Project User Programs Report

Group 43

Name	Autograder Login	Email
Arul Loomba	student483	arulloomba@berkeley.edu
Michael Hom	student324	michaelhom@berkeley.edu
Dmytro Krukovskyi	student387	dkruk@berkeley.edu
David Zechowy	student496	davidzechowy@berkeley.edu

Changes

Argument Passing

When we talked with our TA, we initially agreed to have the alignment after we add the argument pointers to the stack. The order in which we added the arguments to the stack instead mimics the Program startup details in the Pintos documentation. This was the other option we discussed in our review, and it is equally valid.

Address	Name	Data	Type
0xbfffffc	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	stack-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbfffffc	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 (*) ()

1. The arguments from the command line are entered into the stack in reverse order.

2. The stack is aligned by adding an u_int8_t to account for the size of the arguments to have the address of argc to be 16 byte aligned.

3. The pointers of the arguments initially added to the stack are imputed next also in reverse order.

4. Lastly, we add the argv, argc, and the pretend return address of 0.

Process Control Syscalls

We updated the child structure to include a semaphore, wait_sema, for the parent and child to access when editing the statuses of each child. In addition, we added a was_waited flag to check if the child has been waited on once before, as you cannot wait twice for the same child.

An exit status and a normal_exit flag were added in the process structure as well to catch the edge case of the parent exiting before the child or children.

For the execute function, we added a sema_load in the process structure to wait for the child to load the files to execute. To help the execute syscall, the boolean load_success in process structure will store whether the child has successfully loaded the files or not.

For fork syscall we added a page mapping struct so we don't have to scan through every byte in the memory; This makes copying address space much faster.

```
struct page_mapping {
  void* upage; // Virtual address
  void* kpage; // Physical page
  bool writable;
  struct list_elem elem;
};
```

File Operation Syscalls

We made a significant change in adding a brand new structure to further separate the file descriptor with the file which could have multiple pointers redirecting toward it. This implementation allows for the concept of a global file descriptor table to manage the reads and writes on a file entry. The change to make our lives similar is the structure of the process control includes a next_fd in addition to the pintos list of the file descriptor.

Reflection

Arul worked on the argument passing and the practice syscall. Dymtro worked on exec and halt syscall. Everyone contributed to the creation of the wait function. Arul, Dymtro, and David worked on the fork call. In addition, Arul worked on the file operation syscalls. Michael worked on the process syscalls, helped with documentation and creating this report, as well as helping with file operation syscalls.

The working environment was mainly on campus to discuss the coding aspects in a mix of pair programming and dividing the work among how the design document was created, as some members of the team were more familiar with their particular concepts they researched. We strived when working as a team while in-person and were great in communicating when to meet and how much progress we have made as a group. Some things we could have done better is to start earlier on this project as we were rushed near the end, giving us less time to ask questions at office hours, which could have given us more insight into issues we ran into.

Testing

fs-ops-1

This test creates a file, writes data to it, then tests that:

- write() returns the size of written data
- filesize() returns the correct size while the file is open.
- After calling close() once, filesize() returns -1.
- Calling close() a second time does not change that behavior.
- If all checks pass, the process exits with 0.

The main focus is to confirm the behavior of the file operation syscall, filesize, but we confirm that the behavior of write, create, and close remain functional while interacting with filesize. We create a file to check the size of the write as well as the file. We set the initial size of the data passed as well as the file size to predict what the program should return. Then we close the file. Any access to the file after it is closed should error in a controlled fashion, exiting with an -1. The entries in both the local file descriptor table in the process control block and the global file descriptor table are removed when close is removed if there are no other references to the file. The behavior of the filesize should not change after multiple calls after the initial close has occurred.

Output and results of your own Pintos kernel when you run the test case. These files will have the extensions .output and .result.

fs-ops-1.output

```
Copying tests/userprog/fs-ops-1 to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/e1n5iaATyO.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 fs-ops-1
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  95,027,200 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 234 sectors (117 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 137 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 'fs-ops-1' into the file system...
Erasing ustar archive...
Executing 'fs-ops-1':
(fs-ops-1) begin
(fs-ops-1) create fs-ops-1.txt
(fs-ops-1) open fs-ops-1.txt
(fs-ops-1) write fs-ops-1.txt
(fs-ops-1) (fs-ops-1) filesize after write: 100
(fs-ops-1) filesize returns 100
(fs-ops-1) (fs-ops-1) filesize after close: -1
(fs-ops-1) filesize on closed FD returns -1
(fs-ops-1) (fs-ops-1) filesize after second close: -1
(fs-ops-1) filesize on already closed FD returns -1
(fs-ops-1) (fs-ops-1) end
fs-ops-1: exit(0)
```

