CS162 Project0-Pregame

The document of the CS162 Project0 is posted as a link https://cs162.org/static/proj/proj-pregame/, please refer to that document.

Find the faulting instruction

Q1 What virtual address did the program try to access from userspace that caused it to crash?

0xc0000008

Q2 What is the virtual address of the instruction that resulted in the crash?

0x80488ee

Q3 To investigate, disassemble the `do-nothing` binary using `objdump` (you used this tool in Homework 0). What is the name of the function the program was in when it crashed? Copy the disassembled code for that function onto Gradescope, and identify the instruction at which the program crashed.

```
080488e8 <_start>:
 80488e8:     55                push   %ebp
 80488e9:     89 e5                  mov    %esp,%ebp
 80488eb:     83 ec 18               sub    $0x18,%esp
 80488ee:     8b 45 0c               mov    0xc(%ebp),%eax
 80488f1:     89 44 24 04            mov    %eax,0x4(%esp)
 80488f5:     8b 45 08               mov    0x8(%ebp),%eax
 80488f8:     89 04 24               mov    %eax,(%esp)
 80488fb:     e8 94 f7 ff ff         call   8048094 <main>
 8048900:     89 04 24               mov    %eax,(%esp)
 8048903:     e8 d3 21 00 00              call   804aadb <exit>

The instruction is mov    0xc(%ebp),%eax
```

Q4 Find the C code for the function you identified above (hint: it was executed in userspace, so it's either in `do-nothing.c` or one of the files in `proj0/src/lib` or `proj0/src/lib/user`), and copy it onto Gradescope. For each instruction in the disassembled function in #3, explain in a few words why it's necessary and/or what it's trying to do. Hint: see 80x86 Calling Convention

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

The first three instructions are the prologue of the callee, which pushes ebp onto the stack and sets ebp to be the new esp, then allocates stack space for local variables. The next four instructions are to pass the arguments on the stack. Then it pushes the address of the return address on the stack and jumps to the first instruction of the

function. After that, it passes the argument on the stack and calls function.

Q5 Why did the instruction you identified in Q3 try to access memory at the virtual address you identified in Q1? Don’t explain this in terms of the values of registers; we’re looking for a higher level explanation.

Because it wants to access the argument in the stack and pass it as an argument of

function to reuse it.

Step through the crash

Q6 Step into the `process_execute` function. What is the name and address of the thread running this function? What other threads are present in Pintos at this time? Copy their `struct thread`s. (Hint: for the last part `dumplist &all_list thread allelem` may be useful.)

The name of the thread is run_task, and its address is 0xc0020a19; Other threads:

main: {tid = 1, status = THREAD_RUNNING, name = "main", '\000' <repeats 11 times>, stack = 0xc000edbc "001", priority = 31, allelem = {prev = 0xc0039d98 <all_list>, next = 0xc0104020}, elem = {prev = 0xc0039d88 <fifo_ready_list>, next = 0x c0039d90 <fifo_ready_list+8>}, pcb = 0xc010500c, magic = 3446325067};

idle: {tid = 2, status = THREAD_BLOCKED, name = "idle", '\000' <repeats 11 times>, stack = 0xc0104f14 "" , priority = 0, allelem = {prev = 0xc000e020, next = 0xc0039da0 <all_list+8>}, elem = {prev = 0xc0039d88 <fifo_ready_list>, next = 0xc00 39d90 <fifo_ready_list+8>}, pcb = 0x0, magic = 3446325067}

Q7 What is the backtrace for the current thread? Copy the backtrace from GDB as your answer and also copy down the line of C code corresponding to each function call.

```
#0  process_execute (file_name=0xc0007d50 "do-nothing") at ../../userprog/process.c:57
```

```
#1  0xc0020a19 in run_task (argv=0xc0039c8c <argv+12>) at ../../threads/init.c:315 #2
0xc0020b57 in run_actions (argv=0xc0039c8c <argv+12>) at ../../threads/init.c:388 #3
0xc00203d9 in main () at ../../threads/init.c:136;
```

```
sema_init(&temporary, 0); process_wait(process_execute(task)); a->function(argv);
```

