Project 2: User Programs

Presented by

Xiaomo Liu
1 Oct 2009
(update Min Li's slides)

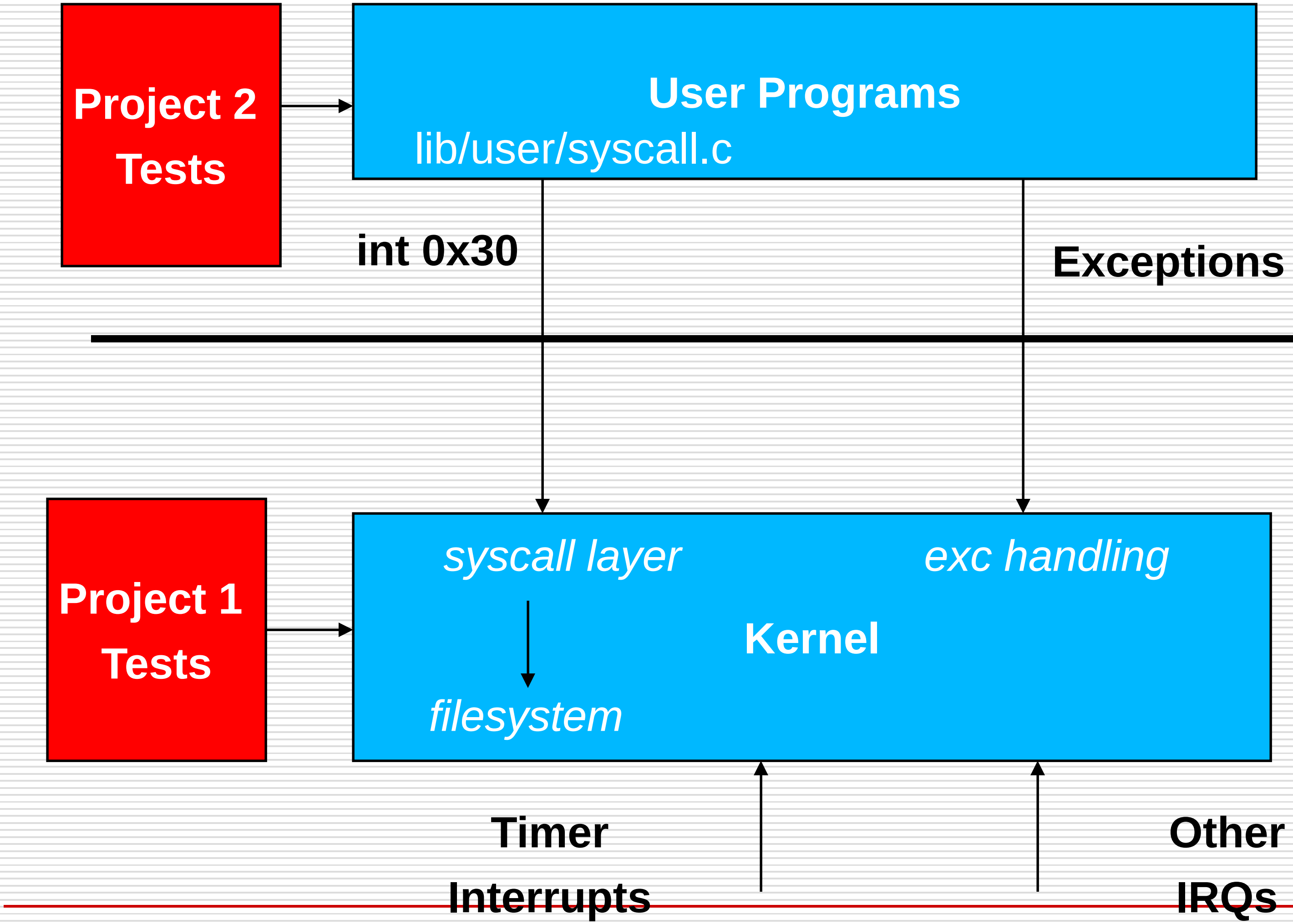
Till now …

- 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

Why Project 2 is not Project 1?



