */
    uint16_t es, :16;     /* Saved ES segment register. */
    uint16_t ds, :16;     /* Saved DS segment register. */

    uint32_t vec_no;      /* Interrupt vector number. */

    uint32_t error_code;  /* Error code. */

    void *frame_pointer;  /* Saved EBP (frame pointer). */
    void (*eip) (void);   /* Next instruction to execute. */
    uint16_t cs, :16;     /* Code segment for eip. */
    uint32_t eflags;      /* Saved CPU flags. */
    void *esp;            /* Saved stack pointer. */
    uint16_t ss, :16;     /* Data segment for esp. */
  };
```



< threads/intr-stubs.S 에 정의된 intr_entry 와 kernel 스택에 형성된 intr_frame 구조체 >

kernel 스택에 intr_frame 을 형성하고 나면 intr_entry 는 kernel 스택의 interrupt_frame 을 가리키고 있는 esp 를 push 하고 intr_handler 를 호출한다. intr_frame 를 가리키는 포인터를 push 한 이유는 intr_handler 에게 인자로서 전달해 주기 위해서다. intr_handler 에서 interrupt number 를 참조하기 위해 사용 되어지고, 이후에 system call handler 에서 user 스택에 저장한 인자를 가져오기 위해 참조된다.

intr_handler 가 호출되면 intr_frame 구조체의 vec_no 멤버를 참조하여 interrupt number 를 가져오며, 해당 interrupt number 에 해당하는 interrupt handler 를 호출하게 된다. 각 interrupt number 에 해당하는 interrupt handler 는 IDT(interrupt descriptor table)에 저장되어 있으며, 이 테이블을 가리키는 주소가 IDT 레지스터에 저장되어, IDT 레지스터를 통해 IDT 에 접근할 수 있다.

```
345 intr_handler (struct intr_frame *frame)
346 {
347   bool external;
348   intr_handler_func *handler;
349
350   /* External interrupts are special.
351      We only handle one at a time (so interrupts must be off)
352      and they need to be acknowledged on the PIC (see below).
353      An external interrupt handler cannot sleep. */
354   external = frame->vec_no >= 0x20 && frame->vec_no < 0x30;
355   if (external)
356     {
357       ASSERT (intr_get_level () == INTR_OFF);
358       ASSERT (!intr_context ());
359
360       in_external_intr = true;
361       yield_on_return = false;
362     }
363
364   /* Invoke the interrupt's handler. */
365   handler = intr_handlers[frame->vec_no];
366   if (handler != NULL)
367     handler (frame);
368   else if (frame->vec_no == 0x27 || frame->vec_no == 0x2f)
369     {
370       /* There is no handler, but this interrupt can trigger
371          spuriously due to a hardware fault or hardware race
372          condition.  Ignore it. */
373     }
```

< threads/interrupt.c: intr_handler(). interrupt number 에 해당하는 handler 를 IDT 에서 찾아 호출한다. >

```
133   idtr_operand = make_idtr_operand (sizeof idt - 1, idt);
134   asm volatile ("lidt %0" : : "m" (idtr_operand));
135
136   /* Initialize intr_names. */
137   for (i = 0; i < INTR_CNT; i++)
138     intr_names[i] = "unknown";
139   intr_names[0] = "#DE Divide Error";
140   intr_names[1] = "#DB Debug Exception";
141   intr_names[2] = "NMI Interrupt";
142   intr_names[3] = "#BP Breakpoint Exception";
143   intr_names[4] = "#OF Overflow Exception";
144   intr_names[5] = "#BR BOUND Range Exceeded Exception";
145   intr_names[6] = "#UD Invalid Opcode Exception";
146   intr_names[7] = "#NM Device Not Available Exception";
147   intr_names[8] = "#DF Double Fault Exception";
148   intr_names[9] = "Coprocessor Segment Overrun";
149   intr_names[10] = "#TS Invalid TSS Exception";
150   intr_names[11] = "#NP Segment Not Present";
151   intr_names[12] = "#SS Stack Fault Exception";
152   intr_names[13] = "#GP General Protection Exception";
153   intr_names[14] = "#PF Page-Fault Exception";
154   intr_names[16] = "#MF x87 FPU Floating-Point Error";
155   intr_names[17] = "#AC Alignment Check Exception";
156   intr_names[18] = "#MC Machine-Check Exception";
157   intr_names[19] = "#XF SIMD Floating-Point Exception";
158 }
```

< threads/interrupt.c 에 정의 되어있는 interrupt list. pintos 부팅 시 intr_init()에 의해 IDT 가 초기화 된다. >

0x30 번인 경우 system call interrupt 에 해당하며, 해당 interrupt handler 는 IDT 의 0x30 번째 인덱스 interrupt gate 에 userprog/syscall.c 에 정의된 syscall_handler()의 함수 포인터가 저장되어 있다. 하지만 위에 interrupt.c 에 정의된 interrupt name 들을 보면 0 부터 19 까지 밖에 없는 것으로 보아 system call interrupt 는 intr_init()에서 기본적으로 초기화 되는 interrupt 는 아닌 것을 알 수 있다. pintos 는 IDT 에 system call interrupt 를 추가하기 위해 부팅 시 userprog/syscall.c: syscall_init() 를 호출하여 IDT 의 0x30 번째 interrupt gate 에 syscall_handler 의 함수포인터를 저장하여 system call interrupt gate 를 추가한다.

```
10 syscall_init (void)
11 {
12   intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
13 }
14
```

< threads/init.c 에서 pintos 부팅 시 호출되어 IDT 에 system call interrupt 를 추가한다. >

여튼, intr_handler()에세 0x30 에 해당하는 interrupt handler 를 찾기 위해 IDT 에서 0x30 번째 interrupt gate 에서 사전에 저장되어 있던 syscall_handler()의 함수 포인터를 가져와서 호출한다.

과제를 하기 이전이라면 syscall_handler()가 구현이 되어있지 않다. 이것을 구현하는 것이 이번 과제의 핵심이다.

syscall_handler()는 user mode 에서 어떤 system call 을 호출했는지 system call number 를 통해 알고 해당 system call 을 수행할 수 있게 해당 루틴으로 잘 분기해주는 역할을 해야 한다. 또한 필요시 user 스택에 저장된 시스템 콜의 인자들을 kernel 스택에 복사하여 해당 함수에 인자를 잘 전달해줘야 한다. 또한 user mode 에게 제한된 privilege 권한을 제공해주는 것이기 때문에, user mode 에서 넘어온 인자들에 대한 메모리 범위 검사를 철저하게 하여 안정성 혹은 보안 이슈가 발생하지 않게 안전한 코드를 작성해야 한다. 아래는 lib/syscall-nr.h 에 정의 되어 있는 system call number enum 구조체이다.

