Project2 : User Program

Prof. Seokin Hong

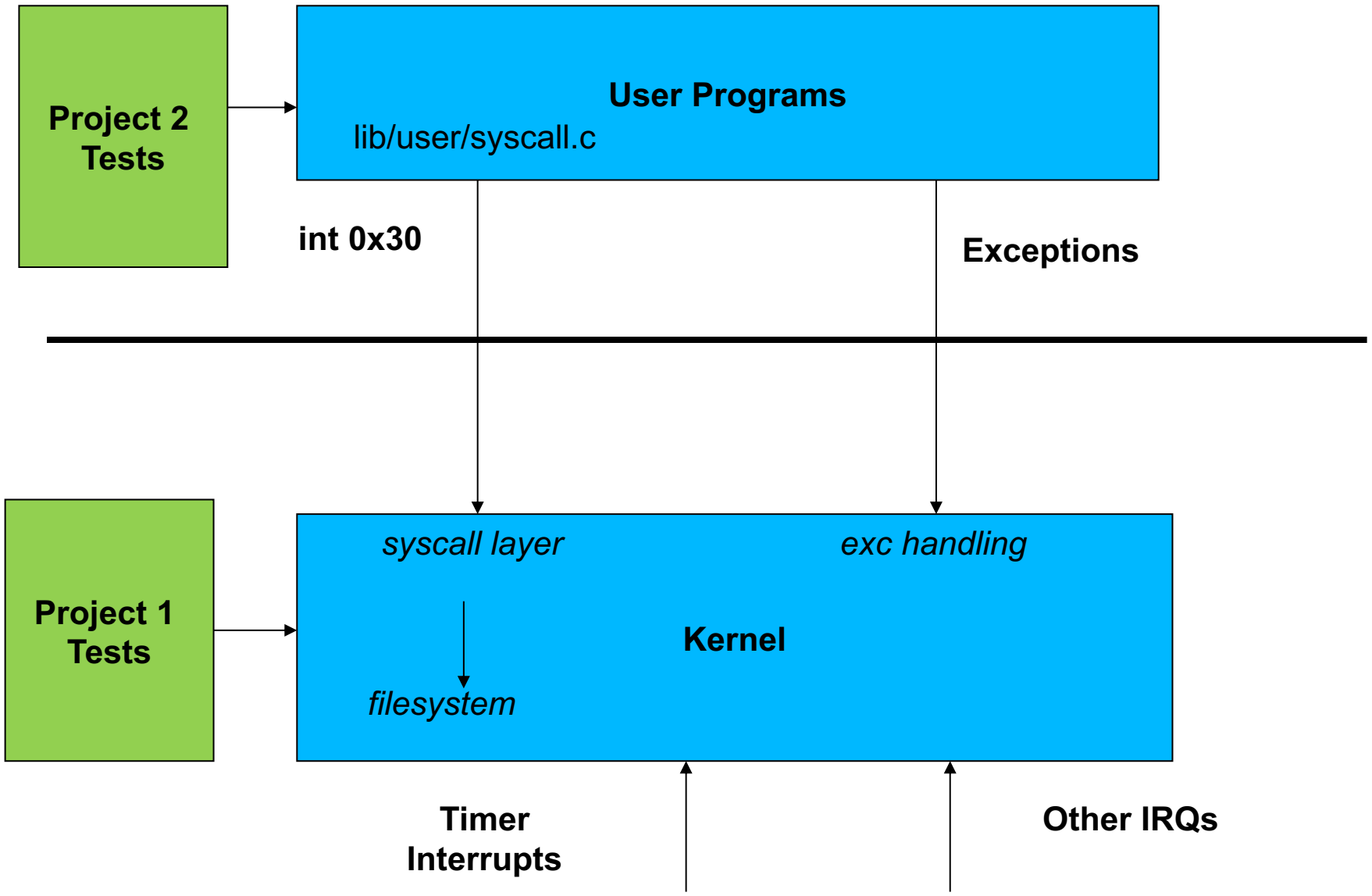
Kyungpook National University

Spring 2019

Till now …

- All code part of Pintos Kernel
- Code compiled directly with the kernel
  - This required that the tests call some functions whose interface should remain unmodified
- From now on, run user programs on top of kernel
  - Freedom to modify the kernel to make the user programs work

Project 1 vs Project 2



What happens if you run a program in the Unix sell?