Sample User Program in C

- In C, a user program test.c can pass argument

  -
    ```
    int main(int argc, char* argv[])
    {
        for(int i=0; i<argc; i++)
        {
            char* arg = argv[i];
        }
    }
    ```
  - ./test arg1 arg2 ...

Sample User Program in C

- test.c can call system libraries

  -
    ```
    #include <stdio.h>
    int main()
    {
        FILE* p_file = fopen("myfile.txt","w");
        if (p_file != NULL) fputs(“fopen”, p_file);
        fclose(p_file);
    }
    ```
  - Get fopen, fputs, fclose by system calls
- Pintos need you to implement
  - Argument passing
  - System calls

Using the File system

- May need to interact with file system
- Do not modify the file system!

- Certain limitations (till Project 4)
  - No internal synchronization
  - Fixed file size
  - No subdirectories
  - File names limited to 14 chars
  - System crash might corrupt the file system

- Files to take a look at: ‘*filesys.h*’ & ‘*file.h*’

Some commands

- Creating a simulated disk
  - pintos-mkdisk filesys.dsk --filesys-size=2
- Formatting the disk
  - pintos -f –q
  - This will only work after your kernel is built !
- Copying the program into the disk
  - pintos -p ../../examples/echo -a echo -- -q
- Running the program
  - pintos -q run ’echo x’
  - Single command:
    - pintos --fs-disk=2 -p ../../examples/echo -a echo -- -f -q run ’echo x’
- $ make check – Builds the disk automatically
  - Copy&paste the commands make check does!

Various directories

- Few user programs:
  - src/examples
- Relevant files:
  - userprog/
- Other files:
  - threads/, filesys/

Requirements

- □ Process Termination Messages
- □ **Argument Passing**
- □ **System calls**
- □ Deny writes to executables

Process Termination

- Process Terminates
  - printf ("%s: exit(%d)\n",...);
  - for eg: args-single: exit(0)

Return Code

Program
name

- Do not print any other message!

Argument Passing

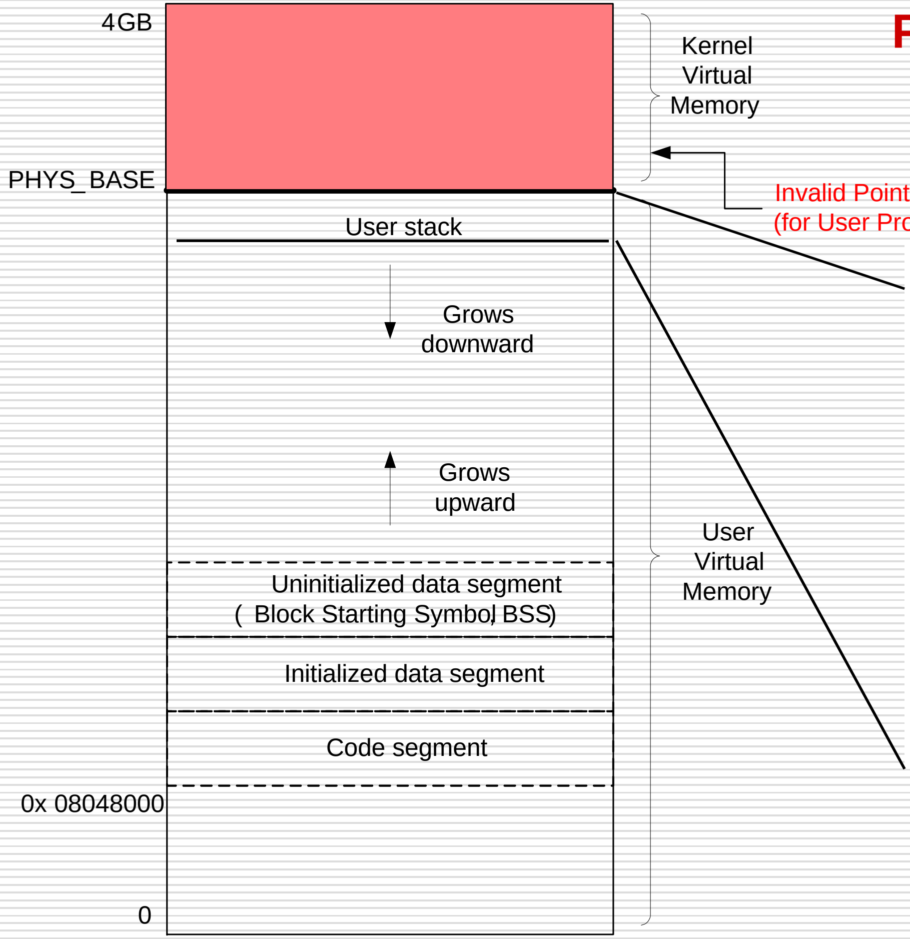
- Pintos currently lacks argument passing. You Implement it!
- Change *esp = PHYS_BASE to *esp = PHYS_BASE – 12 in setup_stack() to get started
- Change process_execute() in process.c to process multiple arguments
- Could limit the arguments to fit in a page(4 kb)
- String Parsing: strtok_r() in lib/string.h

