

# Project #1: User Program

[CSE4070]

Fall 2019

Hyeongu Kang



# Contents

---

## 1. Prerequisites

- Background
- How User Program Works
- Code Level Flow
- Virtual Memory
- System Calls

## 2. Requirements

- Process Termination Messages
- Argument Passing
- System Calls

## 3. Suggested Order of Implementation

## 4. Evaluation

## 5. Documentation

## 6. Submission

# 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 --fileys-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 (fileys): 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 --fileys-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 (fileys): 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**.



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

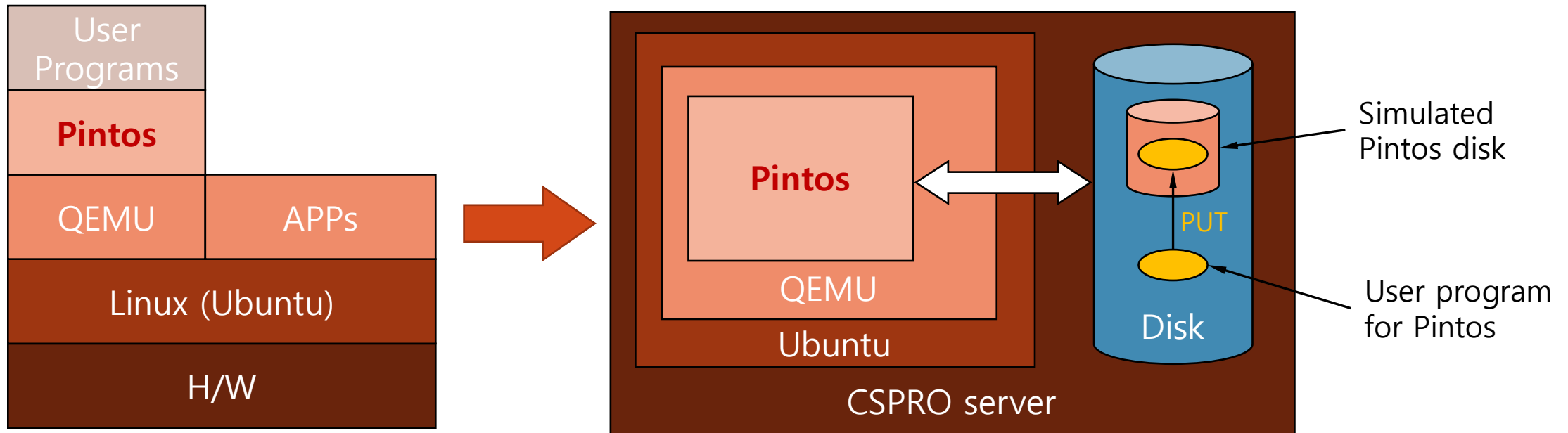
---

- 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/ <b>userprog</b>	process.h / <b>process.c</b> syscall.h / <b>syscall.c</b>	<b>pagedir.h / pagedir.c</b> exception.h / exception.c
src/ <b>threads</b>	<b>thread.h / thread.c</b>	synch.h / synch.c vaddr.h
src/ <b>devices</b>	-	shutdown.h / shutdown.c input.h / input.c
src/ <b>lib</b>	<b>syscall-nr.h</b> <b>user/syscall.h</b> <b>user/syscall.c</b>	-