Q8 Set a breakpoint at `start_process` and continue to that point. What is the name and address of the thread running this function? What other threads are present in Pintos at this time? Copy their `struct thread`s.

```
Name: kernel_thread, address: 0xc0021470; Other threads: pintos-debug: dumplist #0:
0xc000e000 {tid = 1, status = THREAD_BLOCKED, name = "main", '\000' <repeats 11
times>, stack = 0xc000ee7c "" , priority = 31, allelem = {prev = 0xc0039d98 <all_list>, next =
0xc0104020}, elem = {prev = 0xc003b7b8 <temporary+4>, next = 0xc003b7c0
<temporary+12>}, pcb = 0xc010500c, magic = 3446325067} pintos-debug: dumplist #1:
0xc0104000 {tid = 2, status = THREAD_BLOCKED, name = "idle", '\000' <repeats 11 times>,
stack = 0xc0104f14 "" , priority = 0, allelem = {prev = 0xc000e020, next = 0xc010b020},
elem = {prev = 0xc0039d88 <fifo_ready_list>, next = 0xc0039d90 <fifo_r eady_list+8>}, pcb
= 0x0, magic = 3446325067} pintos-debug: dumplist #2: 0xc010b000 {tid = 3, status =
THREAD_RUNNING, name = "do-nothing\000\000\000\000\000", stack = 0xc010bfd4 "",
priority = 31, allelem = {prev = 0xc0104020, next = 0xc0039da0 <all_list+8>}, elem = {prev =
0xc0039d88 <fifo_ready_list>, next = 0xc00 39d90 <fifo_ready_list+8>}, pcb = 0x0, magic =
3446325067} (gdb) dumplist &all_list thread allelem pintos-debug: dumplist #0:
0xc000e000 {tid = 1, status = THREAD_BLOCKED, name = "main", '\000' <repeats 11
times>, stack = 0xc000ee7c "" , priority = 31, allelem = {prev = 0xc0039d98 <all_list>, next =
0xc0104020}, elem = {prev = 0xc003b7b8 <temporary+4>, next = 0xc003b7c0
<temporary+12>}, pcb = 0xc010500c, magic = 3446325067} pintos-debug: dumplist #1:
0xc0104000 {tid = 2, status = THREAD_BLOCKED, name = "idle", '\000' <repeats 11 times>,
stack = 0xc0104f14 "" , priority = 0, allelem = {prev = 0xc000e020, next = 0xc010b020},
elem = {prev = 0xc0039d88 <fifo_ready_list>, next = 0xc0039d90 <fifo_r eady_list+8>}, pcb
= 0x0, magic = 3446325067} pintos-debug: dumplist #2: 0xc010b000 {tid = 3, status =
```

THREAD_RUNNING, name = "do-nothing\000\000\000\000\000", stack = 0xc010bfd4 "", priority = 31, allelem = {prev = 0xc0104020, next = 0xc0039da0 <all_list+8>}, elem = {prev = 0xc0039d88 <fifo_ready_list>, next = 0xc00 39d90 <fifo_ready_list+8>}, pcb = 0x0, magic = 3446325067}

Q9 Where is the thread running `start_process` created? Copy down this line of code.

ef->eip = (void (*)(void))kernel_thread;

