Project #1: User Program

[CSE4070]

Fall 2019

Hyeongu Kang

Contents

Prerequisites

Background

- As we mentioned in 'Pintos Introduction' slides, Pintos is the simple OS which can boot, execute an application, and power off.
- Let's run 'echo' application on Pintos first. **(Run 'make' in src/examples before try this)**

```
~/pintos/src/userprog $ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'
```

Please type "--" (two hyphens) precisely

```
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'echo' into the file system...
Erasing ustar archive...
Executing 'echo x':
Execution of 'echo x' complete.
Timer: 76 ticks
Thread: 0 idle ticks, 76 kernel ticks, 0 user ticks
hda2 (filesys): 26 reads, 172 writes
hda3 (scratch): 83 reads, 2 writes
Console: 818 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

It seems that the 'echo' application have been run properly, but...

Background

- As we mentioned in 'Pintos Introduction' slides, Pintos is the simple OS which can boot, execute an application, and power off.
- Let's run 'echo' application on Pintos first. **(Run 'make' in src/examples before try this)**

```
~/pintos/src/userprog $ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'
```

Please type "--" (two hyphens) precisely

```
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'echo' into the file system...
Erasing ustar archive...
Executing 'echo x':
Execution of 'echo x' complete.
Timer: 76 ticks
Thread: 0 idle ticks, 76 kernel ticks, 0 user ticks
hda2 (filesys): 26 reads, 172 writes
hda3 (scratch): 83 reads, 2 writes
Console: 818 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

We can not see the result of 'echo x' because of **lack of implementation.**

```
[redacted] ~ echo x
x
```

We should be able to see 'x'

Background

- Why can we not see the result of 'echo' command?

Because, in current Pintos,

- **System call** is not implemented
- **System call handler** is not implemented
- **Argument passing** is not implemented
- **User stack** is not implemented

- Basically, current Pintos does not implement many OS functionalities including the aforementioned system call process handling.

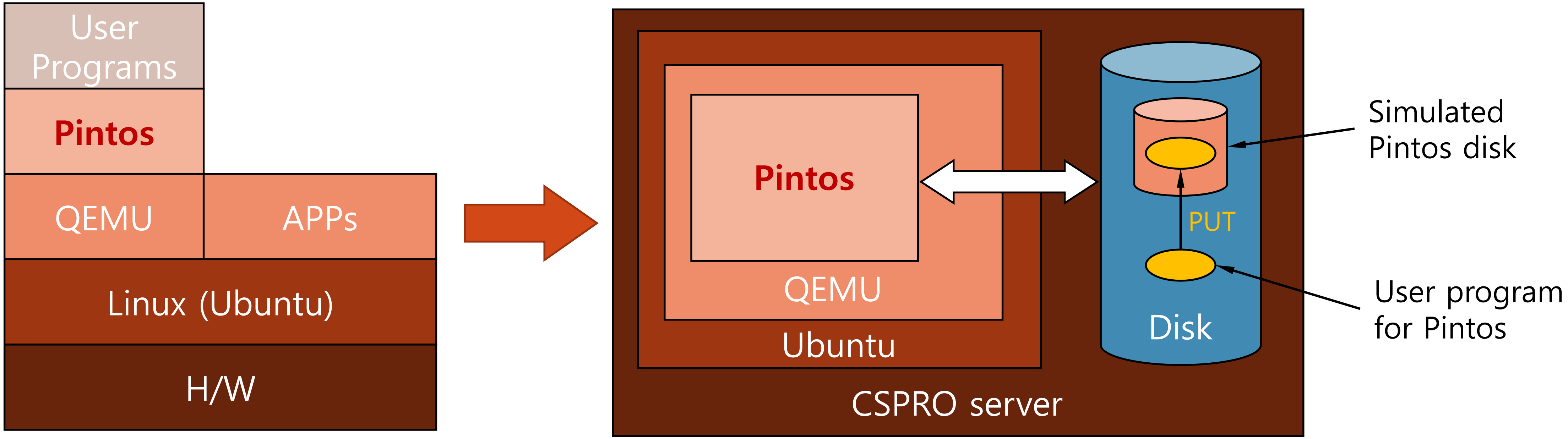
Background

- In this project, students will have to make the Pintos **be able to execute user programs properly.**
  - ➢ Working out of the following directories
- Students will have to modify the following files.

	Files to be modified	Referenced files
src/**userprog**	process.h / **process.c** syscall.h / **syscall.c**	**pagedir.h / pagedir.c** exception.h / exception.c
src/**threads**	**thread.h** / thread.c	synch.h / synch.c vaddr.h
src/**devices**	-	shutdown.h / shutdown.c input.h / input.c
src/**lib**	**syscall-nr.h** **user/syscall.h** **user/syscall.c**	-

How User Program Works

1. Pintos can load and run regular ELF executables.
2. To run user program, you must copy (put) the user program to the simulated file system disk.



How User Program Works

- Let's think of previous example more in detail.

```
~/pintos/src/userprog $ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'
```

"--filesys-size=2": Make simulated Pintos disk which consists of 2MB

"-p ../examples/echo -a echo": Copy '../examples/echo' into the simulated disk and change the name from '../examples/echo' to 'echo'

"--" between echo and -f: Separate pintos options and kernel arguments

"-f": Pintos formats the simulated disk

"-q": Pintos will be terminated after execution of 'echo'

"run 'echo x'": Pintos will execute 'echo' with argument 'x'

How User Program Works

- Let's think of previous example more in detail.

```
~/pintos/src/userprog $ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'
```

✓ 'echo' is the application that writes arguments to the **standard output.**

✓ Thus 'echo' needs the I/O functionality provided by system call in the kernel.

✓ And it also needs user stack implementation which stores arguments and passes it to kernel.

✓ But we don't have any **system call and user stack** implementation now.

That's why we were not able to see the result of 'echo x'

Code Level Flow

$ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run **'echo x'**

threads/init.c

```
int main(){
    …
    syscall_init();
    …
    run_actions(argv);
    …
}
```

threads/init.c

```
Static void run_action(){
    …
    {"run", 2, run_task}
    …
    a->function (argv);
}
```

threads/init.c

```
                                   'echo x'
void run_task(){
    …
    process_wait(process_execute(task));
    …
}
```

useprog/process.c

```
tid_t process_execute(){
    …
    thread_create(…, start_process, …);
    …
}
```

useprog/process.c

```
static void start_process(){
    …
    load(filename);
    ….
}
```

useprog/process.c

```
bool load(filename, ){
    …
    /* parse file name */
    filesys_open(…);
    …
    setup_stack(esp);
    /* construct stack */
}
```


※ Go to 'Argument Passing'

Code Level Flow

$ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run **'echo x'**

Options for pintos utility
'-' for single character option
'--' for multiple characters option
(You can read option executing 'pintos --help')

