LINKÖPINGS UNIVERSITET

TDDE47/TDDE68: Concurrent programming and Operating Systems

Lab B: System call handler

2026-01-15

Dag Jönsson
IDA/SaS

1 Introduction

1.1 Goal

In this lab the goal for you is to build an understanding of what system calls are, how they work under the hood, and the reason behind why system calls are implemented in the way they are. You should also start building an understanding of what dual mode in the context of operating systems are, and what design choices are made because of this.

You will also implement a basic system call handler in Pintos, so that in the end you will have a limited, but fairly functional operating system.

1.2 Reading material

To help you achieve these goals, you should study the following material.

We want to reiterate the following from the previous lab:

Below you have some reading material from Stanford (upstream Pintos). Please do remember that you are taking a course at LiU, and any *instruction* given in the below documentation should be ignored, our instructions take precedence. These links should be treated only as documentation, i.e. ignore any statements like “You will implement this is project 1”.

- Lecture 1
- Course literature chapters:
  - 1.1-1.6: Introduction
  - 2.3-2.4: System calls, System Services
  - 3.1: Process Concept
  - 12.2.3: Interrupts
- Stanford Pintos:
  - Introduction Specifically 1.1.1
  - Project 1: Threads Specifically 2.1.1
  - Project 2: User programs Specifically 3.1.2, 3.1.4, 3.3.4, 3.5, 3.5.2
  - Reference Guide Specifically A.2.1, A.2.2, A.4, A.4.1, A.4.2
- Files:
  - `lib/user/syscall.[h,c]`
  - `lib/syscall-nr.h`
  - `userprog/syscall.c`
  - `struct intr_frame` in `threads/interrupt.h`
  - `filesys/filesys.h`
  - `filesys/file.[h,c]`
  - `threads/thread.h`

The **Files** should be studied in such a way that you know what functions and structures already exist, and have an idea of how they are used. There are usually a comment or two in the source code that you can, and should read, to understand that part of

Pintos. You are of course free to read any other file as you please, but these have been identified as the ones you will need to work with in this lab.

1.3 Preparatory question

To help build up your understanding of the necessary concepts, here are some preparatory questions. They are designed to help you understand what you need to do in Pintos, and will hopefully save you time if you answer these. You should be able to find the answer in material referenced in Section 1.2, or otherwise discuss with your lab assistant. If you get stuck on a question, feel free to skip over it and discuss it with your assistant when time permits.

Question 0: Basic operating systems

a) What is the point of an operating system?
b) What is user space?
c) What is kernel space?
d) What is a user program?
e) What is a software (internal) interrupt?
f) What (in Pintos) is the difference between a thread and process?

Question 1: System calls

a) What is the idea behind system calls?
b) How are system calls made from user space in Pintos? What functions are involved in user space?
c) What function in kernel space is responsible for handling/servicing a system call?
d) How many system calls are already defined in the Pintos user space?
e) Why can't the system calls just be normal functions? Why do they need to execute in kernel space?

Question 2: Interrupts

Pintos only make use one interrupt (`0x30`) for all system calls, so there is only one interrupt handler that handles system calls. With that in mind,

a) How is it possible to distinguish between different system calls made from within this handler?
b) Where are the arguments of a system call stored (if there are any) and how can the system call handler access them?

Question 3: Dual mode

a) Where in memory is the user space stack of a process located? Where is the kernel stack located?
b) What is the reason behind having two stacks instead of just one?
c) How can the kernel access user space memory? In particular, how is it made possible in interrupt handlers?

Question 4: Memory concerns

When a user program makes a system call like `open(filename)` , an address to a string containing the file name is provided as an argument.

a) In what memory (stack) is this string stored?
b) How can the kernel access it? Why?
c) Can you think of any potential problems with having the kernel accessing data through user space provided pointers?
d) In the case of `open`, user space is also expecting a value back (a return value), where would that value be stored so that the user program can access the return value?

Question 5: File system

a) What kind of high level file system operations does Pintos already support?
b) How is a file represented in Pintos?
c) What kind of file operations does Pintos already support?

