Final Project User Programs Report

Group 36

Name	Autograder Login	Email
Sidd Shashi	student63	sshashi@berkeley.edu
Matthew Nguyen	student161	matthew.t.nguyen@berkeley.edu
Aman Doshi	student457	aman.doshi@berkeley.edu
Pritish Parmar	student251	pritishparmar@berkeley.edu

Changes

Argument Passing

1. As we wrote our code in the setup_stack() function inside process.c, we noticed our code started to clutter the function contents and became hard to read. To simplify the function, we moved our code into a setup_stack_helper() which was called by the setup_stack() function, with the relevant arguments passed in. This improved the modularity of the code and made it easier to update, maintain, and test.
2. Within the setup_stack_helper() function, instead of adding necessary padding for 16B alignment with a while loop, we simply computed how many bytes we needed with modular arithmetic, as recommended by our TA Vivek during the design session. This simplified our code and made it easier to comprehend.
3. In our original design document, to form the argv array on the stack, we initially planned to use the memcpy() function to push each string pointer to the stack, one-by-one. Instead of this laborious approach, we decided to fill an array with all the string pointers and used one memcpy() call to push all the argument pointers together, since we already had the size of the array with 4 * (argc + 1). This change simplified our code.

Process Control Syscalls

To implement process control syscalls, we only made a few minor changes against our original design document. All other design ideas were incorporated into our

current code base.

1. For our argument pointer memory validation helper functions, we modified the **valid_pointer(uint8_t *p, size_t size)** function. Instead of validating every single byte from **p** to **p + size - 1**, we just checked the validity of the first and last bytes because if either of them are invalid, the entire pointer must be invalid. This change was recommended by our design TA Vivek, since it reduces the function runtime from O(size) to O(1).
2. We changed our **valid_byte_pointer** and **valid_pointer** helper functions to return a **bool**, instead of having a **void** return type. Originally, these functions would call **process_exit()** if memory was invalid. However, we decided to have the functions return a **bool**, and call **process_exit()** from the calling function if the returned **bool** was false. This allows us to clean up the program state (e.g., free memory) before calling **process_exit**() if argument is stored at invalid memory.

File Operation Syscalls

To implement file operation syscalls, we only made 1 minor change against our original design document. All other design ideas were incorporated into our current code base.

1. To prevent a file operation occurring while loading an executable file, we originally planned to lock the read operation inside the **load()** function in process.c. Instead, as recommended by our TA, we just placed our file operations lock around our call to **syscall_exec()** helper function in our **syscall_handler()** function. This simplifies the code, as the same procedure is used for locking other file operation syscalls.

Floating Point Operations

In our design doc, we wrote an unnecessarily complicated scheme for saving our FPU which involved storing the FPU in the **thread struct** during a thread switch instead of in the **switch_threads_frame struct**. After our design review, we realized this complexity and modified our code as follows.

1. We stored our FPU in the **switch_threads_frame struct** because we now understand that a new **switch_threads_frame struct** gets created upon each thread switch, so there is no need to store the FPU in each thread struct. Also, storing in the **switch_threads_frame struct** greatly simplified the offsets we had to calculate for the **fsave** and **frstor** calls.

Also, when designing, we did not consider **start_process()** enters the user process by simulating an interrupt and loading the FPU from an interrupt frame.

2. To implement the code, we first initialized the FPU, then saved the FPU to an interrupt frame so it can be correctly loaded.

Similarly, in our **thread_create()** function, we did not consider the fact that every new thread needs a newly initialized FPU.

3. To implement this functionality, we saved the current thread's FPU, initialized a new FPU, then saved the FPU into the new thread's **switch_threads_frame struct** (so it can be loaded during a context switch), then restored the current thread's FPU. To write the assembly code in C, we reference examples provided in the testing suite.

Reflection

Overall Working Environment

Throughout the project, our group was extremely organized and diligent. We consistently worked toward deadlines without procrastinating by setting aside time throughout the weeks since project release. We outlined long-term deadlines and broke them down into smaller pieces that we could finish on a daily schedule, and we stuck to the deadlines we wrote for the most part. We set up a text group chat in which we would consistently text daily about working on the project and we met in-person/called to work at least every couple of days. The overall group dynamic was positive.

What Went Well

For us, a clear, respectful channel of communication allowed us to effectively and frequently meet with each other throughout the project. Everyone in the group listened to each others' opinions and created an environment where we were all comfortable working with and bouncing ideas off each other.