```
4  /* System call numbers. */
5  enum
6    {
7      /* Projects 2 and later. */
8      SYS_HALT,                   /* Halt the operating system. */
9      SYS_EXIT,                   /* Terminate this process. */
10     SYS_EXEC,                   /* Start another process. */
11     SYS_WAIT,                   /* Wait for a child process to die. */
12     SYS_CREATE,                 /* Create a file. */
13     SYS_REMOVE,                 /* Delete a file. */
14     SYS_OPEN,                   /* Open a file. */
15     SYS_FILESIZE,               /* Obtain a file's size. */
16     SYS_READ,                   /* Read from a file. */
17     SYS_WRITE,                  /* Write to a file. */
18     SYS_SEEK,                   /* Change position in a file. */
19     SYS_TELL,                   /* Report current position in a file. */
20     SYS_CLOSE,                  /* Close a file. */
21
22     /* Project 3 and optionally project 4. */
23     SYS_MMAP,                   /* Map a file into memory. */
24     SYS_MUNMAP,                 /* Remove a memory mapping. */
25
26     /* Project 4 only. */
27     SYS_CHDIR,                  /* Change the current directory. */
28     SYS_MKDIR,                  /* Create a directory. */
29     SYS_READDIR,                /* Reads a directory entry. */
30     SYS_ISDIR,                  /* Tests if a fd represents a directory. */
31     SYS_INUMBER                 /* Returns the inode number for a fd. */
32   };
```

< lib/syscall-nr.h 에 정의 되어 있는 system call number >

위의 enum 구조체를 참고하여, 이번 과제에서 몇가지 시스템콜을 구현해 볼 것이다.

2. 소스 코드 설명

```
 7 /* 보다 직관적인 check_address() 를 작성하기 위해
 8    각 메모리 영역 시작 주소 값을 USER_START, KERNEL_START로 정의함. */
 9 #define USER_START      0x08048000
10 #define KERNEL_START    0xc0000000
11
```

```
93 // 인자로 받아온 포인터 주소가 kernel영역 메모리 주소면 해당 프로세스를 종료시킨다.
94 void check_address (void *addr)
95 {
96   if (!( (void*)USER_START <= addr && addr < (void*)KERNEL_START ))
97   {
98     printf ("invailed parameter.\n");
99     exit (-1);
100  }
101 }
```

< void check_address (void *addr) >

system call 은 user process 에게 일시적으로 privilege level 을 높여 준다. kernel mode 에서는 user mode 의 메모리 영역에 접근을 할 수 있되, user mode 의 코드를 실행시켜서는 안되며, user mode 에서는 kernel 의 메모리에 읽거나 쓰는 접근 행위를 하면 안된다. 그래서 user mode 에서 넘어오는 모든 인자들에 대해 메모리 범위를 검사를 한다.

가령 system call handler 가 user stack 에서 system call 에 전달될 인자들을 읽어온다면, 이 읽어온 인자들에 대해 실제로 user mode memory 영역에 있는 데이터를 읽어오는지 해당 인자가 저장된 위치를 check_address() 로 검사해야 하고, 해당 인자가 포인터 변수라면 그 포인터 변수가 kernel memory 영역을 가리켜 kernel 메모리 영역에 접근하는지 check_address()로 검사해야 한다.

check_address()는 포인터 인자가 정상적으로 user memory 영역을 가리키는지 검사를 하고, 해당 포인터가 kernel memory 영역을 가리키거나 user process 메모리 공간을 벗어난 곳을 가리키면 해당 프로세스를 종료시킨다.

```
103 /* 유저모드 스택에 저장된 시스템 콜 함수에 대한 인자를 커널 스택에 복사한다.
104    esp 위치에 system call number가 저장되어 있고, esp+4 위치부터 인자들이 저장되어 있다. */
105 void get_argument (void *esp, int *arg, int count)
106 {
107   int i;
108   int *user_arg = (int*)esp + 1;
109
110   for (i = 0; i < count; i++)
111   {
112     check_address (&user_arg[i]); // 유저모드 메모리 주소에서 넘어온 값이 맞는지 확인
113     arg[i] = user_arg[i];
114   }
115
116 }
```

< void get_argument (void *esp, int *arg, int count) >

시스템 콜도 결국 작동하는 privilege level 만 다를 뿐 함수로서 작동하기 때문에 system call 함수에 전달 될 인자가 필요할 수 있다. 하지만 user mode 에서 system call API 를 통해 인자를 전달해도 calling convention 에 의해 해당 인자들은 user stack 에 저장 된다. system call handler 는 user stack 에 저장된 system call 인자들을 kernel stack 으로 복사해와서 system call 함수에게 전달을 해주어야 한다. 이 과정에서 user stack 에 저장된 인자의 배열을 읽어 들여 kernel stack 의 배열에 복사해와야 하는데, 이를 위해 필요한 함수가 get_argument ()이다.

get_argument() 함수는 user stack 에 접근하여 저장된 인자를 읽어와야 한다. user stack 을 가리키는 메모리 주소는 system call handler 에게 전달된 intr_frame 구조체 포인터를 통해 esp 멤버 값을 참조하여 알 수 있다. user stack 의 끝을 나타내는 esp 에는 system call number 가 저장되어 있고, esp+4 부터 system call 에 전달 될 인자들이 저장되어 있다.

user stack 에서 각 인자를 가져올 때 실제 user memory 영역에 저장된 값을 읽어오는 것이 맞는지 check_address()로 인자가 저장된 주소를 검사를 한다. 그러고 나서 get_argument()가 인자로 받은 커널 스택의 배열을 가리키는 arg 에 순차적으로 배열로서 저장한다. get_argument()가 arg 가 가리키는 커널 스택 공간에 배열을 복사하면, get_argument() 호출 이후 syscall_handler()의 system call 처리 루틴에서 arg 배열을 참조해 system call 함수에게 인자를 전달해 줄 수 있다.