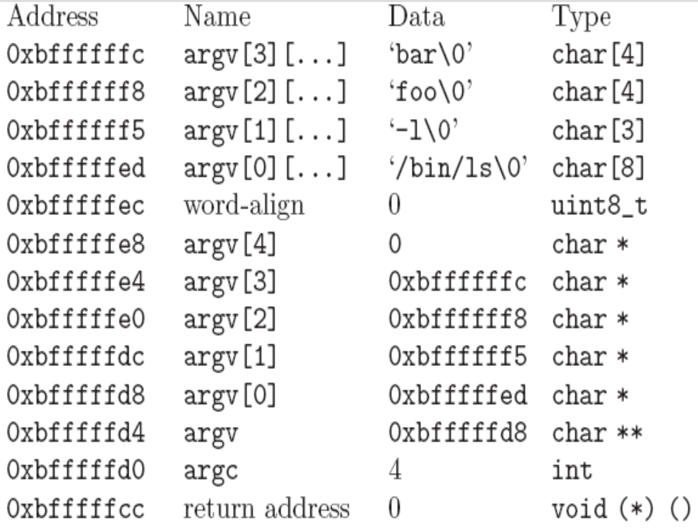
```
pgm.c
main(int argc,
          char *argv[]) {
          ...
    }
$ pintos run ‘pgm alpha beta’
argc = 3
argv[0] = “pgm”
argv[1] = “alpha”
argv[2] = “beta”
```