# 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 --fileys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'
```

**"--fileys-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.

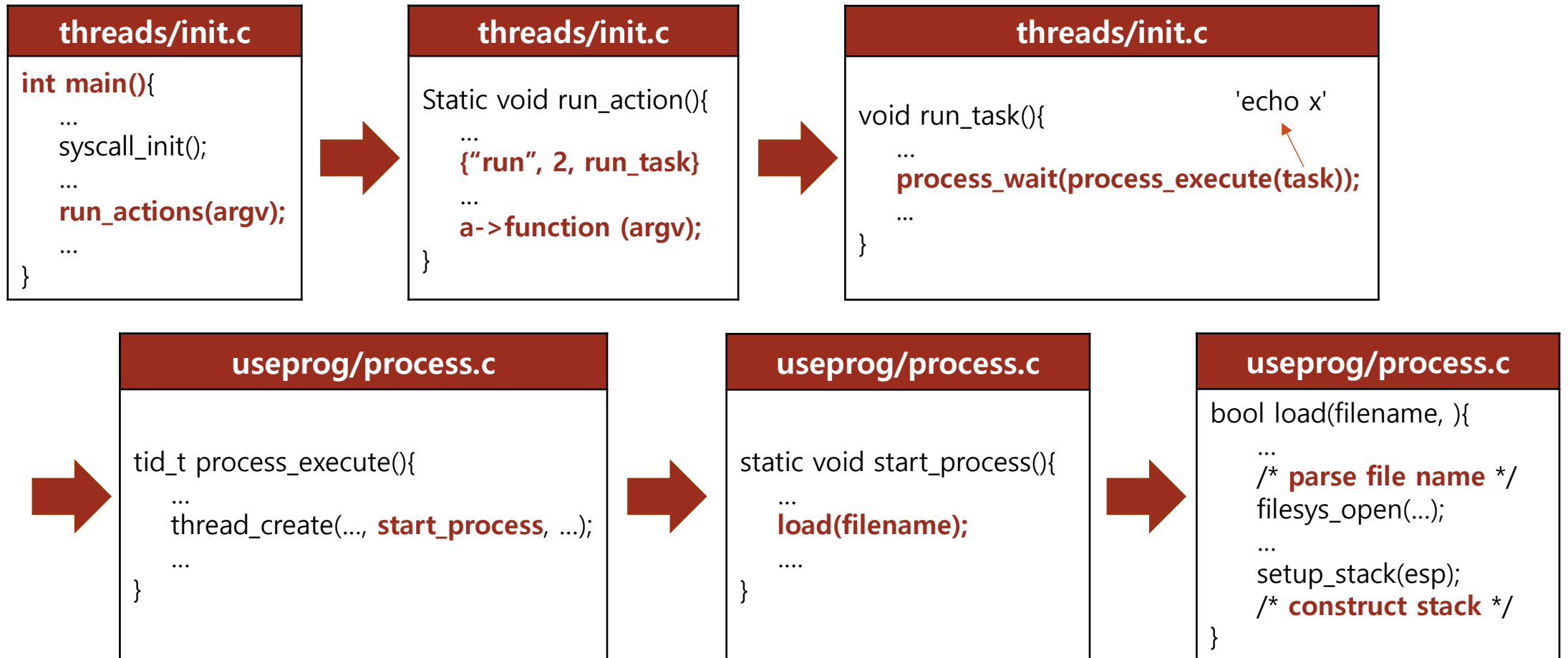
```
~/pintos/src/userprog $ pintos --fileys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'
```

- ✓ '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'**

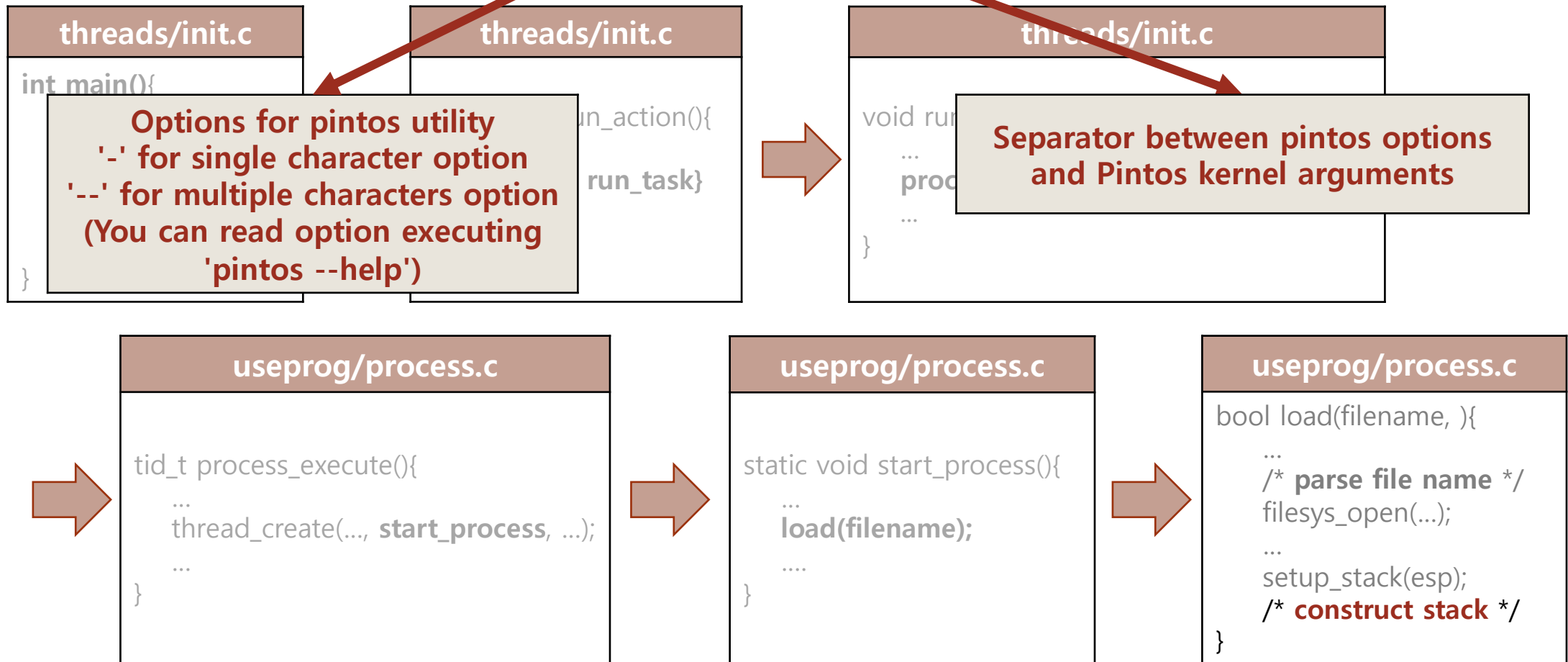
# Code Level Flow

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



