LINKÖPINGS UNIVERSITET

TDDE47/TDDE68: Concurrent programming and Operating Systems

Lab D: Memory and thread safety

2026-01-15

Dag Jönsson
IDA/SaS

1 Introduction

1.1 Goal

In this lab you will be working on making your operating system safer, both in terms of memory and thread safety.

You might recall that you were told to not trust pointers originating from user space? And you might have read something about the file system not being thread safe yet. You will be fixing both of those shortcomings now!

1.2 Reading material

- Lectures 7, 8, 9
- Course literature chapters:
  - Something I'm sure
- Stanford Pintos:
  - Project 2: User programs Specifically 3.1.4, 3.1.5
  - Project 3: Virtual Memory Specifically 4.1.2-4.1.2.3
  - Project 4: File systems Specifically 5.1, 5.3.5
  - Reference Guide Specifically A.6, A.7
- Files:
  - `threads/vaddr.h`
  - `userprog/pagedir.h`
  - `filesys/filesys.[h,c]`
  - `filesys/free-map.[h,c]`
  - `filesys/directory.[h,c]`
  - `filesys/file.[h,c]`
  - `filesys/inode.[h,c]`

1.3 Preparatory question

Question 0: Memory concerns

a) What specific address is the *start* of kernel space? Is it defined by a name?
b) What specific address is the *end* of user space? Where does it start?
c) Why are you not allowed to dereference `NULL` ? What address does `NULL` "point" to?
d) Where should pointers coming from a user space program point to? Where should they absolutely not point to?
e) What could go wrong if we allowed user programs to pass invalid pointers to the system call handler?

Question 1: Memory paging

a) How many levels of paging does Pintos utilize?
b) What is a page in the context of memory?
c) What is a memory frame? How does it relate to pages?
d) How is a virtual address translated into a physical address?

e) How large is a single page in Pintos?
f) How can you check what page a given address belongs in?
g) What is a page table? How is it represented in Pintos?
h) Where is the current thread's page table stored?
i) How can you check if a given address is mapped to a page belonging to the current thread?
j) If you know that a specific address is mapped to the current thread, what do you know about the page that the address is in?
k) How many addresses in page do you need to validate to know that the whole page is valid?

Question 2: Pointer validation

a) How many pointers originate from the user program in the system call `write()` ? What type of data do they point to?
b) How many pointers originate from the user program in the system call `open()` ? What type of data do they point to?
c) Can the user program modify the stack pointer?
d) What criteria need to be fulfilled for a *buffer* to be valid?
e) What criteria need to be fulfilled for a *string* to be valid? (Cstrings)
f) Write down what pointers each system call take as argument, and also what type of validation need to be done for the given pointer. (buffer or string)

By now you should have all the knowledge you need to create a solution for Assignment A.

Question 3: File system

a) Give a short description of what responsibility the below structures within the Pintos file system has:
   - filesys
   - free-map
   - directory
   - file
   - inode
b) What functions are involved when creating/removing a file?
c) What functions are involved when opening/closing a file?
d) What functions are involved when reading/writing a file?
e) Go through questions b-d, and write down what shared state exist between these functions.
f) How many `dir` structs are created when we open 5 separate files?
g) Describe, in your own words, what the reader-writer problem is.
h) What functions and structures would be of interest if you were to implement a solution to the reader-writer problem in the Pintos file system?

2 Assignment A: Pointer validation

As you might recall from lab B, you were explicitly told to not worry about pointer validation. You might also remember that there were a handful of tests that aren't passing from the last lab. This is something you will fix now.

With your answers from the prep questions, implement pointer validation in the system call handler.

2.1 Testing

When you have implemented a working solution, those last tests should pass.

Once you pass all the current tests, there are more! The file `userprog/Make.vars` holds the configuration for what tests are run. You should modify it so that the `tests/filst` run as well, you can do this by removing the `#` just before `tests/filst`. Do a `make clean; make -j8 check` and you will run a couple of more tests that try really hard to break you validation. Fix any potential bug found by these new tests and you should be good to move on.

3 Assignment B: File system synchronization

Up to this point, the file system has been thread-*unsafe*, which is the goal to fix in this assignment. After this, your operating system will be fairly competent, and could potentially be used as a OS somewhere.

With the answers from the prep questions in mind, go through the file system and add synchronization where it's required. Some points to keep in mind:

- The block level (device level) is already synchronized, and you don't have to think about it.
- Synchronize as small a part as possible. It's not acceptable to just put a single global lock in the file system and acquire it as soon as you do anything with the file system.
- Functions that are not working on the same shared resource should not interfere with each other.
- The file system should allow a file on disk to be read concurrently by an arbitrary amount of threads. If a thread is writing to a file on disk, that specific file is not allowed to be read from or written to by a different thread.
- Two or more separate files being read/written should not interfere with each other.

3.1 Testing

To test your synchronization of the file system, you first need to enable the test for that. In `userprog/Make.vars`, remove the `#` before `tests/filesys/base` to enable them, run a `make clean; make -j8 check` and make sure all tests pass.

Once those are passing, we have a couple more to round the test suite out. Make a final change to `userprog/Make.vars`, remove the last `#` before `tests/klaar`, and run the test suite again. The number of passing tests should be 90.

These last tests are a bit scattered in what they tests, but there are two that are slightly more involved than the other. `tests/klaar` includes the `pfs` test which will take a minute or so to run, but will stress your file system synchronization pretty well. It is also notorious for timing out, and usually it's because the pointer validation is too slow. This is not a bug, but a feature. Your pointer validation need to be fairly efficient, and not do a lot of unnecessary work for each buffer/string. `tests/dagjo` have `recursor-ng` that stress your `exec-wait-exit` solution. Do not be surprised if you have to go back and fix a bug that has gone unnoticed until now.

Finally, when you feel you have fixed all the bugs, make sure to run `pintos-check-forever -a` to run the test suite continuously. The `-a` flag is just to stop running the test suite if anything fails, so you can debug the problem. The rationale behind rerunning the test suite over and over again is because we can't actually be sure that your synchronization is decent by just running a test once, we need to run it several times, with hopefully slightly different timings each time so that any issues can crop up. A decent number of passing iterations would be 30-40.

4 Demonstration

For the demonstration you are expected to:

- Show that your solution passes the relevant tests in the test suite. There should not be any test that flips between FAIL and PASS between test runs.
- Be able to answer questions of similar character as the prep questions.
- Explain how your pointer validation works.
- Explain what files/functions/structures you have synchronized and why.