Example taken from Abdelmounaam Rezgui’s presentation

Memory layout

PHYS_BASE = 3GB



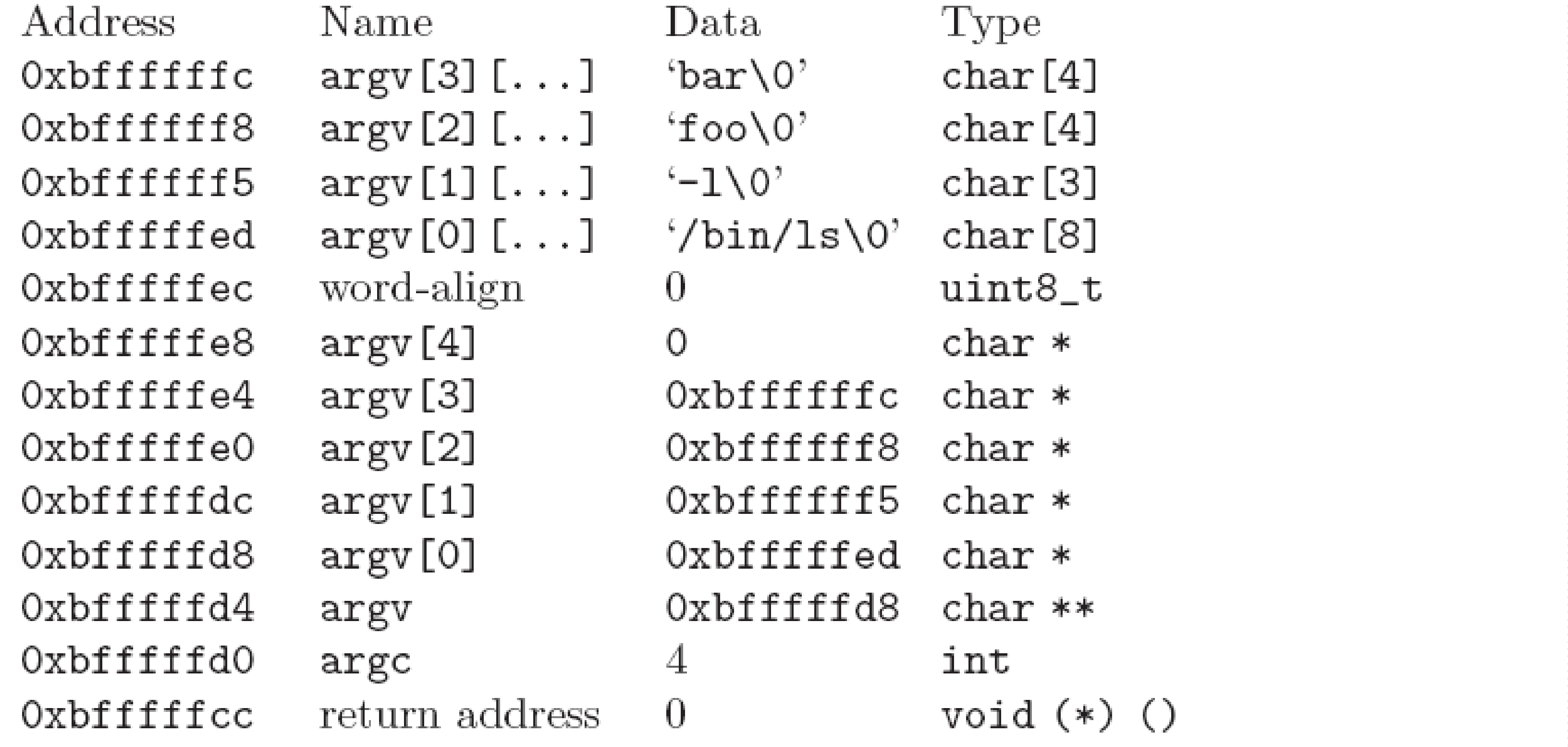


Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	argv[1][...]	'-l\0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	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 (*) ()