Separator between pintos options and Pintos kernel arguments

threads/init.c

```
int main(){
}
```

threads/init.c

```
...un_action(){
    run_task}
```

threads/init.c

```
void ru...
    ...
    proc...
    ...
}
```

useprog/process.c

```
tid_t process_execute(){
    ...
    thread_create(..., start_process, ...);
    ...
}
```

useprog/process.c

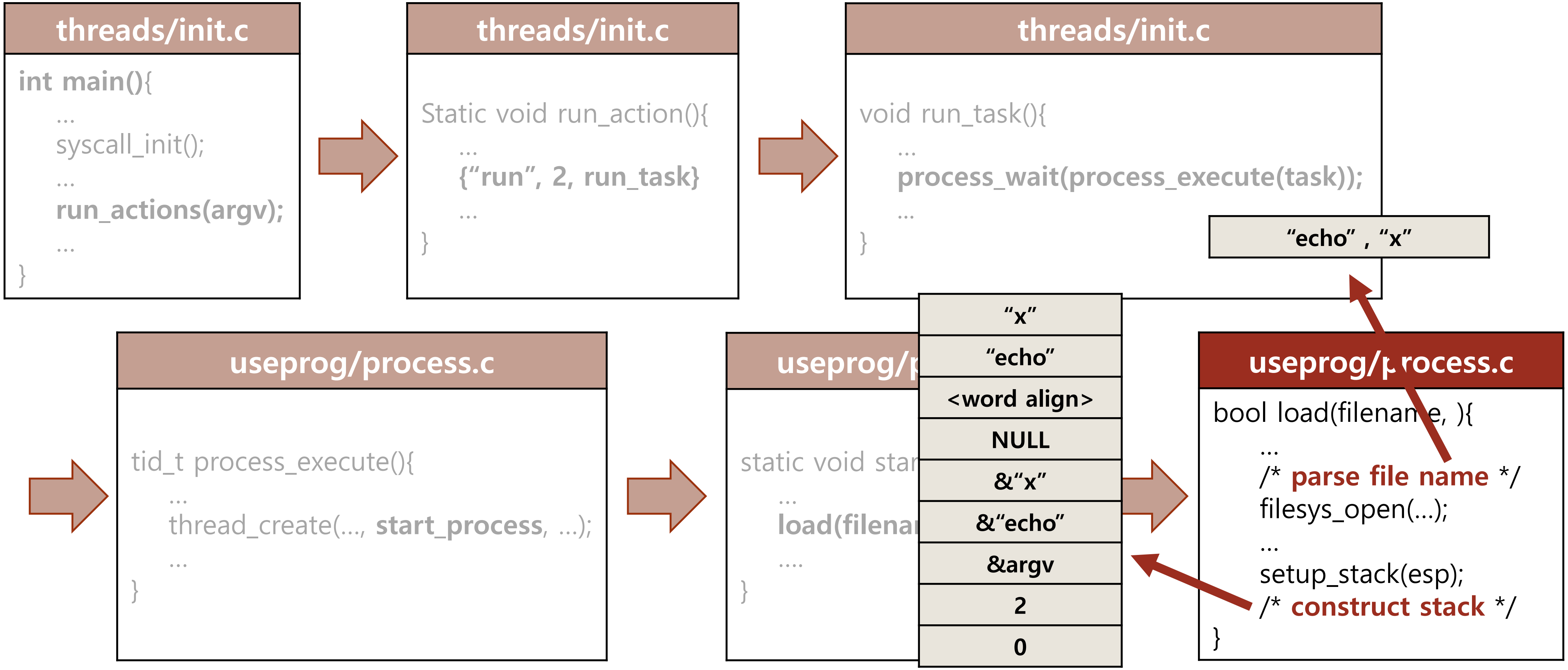
```
static void start_process(){
    ...
    load(filename);
    ....
}
```

useprog/process.c

```
bool load(filename, ){
    ...
    /* parse file name */
    filesys_open(...);
    ...
    setup_stack(esp);
    /* construct stack */
}
```

Code Level Flow

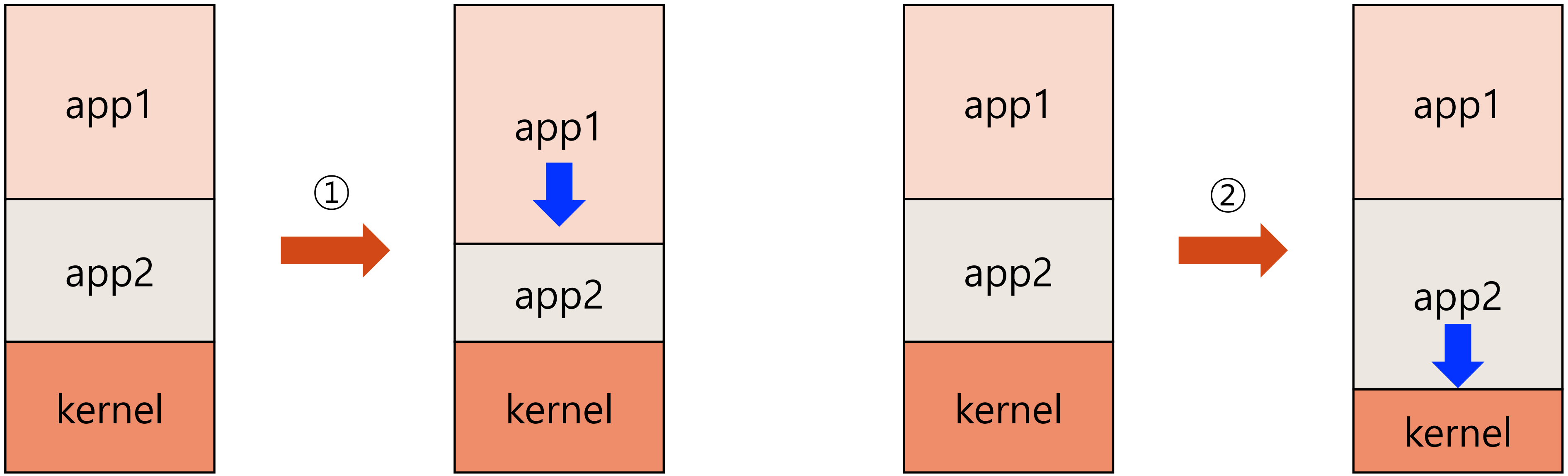
$ pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run **'echo x'**



Setup user stack based on 80x86 calling convention

Virtual Memory

- Now, we know that Pintos lacks system call and user stack implementation.
- These imply that **Pintos divides memory into two region, user memory and kernel memory**.
- If we use these memory area directly, it's **hard to manage memory**.
- For example, there is possibility that ①**each process can be harmed by each other** and a process can ②**harm the kernel code** which is important to run operating systems.