```
29 static void
30 syscall_handler (struct intr_frame *f UNUSED)
31 {
32   int syscall_num;
33   int arg[5] = {0, };
34   int *esp = f->esp;
35
36   /* esp가 유저모드의 메모리 주소 영역인지 체크한다.
37      유저모드에서 넘어온 모든 인자의 주소와 포인터는
38      check_address() 로 메모리 범위를 검사해줘야 한다.
39      현재 상태는 kernel모드인 privilege 모드이므로,
40      커널 입장에서 비신뢰적인 유저모드 코드에서 넘어온 인자를 검사하지 않으면,
41      유저모드에서 커널 메모리 영역 접근을 의도할 수 있다. */
42   check_address (esp);
43
44
45   /* 유저모드에서 int 0x30 (시스템콜 인터럽트) 를 발생시키기 전에 스택에 인자로 넘겨준
46      system call number를 가져온다.  */
47   syscall_num = *esp;
48
```

```
49   // system call number는 src/lib/syscall-nr.h에 enum 구조체에 정의되어 있다.
50   // 각 system call number별로 switch문에서 분기하여 system call을 수행한다.
51   switch (syscall_num)
52   {
53
54     case SYS_HALT :
55         halt ();
56         break;
57
58     case SYS_EXIT :
59         get_argument (esp, arg, 1);
60         exit (arg[0]);
61         f->eax = arg[0];
62         break;
63
77     case SYS_CREATE :
78         get_argument (esp, arg, 2);
79         check_address (arg[0]);
80         f->eax = create ((const char*)arg[0], (unsigned)arg[1]);
81         break;
82
83     case SYS_REMOVE :
84         get_argument (esp, arg, 1);
85         check_address (arg[0]);
86         f->eax = remove ((const char*)arg[0]);
87         break;
88
89   }
```

< syscall_handler (struct intr_frame *f) , 시스템 콜 핸들러 >

syscall_handler()는 0x30번 interrupt에 대해 interrupt handler에 의해 호출될 함수이다. system call number를 user stack에서 읽어 와 어떤 시스템 콜을 호출할지 찾고 해당 시스템 콜을 실행시켜준다.

system call number는 user stack에 저장되어 있다. user mode에서 제일 마지막으로 push한 값이 system call number이므로 intr_frame 구조체의 esp 멤버에 저장된 esp의 값이 결국 system call number를 가리킨다. 그래서 syscall_num = *(esp) 코드에서 user stack에서 system call number를 읽어올 수 있는 것이다.

syscall_num 값을 읽어왔으면 syscall_num 값에 따라 해당 system call을 호출할 수 있게 switch문으로 분기하여 처리한다. lib/syscall-nr.h의 system call enum 구조체에 의해 정의된 system call number에 따라 switch문으로 분기하여 해당 루틴에서 적절하게 get_argument()로 user stack에서 인자를 읽어와 해당 system call을 호출하는 처리를 해준다. 각 switch case의 루틴에선 해당 system콜의 인자의 개수와 인자의 타입을 고려하여 system call을 실행 할 수 있게 하였다.

```
118 // pintos를 종료하는 시스템 콜 함수
119 void halt ()
120 {
121   shutdown_power_off ();
122 }
```

< halt() 시스템 콜. 핀토스를 종료한다. >

halt() 시스템 콜을 호출하면 핀토스를 종료할 수 있게 shutdown_power_off()를 호출한다.

```
124 // 현재 돌아가는 프로세스를 종료시키는 시스템 콜 함수
125 void exit (int status)
126 {
127   struct thread *current_thread = thread_current ();
128   current_thread->exit_status = status;
129   printf ("process : %s exit(%d)\n", current_thread->name, status);
130   thread_exit ();
131 }
```

< exit() 시스템 콜. 현재 실행 중인 프로세스를 종료 시킨다. >

exit() 시스템 콜을 호출하면 현재 실행 중인 프로세스를 종료할 수 있게 thread_exit()를 호출한다. 위 사진에서는 3 번째 과제에서 추가로 구현된 부분을 포함하고 있다.

```
133 // 디스크에서 파일을 생성하는 시스템 콜 함수
134 bool create (const char *file, unsigned initial_size)
135 {
136   return filesys_create (file, initial_size);
137 }
```

< create() 시스템 콜. 디스크에 파일을 생성한다. >

create() 시스템 콜을 호출하면 사용자가 디스크에 파일이름이랑 초기 사이즈를 정해서 디스크에 파일을 만들 수 있도록 filesys_create()를 호출한다.

```
139 // 디스크에서 파일을 지우는 시스템 콜 함수
140 bool remove (const char *file)
141 {
142   return filesys_remove(file);
143 }
```

< remove() 시스템 콜. 디스크에 지우고자 하는 파일을 제거할 수 있다. >

remove() 시스템 콜을 호출하면 사용자가 디스크에 있는 지우고자 하는 파일을 지울 수 있도록 filesys_remove() 함수를 호출해준다.

4. 과제 수행 결과

```
 3 #include "tests/lib.h"
 4 #include "tests/main.h"
 5
 6 void
 7 test_main (void)
 8 {
 9   exit (57);
10   fail ("should have called exit(57)");
11 }
```

핀토스 테스트 파일 중 /tests/userprog/exit.c 를 컴파일 한 exit파일을 핀토스에서 실행해보아 테스트를 했다. 정상 실행 된다면, process : (프로세스 이름) exit(57) 이 출력될 것이다.

실행 결과 :

