Testing

QUESTIONS

Pintos already contains a test suite for file system functionalities, but it does not cover the buffer cache. For this project, you must implement **two** of the following test cases:

1. Test your buffer cache's effectiveness by measuring its cache hit rate. First, reset the buffer cache. Next, open a file and read it sequentially, to determine the cache hit rate for a cold cache. Then, close it, re-open it, and read it sequentially again, to make sure that the cache hit rate improves.
2. Test your buffer cache's ability to coalesce writes to the same sector. Each block device keeps a `read_cnt` counter and a `write_cnt` counter. Write a large large file at least 64 KiB (i.e. twice the maximum allowed buffer cache size) byte-by-byte. Then, read it in byte-by-byte. The total number of device writes should be on the order of 128 since 64 KiB is 128 blocks.
3. Test your buffer cache's ability to write full blocks to disk without reading them first. If you are, for example, writing 100 KiB (200 blocks) to a file, your buffer cache should perform 200 calls to `block_write`, but 0 calls to `block_read` since exactly 200 blocks worth of data are being written. Read operations on inode metadata are still acceptable. As mentioned earlier, each block device keeps a `read_cnt` counter and a `write_cnt` counter. You can use this to verify that your buffer cache does not introduce unnecessary block reads. **If your buffer cache does not have this property, then implement the other two options listed above.**

You should focus on writing tests for general buffer cache features, rather than writing tests for your specific implementation of the buffer cache. You should write your test cases with a minimal set of assumptions about the underlying buffer cache implementation, but you are permitted to make as many basic assumptions about the buffer cache as you need to, since it is very difficult to write buffer cache tests without doing so. Use your best judgement and create test cases that could potentially be adapted to a different group's project without rewriting the whole thing. Once you

finish writing your test cases, make sure that they get executed when you run `make check` in the `filesys/` directory.

Implementation details

You should add your two test cases to the `filesys/extended` test suite, which is included when you run `make check` from the filesys directory. All of the filesys and userprog tests are "user program" tests, which means that they are only allowed to interact with the kernel via system calls. **Since buffer cache information and block device statistics are NOT currently exposed to user programs, you must create new system calls to support your two new buffer cache tests.** You can create new system calls by modifying these files (and their associated header files):

`lib/syscall-nr.h`

> Defines the syscall numbers and symbolic constants. This file is used by both user programs and the kernel.

` lib/user/syscall.c`

> Syscall functions for user programs.

`userprog/syscall.c`

> Syscall handler implementations.

Some things to keep in mind while writing your test cases:

- User programs have access to a limited subset of the C standard library. You can find the user library in `lib/`.
- User programs cannot directly access variables in the kernel.
- User programs do not have access to `malloc`, since `brk` and `sbrk` are not implemented. User programs also have a limited stack size. If you need a large buffer, make it a static global variable.
- Pintos starts with 4MB of memory and the file system block device is 2MB by default. Don't use data structures or files that exceed these sizes.
- Your tests should use `msg()` instead of `printf()` (they have the same function signature).

Adding file system tests to Pintos

You can add new test cases to the `filesys/extended` suite by modifying these files (all inside `tests/filesys/extended`):

`Make.tests`

> Entry point for the `filesys/extended` test suite. You need to add the name of your test to the `raw_tests` variable, in order for the test suite to find it.

`my-test-1.c`

> This is the test code for your test (you are free to use whatever name you wish, "my-test-1" is just an example). Your test should define a function called `test_main`, which contains a user-level program. This is the main body of your test case, which should make syscalls and print output. Use the `msg()` function instead of `printf`.

`my-test-1.ck`

> Every test needs a .ck file, which is a Perl script that checks the output of the test program. If you are not familiar with Perl, don't worry! You can probably get through this part with some educated guessing. Your check script should use the subroutines that are defined in `tests/tests.pm`. At the end, call `pass` to print out the "PASS" message, which tells the Pintos test driver that your test passed.

`my-test-1-persistence.ck`

> Pintos expects a second .ck file for every `filesys/extended` test case. After each test case is run, the kernel is rebooted using the same file system disk image, then Pintos saves the entire file system to a tarball and exports it to the host machine. The `*-persistence.ck` script checks that the tarball of the file system contains the correct structure and contents. **You do not need to do any checking in this file, if your test case does not require it.** However, you should call `pass` in this file anyway, to satisfy the Pintos testing framework.

Copyright © 2022 CS 162 staff.