- cp -r /usr/bin/temp .

1) Shell parses user input
2) Shell calls fork() and execve(“cp”, argv, env)
3) **cp** uses file system interface to copy files
4) **cp** may print messages to stdout
5) **cp** exits

Problems in the current implementation of Pintos

- Pintos already implements a basic program loader
  - Can parse ELF executables and start them as a process with one thread
  - Loaded programs can be executed

- But this system has problems:
  - User processes crash immediately
  - System calls have not been implemented

Project Goal

- Extend Pintos to enable to run user programs
- Allow more than one process to run at a time on Pintos

- **Part1:** Argument Passing
- **Part2:** System Call

Project Goal

- **Implement argument passing**
  - Example : “$ ls” may work on Pintos. But., “$ ls -l –a” doesn’t work
  - You must pass argv and argc to user programs

- **Implement the Pintos system call APIs**
  - Process management: exec(), wait(), exit()
  - OS shutdown: halt()
  - File I/O: open(), read(), write(), close()

Project Requirement

- **Passing command-line arguments to program**
- Safe memory access
- **A set of system calls**
  - See the project description document (word file) or section 3.3.4 of the Pintos official document (https://web.stanford.edu/~ouster/cgi-bin/cs140-winter13/pintos/pintos.pdf).
- Process termination messages
- Denying writes to files in user as executables

Part1:
Argument Passing and Stack setup

Background

- Pintos main() function

threads/init.c

• main() => run_actions(**argv**) after booting

• run_actions => run_task(**argv**)

• the task to run is argv[1]

• run_task => process_wait(**process_execute(task)**)

userprog/process.c

• process_execute() creates a thread that runs start_process(filename…) => **load**(filename…)