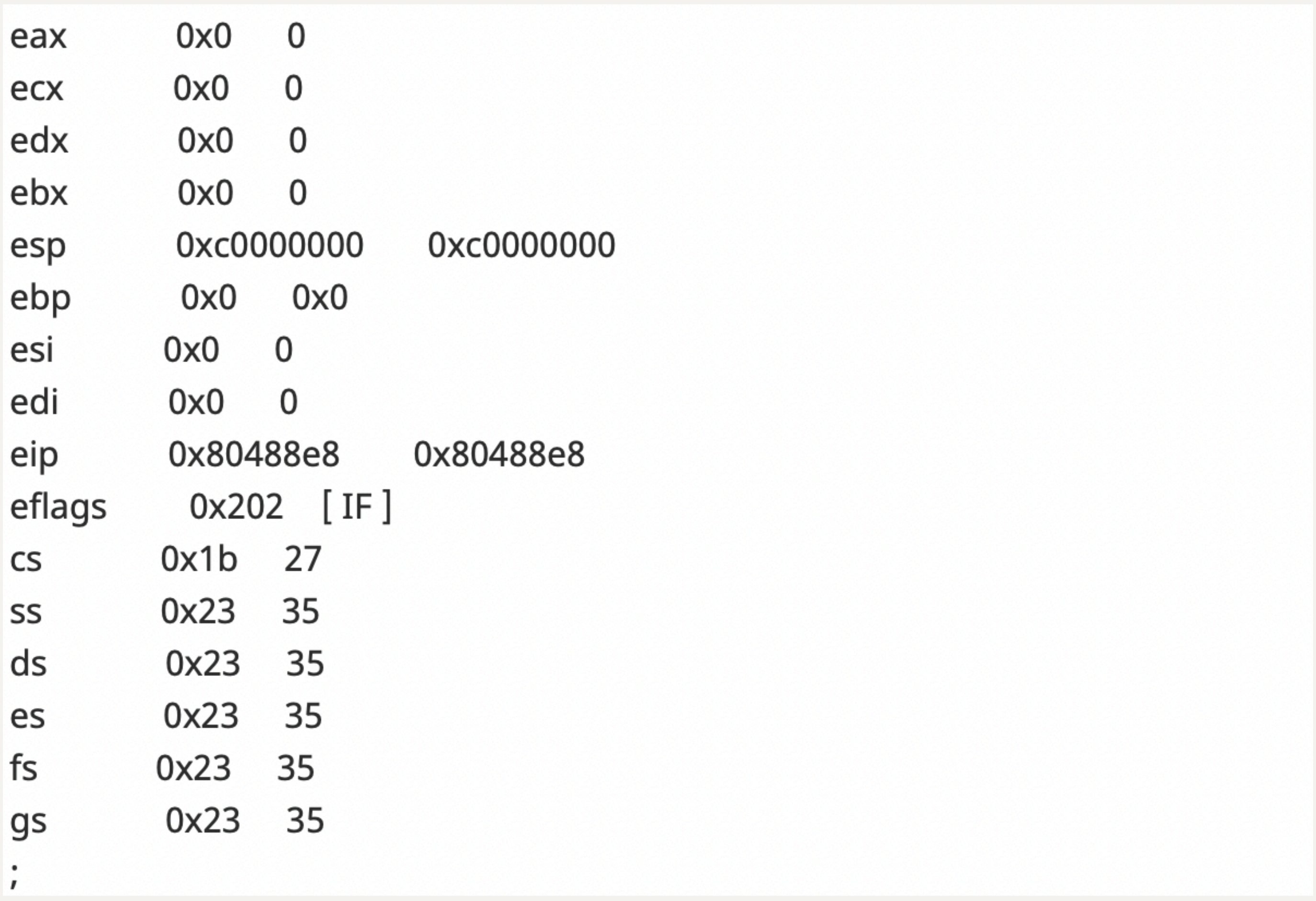
Q10 Step through the `start_process()` function until you have stepped over the call to `load()`. Note that `load()` sets the `eip` and `esp` fields in the `if_` structure. Print out the value of the `if_` structure, displaying the values in hex (hint: `print/x if_`).