Figure taken from Abdelmounaam Rezgui's presentation

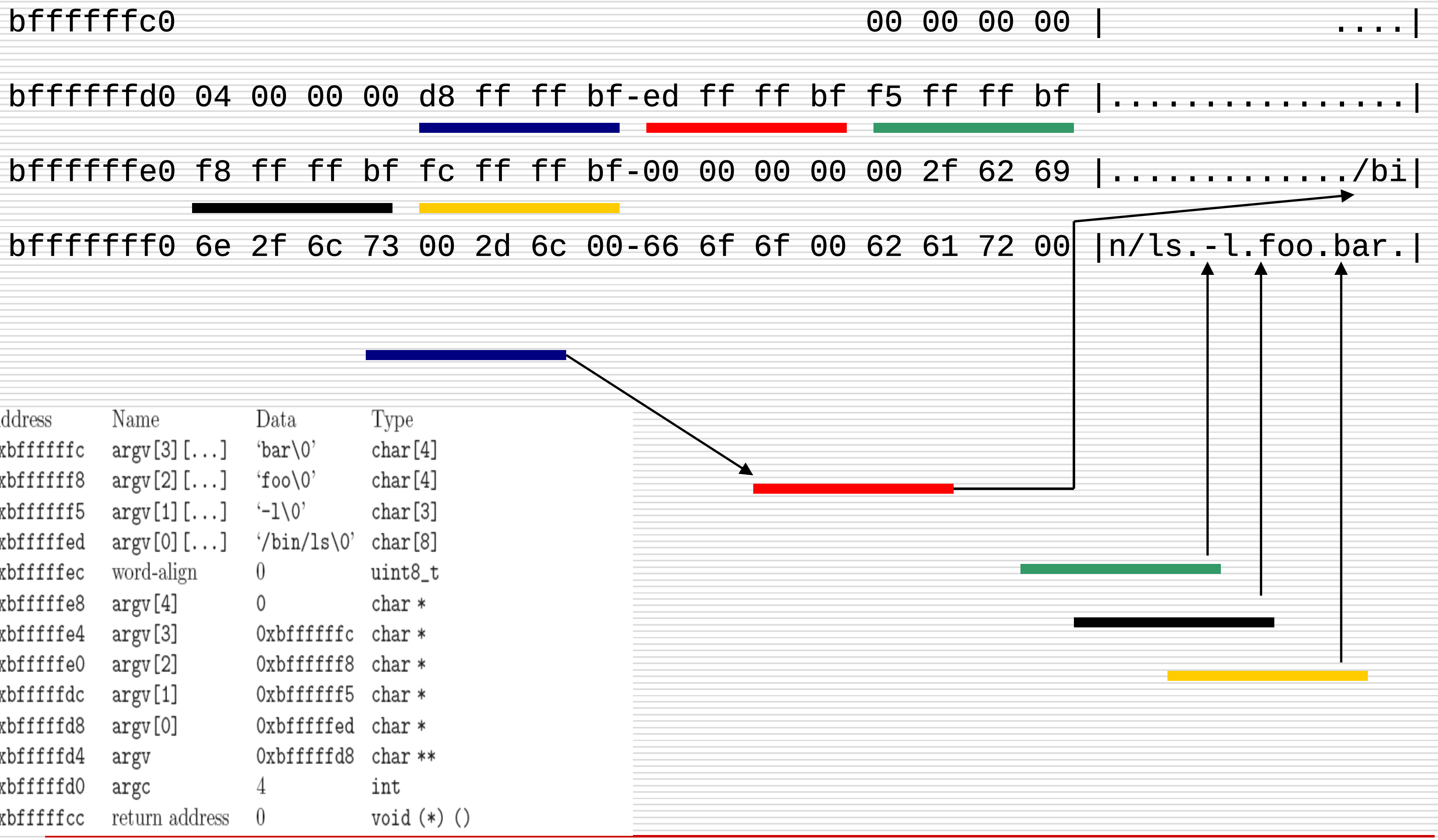
Setting up the Stack

How to setup the stack for the program: /bin/ls –l foo bar



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 (*) ()