```
Loading.........
Kernel command line: run exit
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':
Execution of 'exit' complete.
process : exit exit(57) ←
```

User Program

- 프로세스 계층 구조

1. 과제 목표

프로세스 간의 부모와 자식 관계를 구현하고, 부모가 자식프로세스의 종료를 대기하는 기능을 구현한다.

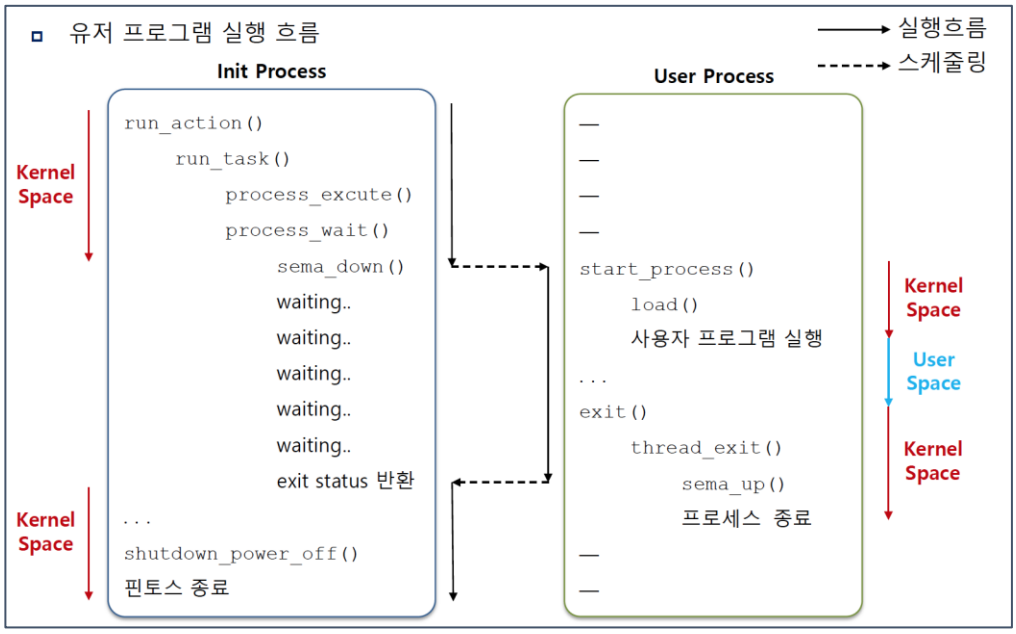
2. 과제 설명

- 핀토스는 프로세스 간에 부모와 자식 관계를 명시하는 정보가 없다. 부모와 자식의 구분이 없고, 자식 프로세스의 정보를 알지 못하기 때문에 자식 프로세스가 정상적으로 제 프로그램을 수행하기 전에 부모 프로세스, 즉 커널이 종료되는 현상이 발생할 수 있다.
- 프로세스 제어 블록(struct thread)에 부모 프로세스와 자식 프로세스들을 가리키는 자료구조 추가한다.
- PCB에 있는 부모프로세스와 자식프로세스 정보를 다루는 함수를 제작한다.

(자식 프로세스들은 리스트로 구현하고, 자식 리스트에서 원하는 프로세스 검색, 삭제 하는 함수를 구현한다.)

- 세마포어를 이용하여, exec(), wait() 시스템콜을 구현한다.

3. 코드 수정 및 추가 함수



이번 과제에서의 핵심은 위의 그림과 같이 부모 프로세스가 자신이 생성한 자식프로세스의 종료를 기다리고 실행을 재개하는 것을 구현하는 것이므로, 유저프로그램 실행 흐름을 따라 코드 설명을 해보았다.

(1) (thread.h) 프로세스 디스크립터(struct thread)에 필드 추가

```
struct thread *parent; //부모프로세스의 디스크립터
struct list_elem child_elem;//자식리스트 element
struct list child_list; //자식리스트
bool loaded; //프로세스의 프로그램 메모리탑재 유무
bool exited; //프로세스가 종료유무확인
struct semaphore exit_sema; //exit 세마포어
struct semaphore load_sema; //load 세마포어
int exit_status; //exit 호출시 종료status
```

부모 자식 관계를 표현하기 위해, 부모 스레드의 PCB인 struct thread A에서 B의 자식 리스트를 관리할 수 있도록 자식 리스트를 추가하고 자식 스레드의 PCB인 struct thread B에는 부모 프로세스의 PCB를 가리키는 포인터를 설정한다.

이렇게 추가한 필드들은 PCB(struct thread)가 생성될 때 **초기화**를 해줘야 한다.

1. 스레드 생성 시 자식 리스트 초기화 : thread.c의 init_thread()함수 내부에서 list_init(&t->child_list);를 통해 자식 리스트를 초기화해준다.
2. 프로세스 디스크립터의 정보 초기화 및 부모 프로세스의 자식 리스트에 추가 : process_execute()안의 thread_create()함수 호출을 통해 스레드가 생성되는데, thread_create()함수 내부를 보면 struct thread *t = palloc_get_page(PAL_ZERO); 를 통해 PCB를 저장할 page를 할당 받고나서, 추가된 PCB멤버들을 초기화한다.

```
/* 부모프로세스저장*/
t->parent = thread_current();
/* 프로그램이 로드되지않음*/
t->loaded = false;
/* 프로세스가 종료되지않음*/
t->exited = false;
/* exit 세마포어0으로초기화*/
sema_init(&t->exit_sema, 0);
/* load 세마포어0으로초기화*/
sema_init(&t->load_sema, 0);
/* 부모프로세스의 자식리스트에추가*/
list_push_back(&thread_current()->child_list, &t->child_elem);
```

(2) 자식 프로세스 디스크립터를 검색하는 함수 (process.c)

자식 리스트를 검색하여 해당 pid에 맞는 프로세스 디스크립터를 반환한다.

pid를 갖는 프로세스 디스크립터가 존재 하지 않을경우 NULL을 반환한다.

```
struct thread *get_child_process(int pid)

{
  struct thread *cur = thread_current();
  struct thread *tmp_t;
  struct list_elem *elem;