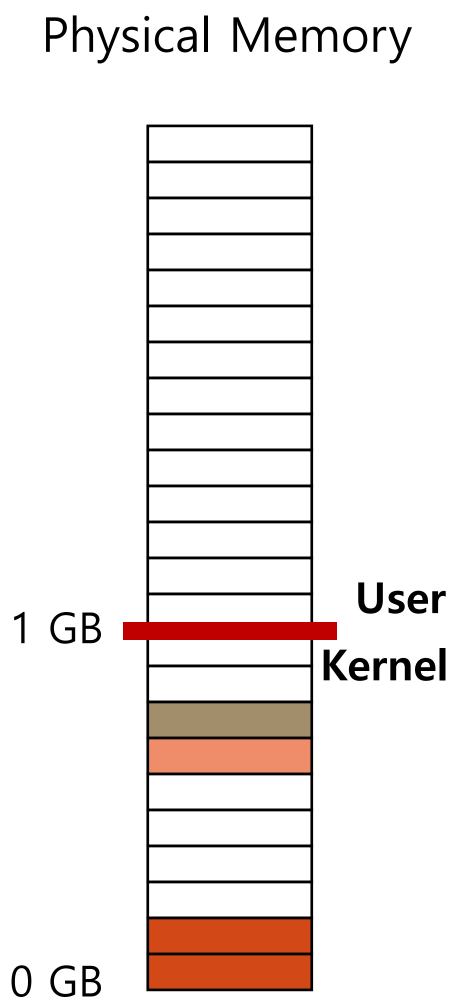
Virtual Memory

- To prevent these problems, operating systems adopt **virtual memory system**.
- Because of virtual memory, **each process can have their own memory area** and use it as if the process occupies whole memory.
- **Pintos also manages memory regions by virtual memory**.
- Virtual memory is also divided into two regions: **user virtual memory and kernel virtual memory**.

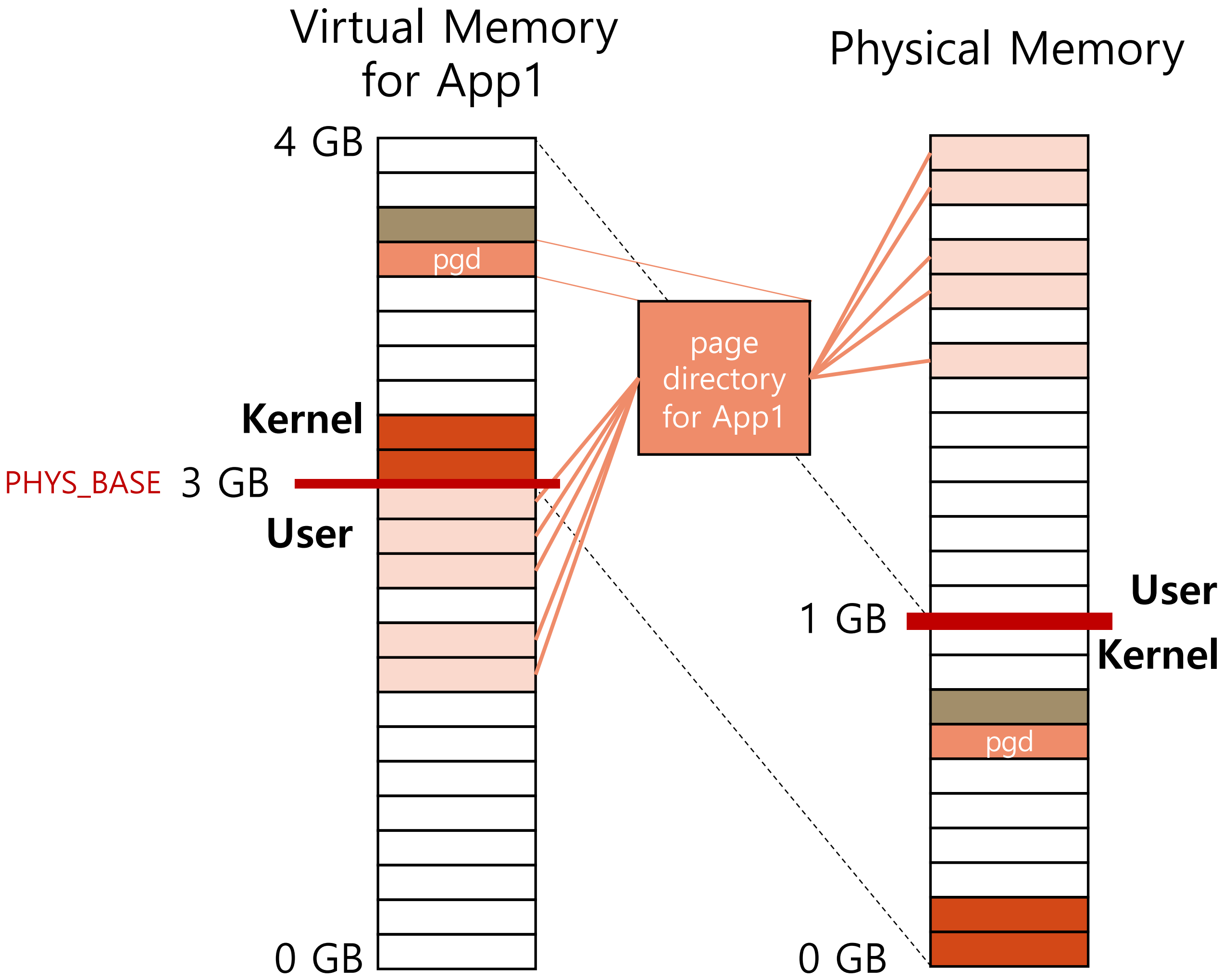
Virtual Memory in Pintos

1. Each process has its own user virtual memory.
2. Pintos allocates 1 GB to kernel as global memory.
   (PHYS_BASE (3 GB) ~ 4 GB in virtual memory)