Setting up the Stack... Contd



```
bffffffc0                                           00 00 00 00 |            ....|
bffffffd0 04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffffe0 f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |............./bi|
bfffffff0 6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

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 (*) ()

Synchronization

- Synchronization between parent and children processes
  - Ensuring child process Loading new executables successfully

Requirements

- Process Termination Messages
- **Argument Passing**
- **System calls**
- Deny writes to executables

System Calls

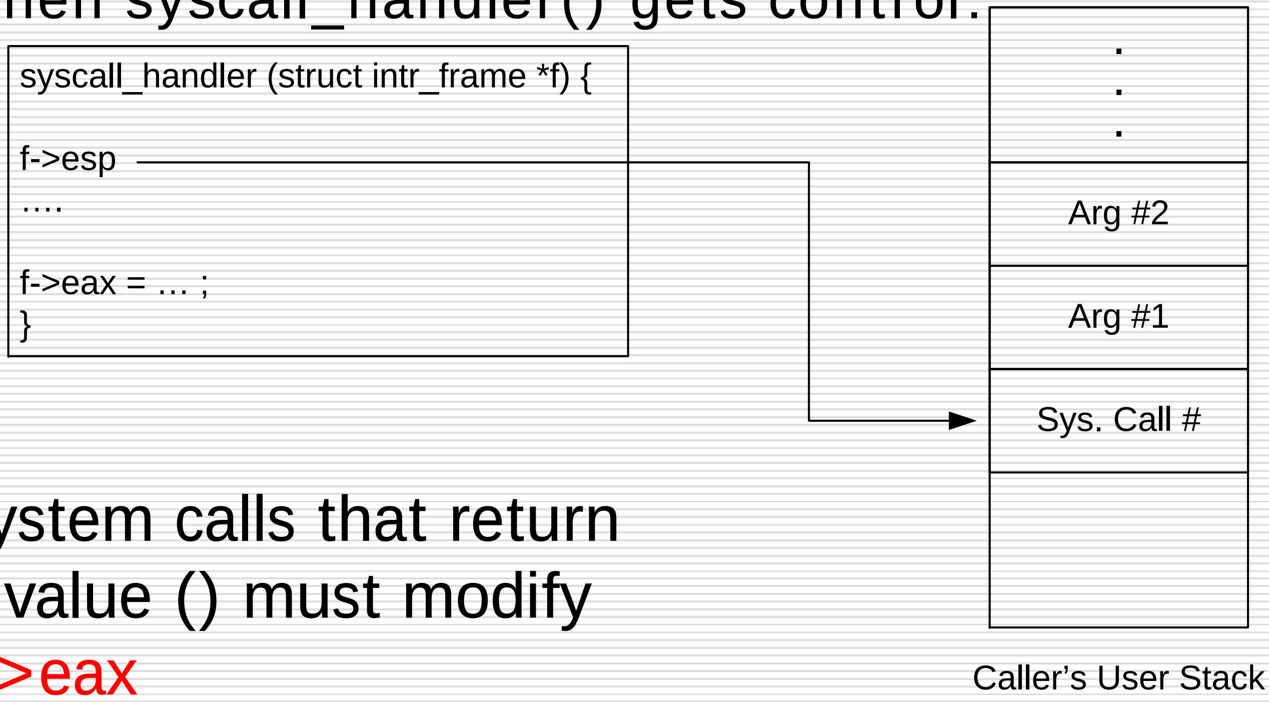
- □ Pintos lacks support for system calls currently!
- □ Implement the system call handler in userprog/syscall.c
- □ System call numbers defined in lib/syscall-nr.h
- □ Process Control: exit, exec, wait
- □ File system: create, remove, open, filesize, read, write, seek, tell, close
- □ Others: halt

Syscall handler currently …

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

Continued...

- A system call has:
  - System call number
  - (possibly) arguments
- When syscall_handler() gets control:



```
syscall_handler (struct intr_frame *f) {

f->esp
….