Question 6: File descriptors

When a user program requests a file to be made ready to work with, they make a call to `open(filename)`, and get back a file id, known as a *file descriptor*, shortened to *fd*. These file descriptors are then use by the user program to reference a file for example when it want to read from it or write to it.

a) What kind of data type would be a good fit for a file descriptor?
b) Why do operating systems use file descriptors? Why not just return a pointer to the file?
c) How could the kernel keep track of what file a given file descriptor corresponds to, given that file descriptors are per process? How does the kernel keep track of the process?
d) How do we get access to the current thread?
e) What would be a simple way of creating the file descriptor?
f) Are there any special file descriptors already defined in Pintos? Which ones in that case? What are they used for?

2 Assignment: System call handler

Now that you have started to build an understanding of how a user program can communicate with the kernel, and the rationale behind the separation, we can start to think about the coding assignment.

You are not expected to write a file system, or implement interrupt handling, since almost everything you need is already implemented. You are expected to wire up the kernel side of system call handling, by writing code that do the following steps:

- Check what system call was made.
- “Extract” the correct number of arguments for the given system call.
- Orchestrate the calling of the functions that are needed to make the system call happen.
- Store any potential new state in the thread context.

- Return the expected value to the user program depending on outcome of the above.

You *will* need to create a solution to manage *file descriptors*, look back to your answers in Section 1.3.

The list of system calls you are expected to implement is found in Section 2.2.

You can find a list of requirements and limits in Section 2.3.

There is a test program you should use as a starting point, see Section 2.4.

2.1 Tips for an "easier" time through the lab series

Read before you code. Every year we have a group of students who are way to eager to start writing code, because they want to finish quickly. However, they end up spending *more* time fixing and debugging their code than the students who take their time reading and understanding the problem, and then create a solution.

Try to write your code in a structured way. The system call handler is something you will revisit throughout the lab series, so it will save you time in the long run if you try to maximize the readability and extendability of your solution. Prefer to write clear code above clever, short code. Your future self will thank you!

Also think of the Pintos project as *your* project now. If you want to add new functions to solve a problem, go right ahead. It's *your* project. But at the same time, for your own sake, don't change everything just because you can. The problems you are trying to solve might already be halfway solved, you just have to call the right functions.

Keep in mind that the instructions aren't telling you exactly how to solve the problems presented to you by design, because it's up to you to learn, test and research on your own. The assistants are there to help guide you, but will not solve the problem for you.

If you feel unsure if you are expected to do something or not, discuss it with your lab assistant before spending too much time on it to avoid unnecessary frustration.

Use git efficiently! A suggestion is to work on each lab in their separate branches, and once you've demonstrated (and fixed any potential issues spotted during demonstration) you merge your solution into `main` and hand in your solution based on the `main` branch. Creating a branch per lab makes it easy for you to "bookmark" your work for that lab, and makes it easier to start work on the next lab. If you need to fix anything spotted after handing in you can easily `checkout` the old branch, implement and test your fixes, and then making use of `merge` / `rebase` to get those changes into your new working branch.

Commit often, suggestion is to commit once you've implemented and tested a solution for a smaller problem (for example a specific syscall). If you run into an issue and you are unable to figure out why you have the issue, being able to go back to a working state and starting over is sometimes faster than debugging.

If you add any code outside of the `pintos/userprog` directory, you should wrap your code in the following:

```
#ifdef USERPROG
... YOUR CODE ...
#endif
```

This is to make sure that your changes doesn't affect the different modules in Pintos if we decide to compile without a user space (what `pintos/userprog` represents). For example, you can compile just the kernel, `pintos/threads`, and run tests on those parts specifically.

And finally, read through the assignment in full first, create a strategy (suggestion is to solve the 'easiest' problem first), and work methodically forward.

2.2 System calls

Read through the list before you start to code!

In this assignment, you need to implement the kernel side of the following system calls:

- `void sleep (int millis)`

  Makes the current process sleep for `millis` milliseconds. You can find some useful functions in `devices/timer.h`.

  Note: This system call does not exist in user space yet, you will need to add it in user space as well. This is the only exception, the rest are already wired up in user space.

