Project 1: User Programs Final Report

Group Members

Xinyu Fu (fuxy@berkeley.edu)
James Tang (jamestang@berkeley.edu)
Lanxiang Hu (hlxde1@berkeley.edu)

Student Testing Report

Test 1: CHILD_FILE()

We noticed that tests such as `multi-child-fd.c` along with `child_close` test the functionality that child processes do not inherit parent fd's, and so closing a file opened by the parent in a child process fails. In this test, having opened a file in the parent process, we want to check whether a subprocess that tries to open and close the same file could succeed. Lastly, we close the file in the parent process as well, which must also succeed since the parent process and the child process are supposed to handle the file independently.

child_file.output:

```
Copying tests/userprog/child_file to scratch partition...
Copying ../../tests/userprog/sample.txt to scratch partition...
Copying tests/userprog/c-open-close to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/FkSU5lW3qc.dsk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading............
Kernel command line: -q -f extract run child_file
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  131,072,000 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 205 sectors (102 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 224 sectors (112 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'child_file' into the file system...
Putting 'sample.txt' into the file system...
Putting 'c-open-close' into the file system...
Erasing ustar archive...
Executing 'child_file':
(child_file) begin
(child_file) open "sample.txt"
(c-open-close) begin
(c-open-close) end
c-open-close: exit(0)
(child_file) wait(exec()) = 0
(child_file) close "sample.txt" in parent
(child_file) end
child_file: exit(0)
Execution of 'child_file' complete.
Timer: 73 ticks
Thread: 11 idle ticks, 60 kernel ticks, 2 user ticks
hda2 (filesys): 159 reads, 456 writes
hda3 (scratch): 223 reads, 2 writes
Console: 1153 characters output
Keyboard: 0 keys pressed
```

```
Exception: 0 page faults
Powering off...
```

child_file.result:

```
PASS
```

Potential Pintos Bug

The test created a buffer called `child_cmd` to store the command for the child process. If the buffer was not created properly, either in invalid memory or not with enough memory, then the child command might not be executed as expected.
It's possible that the parent process would exit with exit code = -1 and would not go into executing the child at all in case when `open()` was unsuccessful, e.g.: if "sample.txt" isn't created beforehand.

Reflection on Writing Tests in Pintos

Test 2: SYS_SEEK()

We added a unit test for `sys_seek()` function in `/tests/userprog/test-seek.c`, which first opens the file sample.txt, then call `seek()` to point fd to the 5th byte from the beginning of the file. I checked whether the program will match it to the 5th character in sample.txt (which is 'i'). If the character mismatches, we fail the test.

test-seek.output:

```
Copying tests/userprog/test-seek to scratch partition...
Copying ../../tests/userprog/sample.txt to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/R0mT_valzj.dsk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading..........
Kernel command line: -q -f extract run test-seek
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  383,385,600 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 181 sectors (90 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 106 sectors (53 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'test-seek' into the file system...
Putting 'sample.txt' into the file system...
Erasing ustar archive...
Executing 'test-seek':
(test-seek) begin
(test-seek) open "sample.txt" for verification
(test-seek) seek position 5 "sample.txt"
(test-seek) close "sample.txt"
(test-seek) end
test-seek: exit(0)
Execution of 'test-seek' complete.
Timer: 76 ticks
Thread: 8 idle ticks, 67 kernel ticks, 2 user ticks
hda2 (filesys): 92 reads, 218 writes
hda3 (scratch): 105 reads, 2 writes
Console: 1049 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

test-seek.result:

```
PASS
```

Potential Pintos Bug

The test created a buffer called `block` to store the byte read after `seek` was called after position 5 of the file. If the buffer was not created properly, either in invalid memory or not with enough memory, then the subsequent if statement checking whether block[0] matches our expectation might incur errors.
In this test, a single letter is checked to see whether the result of read after the seek syscall (character `i` and it is stored as the first element in the buffer `block`) matches our expectation. However, it's possible for any arbitrary byte, say the 10th byte or 2043th byte, to be `i` as well and be read as the next byte after calling `seek`.

Changes Made Since Design Doc

Task 1

In our original design, our intention was to modify the `start_process` function in parsing the command-line arguments and push them onto the stack for the user process. However, it was later discovered by us that the `esp` pointer is not easily accessible outside of the `load` function as the `_if` interrupt frame is modified. As a result, we implemented all stack-frame related processes in the `load` function, so it would return a finalized `esp` after the arguments have been pushed.

Moreover, in order to explicitly print out `exit(-1)` when an exception like page fault occurs and sys_exit(-1) is called, we added the exit instruction `terminate(-1)` which contains the error information in `exception.c` so that the exit message can be properly printed out even if a process exits due to a fault.

Task 2

For the `exec` syscall, since the files opened and accessed by the parent and child thread should be independent of each other even if they are of the same name, we need an independent copy of the file name, assigned as `fn_copy`, for the child thread which is to be parsed and modified by `strtok_r()` in argument passing. We also copied the file name an extra time as `wi_fn' so that an intact copy can be passed into the` wait_info `in successfully executing the` load` function with the correct file name.

In implementing the `wait` syscall, we made some changes to our `wait_info` struct shared between parent the child, the modified version looks like:

struct wait_info {
struct semaphore sema; // 0=child alive, 1=child dead.
char* file_name; // File name of the program, used in loading the executable.
int ref_cnt; // 2 - both alive, 1 - either child or parent alive, 0 - both dead
struct lock lock; // allows mutual exclusion when modifying members in the struct
int exit_code; // Pointer to the child's exit_code.
bool loaded; // whether the child process have successfully loaded executable
bool waited; // whether the parent is has waited for the child
tid_t tid; // tid of the child thread
struct list_elem elem; // for creating a list of children_wait_info
};

We added the field `file_name` passed down to the struct from `process_execute` since we modified the `start_process` function so that it takes in the struct `wait_info` rather than the file name. This allows us to update the `wait_info`'s semaphore and `loaded` boolean in communicating with the parent thread. And we store the file name in the struct so that it can be further used when needed. This way, we utilize the same struct `wait_info` to deal with the case where the parent waits for the child to load, as well as the case where the parent waits for the child to exit.

Task 3

struct fd_row {
int fd;
struct list_elem elem;
struct file* open_file_object;
};

struct thread {
/* Owned by thread.c. / *tid_t tid;* / Thread identifier. / *enum thread_status status;* / Thread state. / *char name[16];* / Name (for debugging purposes). / *uint8_t* stack; /* Saved stack pointer. / *int priority;* / Priority. / *struct list_elem allelem;* / List element for all threads list. / *struct wait_info* wait_info; /* wait_info struct for current thread. / *struct list children_wait_info;* / list of wait_info for children threads. / *struct list fdt;* / Fd table consists of a list of table entries. / *int next_aval_fd;* / Keeps track of the next available fd to assign. / *struct file* cur_file; /* record current file */
}
Instead of using an array to store each open file object, we used linked lists and created a struct called `fd_row` (represents a row in the fd table) which contained a file descriptor, a list element, and an open file object. In the struct `thread`, we also added the field `fdt` which represents a fd table that contains a list of file descriptor entries.
In addition, we added the field `next_aval_fd` in the `thread` struct, incrementing it every time we open a new file, so that we can use a unique file descriptor whenever we open a file.

Reflection

What each of us did:

- Xinyu Fu: implemented task 3
- James Tang: implemented practice, halt, exit, and wait syscall.
- Lanxiang Hu: implemented task1, exec syscall and student tests.

What went well:

1. Overall, the project went very smoothly in terms of communication, splitting up the tasks, and putting all our code together. We were able to utilize zoom meetings to get everyone involved in the coding process.

2. All of us had a thorough understanding of the course material as well as solid coding skills needed to complete the project.

3. We all wrote code that was easy to understand with lots of comments to facilitate communication. This saved a lot of time in the debugging process because we had to look back at the code and figure out where the bug was.

What could be improved:

1. A lot of time was spent on issues with github. We should get more familiar with it and utilize the branch functionality to ensure everyone has their code up-to-date.

2. We underestimated how long the debugging process would take and ended up submitting the project using 2 slip days. This is a sign that we should start earlier next time and start debugging as early as possible. We will never know how many bugs there are until we run tests.

3. Following the point above, we also underestimated the time it takes to write a test, which led to some frustration by the end of the project. We struggled a bit with comprehending the pintos test mechanism and modifying the makefile. In the future, we should think about writing tests when we are coding.