Virtual Memory: Launch Application

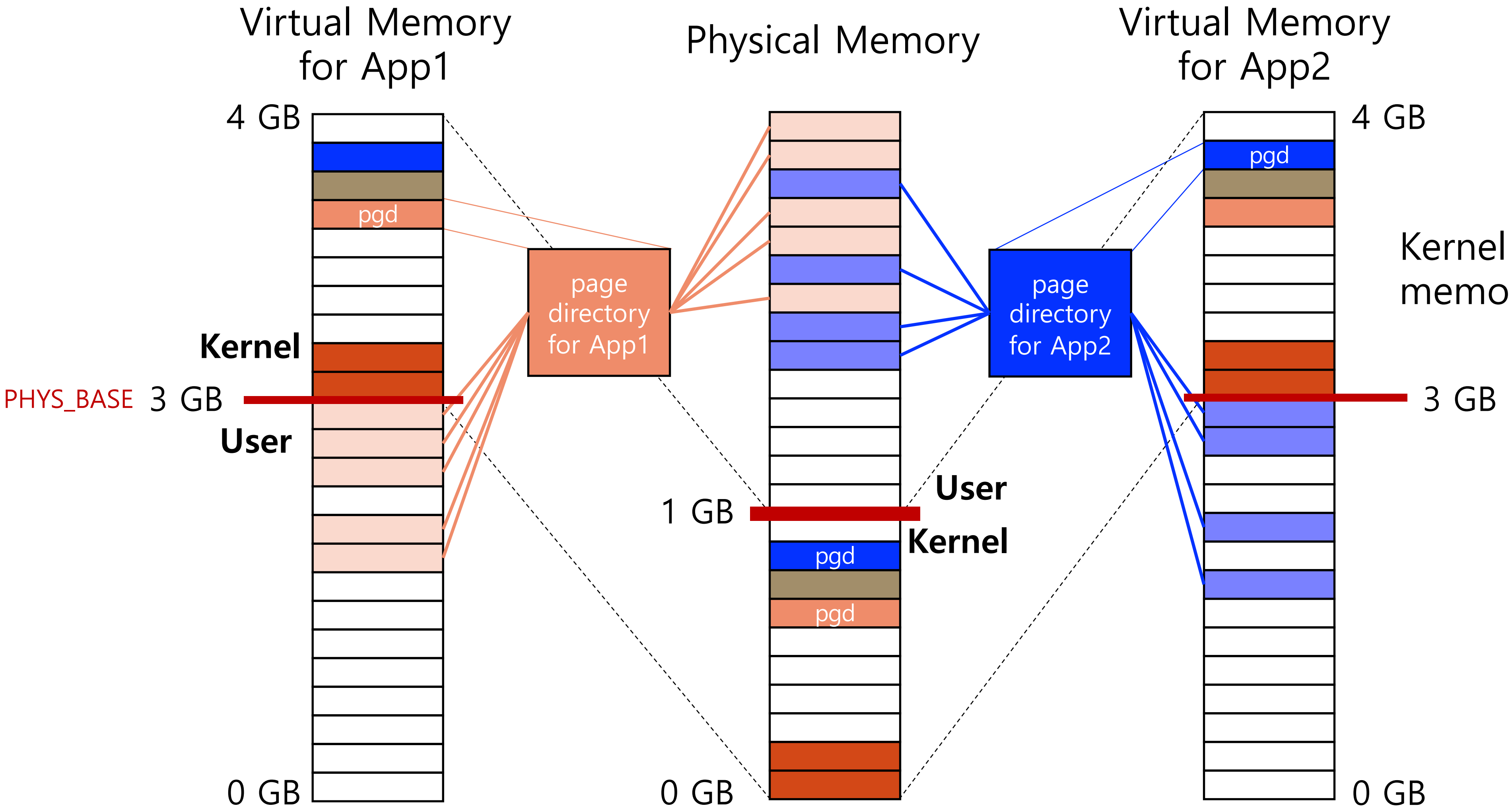


Virtual Memory: Launch Application



Kernel virtual memory is global.

Virtual Memory: Launch Application



Kernel virtual memory is global.

Virtual Memory in Pintos

1. Each process has its own user virtual memory.
2. Pintos allocates 1 GB to kernel as global memory.
   (PHYS_BASE (3 GB) ~ 4 GB in virtual memory)
3. Memory unit is a page in Pintos, which is size of 4 KB.
4. One page is allocated for each thread.
5. User program can access physical memory by translating virtual address via page directory and page table.
   (Refer to A.7 'Page Table')

Virtual Memory

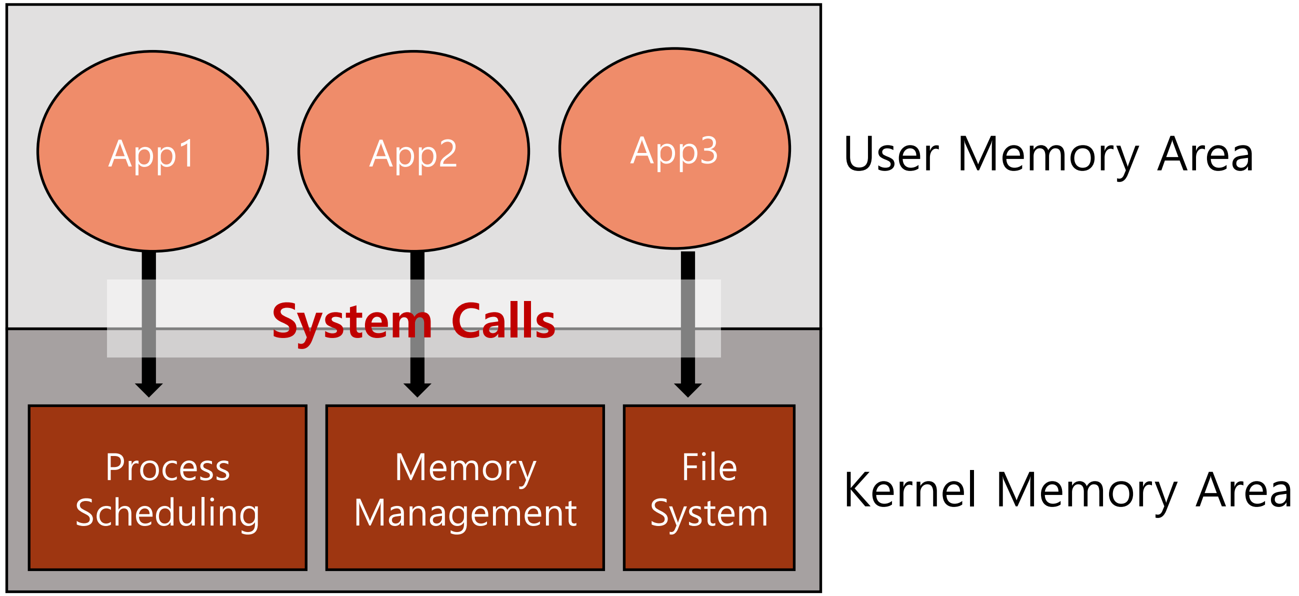
- Functions for page
  1) threads/vaddr.h
     - ✓ `is_user_vaddr(), is_kernel_vaddr()`
       Check that given virtual address is an user/kernel virtual address
     - ✓ `ptov(), vtop()`
       Translate physical address to kernel virtual address and vice versa
  2) threads/palloc.c
     - ✓ `palloc_get_page()`
       Get page from user/kernel memory pool
  3) userprog/pagedir.c
     - ✓ `pagedir_create()`
       Create page directory
     - ✓ `pagedir_get_page()`
       Look up the physical address that corresponds to user virtual address in page directory
     - ✓ `pagedir_set_page()`
       Add mapping in page directory from user virtual address to the physical page

System Calls

- As we've seen, Pintos divides memory into user virtual memory and kernel virtual memory to protect each process and kernel code.
- Along with the concept of virtual memory, OSes prevent user program from accessing the kernel memory area which contains core functionalities.
- Then, how user program uses kernel's functionality?
- OSes provide **system calls** to solve this problem.