  for (elem = cur->child_list.head.next; elem != cur->child_list.tail.prev; elem =
elem->next)
  {
    tmp_t = list_entry (elem, struct thread, child_elem);
    if (pid ==  tmp_t->tid)
      {
        return tmp_t;
      }
  }
  return NULL;
}
```

wait() 시스템콜은 인자로 자식프로세스의 pid를 받아서 해당 pid의 프로세스의 종료를 기다려준다. 해당 함수는 인자로 받은 pid에 해당하는 pcb의 주소값을 반환을 해준다.

인자로 받은 pid에 해당하는 프로세스를 찾는 과정은 전체 리스트가 아닌 자신의 child_list에서 찾는 것이 빠르다.

(3) 자식 프로세스 PCB를 삭제하는 함수 (process.c)

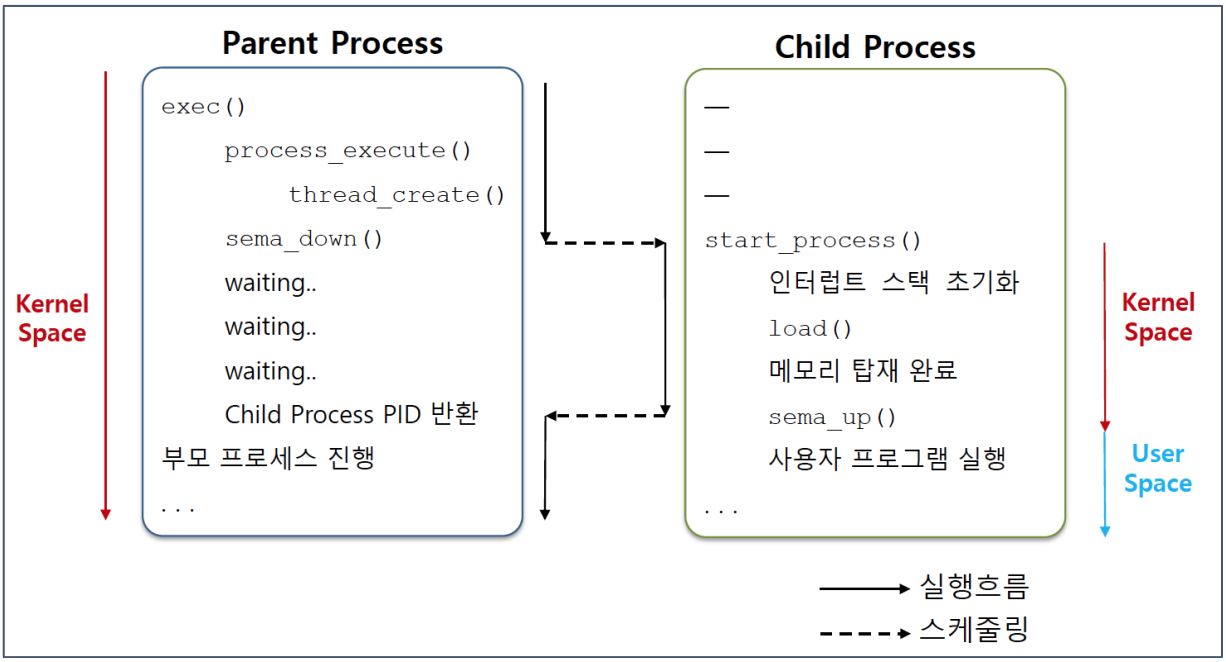
자식의 프로세스 PCB(pcb)를 부모 프로세스의 자식 리스트에서 제거하고, 해당 PCB를 저장하기 위해 할당된 메모리를 해제하는 함수이다.

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

(1), (2), (3)을 이용하여 wait 함수와 exec함수를 구현한다. 아래에서 설명할 함수들은 위에 언급한 함수들과 세마포어를 통해 부모프로세스와 자식프로세스 간에 실행 순서를 조정한다.

(4) exec()

exec()호출시, child process 가 프로그램을 메모리에 탑재 완료할 때까지 부모가 대기하도록 실행 흐름을 변경하여 작성한다.




```
tid_t exec(const char *cmd_line)
{
  struct thread *child = NULL;
  pid_t child_pid = 0;

  // cmd_line의 프로그램과 인자들을 실행할 자
  child_pid = process_execute (cmd_line);
  // 커널은 생성된 자식프로세스의 pid에 해당하
  child = get_child_process(child_pid);

  // 자식 프로세스로 실행흐름이 변경되어, 자식
기한다
  sema_down (&child->load_sema);
  if (&child->exit_status == -1) { // 여기
                                   // 로드
    return -1;
  }

  return child_pid;


}
```

⇒ 그림과 코드를 같이 보며 진행한다.

cmd_line의 프로그램과 인자들을 실행할 자식 프로세스를 생성하고, 생성된 자식프로세스의 pid에 해당하는 pcb를 찾는다.
그리고 pcb의 load 세마포어에 접근하여 자식 프로세스로 실행 흐름이 넘어가게 된다. 부모 프로세스는 BLOKED상태로 대기하며, 프로그램 load 실패시 -1을 리턴하고, load성공시 자식 프로세스의 pid를 리턴한다.

process.c의 start_process() 함수에서 메모리 탑재 성공 시, 유저 프로그램을 실행하고, 실패 시 스레드를 종료한다.

탑재 성공 시, 자식 프로세스가 종료 될 때 thread_exit() 내부의 sema_up(load_sema)을 호출하게 되며 부모 프로세스가 BLOCKED에서 READY상태로 바뀌고, schedule() 함수가 호출되어 스케줄러