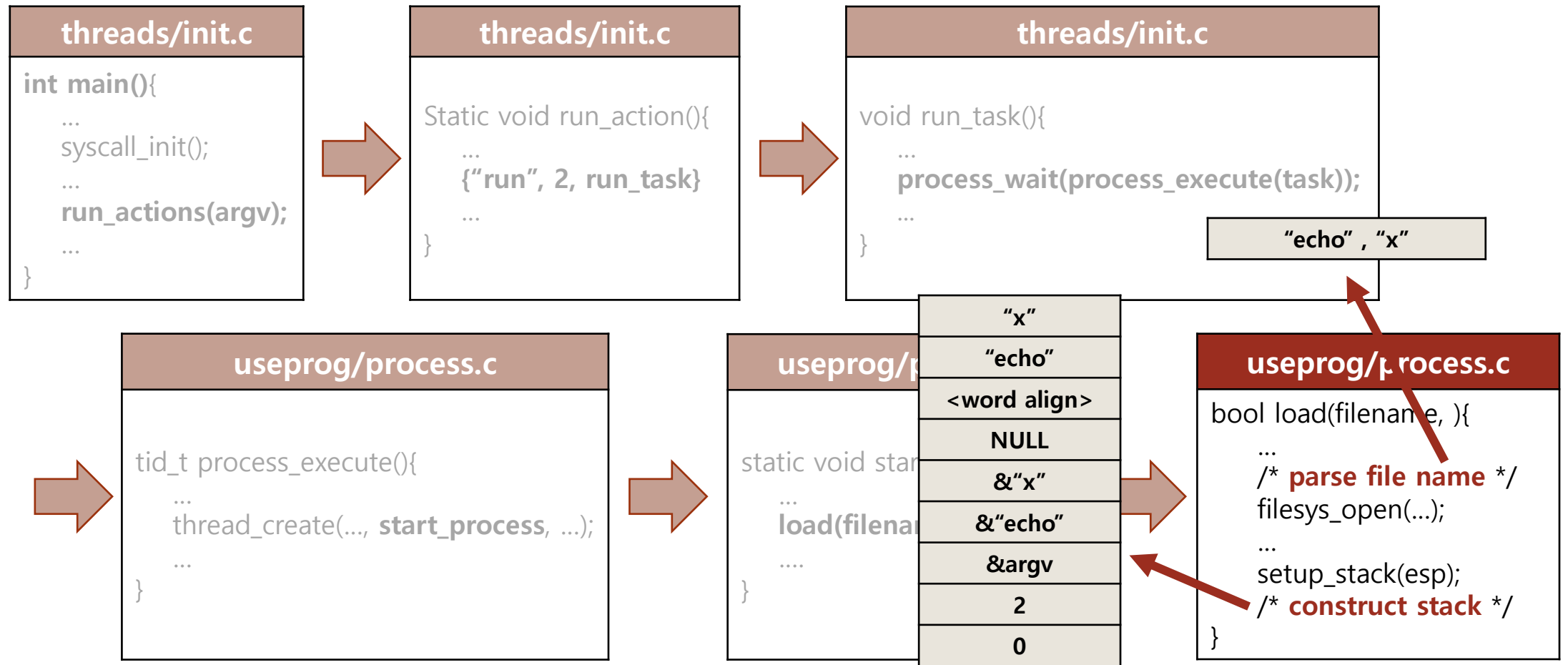
# Code Level Flow

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



# Code Level Flow

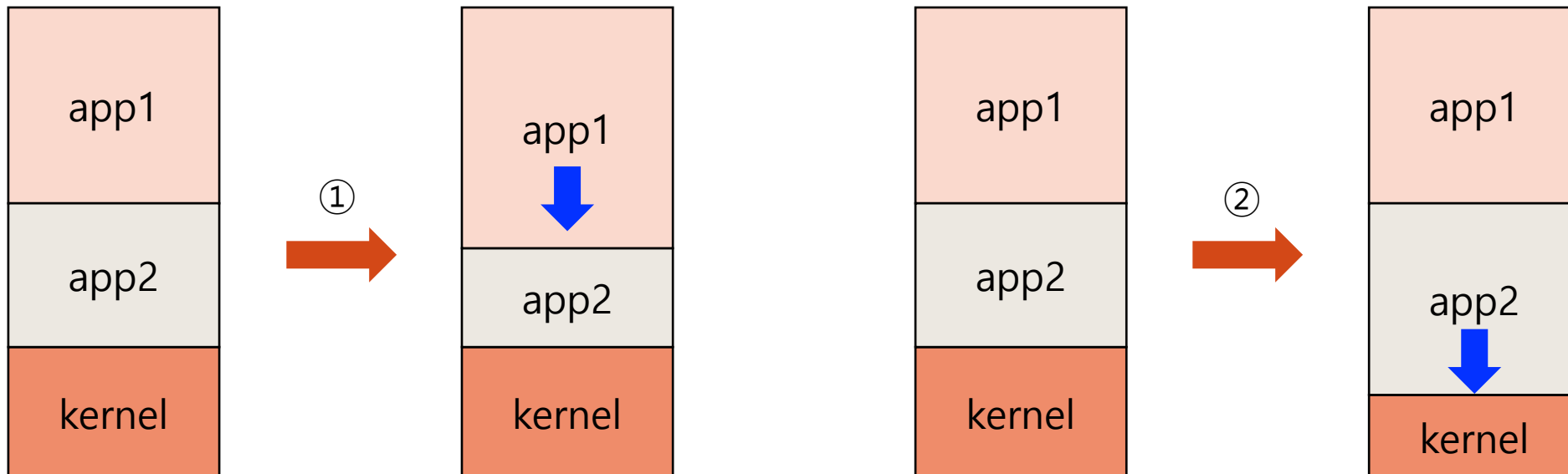
\$ pintos --fileysys-size=2 -p ../examples/echo -a echo -- -f -q run **'echo x'**



Setup user stack based on 80x86 calling convention 13

# 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

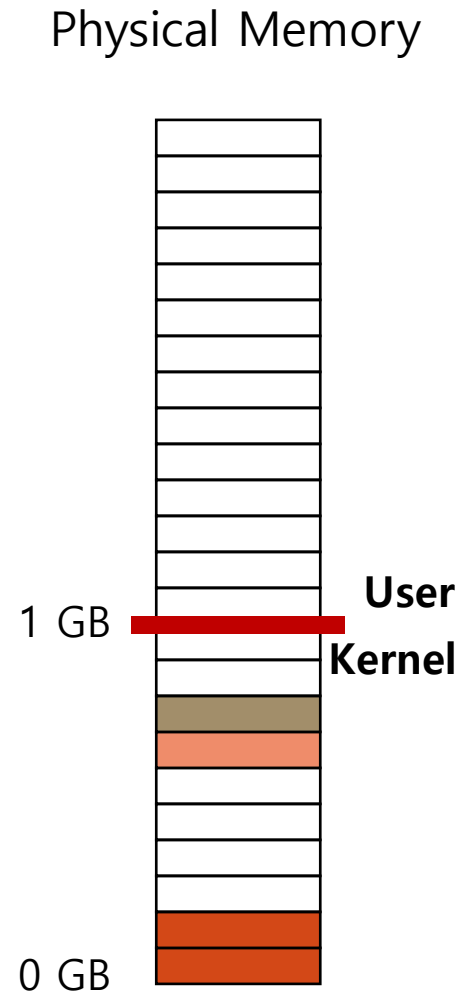
---

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)

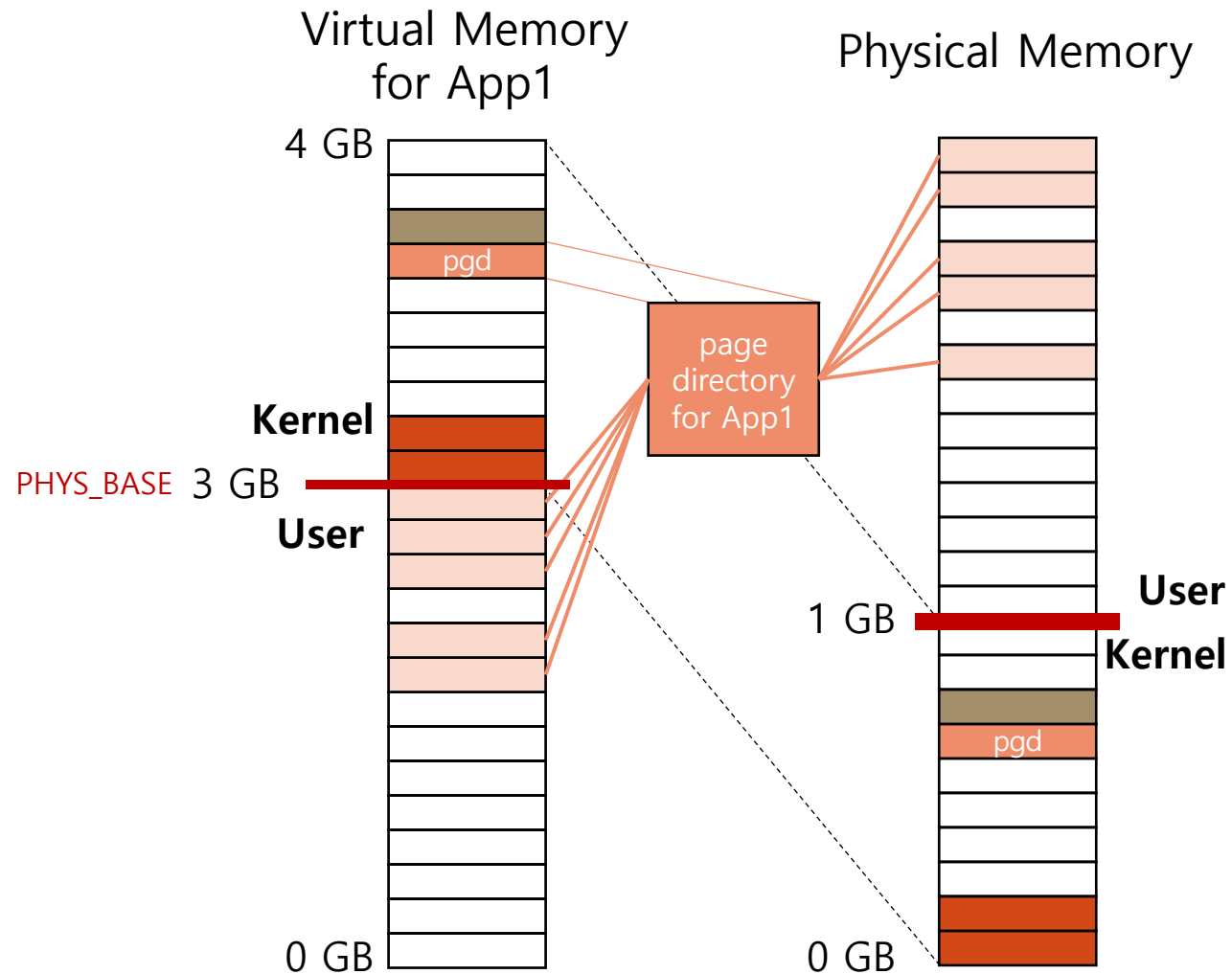


# Virtual Memory: Launch Application

---