• load() sets up the stack, data, and code, as well as the start address

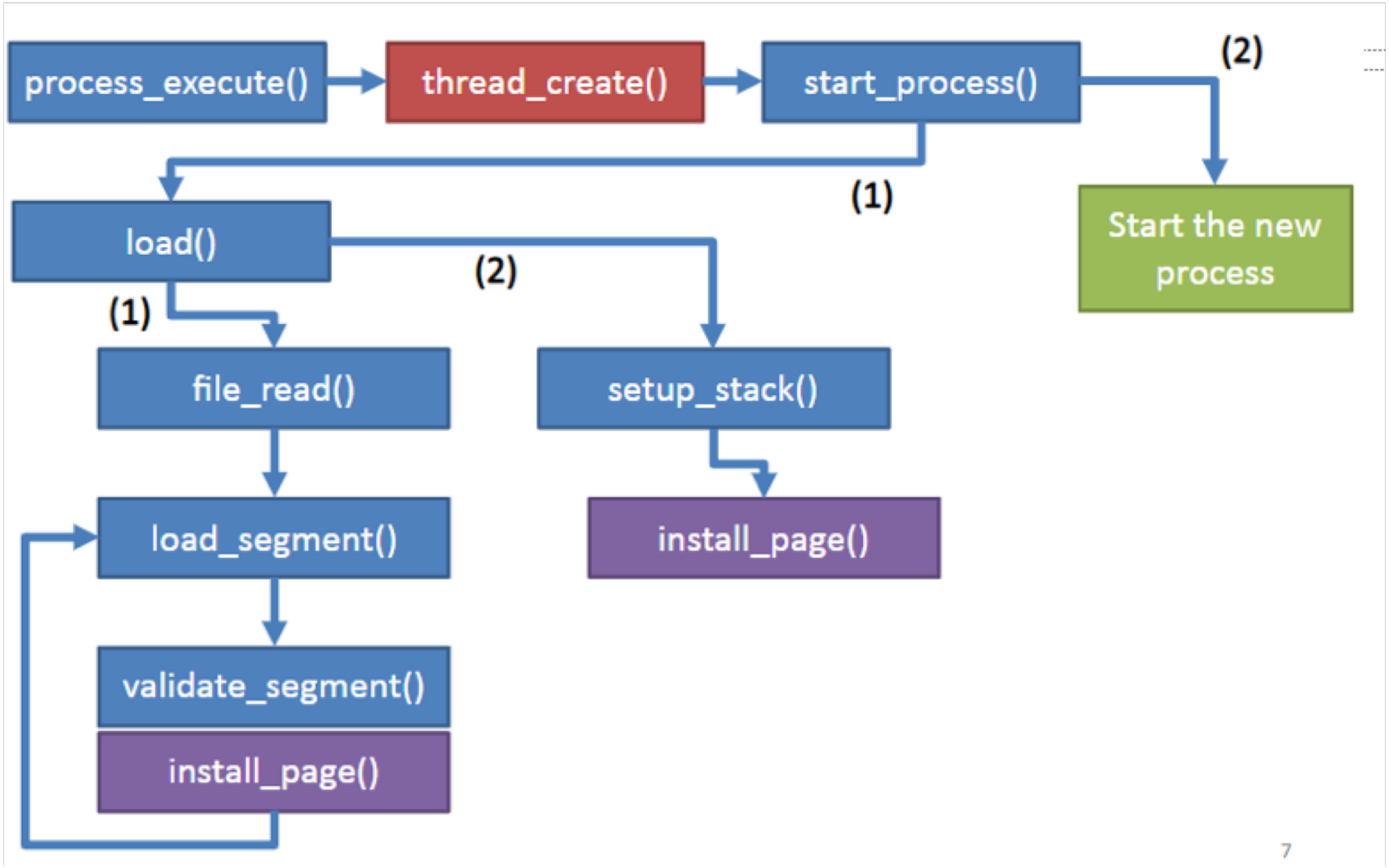
Background (cont'd)

- Program Start
  - _start() is the entry point for user programs
    - Linked with user programs
    - In lib/user/entry.c
    - _start() is a wrapper around main()
  - **main() requires two arguments: argc, argv**

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

Background (cont'd)

- Program Loading Flowchart



Background (cont'd)