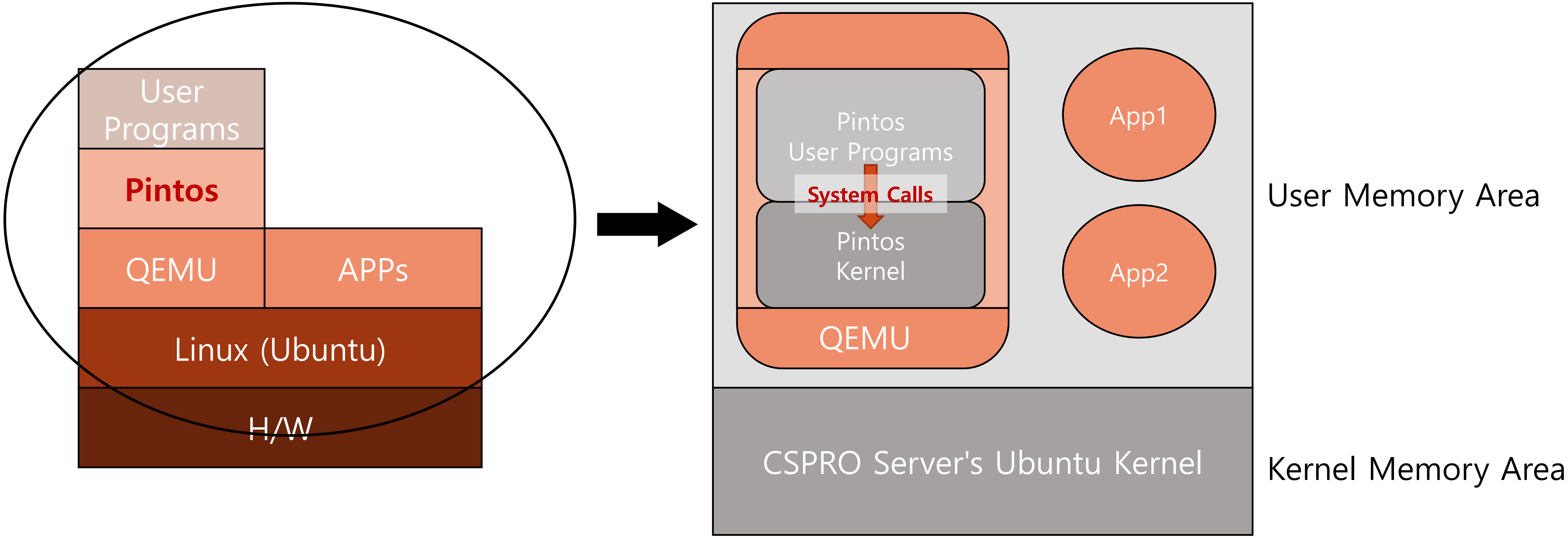
System Calls

- For safety, operating system provides two types of mode, user and kernel mode.
- **When user program is run in user mode, it can not execute new program and access memory or disk**.
- These operations are performed in **kernel mode**.
- Operating system provides **system calls** to enter kernel mode.



System Calls

- Pintos provides **user level interface** of system calls in 'lib/user/syscall.c' and **skeleton of system call handler** in 'userprog/syscall.c'.



System Calls

- Procedure of system call in Pintos
  - ➢ User programs call system call function.

```c
/* cat.c

   Prints files specified on command line to the console. */

#include <stdio.h>
#include <syscall.h>

int
main (int argc, char *argv[])
{
  bool success = true;
  int i;

  for (i = 1; i < argc; i++)
    {
      int fd = open (argv[i]);
      if (fd < 0)
        {
          printf ("%s: open failed\n", argv[i]);
          success = false;
          continue;
        }
```

open() system call

* source code: examples/cat.c

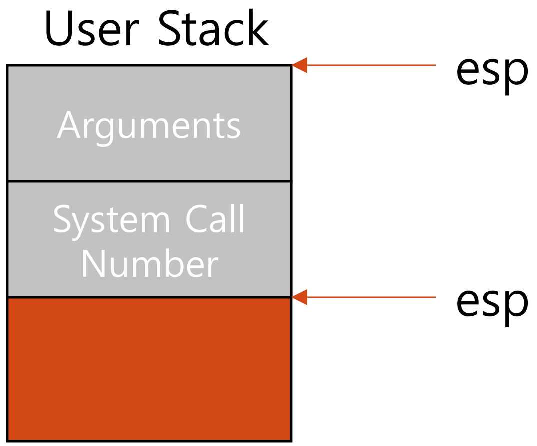
System Calls

- Procedure of system call in Pintos
  - **System call number** and any additional **arguments** are pushed on caller's stack.
  - Invoke interrupt for system call by using **`'int $0x30'`** instruction.

```c
int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}
```

```c
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                   \
        ({                                                       \
          int retval;                                            \
          asm volatile                                           \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                   \
               : [number] "i" (NUMBER),                          \
                 [arg0] "g" (ARG0)                               \
               : "memory");                                      \
          retval;                                                \
        })
```

After returning from system call handler, restore stack pointer



* source code: lib/user/syscall.c

System Calls

- Procedure of system call in Pintos
  - ➤ Set the stack for interrupt and call interrupt handler.

```
.func intr_entry
intr_entry:
    /* Save caller's registers. */
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal

    /* Set up kernel environment. */
    cld          /* String instructions go upward. */
    mov $SEL_KDSEG, %eax    /* Initialize segment registers. */
    mov %eax, %ds
    mov %eax, %es
    leal 56(%esp), %ebp /* Set up frame pointer. */

    /* Call interrupt handler. */
    pushl %esp
.globl intr_handler
    call intr_handler
    addl $4, %esp
.endfunc
```

Call interrupt handler

* source code: threads/intr-stubs.S

System Calls

- Procedure of system call in Pintos
  - `intr_handler()` calls system call hander.

```c
void
intr_handler (struct intr_frame *frame)
{
  bool external;
  intr_handler_func *handler;

  /* External interrupts are special.
     We only handle one at a time (so interrupts must be off)
     and they need to be acknowledged on the PIC (see below).
     An external interrupt handler cannot sleep. */
  external = frame->vec_no >= 0x20 && frame->vec_no < 0x30;
  if (external)
    {
      ASSERT (intr_get_level () == INTR_OFF);
      ASSERT (!intr_context ());

      in_external_intr = true;
      yield_on_return = false;
    }

  /* Invoke the interrupt's handler. */
  handler = intr_handlers[frame->vec_no];
  if (handler != NULL)
    handler (frame);
```

Interrupt handler for **system call handler** have already been registered※ while Pintos was booting

※ Refer to the following function calls in case you're curious:
1) `main()` in 'threads/init.c' calls `syscall_init()` which is in 'userprog/syscall.c'
2) `syscall_init()` calls `intr_register_int()` in 'threads/interrupt.c'

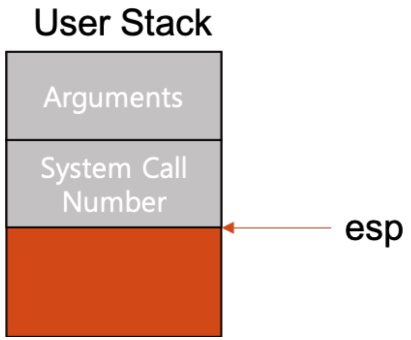
* source code: threads/interrupt.c

System Calls

- Procedure of system call in Pintos
  - ➢ `syscall_handler()` gets control and it can access the stack via **'esp'** member of the `struct intr_frame` (in threads/interrupt.h).
  - ➢ 80x86 convention stores return value of system call in EAX register so that we can store the return value in **'eax'** member of the `struct intr_frame`.

```c
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

※ Pintos provides skeleton of system call handler
We will develop this in this project!