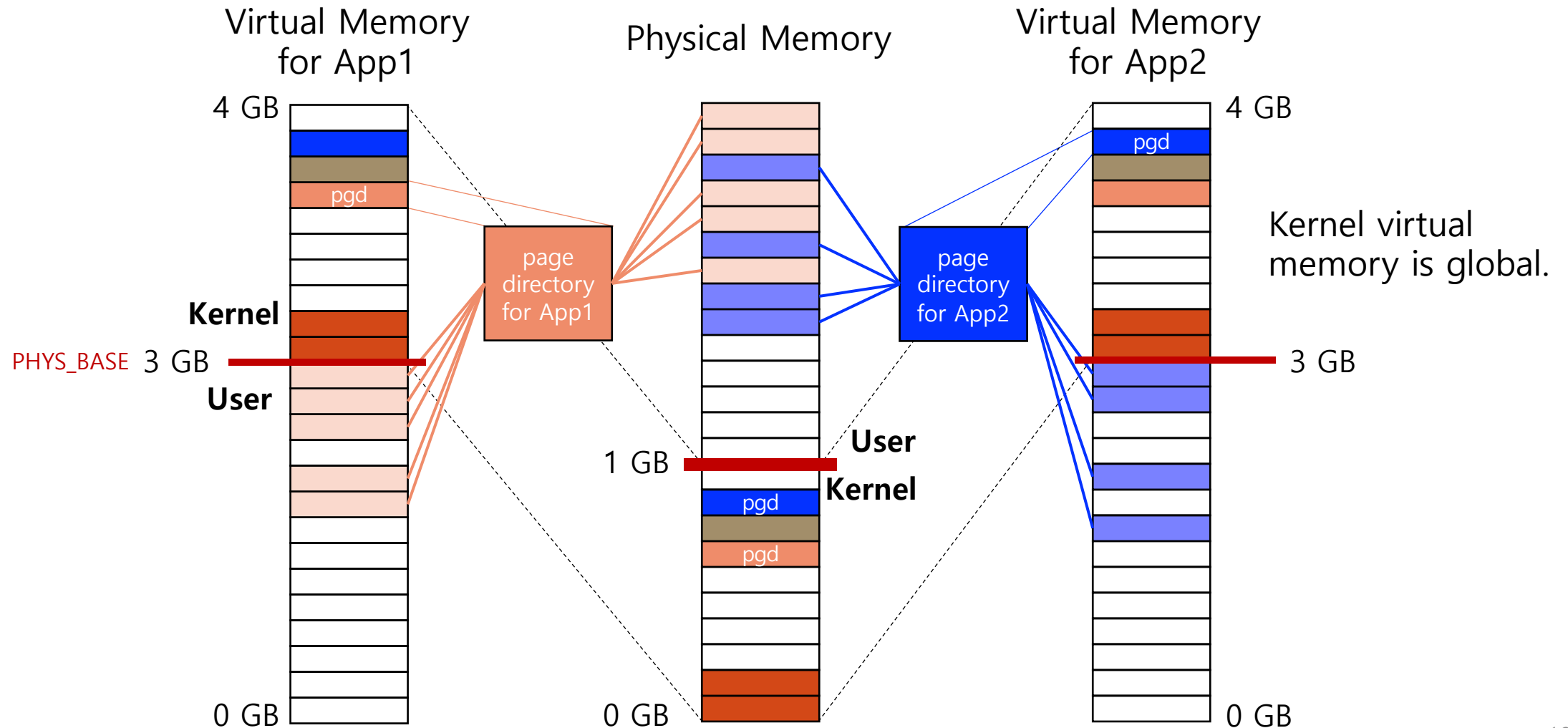


# Virtual Memory: Launch Application



Kernel virtual memory is global.

# Virtual Memory: Launch Application



# 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()`, `vtov()`

- 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

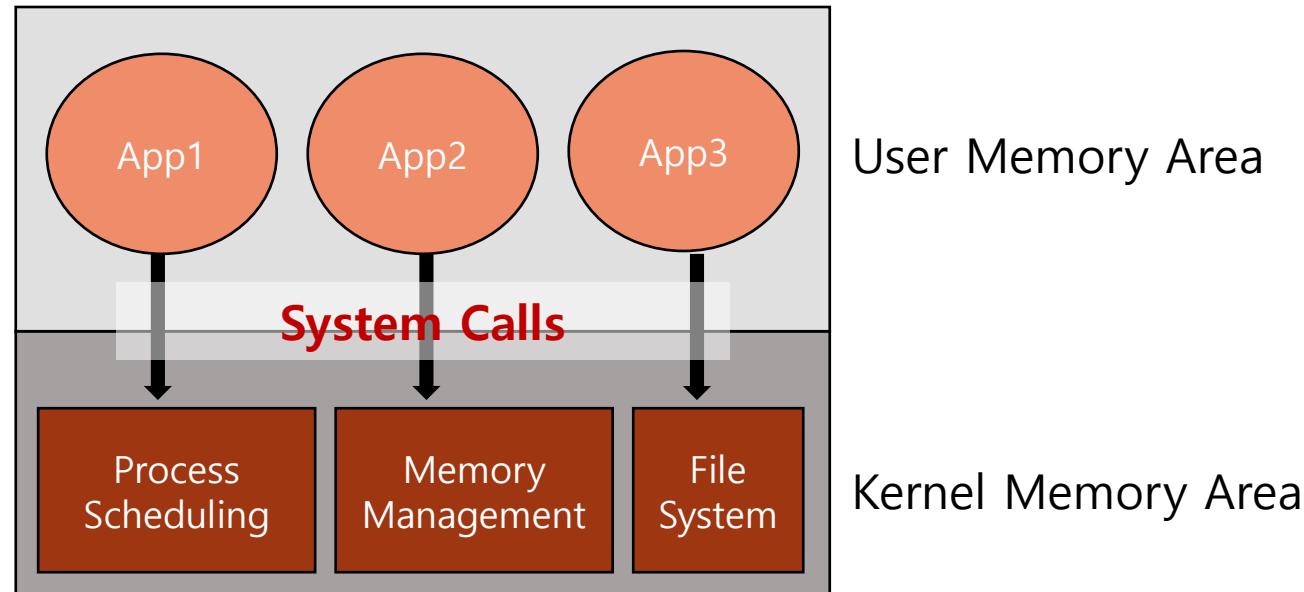
# System Calls

---

- 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.

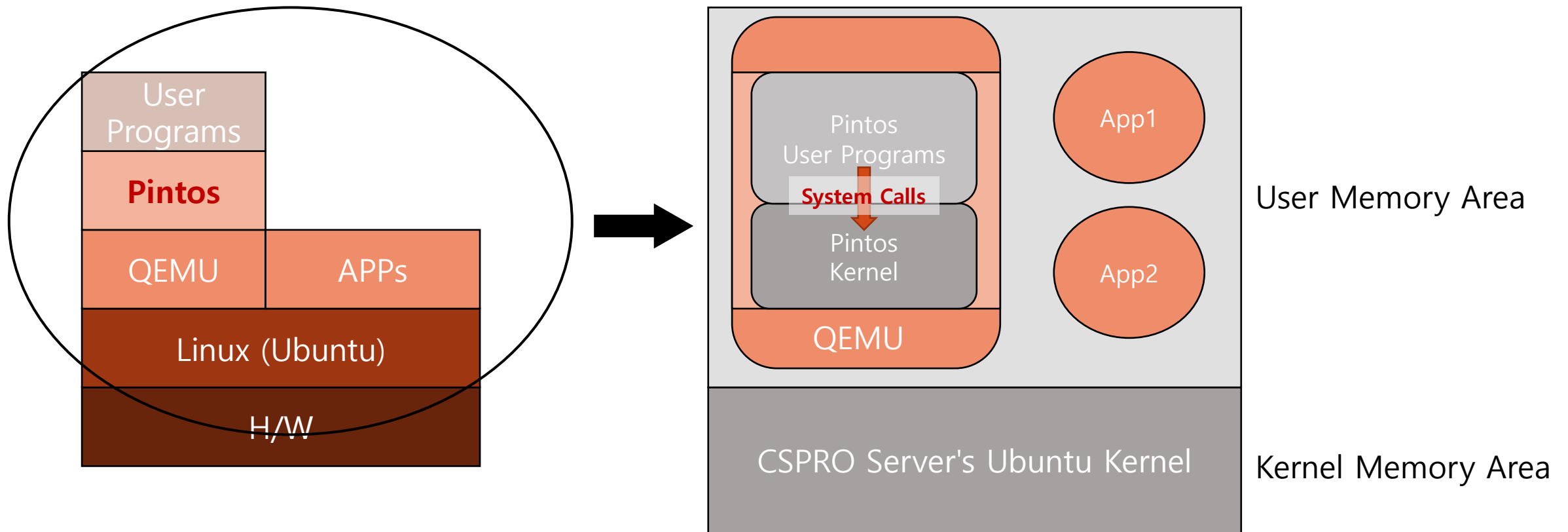
# System Calls

- 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.