```
success = load(file_name, &if_.eip, &if_.esp);
thread_current()->loaded = success;
sema_up(&thread_current()->load_sema);//부모 프로세스의 상태가 BLOKED->READY 됨.
```

에 의해 선택되어 질 때 RUNNING상태로 변경되며, 실행 흐름이 다시 부모프로세스로 넘어가게

된다.

메모리 탑재 실패 시 프로세스 디스크립터에 메모리 탑재 실패,
메모리 탑재 성공 시 프로세스 디스크립터에 메모리 탑재 성공 여부를 담는다.

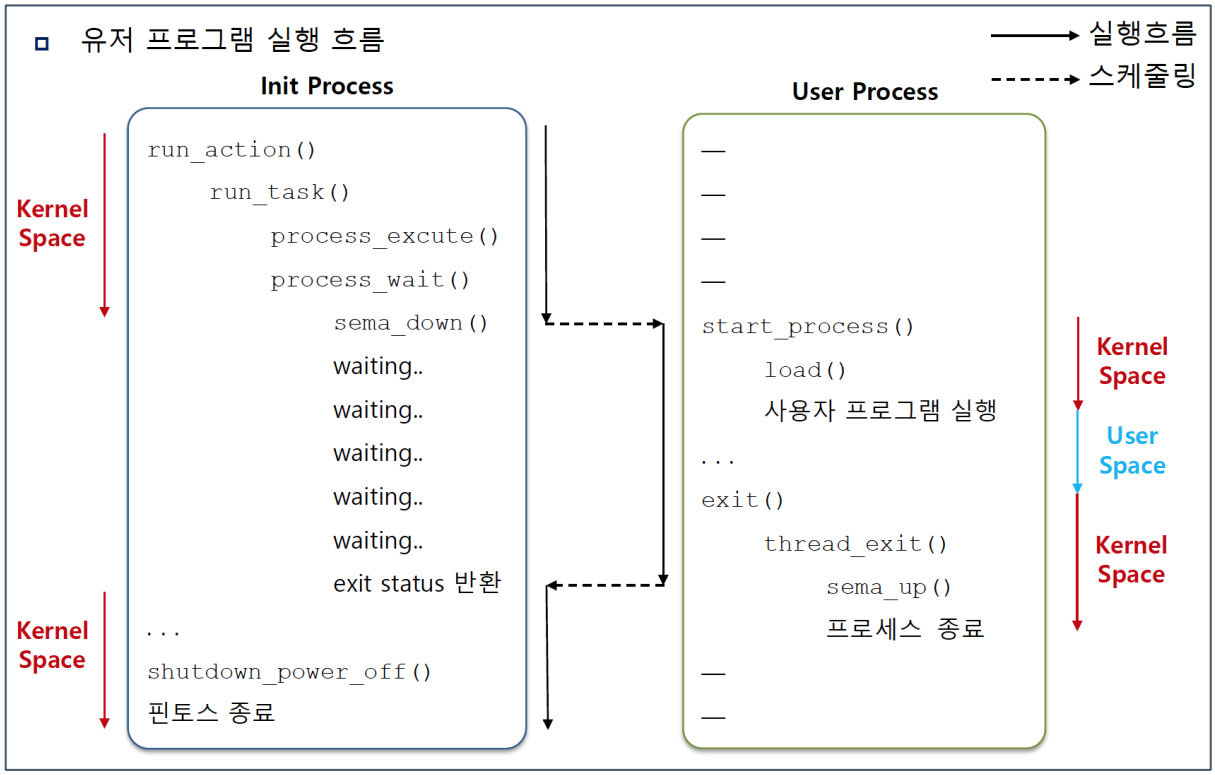
```
if (!success) {
  thread_current()->exit_status = -1;
  thread_exit ();
} else {
  thread_current()->exit_status = 0;
  thread_current()->exited = false;

  argument_stack(parse, token_count, &if_.esp);
}
```

마지막으로, 구현한 exec시스템 콜을 시스템 콜 핸들러에 추가한다.

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

(5) wait()

현재 process_wait() 는 -1을 리턴하여 init process는 유저프로세스가 종료될때까지 대기하지 않고 핀토스를 종료해버린다. process_wait()기능을 구현하여 자식프로세스가 모두 종료될 때 까지 부모 프로세스는 대기(blocked state)를 하고, 자식 프로세스가 올바르게 종료가 됐는지 확인하는 함수를 구현한다.



```
int
process_wait (tid_t child_pid)
{
  struct thread *child = NULL;
  int status = 0;

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

  if (!child->exited)
    sema_down(&child->exit_sema);

  status = child->exit_status;
  remove_child_process(child);

  return status;

}
```

이 그림과 같이 부모 프로세스가 대기 또는 재개될 수 있도록 수정하였다. 생성된 자식 프로세스의 pid를 인자로 받아, pcb 안의 세마포어에 접근할 수 있도록 get_child_process(child_pid)를 통해 pcb의 주소값을 반환받는다. 그리고 자식 프로세스가 종료될 때까지 sema_down을 이용하여, 부모프로세스가 대기하게 된다. 자식프로세스가 종료되면 자식프로세스의 종료코드를 받아오고, 디스크립터를 삭제한 후, 자식프로세스의 exit_status를 리턴한다. (process.c)

(thread.c) thread_exit() : 유저 프로세스(자식 프로세스) 종료 시 thread_exit() 호출을 하는데, 유저 프로세스가 종료되면 부모 프로세스가 대기 상태를 이탈 후 다시 재개하게 된다. 또한, 프로세스 디스크립터에 프로세스의 종료를 알려야 한다.

```
  thread_current()->exited = true;
  if (!(thread_current()->tid == 1 || thread_current()->tid == 2))
  {
    sema_up(&thread_current()->exit_sema);
  }


  thread_current ()->status = THREAD_DYING;
  schedule ();
  NOT_REACHED ();
}
```

그리고, **(thread.c) thread_schedule_taill()함수** 안의 palloc_free_page(prev); 를 주석 처리해줘야 한다. (프로세스 디스크립터를 삭제하지 않도록 수정해야한다.) 왜냐하면 이전에 실행되었던 thread가 종료 되었다면 부모프로세스의 자식리스트에서 삭제하는 부분인데, 이미 wait() 시스템콜에서 그 thread의 pcb를 지워줬었기 때문에 삭제된 것을 다시 삭제하려는 시도가 되어 에러가 나게 되기 때문이다.

(6) exit()

마지막으로 exit(), wait() 시스템콜을 수정한다. 프로그램 종료 시 exit()시스템 콜을 호출하게 된다.

정상적으로 종료가 됐는지 확인하기 위해 exit status에 종료 값을 저장한다.

```c
void exit(int status)
{
  struct thread *cur = thread_current();
  cur->exit_status = status;

  /* 스레드이름과 exit status 출력*/
  printf("thread name : %s, status : %d\n", thread_name(), status);
  /* 스레드종료*/
  thread_exit();
}
```

(7) wait()

procsee_wait()을 사용하여, 자식 프로세스가 종료 될 때까지 대기한다.

```c
void wait(tid_t pid)
{
  /* process_wait()사용, 자식프로세스가종료될때까지대기*/
  process_wait(pid);
}
```

User Program
- 파일 디스크립터

1. 과제 목표

파일 디스크립터 및 관련 시스템 콜들을 구현한다.

2. 과제 설명

핀토스에는 파일 디스크립터 부분이 누락되어 있다. 파일 입출력을 위해서는 파일 디스크립터의 구현이 필요하다.

3. 코드 수정 및 추가 함수

(1) (thread.h) 필드 추가 & (thread.c) 필드 초기화

커널 메모리에 File Descriptor Table을 할당 후 thread 자료구조의 fdt 필드가 각 File Descriptor 테이블을 가리키도록 구현하기위해, struct thread 안에 **struct file **FDT; int next_fd;** 를 추가해주었다. 그리고 추가한 필드들을 앞의 과제들과 마찬가지로 스레드 생성시에 File Descriptor를 초기화해준다. **t->FDT = palloc_get_page (PAL_ZERO); t->next_fd = 2;**

0과 1은 이미 STDIN과 STDOUT이 사용 중이므로 2번째 엔트리부터 파일 디스크립터를 할당 받을 수 있도록 2로 초기화 해주었다.

(2) process_add_file

struct file(파일 객체) 가리키는 포인터를 File Descriptor 테이블에 추가해주는 함수이다. 추가된 파일 객체의 File Descriptor 값을 반환하고, 다음 File Descriptor 값을 1증가시킨다. FDT의 최대 개수(FILE_MAX)를 넘어가게 되면 mod연산을 통해서 FILE_MAX 크기만큼(한 바퀴)을 완전히 돌 수 있도록 한다. 추가를 하면, 그 fd값을 반환하고 FDT가 꽉 찼을 경우 -1을 반환하도록 하였다.