f->eax = … ;
}
```

Caller's User Stack

- System calls that return a value () must modify f->eax

Figure taken from Abdelmounaam Rezgui's presentation

System calls – File system

- Decide on how to implement the file descriptors
  - O(n) data structures… perfectly fine!
- Access granularity is the entire file system
  - Have 1 global lock!
- write() – fd 1 writes to console
  - use putbuf() to write entire buffer to console
- read() – fd 0 reads from console
  - use input_getc() to get input from keyboard
- Implement the rest of the system calls

System calls – Process Control

- wait(pid) – Waits for process pid to die and returns the status pid returned from exit
- Returns -1 if
  - pid was terminated by the kernel
  - pid does not refer to child of the calling thread
  - wait() has already been called for the given pid
- exec(cmd) – runs the executable whose name is given in command line
  - returns -1 if the program cannot be loaded
- exit(status) – terminates the current program, returns status
  - status of 0 indicates success, non zero otherwise

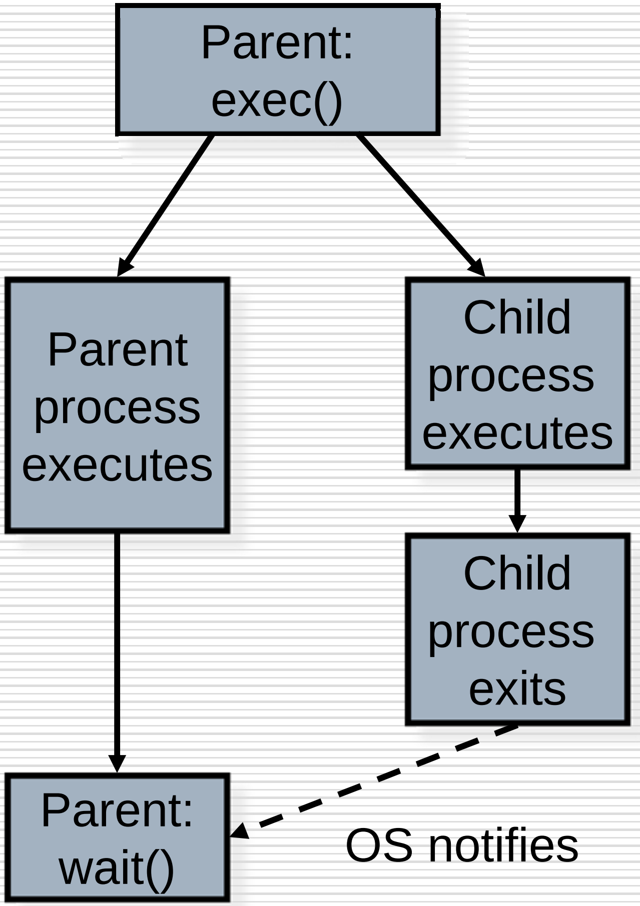


Figure taken and modified from Dr. Back's lecture – CS3204 - Fall 2006

Process Control: wait

- Implement process_wait() in process.c
- Then, implement wait() using process_wait()
- Cond variables and/or semaphores will help
  - Think about what semaphores may be used for and how they must be initialized

- Some Conditions to take care!
  - Parent may or may not wait for its child
  - Parent may call wait() after child terminates!

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

```
main() {
int i; pid_t p;
p = exec("pgm a b");
// i = wait (p);
}
```

Memory Access

- System calls can have memory access
  - e.g open(), read(), write()
  - have a look at tests cases *-bad-ptr.c
- In open-bad-ptr.c
  - void test_main (void)

```
{
        msg ("open(0x20101234): %d”,
                open ((char *) 0x20101234));
        fail ("should have called exit(-1)");
}
```

Memory Access (contd')

- Invalid pointers must be rejected. Why?
  - Kernel has access to all of physical memory including that of other processes
  - Kernel like user process would fault when it tries to access unmapped addresses
- User process cannot access kernel virtual memory
- User Process after it has entered the kernel can access kernel virtual memory and user virtual memory
- How to handle invalid memory access?

Memory Access (contd')

- Two methods to handle invalid memory access
  - Verify the validity of user provided pointer and then dereference it
    - Look at functions in userprog/pagedir.c, threads/vaddr.h
    - Strongly recommended!
  - Check if user pointer is below PHYS_BASE and dereference it
    - Could cause page fault
    - Handle the page fault by modifying the page_fault() code in userprog/exception.c
  - Make sure that resources are not leaked

Some Issues to look at…

- Check the validity of the system call parameters
- Every single location should be checked for validity before accessing it. For e.g. not only f->esp, but also f->esp +1, f->esp+2 and `f->esp+3 should be checked
- Read system call parameters into kernel memory (except for long buffers)
  - copy_in function recommended!

Denying writes to Executables

- Use file_deny_write() to prevent writes to an open file
- Use file_allow_write() to re enable write
- Closing a file will automatically re enable writes

Suggested Order of Implementation

- Change *esp = PHYS_BASE to *esp = PHYS_BASE – 12 to get started
- Implement the system call infrastructure
- Change process_wait() to a infinite loop to prevent pintos getting powered off before the process gets executed
- Implement exit system call
- Implement write system call
- Start making other changes

Misc

- Deadline: 19 Oct, 11:59 pm
- Do not forget the design document
  - Must be done individually
- Good Luck!