```
1 /* cat.c
2
3  Prints files specified on command line to the console. */
4
5 #include <stdio.h>
6 #include <syscall.h>
7
8 int
9 main (int argc, char *argv[])
10 {
11     bool success = true;
12     int i;
13
14     for (i = 1; i < argc; i++)
15     {
16         int fd = open (argv[i]);
17         if (fd < 0)
18         {
19             printf ("%s: open failed\n", argv[i]);
20             success = false;
21             continue;
22         }
```

open() system call

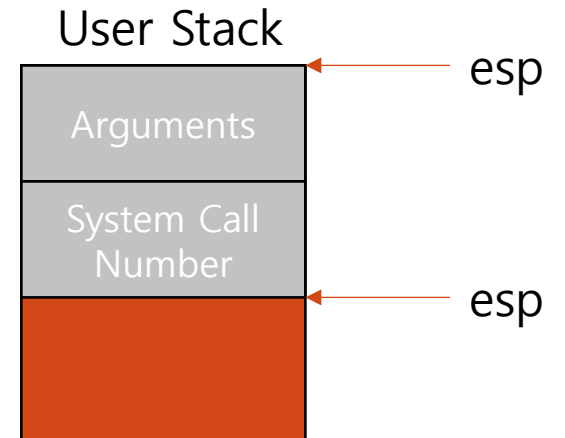
# System Calls

- 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.