```c
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t ebx;                 /* Saved EBX. */
    uint32_t edx;                 /* Saved EDX. */
    uint32_t ecx;                 /* Saved ECX. */
    uint32_t eax;                 /* Saved EAX. */
    void *esp;                    /* Saved stack pointer. */
    uint16_t ss, :16;             /* Data segment for esp. */
  };
```

* source code: userprog/syscall.c

Requirements

Process Termination Messages

1. When user program terminates, kernel prints termination messages.
   Output form is as follows:

   Process Name: exit(exit status)₩n

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(exec-once) begin
(child-simple) run
child-simple: exit(81)
(exec-once) end
exec-once: exit(0)
EOF
```

<tests/userprog/exec-once.ck>

Refer to the following functions
threads/thread.c: thread_exit()
userprog/process.c: process_exit()

2. Refer to Pintos manual 3.3.2

Process Termination Messages

- How can we get a process name?
  - Refer to `struct thread`

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;
    enum thread_status status;
    char name[16];
```

Process Termination Messages

- How is user program terminated?
  - When ELF user program runs, `_start( )` in lib/user/entry.c is called at first.

```
void
_start (int argc, char *argv[])
{
  exit (main (argc, argv));
}
```

  - After executing the program, `exit( )` system call is called.
  - But Pintos only provides exit() system call API, exit() system call is not implemented.

Process Termination Messages

- How is user program terminated?
  - When ELF user program runs, `_start( )` in lib/user/entry.c is called at first.

```
void
_start (int argc, char *argv[])
{
  exit (main (argc, argv));
}
```

  - Flow of function calls

    exit() in lib/user/syscall.c

    -> syscall1 (SYS_EXIT, status) in lib/use/syscall.c

    -> syscall_handler() in userprog/syscall.c

    -> thread_exit() in threads/thread.c

    -> process_exit() in userprog/process.c

    Refer to slide pg. 25-29

Argument Passing

1. User program can have multiple arguments.

```
[redacted] ~ /bin/ls -l foo bar
-rw-r--r-- 1 root root 0 Sep 11 02:59 bar
-rw-r--r-- 1 root root 0 Sep 11 02:58 foo
```

2. **Parse the arguments and allocate it to memory according to 80x86 calling convention.**
   - Refer to the next slides and Pintos manual 3.5
3. Assume that the length of arguments is less than 4 KB.
   - Test programs use less than 128 Bytes as arguments.

```
PHYS_BASE +----------------------------------+
          |            user stack            |
          |                 |                |
          |                 |                |
          |                 V                |
          |          grows downward          |
          |                                  |
          |                                  |
          |                                  |
          |                                  |
          |           grows upward           |
          |                 ^                |
          |                 |                |
          |                 |                |
          +----------------------------------+
          | uninitialized data segment (BSS) |
          +----------------------------------+
          |     initialized data segment     |
          +----------------------------------+
          |           code segment           |
0x08048000 +----------------------------------+
          |                                  |
          |                                  |
          |                                  |
          |                                  |
          |                                  |
        0 +----------------------------------+
```

<User virtual memory in Pintos>

Argument Passing

- "/bin/ls –l foo bar" will be parsed into "/bin/ls", "-l", "foo", "bar"

Argument Passing

- "/bin/ls –l foo bar" will be parsed into "/bin/ls", "-l", "foo", "bar"

```
  0xC0000000 (PHYS_BASE)
- 0x00000004
-------------
  0xBFFFFFFC
```

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	uint8_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 (*) ()

- You can start implementation of argument passing after the following function.
  - ✓ userprog/process.c : static bool **setup_stack**(void **esp)
  - ✓ Refer to 'Code Level Flow' in the previous chapter

Argument Passing

- "/bin/ls –l foo bar" will be parsed into "/bin/ls", "-l", "foo", "bar"
  - ✓ Push arguments at the top of the stack

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	uint8_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 (*) ()

Argument Passing

- "/bin/ls –l foo bar" will be parsed into "/bin/ls", "-l", "foo", "bar"
  - ✓ Push address of each argument

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	uint8_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 (*) ()

NULL pointer sentinel (required by C standard)

Argument Passing

- "/bin/ls –l foo bar" will be parsed into "/bin/ls", "-l", "foo", "bar"
  - ✓ result of **hex_dump()** : This function is very useful for debug (in src/lib/stdio.c).

```
bffffc0                                              00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |............./bi|
bfffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

0xbffffff8 0xbffffffc foo\0 bar\0

Top of the stack

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	uint8_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 (*) ()

Argument Passing

- In **userprog/process.c**, there is `setup_stack( )` which allocates a minimal stack page (4KB).
- In the given code, it fetches the start address of user program after `setup_stack( )`.
- Since the given code only allocates stack page, we need to make up the stack after `setup_stack( )`.
- Make up the stack referring to **"3.5 80x86 Calling Convention"** in Pintos manual

```c
  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;

  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;
```

Write codes here!

System Calls

1. Implement the following system calls
   (**Requirements** of each system call is described in **Pintos manual 3.3.4**)
   - **`halt`, `exit`, `exec`, `wait`, `read`, `write`**
     (Pintos exec is different from UNIX exec)
   - 2 new system calls (**fibonacci, sum_of_four_int**)
   - **`read`** and **`write`** are special case in this project (refer to the following item)

2. System calls related with file system don't need to implement in this project.
   - create, remove, open, filesize, read, write, seek, tell, close
   - But, **`read` and `write` should perform standard input/output at least**.

System Calls: General System Calls

- halt()
  1) Terminates Pintos by calling **`shutdown_power_off()`**
- exit()
  1) Terminates the current user program, returning status to the kernel

System Calls: General System Calls

- exec()
  1) Create child process
  2) Refer to `process_execute()` in userprog/process.c
- wait()
  1) What wait() system call should do is **wait child process until it finishes its work**.
  2) Check child thread ID is valid
  3) Get the exit status from child thread when the child thread is dead
  4) To prevent termination of process before return from wait(), You can use **busy waiting technique*** or **`thread_yield()`** in threads/thread.c

* Busy waiting means the codes **do nothing** and just keep checking condition over and over

System Calls: General System Calls

- write() and read()
  1) File Descriptor (Similar as FILE* in standard C)
     - ✓ Return value of `open( )`, `create( )`
     - ✓ Each thread has its own FD in Pintos.
  2) File Descriptor of STDIN, STDOUT
     - ✓ STDIN = 0, STDOUT = 1
  3) Employ the following functions to implement read(0)
     - ✓ pintos/src/devices/input.c : uint8_t **input_getc**(void)
  4) Employ the following functions to implement write(1)
     - ✓ pintos/src/devices/input.c : uint8_t **input_putc**(void)

System Calls: Code Level Flow

- Let's start from **`main()`** in threads/init.c
- **`run_actions (argv);`** will be invoked.

```
/* Run actions speci
run_actions (argv);

/* Finish up. */
shutdown ();
thread_exit ();
```


* threads/init.c

System Calls: Code Level Flow

- Focus on {"run", 2, run_task}
- **`a->function (argv);`** invokes **`run_task()`**.

```
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;
      int argc;
      void (*function) (char **argv);
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
```

```
while (*argv != NULL)
  {
    const struct action *a;
    int i;

    /* Find action name. */
    for (a = actions; ; a++)
      if (a->name == NULL)
        PANIC ("unknown action `%s' (use
      else if (!strcmp (*argv, a->name))
        break;

    /* Check for required arguments. */
    for (i = 1; i < a->argc; i++)
      if (argv[i] == NULL)
        PANIC ("action `%s' requires %d a

    /* Invoke action and advance. */
    a->function (argv);
    argv += a->argc;
  }
```