Also, breaking down tasks into smaller, more manageable components allowed us to not be intimidated by the size of the project and stay consistent with our progress. Even if we did not make great progress over one work session, we would make sure to make some progress every few days to make sure we stayed on track. In the end,

our small tasks added up and we managed to finish the project code stress-free with a few days to spare.

Areas of Improvement
One thing that we can certainly work on moving forward is working more in collaboration. There were a few times where we would spend time designing or coding individually, without thinking things through with the rest of our group. Though we would eventually share what we did with others, we found that when we collaborated on tasks, we worked more efficiently and our understanding would improve because we would bounce ideas off each other. Bugs were easier to find because another set of eyes would be able to find things that the first couldn't. Also, when people worked in isolation, their understanding on what they worked on would become concentrated so others wouldn't fully understand how things came together.

Siddharth Shashi:
Sidd designed the file operation syscalls and floating point operations. For writing code, he worked with Matthew and Aman on argument passing, worked with Aman and Pritish on process control syscalls, and worked with Matthew and Aman on floating point operations. He also outlined long and short term deadlines for task completion and organized meetings for group work.

Aman Doshi:
In the design document, Aman primarily worked on the process control syscalls. Additionally, he also provided support for argument passing and file operation syscalls. In argument passing, he helped Matthew locate where code should be written and plan high-level pseudocode. In file operation syscalls, he helped Sidd with determining edge cases and planning helper functions.

For writing code, Aman primarily worked on skeleton code for all syscalls and code for file operation sycalls. Alongside Matthew, Aman also developed the memory validation helper functions, which validate syscall arguments. Additionally, he helped Matthew and Sidd debug argument passing syscalls. He also helped Pritish and Sidd debug the **wait** and **exec** syscalls. Moreover, he contributed with Matthew, Sidd, and Pritish to implement floating point operations.

Matthew Nguyen:

Matthew completed the argument passing section of the design doc. He also helped design the file operations syscalls section of the design doc. More specifically, he helped determine which pre-defined functions to call for each file operation syscall, helped come up with the idea of putting the file descriptor table in the process struct, and helped come up with the idea of using a counter variable called next_fd for tracking the next unused file descriptor.

For writing code, he worked with Sidd and Aman on argument passing and worked with Aman on file operation syscalls. More specifically, Matthew helped clean up the pointer validation helper function used for syscalls and also wrote the **read** and **write** syscalls with Aman. Matthew also helped Aman, Sidd, and Pritish to implement floating point operations. Lastly, Matthew designed and wrote the seek-normal test case which tests the basic functionality of the **seek** syscall.

P**ritish Parmar:**
Pritish worked on creating function layouts and adding comment description for functions in syscall.c. He also worked on process control system calls like exec. Lastly, he worked on the concept check questions, and coming up with test ideas, finishing off by actually adding the test files and implementing the test.

Testing

Test 1: seek-normal
Description
Tests the basic functionality of the **seek** syscall to see if it correctly changes the position of the file reader/writer. We created this test because there was previously no test in Pintos that tested the **seek** syscall.

Overview and Expected Output
This test creates and opens a file called “test.txt” and then writes the string “hello world” into the file. The test then attempts to use the **seek** syscall to set the position of the file to position 6. Finally, the test reads the remaining characters from the file and checks to make sure that the string that was read is “world”. If the string that was read is “world” as desired, then the test will pass output “PASS”, otherwise it will fail and output "Seek syscall failed because file position was not set to 6 as expected".