```
102 int
103 open (const char *file)
104 {
105     return syscall1 (SYS_OPEN, file);
106 }
```

```
17 /* Invokes syscall NUMBER, passing argument ARG0, and returns the
18    return value as an 'int'. */
19 #define syscall1(NUMBER, ARG0)
20 ({
21     int retval;
22     asm volatile
23         ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp"
24          : "=a" (retval)
25          : [number] "i" (NUMBER),
26            [arg0] "g" (ARG0)
27          : "memory");
28     retval;
29 })
30
```

After returning from system call handler, restore stack pointer



# System Calls

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

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

Call interrupt handler

# System Calls

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

```
344 void
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);
```

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'

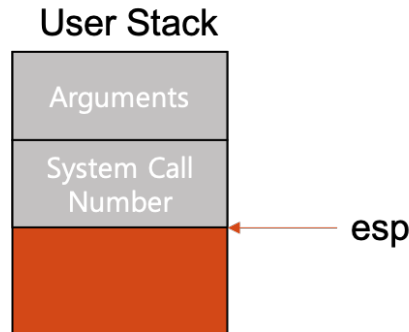
\* source code: threads/interrupt.c

# 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`.

```
15 static void
16 syscall_handler (struct intr_frame *f UNUSED)
17 {
18     printf ("system call!\n");
19     thread_exit ();
20 }
```

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



```
20 struct intr_frame
21 {
22     /* Pushed by intr_entry in intr-stubs.S.
23        These are the interrupted task's saved registers. */
28     uint32_t ebx;          /* Saved EBX. */
29     uint32_t edx;          /* Saved EDX. */
30     uint32_t ecx;          /* Saved ECX. */
31     uint32_t eax;          /* Saved EAX. */
54     void *esp;             /* Saved stack pointer. */
55     uint16_t ss, :16;       /* Data segment for esp. */
56 };
```

\* 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**

```
1 # -*- perl -*-
2 use strict;
3 use warnings;
4 use tests::tests;
5 check_expected ([<'EOF']);
6 (exec-once) begin
7   (child-simple) run
8   child-simple: exit(81)
9   (exec-once) end
10  exec-once: exit(0)
11 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/user/syscall.c`
  - > `syscall_handler()` in `userprog/syscall.c`
  - > `thread_exit()` in `threads/thread.c`
  - > `process_exit()` in `userprog/process.c`

