Operating System: Project 2

Instructed by *Wei Xu*

Due on Nov 19, 2020

Tests for File Management Syscalls

The listed C programs test different features of the file management syscalls.

`testFileBasic.c`

This program tests the basic functionalities of `creat`, `open`, `read`, `write`, `close` and `unlink`:

- open `testFileBasic0.file`, read the content (“Reading.”), and then close the file;
- create `testFileBasic1.file`, write the content (“Writing.”), and close the file; then reopen it for reading to verify whether we have successfully written;
- simultaneously handle two files `testFileBasic1.file` and `testFileBasic2.file` (creating, writing some new content (“Rewrite.”) and closing);
- check the content of `testFileBasic1.file` to verify that `creat` successfully overwrites an exiting file;
- delete `testFileBasic2.file`.

`testFileMultiple.c`

This program tests whether the syscalls correctly manage multiple files:

- try to create 16 files (`testFileMultiple**.file`); since two file descriptors have been occupied by `stdin` and `stdout`, the last two attempts should fail and return -1, while other attempts should return the correct descriptors (in default settings, 2 through 15 are allocated);
- write some initial content (“Testing.”) to all 14 files and close them;
- recreate the first 7 files with some new content (“Rewrite.”);
- open all 14 files, examine their contents and close them.

`testFileBig.c`

This program tests whether we can correctly read and write big files (files longer than one page):

- first write 2048 characters into a newly created file `testFileBig.file`, and close it;
- then reopen it and read its contents for verification (we cannot print due to restrictions of `stdout`).

`testFileError.c`

This program tests some common erroneous file operations (for which the kernel should not crash):

- open and unlink non-existing files (should return -1);
- read and write unallocated file descriptors (should return -1);
- test for the case where read/write length is larger than the buffer size (in the last case, even writing the entire virtual memory into a file); however, this should not trigger any exception with C, since memory leak is not regarded as an error (as long as the memory addresses are valid);
- test for the case where read/write length is 0;
- try to close `stdin` and `stdout` (note that after closing `stdout`, we cannot output any characters to the console, and any further `write` syscall will lead to errors).

`testFileUnlink.c`

This program tests for the case where we try to unlink an open file (postponed deletion until closing is expected) and read file contents before closing it (this is valid, since unlink does nothing before closing).

However, it seems that the file system does not support delayed unlink, so the `unlink` syscall directly fails and returns -1 (unsuccessful deletion), while the program runs straight to the end. As discussed online, this should be regarded as a bug of NachOS internals (beyond our implementations).

`testFileConflict.c` and `testFileConflictChild.c`

This program tests for the case where multiple processes try to open (or even overwrite) the same file. In a typical OS, multiple opening is allowed, but overwriting should be illegal. However, NachOS does not guarantee to handle this correctly, which should also be regarded as a bug of NachOS internals.

Tests for Process Management Syscalls

`child_*.c`

`child_1` invokes and joins `child_2` first. Then `child_1` does its own output. Finally, it invokes but does not join `child_3`, which contains an infinite loop.

If directly call `child_1`, the bash will never halt. If call `child_1` in `sh`, the process `child_3` will run in background.

`text_halt.c`

This program plays the role as an evil user who invokes `halt()`. In this test, the OS prints error message to console and the bash does not really halt.

`textExecNesting.c`

This program tests for nested `exec` calls, which receives a parameter n. When $n > 0$, it runs another instance of itself with parameter $(n-1)$; when $n = 0$, it executes `testFileMultiple.c`. Due to the limited memory size by default, we shall never run 5 processes simultaneously, so only the case $n = 1$ succeeds, while any larger n will results in early returning (and no-free-memory error messages).

Test for Memory Management

`testMemoryStack.c` and `testMemoryStatic.c`

This program tests for the layout of different sections within memory. Note that the 10000-byte buffer is allocated in the stack in `testMemoryStack.c`, and it is allocated in the static section in `testMemoryStatic.c`. Since the stack always covers 8 pages, the stack version will overflow, and overwrite part of the file name (`char*` in stack) so that the file opening operation fails; while the static version runs smoothly. *(This is an interesting overflow "bug" we met in testing; we only include it here for fun.)*

Test for Lottery Scheduler

Basically, we reuse the tests for schedulers in project 1.

`Lock.schedulerTest() in Lock.java`

We construct 9 threads which are waiting for a lock that is held by a thread with relatively low priority. Meanwhile, a thread with higher priority consumes CPU time as long as the lock is not free and it revokes the *yield* function periodically. Without priority donation, thread 0 will be stuck forever.

`Condition2.schedulerTest()`

Test (`Condition2.schedulerTest`) relies on the Consumer-Producer model to test whether the queue is correctly organized when the number of consumers get larger.