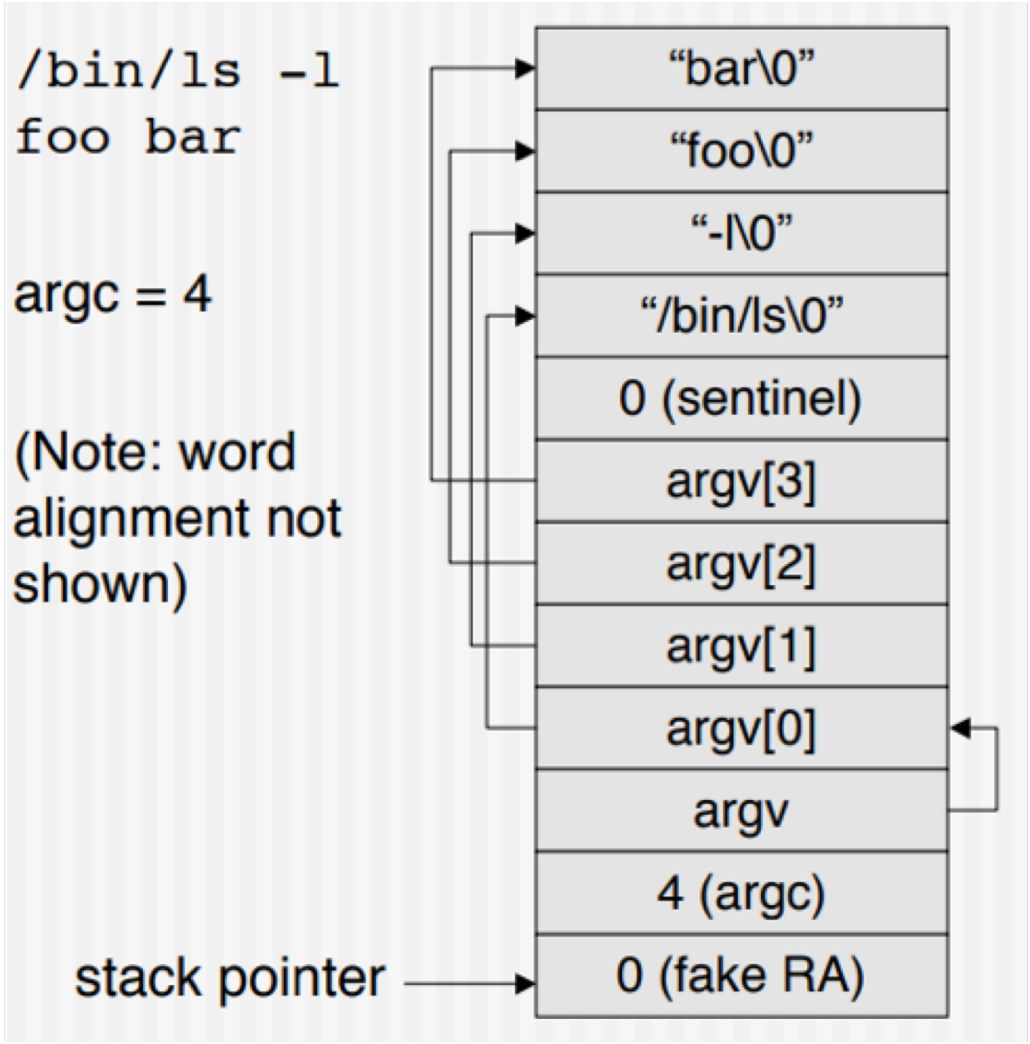
- **Argument Passing**
  - Before a user program starts executing, the kernel must push the function's arguments onto the stack
  - This involves breaking the command-line input into individual words
  - **how to handle arguments for the following example command?**

    ex) /bin/ls -l foo bar
    - First, break the command into words: /bin/ls, -l, foo, bar.
    - Second, place the words at the top of the stack.
    - Third, push the address of each string plus null pointer

  - So, you need to implement the string parsing (might in process_execute())
    - One solution is to use strtok_r() in lib/string.c

Background (cont'd)

- **Argument Passing**



- Push the words onto the stack
- Push a null pointer
- Push the address of each word in right-to-left order
- Push argv and argc
- Push 0 as a fake return address
- hex_dump() function in <stdio.h> may be useful to seeing the layout of your stack!

What you need to implement..

1. Emulate process_wait()

- The current implementation of process_wait() just return -1.

```
int process_wait(tid_t child_tid)
{
    return -1;
}
```

- Thus, you need to emulate a simple process wait function. The easiest way is to simply continually yielding

```
int process_wait(tid_t child_tid)
{
    while(true)
    {
        thread_yield();
    }
}
```

What you need to implement..

2. Parsing argument and setting up stack

- setup_stack() only takes in a void** esp (the stack pointer).

```
static bool
setup_stack (void **esp)
{
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }
  return success;
}
```

What you need to implement..

2. Parsing argument and setting up stack (Cont'd)

- So, you need to extend setup_stack() to add more arguments and pass in any other information you want
- You'll need to add on to the function before you return success. Don't modify what's already there.
- You will need to move the stack pointer and write data to the pointer.
- Remember that since it's a stack, you need to write everything in reverse order by decrementing the pointer

What you need to implement..

2. Parsing argument and setting up stack (Cont'd)

- General step

1) Parse the filename using strtok_r

2) Write each argument (including the executable name) in reverse order to the stack.

```
char argument[] = "arg1\0"
*esp -= strlen(argument);
memcpy(*esp, argument, strlen(argument));
```

3) Write the necessary number of 0s to word-align to 4 bytes.

```
Int word_align = 0, 1, 2, or 3;
*esp -= word_align;
memset(*esp, 0, word_align);
```

What you need to implement..