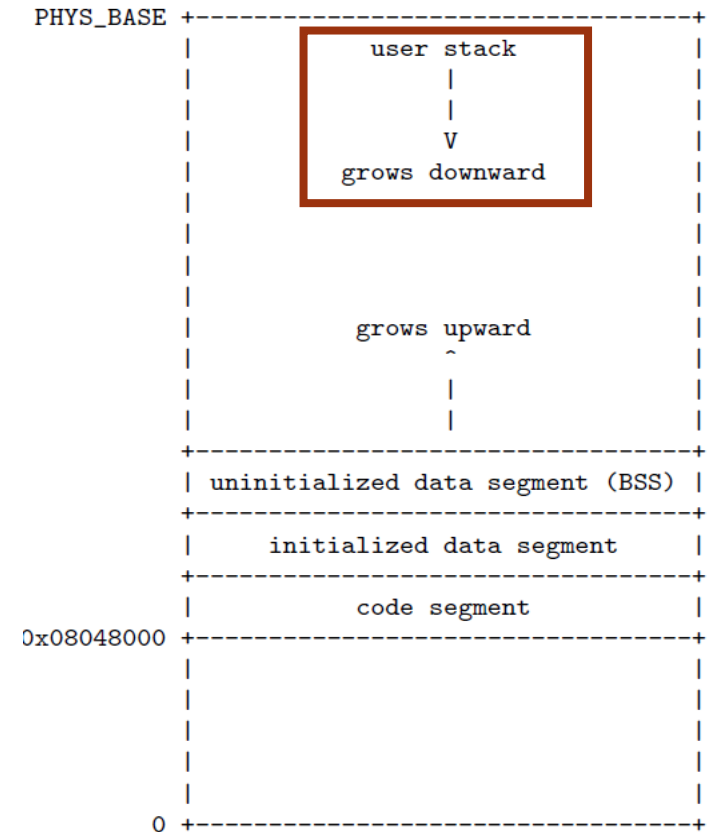
**Refer to slide pg. 25-29**

# Argument Passing

1. User program can have multiple arguments.

```
~ /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.



<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"`

<div>0xC0000000 (PHYS_BASE)</div> <div>- 0x00000004</div> <div>-----</div> <div>0xBFFFFFFC</div>	Address	Name	Data	Type
	0xbfffffffcc	argv[3][...]	'bar\0'	char[4]
	0xbfffffffb8	argv[2][...]	'foo\0'	char[4]
	0xbfffffffb5	argv[1][...]	'-l\0'	char[3]
	0xbfffffffed	argv[0][...]	'/bin/ls\0'	char[8]
	0xbfffffffec	word-align	0	uint8_t
	0xbffffffe8	argv[4]	0	char *
	0xbffffffe4	argv[3]	0xbfffffffcc	char *
	0xbffffffe0	argv[2]	0xbfffffffb8	char *
	0xbffffffdc	argv[1]	0xbfffffffb5	char *
	0xbffffffd8	argv[0]	0xbfffffffed	char *
	0xbffffffd4	argv	0xbffffffd8	char **
	0xbffffffd0	argc	4	int
	0xbffffffcc	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
0xbfffffffcc	argv[3][...]	'bar\0'	char[4]
0xbfffffffb8	argv[2][...]	'foo\0'	char[4]
0xbfffffffb5	argv[1][...]	'-l\0'	char[3]
0xbfffffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffffec	word-align	0	uint8_t
0xbffffffe8	argv[4]	0	char *
0xbffffffe4	argv[3]	0xbfffffffc	char *
0xbffffffe0	argv[2]	0xbffffffb8	char *
0xbffffffdc	argv[1]	0xbffffffb5	char *
0xbffffffd8	argv[0]	0xbffffffed	char *
0xbffffffd4	argv	0xbffffffd8	char **
0xbffffffd0	argc	4	int
0xbffffffcc	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
0xbfffffffcc	argv[3][...]	'bar\0'	char[4]
0xbfffffff8	argv[2][...]	'foo\0'	char[4]
0xbfffffff5	argv[1][...]	'-l\0'	char[3]
0xbffffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbffffffec	word-align	0	uint8_t
0xbffffffe8	argv[4]	0	char *
0xbffffffe4	argv[3]	0xbfffffffcc	char *
0xbffffffe0	argv[2]	0xbfffffff8	char *
0xbffffffdc	argv[1]	0xbfffffff5	char *
0xbffffffd8	argv[0]	0xbffffffed	char *
0xbffffffd4	argv	0xbffffffd8	char **
0xbffffffd0	argc	4	int
0xbffffffcc	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`).

```

bfffffff00 04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |.....|
bfffffff04 f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |...../bi|
bfffffff08 6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
bfffffff0c                                     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]
0xbffffffd	argv[0][...]	'/bin/ls\0'	char[8]
0xbffffffe	word-align	0	uint8_t
0xbffffffe8	argv[4]	0	char *
0xbffffffe4	argv[3]	0xbffffffc	char *
0xbffffffe0	argv[2]	0xbffffff8	char *
0xbffffffdc	argv[1]	0xbffffff5	char *
0xbffffffd8	argv[0]	0xbffffffd	char *
0xbffffffd4	argv	0xbffffffd8	char **
0xbffffffd0	argc	4	int
0xbffffffcc	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

```
304  /* Set up stack. */
305  if (!setup_stack (esp))
306      goto done;
307
308  /* Start address. */
309  *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 specific to this thread */  
run_actions (argv);  
  
/* Finish up. */  
shutdown ();  
thread_exit ();
```

# 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;
}
```

# 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



# 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 = pallocc_get_page (0);
    if (fn_copy == NULL)
        return TID_ERROR;
    strcpy (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)
        pallocc_free_page (fn_copy);
    return tid;
}
```

# 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));
}
```

# 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()**.

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

# System Calls: Code Level Flow

---

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