* threads/init.c

System Calls: Code Level Flow

- **run_task()** invokes **process_execute()**.

```
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

What does 'task' contain?
Does it contain only file name?
Refer to print statement
strtok_r() in lib/string.c will help you

* threads/init.c

System Calls: Code Level Flow

- **thread_create()** will enroll user program name and function to launch user program, **start_process()**.

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

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```


* userprog/process.c

System Calls: Code Level Flow

- When process scheduling is invoked, the child process (user program) will be executed by wrapper function of **`_start()`** in lib/user/entry.c

```
void
_start (int argc, char *argv[])
{
  exit (main (argc, argv));
}
```


* lib/user/entry.c

System Calls: Code Level Flow

- When ELF executable (user program) is finished, **`exit()`** system call is called.
- But **`exit()`** system call is not implemented yet.
- What **`exit()`** system call should do is invoking **`thread_exit()`** which invokes **`process_exit()`**.
- Anyway, after **`exit()`** system call, it returns to **`process_wait()`**.

```c
void
_start (int argc, char *argv[])
{
  exit (main (argc, argv));
}
```


* lib/user/entry.c

System Calls: Code Level Flow

- Parent process which ran **process_execute()** should be waiting in **process_wait()** until the child process is finished.

```c
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```


* threads/init.c

System Calls: Source Codes

1. lib/user/syscall.h and lib/user/syscall.c
   - APIs for general system calls are already given in Pintos code.
   - **You don't need to add something for general system call APIs**.
2. userprog/syscall.h
   - There is only one prototype `syscall_init()` which register system call interrupts when Pintos was booted.
   - You would write prototype of general system calls here.
3. userprog/syscall.c
   - You have to make **`syscall_handler()`** handle system calls.
   - If you have done argument passing, you can get system call number from **`intr_frame *f`**.
   - **`esp`** member of **`intr_frame *f`** contains system call number.
     (You can refer to lib/syscall-nr.h to check each system call number)
   - And then you can use switch statement to classify system calls.
     (What really these system calls do would be written here)

Additional System Calls

- **Implement new system calls into Pintos**
  1. **int fibonacci(int n)**
     - ✓ Return N th value of Fibonacci sequence
  2. **int sum_of_four_int(int a, int b, int c, int d)**
     - ✓ Return the sum of a, b, c and d

  ※ Please use the correct name of system calls.

Additional System Calls

- **Write user level program which uses new system calls**

  1. Make sum.c in pintos/src/examples
  2. Write simple example by using new system calls
  3. **Name of execution file should be 'sum'**
  4. Usage : ./sum [num 1] [num 2] [num 3] [num 4]

     Function : Print the result of 'fibonacci' system call using [num 1] as parameter

     Print the result of 'sum_of_four_int' system call using [num 1, 2, 3, 4] as parameter

     Example :

     ```
     $ ./sum 10 20 40 56
     55 126
     ```

  5. Run the following commands after implementing new system calls

     ```
     pintos/src/userprog/build$ pintos –p ../../examples/sum/ -a sum -- -f –q run ‘sum 10 20 40 56’
     ```

Additional System Calls

1. How to identify system call number?
   - ✓ Refer to 'lib/syscall-nr.h'
2. How to return system call's result?
   - ✓ Check argument `'struct intr_frame'` of **syscall_handler()** in syscall.c
   - ✓ Refer to 'System Calls' in Prerequisites

Additional System Calls

- To compile newly added user program, "sum", you need to modify **Makefile** in **src/examples**.
- Refer to how other user programs are written in **Makefile**

Additional System Calls: Source Codes

1. lib/user/syscall.h
   - Write prototype of 2 new system call APIs
2. lib/user/syscall.c
   - Define new `syscall4()` function for `sum_of_four_int()` system call API
   - Define `fibonacci()` and `sum_of_four_int()` system calls APIs
3. lib/syscall-nr.h
   - Add system call numbers for 2 new system calls
4. userprog/syscall.h
   - Write prototype of 2 new system calls
5. userprog/syscall.c
   - Define `fibonacci()` and `sum_of_four_int()` system calls
   - What really these system calls do would be written here.

Accessing User Memory

1. User program can pass a invalid pointer.
   - **NULL** pointer such as **`open (NULL);`** in tests/userprog/open-null.c
   - **Unmapped** virtual memory
   - Pointer to **kernel** address space
2. **Invalid pointers must be rejected** without harm to kernel or other running process.
3. It can be implemented in 2 ways:
   1) Verify the **validity of a user-provided pointer**, then dereference it.
   2) Check only that a user pointer points below PHYS_BASE, then dereference it.
      If the pointer is invalid, it will cause a "page fault" that you can handle by modifying the code **page_fault()** in 'userprog/exception.c'
4. Refer to Pintos manual 3.1.5

Accessing User Memory

```
0xc0000000 (3 GB) PHYS_BASE +----------------------------------+
                            |            user stack            |
                            |                 |                |
                            |                 |                |
                            |                 V                |
                            |          grows downward          |
                            |                                  |
                            |                                  |
                            |                                  |
                            |                                  |
                            |           grows upward           |
                            |                 ^                |
                            |                 |                |
                            |                 |                |
                            +----------------------------------+
                            | uninitialized data segment (BSS) |
                            +----------------------------------+
                            |     initialized data segment     |
                            +----------------------------------+
                            |           code segment           |
                 0x08048000 +----------------------------------+
                            |                                  |
                            |                                  |
                            |                                  |
                            |                                  |
                            |                                  |
                          0 +----------------------------------+
```

Refer to Pintos manual 3.1.4

Mapped User Virtual Memory

Unmapped User Virtual Memory

Accessing User Memory

- To verify the validity of a user-provided pointer, you can use functions in **userprog/pagedir.c** and **threads/vaddr.h**
- `pagedir_get_page()` returns the kernel virtual address corresponding **(mapped)** to the given user virtual address.
  -> Check **Unmapped** virtual memory by using this function
- `is_user_vaddr()` and `is_kernel_vaddr()` check that the given virtual address is user virtual address and kernel virtual address respectively.
  -> Check pointer to **kernel** address space
- Employ these functions to verify the validity of given pointer

Suggested Order of Implementation

Refer to Pintos manual 3.2

- You can not see the result until you implement core functionalities.
- Thus, read this slides and Pintos manual first, design the structures and then start to implement.

1) **Argument Passing**: After implementing it, check the result using hex_dump()
2) **User Memory Access**: Make a plan for protecting user memory accesses from system calls
3) **System Call Handler**: Implement syscall_handler() to handle system call
4) **System Call Implementation**: Implement **exec(), exit(), write(), read()** first
5) **Additional Implementation**: Implement fibonacci(), sum_of_four_int()

※ Refer to source codes in 'src/tests/userprog'

Suggested Order of Implementation

Refer to Pintos manual 3.2

- You can not see the result until you implement core functionalities.
- Thus, read this slides and Pintos manual first, design the structures and then start to implement.