Execution of 'fs-ops-1' complete.
Timer: 65 ticks
Thread: 5 idle ticks, 26 kernel ticks, 34 user ticks
hda2 (filesys): 96 reads, 284 writes
hda3 (scratch): 136 reads, 2 writes
Console: 1311 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...

fs-ops-1.result

PASS

Test 1: fs-ops-1 (Close and Filesize) Test

1. Improper FD Removal on Close
    a. Description:
    If kernel did not remove the file descriptor from the process's file descriptor table when close() is called (i.e. it left the descriptor still valid), then after closing the file, filesize(fd) would still return the file's size (100) instead of –1.
2. Bug 2: Reusing Closed File Descriptor Without Marking Invalid
    a. Description:
    If my kernel reused or did not mark a closed file descriptor as invalid, then a second close() might succeed or leave the FD in a valid state, causing filesize(fd) to return a valid size (or even the original size) rather than –1.

---

fs-ops-2

This test creates a file, writes data into it, and then tests the following:

- tell() returns the correct file offset after writing, seek() correctly updates the file position.
- That reading after seek yields the expected data.
- That seeking past the end-of-file results in a read of 0 bytes.
- That after closing the file, tell() returns -1.
- If all checks pass, the process exits with 0.

The main focus is to confirm the behavior of the file operation syscalls, seek and tell, but we confirm that the behavior of write, create, and close remain functional while interacting with the functions, seek and tell. The seek function should edit the local and global file descriptor table to move the offset for future reads or writes on the file. The tell function is reporting where the current location is to give users a simple way to check where the file offset is. Our tests catch normal operations of the functions as well as when we run into the edge case of after the file closes. We decided to have the tell function exit with -1, so this is what we look for in our test.

Output and results of your own Pintos kernel when you run the test case. These files will have the extensions .output and .result.

fs-ops-2.output

```
Copying tests/userprog/fs-ops-2 to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/fo97auqc_W.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 fs-ops-2
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  78,540,800 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 234 sectors (117 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 138 sectors (69 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 'fs-ops-2' into the file system...
Erasing ustar archive...
Executing 'fs-ops-2':
(fs-ops-2) begin
(fs-ops-2) create fs-ops-2.txt
(fs-ops-2) open fs-ops-2.txt
(fs-ops-2) write fs-ops-2.txt
(fs-ops-2) (fs-ops-2) tell after write: 11
(fs-ops-2) tell returns 11
(fs-ops-2) (fs-ops-2) tell after seek to 5: 5
(fs-ops-2) tell returns 5
(fs-ops-2) read returns 3 bytes after seek
(fs-ops-2) data read after seek is 'FGH'
(fs-ops-2) (fs-ops-2) tell after seek to 100: 100
(fs-ops-2) tell returns 100 after seek to 100
(fs-ops-2) read returns 0 when reading at EOF
(fs-ops-2) (fs-ops-2) tell on closed FD: -1
(fs-ops-2) tell on closed FD returns -1
(fs-ops-2) (fs-ops-2) end
fs-ops-2: exit(0)
Execution of 'fs-ops-2' complete.
Timer: 63 ticks
Thread: 3 idle ticks, 22 kernel ticks, 38 user ticks
hda2 (filesys): 99 reads, 286 writes
hda3 (scratch): 137 reads, 2 writes
Console: 1486 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

fs-ops-2.result

```
PASS
```

For fs-ops-2 (Seek and Tell) Test

1. Bug 1: Faulty File Pointer Update on Seek
    a. Description:
    If my kernel did not update the file pointer correctly when a seek() call is made, then tell() would return the old offset rather than the new position.
2. Bug 2: FD Not Invalidated on Close for Tell
    a. Description:
    If kernel did not invalidate the file descriptor on close instead of removing it, then the test case would output a valid offset (for example, '(fs-ops-2) tell on closed FD: 100') instead of '(fs-ops-2) tell on closed FD: -1'.

In addition, tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? What did you learn from writing test cases?

- Writing tests for Pintos was fairly challenging. We had to dive into how the kernel was working and whether or not certain test requirements would result in certain errors, especially because of system calls, process management, and file system operations. It also forced us to understand how all the different processes, file descriptors, and memory management interacted, and it revealed small bugs that might have been missed otherwise. We spent considerable time designing tests and also inputting them and making sure they passed. We have learned how to better create tests, integrate them with the pintos code and overall improve the quality of the code.