2. Parsing argument and setting up stack (Cont'd)

- General step **(Cont'd)**

  4) Write the last argument, consisting of four bytes of 0's

  5) Write the addresses pointing to each of the arguments. You need to figure out how to reference the addresses after writing all the arguments. These are char*s.

```
*esp -= sizeof(char*);
memcpy(*esp, address, sizeof(char*));
```

  6) write the address of argv[0]. This will be a char**

```
*esp -= sizeof(char**);
memcpy(*esp, address_argv0, sizeof(char**));
```

What you need to implement..

2. Parsing argument and setting up stack (Cont'd)

- General step **(Cont'd)**

7) Write the number of arguments (argc)

8) Write a NULL pointer as the return address.

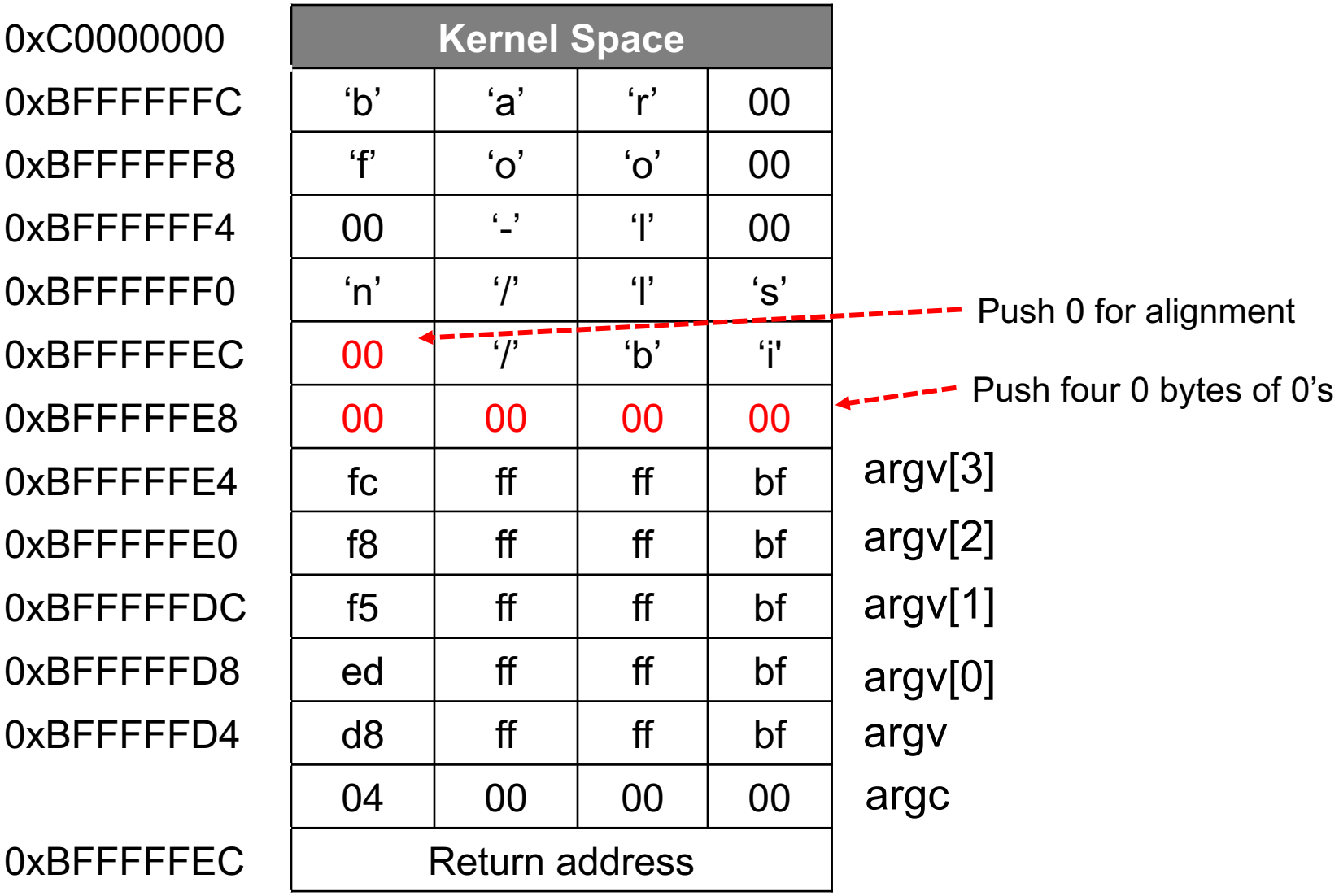
*Hex dump results for a test program (args-multiple)