```
int process_add_file (struct file *f) {
  struct thread *cur = thread_current ();
  // 파일 객체가 저장될 FDT entry번호를 저장함.
  int current_fd = cur->next_fd;
  // next_fd 번째 자리 FDT entry에 해당 파일 객체 포인터를 저장함.
  cur->FDT[cur->next_fd] = f;

  do {
    cur->next_fd = (cur->next_fd + 1) % (FILE_MAX);
    /* 한바퀴 돌고도 빈 FDT entry를 못찾으면 -1 리턴 */
    if (cur->next_fd == current_fd)
      return -1;
  } while (cur->FDT[cur->next_fd] != NULL);

  // 리턴 값으로 해당 파일을 저장한 FD number를 리턴함
  return current_fd;
}
```

(3) process_get_file

해당 엔트리에 파일이 있으면 파일 객체 포인터가 리턴되고, 파일 객체가 없는 경우에는 해당 엔트리에 null값이 저장되어 있으므로 null이 리턴된다.

```
struct file * process_get_file (int fd) {
  return thread_current ()->FDT[fd];
}
```

(4) process_close_file

fd에 해당하는 파일을 file_close( )를 호출하여 닫고, 파일 디스크립터 테이블 해당 엔트리를 NULL 값으로 초기화한다.

```
void process_close_file (int fd) {
  struct file *file = process_get_file (fd);
  file_close (file);
  thread_current ()->FDT[fd] = NULL;

}
```

(5) process_exit

유저 프로세스가 파일을 닫지 않고 프로세스를 종료 하려고 할 때, 모든 열린 파일을 닫고 프로세스를 종료해야 한다. 닫기 프로세스가 종료 될 때, 메모리 누수방지를 위해 프로세스에 열린 모든 파일을 닫는다. 그리고 File Descriptor 테이블 메모리를 페이지 해지해준다. 핀토스 부팅 시 생성되는 kernel의 main스레드와 idle스레드에 대해서는 예외처리를 해준다.

```
if (!(cur->tid == 1 || cur->tid == 2)) {

  for (i = 2; i < FILE_MAX; i++) {
    process_close_file (i);
  }
   palloc_free_page(cur->FDT);
}
file_close (cur->run_file);
```

위 설명에서 다룬 수정 및 추가된 코드와 함수들로 파일 관련 시스템콜을 구현할 수 있다.

(6) open()

filesys_open()으로 해당 파일이름과 경로에 해당하는 파일을 열어서 파일객체를 반환한다. 이 과정에서 해당 파일이 없거나 권한에 문제가 있어 열지 못한다면 null을 반환한다.
정상적으로 파일 객체를 반환 받았으면 해당 FDT에 파일 객체 포인터를 추가하여 해당 FD number를 리턴한다.

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

  struct file *file_object;

  // open-null testcase를 통과하기 위한 예외처리
  if (file == NULL) {
    exit(-1);
  } else {
    file_object = filesys_open (file);
  }
  if (file_object == NULL) {
    return -1;
  } else {
    return process_add_file(file_object);
  }
}
```

(7) filesize

파일 디스크립터에 해당하는 파일의 size를 리턴해주는 함수이다.

```
int filesize (int fd) {
  struct file *file_object = process_get_file (fd);

  if (file_object == NULL) {

    return -1;
  } else {
    return file_length (file_object);
  }

}
```

(8) close()

파일 디스크립터 사용이 끝나면 이 시스템콜을 호출하여 해당 fd의 file object를 해제할 수 있다.

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

(9) tell()

파일 객체의 읽고 쓸 position이 어디인지 알려주는 시스템 콜을 구현하였다.

```
unsigned tell (int fd) {
  struct file *file_object = process_get_file (fd);

  if (file_object == NULL) {
    /* 유효하지 않은 파일 디스크립터거나 STDIN이거나 STDOUT인 경우 return -1
       만약 null값을 file_tell()에 넘겨서 호출할 경우 assertion발생*/
    return -1;
  } else {
    /* file_tell() 내부적으로 file_object->pos 값을 읽어와줌 */
    return file_tell (file_object);
  }

}
```

(10) seek()

file object에서 position 멤버를 바꿔줘서, 다음에 read하거나 write할때 시작 위치를 바꿔주는 시스템콜 함수

```c
void seek (int fd, unsigned position) {
  struct file *file_object = process_get_file (fd);

 if (file_object == NULL || position >= file_length (file_object)) {

    /* fd가 STDIN or STDOUT 이거나 파일 객체가 없거나
       position이 파일 크기를 벗어나면 그냥 리턴함 */
    return;
  } else {

    /* 내부적으로 파일 객체의 pos멤버를 position값을 바꾸도록 작동함 */
    file_seek (file_object, position);
    return position;
    // file_seek() 함수가 따로 바꾼 위치 값을 리턴하지 않아서
    //position 인자값 그대로 사용함
  }
}
```

(11) read()

fd가 나타내는 파일의 내용을 읽어주는 시스템콜 함수이다. 개행 문자를 받으면 끊어주고, 해당 파일에서 size크기 만큼 읽고 (혹은 EOF 읽는 지점까지) buffer주소에 읽어온 내용을 저장하며 읽은 byte 수를 반환한다. write와 마찬가지로, 파일에 동시 접근이 일어나는 것을 막기위해 lock을 사용한다.

```c
int read (int fd, void *buffer, unsigned size) {
  struct file *file_object = process_get_file (fd);
  int bytes_read = 0;
  int i;

  if (fd == STDIN) {
    for (i = 0; i < size; i++) {
      ((char*)buffer)[i] = input_getc ();
      if (((char*)buffer)[i] == '\n')
        break;
    }

    bytes_read = i + 1;

  } else {
    lock_acquire (&rw_lock);
    //---------- start critical section -----------
    bytes_read = file_read (file_object, buffer, size);
    //------------- finish critical section ------------
    lock_release (&rw_lock);

 }

  // 읽어온 byte 수를 리턴함
  return bytes_read;

}
```

(12) write()

fd가 나타내는 파일에 내용을 입력할 수 있게 해주는 시스템콜 함수이다.