1) **Argument Passing:** After implementing it, check the result using hex_dump()
2) **User Memory Access:** Make a plan for protecting user memory accesses from system calls
3) **System Call Handler:** Implement syscall_handler() to handle system call
4) **System Call Implementation:** Implement **exec(), exit(), write(), read()** first
5) **Additional Implementation:** Imple

※ Refer to source codes in src/tests/userp

Refer to Code Level Flow

src/userprog/process.c : load()
Check parameters of load()

If you want to check the dump values before implementing process_wait(), insert infinite loop in process_wait() to block process
(You should finish to implement process_wait() later)

Suggested Order of Implementation

Refer to Pintos manual 3.2

- You can not see the result until you implement core functionalities.
- Thus, read this slides and Pintos manual first, design the structures and then start to implement.

1) **Argument Passing**: After implementing it, check the result using hex_dump()
2) **User Memory Access:** Make a plan for protecting user memory accesses from system calls
3) **System Call Handler**: Implement syscall_handler() to handle system call
4) **System Call Implementation**: Implement **exec(), exit(), write(), read()** first
5) **Additional Implementation**: Implement fibonacci(), sum_of_four_int()

※ Refer to source codes in src/tests/userprog

Refer to src/threads/vaddr.h

Recommend to implement the function which checks the validity of given address

Suggested Order of Implementation

Refer to Pintos manual 3.2

- You can not see the result until you implement core functionalities.
- Thus, read this slides and Pintos manual first, design the structures and then start to implement.

1) **Argument Passing**: After implementing it, check the result using hex_dump()
2) **User Memory Access**: Make a plan for protecting user memory accesses from system calls
3) **System Call Handler:** Implement syscall_handler() to handle system call
4) **System Call Implementation**: Implement **exec(), exit(), write(), read()** first
5) **Additional Implementation**: Implement fibonacci(), sum_of_four_int()

※ Refer to source codes in src/tests/userprog

src/userprog/syscall.c : syscall_handler()
Check argument 'struct intr_frame' of syscall_handler() in syscall.c
(struct intr_frame is in src/threads/interrupt.h)

Suggested Order of Implementation

Refer to Pintos manual 3.2

- You can not see the result until you implement core functionalities.
- Thus, read this slides and Pintos manual first, design the structures and then start to implement.

1) **Argument Passing**: After implementing it, check the result using hex_dump()
2) **User Memory Access**: Make a plan for protecting user memory accesses from system calls
3) **System Call Handler**: Implement syscall_handler() to handle system call
4) **System Call Implementation:** Implement **exec(), exit(), write(), read()** first
5) **Additional Implementation**: Implement fibonacci(), sum_of_four_int()

※ Refer to source codes in src/tests/userprog

Synchronization will be needed
(You can use busy waiting)
exit status is -1 when syscall_handler is terminated in abnormal way

Suggested Order of Implementation

Refer to Pintos manual 3.2

- You can not see the result until you implement core functionalities.
- Thus, read this slides and Pintos manual first, design the structures and then start to implement.

1) **Argument Passing**: After implementing it, check the result using hex_dump()
2) **User Memory Access**: Make a plan for protecting user memory accesses from system calls
3) **System Call Handler**: Implement syscall_handler() to handle system call
4) **System Call Implementation**: Implement **exec(), exit(), write(), read()** first
5) **Additional Implementation:** Implement fibonacci(), sum_of_four_int()

※ Refer to source codes in src/tests/userp

Modify the followings:
src/lib/syscall-nr.h
src/lib/syscall.h
src/lib/syscall.c

Evaluation

Refer to Pintos manual 1.2.1

1. **21 of 76 tests** in this project will be graded.
   (Refer to the test case list in the next slide)
2. Total score is 100 which consists of 80 for test cases and 20 for documentation.
3. Grading script (***make grade*** or ***make check*** in src/userprog) provided by Pinots will be used.
4. Refer to **'grade'** and **'results'** files in src/userprog/build after grading
   ('grade' file is only created when you use ***make grade***)

Evaluation

Refer to Pintos manual 1.2.1

5. Test cases are classified in functionality test and robustness test.
6. Refer to the followings for checking each test case's point based on the test type
   - pintos/src/tests/userprog/Rubric.functionality
   - pintos/src/tests/userprog/Rubric.robustness
   - Functionality and robustness gets 50% of total score respectively.

➢ We do not follow the score ratio of types shown in pintos/src/tests/userprog/Grading

Evaluation: Test Cases (21 tests)

Functionality		
No.	Name	Point
1	args-none	3
2	args-single	3
3	args-multiple	3
4	args-many	3
5	args-dbl-space	3
6	exec-once	5
7	exec-multiple	5
8	exec-arg	5
9	wait-simple	5
10	wait-twice	5
11	multi-recurse	15
12	exit	5
13	halt	3
Total		63

Robustness		
No.	Name	Point
1	exec-bad-ptr	3
2	exec-missing	5
3	sc-bad-arg	3
4	sc-bad-sp	3
5	sc-boundary	5
6	sc-boundary-2	5
7	wait-bad-pid	5
8	wait-killed	5
Total		34

Evaluation

Refer to Pintos manual 1.2.1

- If you see src/tests/userprog/Grading, functionality test set takes 35% and robustness test set takes 25% of total score.
- But we do not follow this.
- Each type of test set takes 50% respectively.
- Thus, total score is

$$\left(\frac{\textbf{Functionality points}}{\textbf{63}} \times \textbf{50} + \frac{\textbf{Robustness points}}{\textbf{34}} \times \textbf{50}\right) / \textbf{100} \times \textbf{80}$$

Remaining 20 is for documentation

Documentation

- Use the document file uploaded on e-class
- Documentation accounts for 20% of total score.
  (Development 80%, Documentation 20%)

Submission

- Before submission, check that you performed the following:
  1. **Make clean** before compressing
  2. Submission form (Refer to the next slide)
  3. Once you copy other's codes, you will get **F grade**

Submission

- It is a **team project**.
- Due date : **2019. 11. 3 23:59**
- Submission
  - The form of submission file is as follows:

Name of compressed file	Example (project 1, Group #7)
os_prj1_##.tar.gz	os_prj1_07.tar.gz

  - Refer to 'OS project guide' explaining how to compress Pintos code
  - Only one person of group should submit the file.
  - **You should also submit the design document in hardcopy (AS916).**
    (Due date of hardcopy is same as the compressed file)

Project Schedule

Projects	Points	Contents	Periods	Lectures
Project 0-1	1	Installing Pintos	9/16 – 9/22	Manual will be provided
Project 0-2	3	Pintos Data Structures	9/21 – 10/6	9/21 (Sat.)
Project 1	**6**	**User Programs (1)**	**10/5 – 11/3**	**10/5 (Sat.)**
Project 2	4	User Programs (2)	11/2 – 11/17	11/2 (Sat.)
Project 3	6	Threads	11/16 – 12/8	11/16 (Sat.)

※ Once you copy other's codes, you will get F grade