pintos -v -k -T 60 --bochs --filesys-size=2 -p tests/userprog/args-multiple -a args-multiple -- -q -f run 'args-multiple some arguments for you!'

```
bffffff0                00 00 00 00-05 00 00 00 c0 ff ff bf |    ............|
bfffffc0  da ff ff bf e8 ff ff bf-ed ff ff bf f7 ff ff bf |................|
bfffffd0  fb ff ff bf 00 00 00 00-00 00 61 72 67 73 2d 6d |..........args-m|
bfffffe0  75 6c 74 69 70 6c 65 00-73 6f 6d 65 00 61 72 67 |ultiple.some.arg|
bffffff0  75 6d 65 6e 74 73 00 66-6f 72 00 79 6f 75 21 00 |uments.for.you!.|
```

What you need to implement..

Stack for “/bin/ls -l foo bar” command



Part 2:
System Call

Background

- **System call in Pintos**
  - User programs use system calls to use the kernel services
  - Library function has a system call
    - lib/user/syscall.c
  - All system calls use an interrupt
    - Push argument and system call number
    - Update the stack pointer
    - Invoke an interrupt with 0x30 (interrupt number)
  - When an interrupt occur, the Interrupt handler (syscall_handler) provides appropriate services
    - The handler distinguishes the system call (e.g., open, close, read, …) and the number of arguments for the service

Background (cont'd)

- **System call in Pintos**
  - system call interfaces (library function), lib/user/syscall.c

```
pid_t
exec (const char *file)
{
  return (pid_t) syscall1 (SYS_EXEC, file);
}
```

```
/* 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;                                                       \
        })
```

  - syscall_handler(), in userprog/syscall.c

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

What you need to implement..

In userprog/syscall.c

1. Extend syscall_handler()

- To get system call number and arguments
- Invoke each system call routine

2. Implement each system call routine

- To this end, you need to use some functions provided by Pintos
- For example,
  - Use process_execute() for implementing exec(()
  - Use process_wait() for implementing wait()
  - Use filesys_open() for implementing open()
  - Use file_close() for implementing close()
  - Use file_write() for implementing write()
  - ….

Recommended order of implementation for this project

- **change process_wait() to an infinite loop**
- **Argument passing and setup stack**
- **System call infrastructure**
  - Implement code to read the system call number from the user stack and invoke a system call handler based on it.
- **exit() system call**
- **write() system call**
- **other system calls**
  - **halt(), exit(), exec(), wait(), create(), remove(), open(), filesize(), read(), write(), seek(), tell(), close()**
  - You can find more information about the system call in the section 3.3.4 of the Stanford Pintos document(https://web.stanford.edu/~ouster/cgi-bin/cs140-winter13/pintos/pintos.pdf).
- **Denying writes to executables**

Ctags ← Tool for hacking open source project

- Install
  - $ apt-get install ctags
- Generate “tags” file
  - $ ctags -R *
- Run VIM with “tags” file
  - $ vim -t tags

Keyboard command	Action
`Ctrl-]`	Jump to the tag underneath the cursor
`:ts <tag> <RET>`	Search for a particular tag
`:tn`	Go to the next definition for the last tag
`:tp`	Go to the previous definition for the last tag
`:ts`	List all of the definitions of the last tag
`Ctrl-t`	Jump back up in the tag stack