Results

seek-normal.result

```
PASS
```

seek-normal.result

seek-normal.output

```
Copying tests/userprog/seek-normal to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/V6wgp7Pn_Z.dsk -m 4 -net none -nographic -monitor null
c[?7l[2J[0mSeaBIOS (version 1.15.0-1)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.....................
Kernel command line: -q -f extract run seek-normal
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  124,108,800 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 227 sectors (113 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 136 sectors (68 kB), Pintos scratch (22)
ide1: unexpected interrupt
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'seek-normal' into the file system...
Erasing ustar archive...
Executing 'seek-normal':
(seek-normal) begin
(seek-normal) create "test.txt"
(seek-normal) open "test.txt"
(seek-normal) end
seek-normal: exit(0)
Execution of 'seek-normal' complete.
Timer: 66 ticks
Thread: 8 idle ticks, 25 kernel ticks, 33 user ticks
hda2 (filesys): 95 reads, 282 writes
hda3 (scratch): 135 reads, 2 writes
Console: 1017 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

~/code/group/src/userprog/build/tests/userprog/seek-normal.output

seek-normal.output

Potential Kernel Bugs

Kernel Bug 1:

If the kernel has an off-by-one error where **seek()** doesn't place the file offset at the correct position, then the test case would output "Seek syscall failed because file position was not set to 6 as expected." This is because **read()** wouldn't read the correct 6 bytes corresponding to "world\0".

Kernel Bug 2:

If the kernel performed relative seeking instead of absolute seeking when the **seek()** syscall is called, then the test case would fail and output "Seek syscall failed because file position was not set to 6 as expected". This is because after we write "hello world" to the file "test.txt" the position of the file reader/writer will be at EOF. Then, when we call **seek(handle, 6**), where handle is the fd of "test.txt**",** we would set the position of the file to EOF+6. Therefore, when we call try to read the remaining characters from the file, we would not read "world\0" as expected.

Test 2: remove-normal

Description

Test the **remove** file operation syscall, as it is not fully tested by the existing test suite. The test will rely on the correctness of **create**, **open**, and **write** file operation syscalls, which are all well tested in Pintos.

Overview and Expected Output

We create and open a file "test.txt". We then remove the file. If the call to remove syscall does not return true, the test outputs a fail message "Remove syscall failed to remove test.txt".

Next, we write "hello" to the removed file. Since the file is not closed, this operation should succeed. If the write fails, the test outputs a fail message "Unable to write to file after file removed but not closed".

Lastly, we close the file and then try to reopen it. Closing the file should remove the file, as there are no other references to it. If the file is able to be reopened, then the test outputs a fail message "File opened after file removed and closed".

Otherwise, the test outputs "PASS".

Results

remove-normal.result | remove-normal.output

group > src > userprog > build > tests > userprog > remove-normal.output

```
Copying tests/userprog/remove-normal to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/f1zhbSlkp8.dsk -m 4 -net none -nographic -monitor null
[?7l[2J[0mSeaBIOS (version 1.15.0-1)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.....................
Kernel command line: -q -f extract run remove-normal
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  192,921,600 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 227 sectors (113 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 136 sectors (68 kB), Pintos scratch (22)
ide1: unexpected interrupt
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'remove-normal' into the file system...
Erasing ustar archive...
Executing 'remove-normal':
(remove-normal) begin
(remove-normal) create "test.txt"
(remove-normal) open "test.txt"
(remove-normal) end
remove-normal: exit(0)
Execution of 'remove-normal' complete.
Timer: 61 ticks
Thread: 7 idle ticks, 24 kernel ticks, 31 user ticks
hda2 (filesys): 115 reads, 285 writes
hda3 (scratch): 135 reads, 2 writes
Console: 1036 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...

```

remove-normal.output

remove-normal.result | remove-normal.output

group > src > userprog > build > tests > userprog > remove-normal.result

```
PASS

```

remove-normal.result

Potential Kernel Bugs

1. If the kernel removes the file “test.txt” before all references to it are closed, then the test will fail and output "Unable to write to file after file removed but not closed”. In the test, this is checked by a write operation that happens after the remove operation and before the close operation. Hence, if the write operation

fails, then the file has been removed before it has been closed and the test raises the appropriate error.
2. If the file is able to be opened after it is removed and all references to it are closed, then the test will fail and output "File opened after file removed and closed". According to the spec, once a file is removed and all reference to it are closed, the file cannot be re-opened normally as it no longer exists. This is checked by trying to open the file after the close operation, and checking the open operation returns -1.

Overall Testing Experience

Writing Pintos Tests

In Pintos, to write a test, we must implement test code in a .C file, write the expected output in a .ck file, and then update the make file to run the test and make sure all of its dependencies are added.

Improvements

Writing tests in Pintos is laborious compared to other testing suits our group has used in other classes (e.g., JUnit in CS 61B). For an improved testing experience, the Pintos testing suite should allow the programmer to embed the test and expected result into one C file, instead of splitting the test across 2 files. This change leads to 2 main benefits.

1. There are less files cluttering the test directory, making it easier to locate tests when bugs occur.
2. Writing and updating tests is easier, since all the work is done in a single C file.

What We Learnt

We learned several things from writing tests. For example, we learned that there are several potential kernel bugs (e.g. off-by-one errors) that can be very hard to spot. This taught us the importance of having good test cases to easily find and address these bugs. Furthermore, we learned that there are numerous edge cases that we have to account for when writing a large system like Pintos. This taught us the importance of having a variety of test cases to test these edge cases so that we don't miss anything. Lastly, we learned about the importance of having test cases that provide useful, detailed feedback/information that will allow us to understand the bugs in our code and address them appropriately.