```
/* 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);
}
```

# 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

---

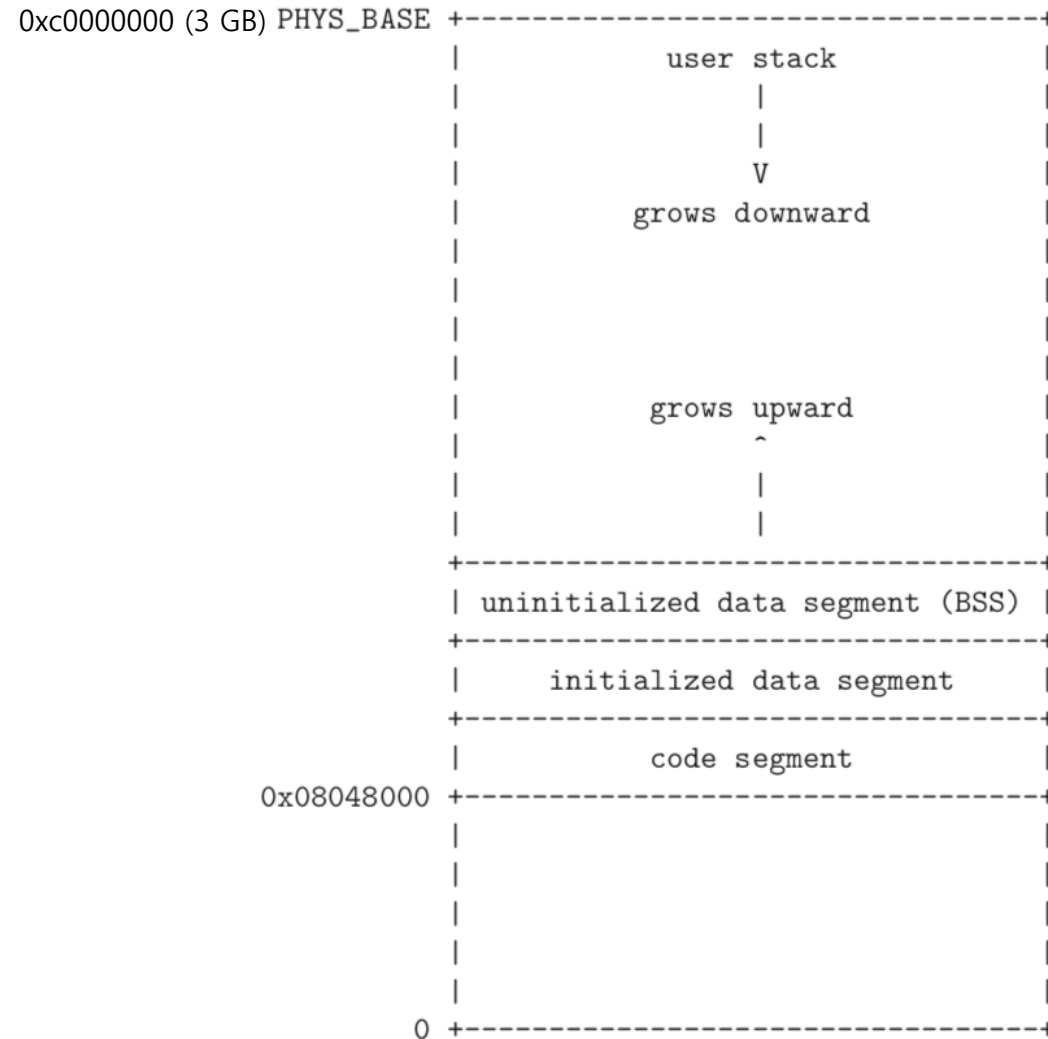
1. lib/user/syscall.h
  - Write prototype of 2 new system call APIs
2. lib/user/syscall.c
  - Define new **syscall4()** function for **sum\_of\_four\_int()** system call API
  - Define **fibonacci()** and **sum\_of\_four\_int()** system calls APIs
3. lib/syscall-nr.h
  - Add system call numbers for 2 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



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:** Implement...

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

## 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/userprog`

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

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`**)

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

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

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

- 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)**

# Submission

---

- **Contents**

- ① Pintos source codes
- ② document: **document.doc** or **document.docx** (Other format is not allowed such as .hwp)

- **Form and way to submit**

- 1) How to make tar.gz file

- Run 'make clean' in pintos/src
    - Make new directory with your group number
    - Copy 'pintos' directory (not pintos/src) into the new directory
    - Copy the document file into the new directory
    - Compress the new directory into os\_prj1\_##.tar.gz **(tar -zcf os\_prj1\_[Group#].tar.gz ./[Group#])**
      - ✓ **For example, if your group number is 7, tar -zcf os\_prj1\_07.tar.gz ./07**
      - ✓ You should use -zcf options for using tar

- 2) Way to submit: Upload the tar.gz file to e-class

- **Submit Hardcopy to AS916 (You should submit softcopy and hardcopy both, if not 3% of point will be deducted)**

- 3) Due date: 2019. 11. 3. 23:59

- ❖ **5% of point will be deducted for a wrong form and way to submit**

- ❖ **Late submission is allowed up to 3 days (~11/6) and 10% of point will be deducted per day**

# 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.)
<b>Project 1</b>	<b>6</b>	<b>User Programs (1)</b>	<b>10/5 – 11/3</b>	<b>10/5 (Sat.)</b>
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**

---

# Appendix



# Way to Submit Your Work

---

- Correct case
  - All the files should be in your 'Group number' directory
  - You should make new directory with your group number and copy pintos directory and document file into the group number directory

```
gr120180383@cspro9:~/99$ ls
document.doc  pintos
gr120180383@cspro9:~/99$
```

# Way to Submit Your Work

---

- Wrong case
  - 'make clean' is not performed in this case (There is build directory)

```
gr120180383@csp9:~/99/pintos/src/userprog$ ls
build      exception.h  gdt.h      Make.vars  pag
exception.c  gdt.c      Makefile   pagedir.c  pro
gr120180383@csp9:~/99/pintos/src/userprog$
```