STDOUT인 경우, 콘솔 화면에 buffer에 있는 내용을 size만큼 출력해주고, file_write()가 fd에 해당하는 파일에서 buffer에다가 size만큼 읽어와서 읽은 byte수만큼 리턴해준다. file object에 pos라는 읽을 위치를 저장해 놓은 멤버가 있어서 내부적으로 이 pos부터 시작해서 size만큼 읽게 된다.

그리고, 파일에 동시 접근이 일어날 수 있으므로 lock을 사용해준다.

```
int write (int fd, void *buffer, unsigned size) {
  struct file *file_object = process_get_file (fd);
  int bytes_write;

  if (fd == STDOUT) {
    putbuf(buffer, size);
    bytes_write = size;
  } else {

    lock_acquire (&rw_lock);
    //------------ start critical section ---------------
    bytes_write = file_write(file_object, buffer, size);
    //------------ finish critical section -------------
    lock_release (&rw_lock);
 }

  return bytes_write;
}
```

여기까지 모두 했다면, page fault 에러 메시지 출력으로 인해 test case가 fail 처리가 되므로, exit(-1)을 호출하도록 수정한다.

User Program

- Denying Writes to Executables

1. 과제 목표

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

2. 과제 설명

- 실행중인 사용자 프로그램의 프로그램 파일이 다른 프로세스에 의해 변경되면, 프로그램이 원래 예상했던 데이터와 다르게 변경된 데이터를 디스크에서 읽어올 가능성이 있다.
- 이런 현상이 발생하게 되면, 현재 실행중인 프로그램이 올바른 연산 결과를 도출할 수 없기 때문에 OS 차원에서 실행중인 프로세스의 프로그램 파일에 대한 쓰기 접근을 막아야한다.

3. 함수 수정

- 프로그램 파일이 프로세스의 가상 주소 영역에 탑재되는 시점을 확인한다.
- 프로세스가 종료되어 프로그램 파일이 더 이상 접근되지 않는 시점을 확인한다.
- 프로그램 파일이 프로세스에 의해 접근되고 있을 때에는 프로그램 파일이 변경되는 것을 방지할 수 있도록 수정한다.
- 프로그램 파일이 메모리에 탑재 될 때 프로그램 파일이 변경되는 것을 예방하고 프로그램 종료 시 데이터 변경이 허락될 수 있도록 수정한다.

(1) (thread.h) 실행중인 파일에 대한 포인터 필드 추가

```
struct file *run_file;
```

(2) (process.c) load()

ELF 파일을 열어서 프로세스에 올려 실행하는 동안 다른 프로세스가 해당 ELF파일에 접근하면 원래 실행하고자 하는 파일과 내용이 달라져 무결성이 깨질 수 있으므로 ELF 파일을 실행파일로서 작동시키는 중에는 다른 프로세스로부터 접근되어 write를 하지 못하도록 해당 파일을 보호해야한다. file_deny_write()으로 해당 파일 객체를 넘기면 해당 파일 객체의 deny_write 멤버가 true로 체크되어 해당 파일이 executable파일로서 실행 중인 것을 구분하여 주고, 해당 파일 객체의 inode 객체의 deny_write_count멤버를 1늘려 해당 파일을 실행중인 프로세스의 개수를 표시하여 준다.

해당 파일을 실행중인 프로세스가 나중에 종료되면 해당 실행 중인 thread구조체를 참조하여 해당 파일 객체를 file_close() 를 호출하여 닫아주는데, 이 함수 내부적으로 file_allow_write() 를 호출하게 되어 있다. 그러면 해당 파일 객체의 deny_write 멤버를 false로 바꿔 해당 파일이 쓰기가 가능하단 것을 표시하고 (근데 어차피 이 객체는 해제될 것이다.) 커널에 파일 별로 유일하게 존재하는 inode구조체에 deny_write_count를 1 줄여서 해당 파일을 실행파일로서 실행중인 프로세스가 1개 줄었다는 것을 표시한다. 실행 할 프로그램 파일을 열 때 데이터 변경을 예방 하도록 수정하여야 한다.

```
lock_acquire (&rw_lock);
/* Open executable file. */
file = filesys_open (file_name);
if (file == NULL)
  {
    printf ("load: %s: open failed\n", file_name);
    goto done;
  }

file_deny_write (file);
t->run_file = file;
```

```
done:
 lock_release (&rw_lock);
 return success;
}
```

(3) process_exit()

load() 함수에 의해 열린 ELF파일 객체를 프로세스 종료 할때 해제한다. 이 파일을 해제함으로 다른 프로세스가 ELF파일에 write할 수 있게 허용해준다. (file_close()함수의 내부적으로 file_allow_write() 함수를 호출하게 되어있다.) 이 ELF로 아직 실행 중인 다른 프로세스가 있다면 그 프로세스들이 모두 종료될 때까지 기다려야 한다. (inode구조체의 deny_write_count가 0이 될 때까지)

```
if (!(cur->tid == 1 || cur->tid == 2)) {

  for (i = 2; i < FILE_MAX; i++) {
    process_close_file (i);
  }
   palloc_free_page(cur->FDT);
}
file_close (cur->run_file);
```

<User Program 최종 결과>

1. make grade의 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%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY OF INDIVIDUAL TESTS
```

2. make check 의 결과

```
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.
make[1]: Leaving directory `/home/ckdqja0225/pintos/project/src_changbeom/userprog/build'
ckdqja0225@ubuntu:~/pintos/project/src_changbeom/userprog$
```

3. bitbucket 주소

git clone

https://seomooneunji@bitbucket.org/pintos_ejcb/project.git

에서 **" src_changbeom 폴더 "** 입니다.