$1 = {edi = 0x0, esi = 0x0, ebp = 0x0, esp_dummy = 0x0, ebx = 0x0, edx = 0x0, ecx = 0x0, eax = 0x0, gs = 0x23, fs = 0x23, es = 0x23, ds = 0x23, vec_no = 0x0, error_code = 0x0, frame_pointer = 0x0, eip = 0x80488e8, cs = 0x1b, eflags = 0x202, esp = 0xc0000000, ss = 0x23}

Q11 The first instruction in the `asm volatile` statement sets the stack pointer to the bottom of the `if_` structure. The second one jumps to `intr_exit`. The comments in the code explain what's happening here. Step into the `asm volatile` statement, and then step through the instructions. As you step through the `iret` instruction, observe that the function "returns" into userspace. Why does the processor switch modes when executing this function? Feel free to explain this in terms of the values in memory and/or registers at the time `iret` is executed, and the functionality of the `iret` instruction.

The reason why the processor switches to user mode is that it has prepared the PCB, and needs instruction to enter the user mode to start executing the program. The functionality of iret is to return the control from the interrupt handler to the caller.

Q12 Once you've executed `iret`, type `info registers` to print out the contents of registers. Include the output of this command on Gradescope. How do these values compare to those when you printed out `if_`?



```
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xc0000000       0xc0000000
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x80488e8        0x80488e8
eflags         0x202    [ IF ]
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x23     35
gs             0x23     35
;
```

They are all the same.

Q13 Notice that if you try to get your current location with `backtrace` you'll only get a hex address. This is because because `pintos-gdb ./kernel.o` only loads in the symbols from the kernel. Now that we are in userspace, we have to load in the symbols from the Pintos executable we are running, namely `do-nothing`. To do this, use `loadusersymbols tests/userprog/do-nothing`. Now, using `backtrace`, you'll see that you're currently in the `_start` function. Using the `disassemble` and `stepi` commands, step through userspace instruction by instruction until the page fault occurs. At this point, the processor has immediately entered kernel mode to handle the page fault, so `backtrace` will show the current stack in kernel mode, not the user stack at the time of the page fault. However, you can use `btpagefault` to find the user stack at the time of the page fault. Copy down the output of `btpagefault`.

```
#0  _start (argc=-268370093, argv=0xf000ff53) at ../../lib/user/entry.c:6

#1  0xf000ff53 in ?? ()
```

Debug

Q14 Modify the Pintos kernel so that `do-nothing` no longer crashes. Your change should be in the Pintos kernel, not the userspace program (`do-nothing.c`) or libraries in `proj0/src/lib`. This should not involve extensive changes to the Pintos source code. Our staff solution solves this with a single-line change to `process.c`. Explain the change you made to Pintos and why it was necessary. After making this change, the `do-nothing` test should pass but all others will still fail.

I changed the assignment of *esp to PHYS_BASE-20 in setup_stack() method. The reason is that the space spared for the intr_frame is not enough.

Q15 It is possible that your fix also works for the `stack-align-0` test, but there are solutions for `do-nothing` that do not. Take a look at the `stack-align-0` test. It behaves similarly to `do-nothing`, but it returns the value of `esp % 16`. Write down what this program should re- turn (hint: this can be found in `stack-align-0.ck`) as well as why this is the case. You may wish to review stack alignment from Discussion 0.) Then make sure that your previous fix for `do-nothing` also passes `stack-align-0`.

It should return 12. The reason is that it needs 12 B padding before arguments for stack alignment.

Q16 Re-run GDB as before. Execute the `loadusersymbols` command, set a breakpoint at `_start`, and continue, to skip directly to the beginning of userspace execution. Using the `disassemble` and `stepi` commands, execute the `do-nothing` program instruction by instruction until you reach the `int $0x30` instruction in `proj0/src/lib/user/syscall.c`. At this point, print the top two words at the top of the stack by examining memory (hint: `x/2xw $esp`) and copy the output.

```
0xbfffff98:    1      162
```

Q17 The `int $0x30` instruction switches to kernel mode and pushes an interrupt stack frame onto the kernel stack for this process. Continue stepping through instruction-by-instruction until you reach `syscall_handler`. What are the values of `args[0]` and `args[1]`, and how do they relate to your answer to the previous question?

The value of args[0] is 3222323096=0xc010bf98.

The value of args[1] is 3222323096=0xc010bf98.

It is on the top relative to the registers' addresses.