- `void halt (void)`

  Shuts down the whole system. Use `shutdown_power_off()` for that, which is declared in `devices/shutdown.h`. This system call should **not** be used to terminate any potential programs you write yourself. I.e. `halt` does not replace `exit`.

- `bool create (char const *file, unsigned initial_size)`

  Creates a new file called `file`, with the initial size set to `initial_size`. Returns `true` if it was created successfully, `false` otherwise.

- `int open (char const *file)`

  Opens the file called `file`. Returns a non negative integer handle, called *file descriptor* (*fd*), or −1 if the file could not be opened. Each call to `open` should create a new fd if possible.

- `void close (int fd)`

  Closes the file corresponding to the given `fd`, freeing up any memory allocated in the kernel.

- `int write (int fd, void const *buffer, unsigned size)`

Writes `size` bytes from `buffer` into an already open file corresponding to `fd`. Returns the number of bytes actually written or −1 if the file could not be written to.

Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behaviour is to write as many bytes as possible up to end-of-file and return the actual number of written bytes, or −1 if no bytes could be written.

Tip: Implement write to `STDOUT` first, then tackle writing to files. In fact, you should implement writing to `STDOUT` before tackling files at all, since it will make it possible for user programs to use `printf`.

Hint: You can find some useful functions for `STDOUT` in `lib/kernel/stdio.h`.

- `int read (int fd, void const *buffer, unsigned size)`

  Reads `size` bytes from an already open file corresponding to `fd` into `buffer`. Returns the number of bytes actually read, or −1 if the file could not be read (due to a condition other than end of file). If reading from `STDIN`, the user should be able to see what they type while typing, just like you would expect from the terminal on any Linux machine. You do not need to implement erasing characters if *backspace* is pressed.

  Tip: Just like with write, implement reading from `STDIN` before tackling files. This is to make it possible for user programs to get input from the user.

  Hint: You can find some useful functions for `STDIN` in `devices/input.c`.

- `bool remove (char const *file_name)`

  Removes the file with the name `file_name`. Returns `true` if successful, `false` otherwise.

- `int filesize (int fd)`

  Return the file length of the file corresponding to `fd`.

- `void seek (int fd, unsigned position)`

  Sets the current position in the file corresponding to `fd` to `position`. If the position exceeds the file size, the position should be unchanged.

- `unsigned tell (int fd)`

  Returns the current position in the file corresponding to `fd`.

- `void exit (int status)`

  Terminates the current user program, passing the exit status to the kernel (you don't have to worry about the status for now, you can just store it in the `struct pi` for now).

  Remember to free all the resources (e.g. close all open files) that are associated with the process and wont be needed anymore. If you need to add some code

to accomplish this, you should consider that Pintos itself calls `thread_exit()` to destroy and cleanup threads, that function in turn calls `process_cleanup()` . Think about where you should place your cleanup code.

You will revisit `exit` in later labs.

2.3 Requirements and limits

- Your solution should be written in C.
- Every variable should be initialized to a known good value before using it.
- A single process should be able to open 32 concurrent files. If we have two processes running, they should individually be able to have 32 files open.
- There should be no memory leaks. Consider that this is an operating system you are building, which means there is no safety net. If you lose track of some memory, that memory is lost until you reboot the machine.
- You do not have to worry about more than 1 thread/process for now, which means you don't have to think about synchronization just yet. You should however avoid creating solutions that only works with 1 thread. Avoid singleton constructions.
- Any *pointer* passed by a user program is valid, for now. You do not have to implement any pointer validation yet. I.e, any pointer given by the user program can be safely dereferenced. You can also expect any strings passed by the user program to be valid C-strings.
- You *will* need to check the validity of the *values* given by user programs though. If a system calls expects an integer value between 0 and 34, you should check that before using the values. Invalid values should result in a "fail"-result be sent back, if a return is expected.

2.4 Testing

To test your solution, you can run the `examples/basic_syscall.c` program. See the comment at the top of the file on how to run it. When you have a working system call handler the test should **not** print out any warnings or errors, and finish with the following printout: `If you got this far, you've passed all the tests this file offers!`

If you wish to write your own programs to test smaller problems or specific cases, feel free to do so. Make sure to place them in `pintos/examples` , and you will have to modify the `examples/Makefile` to make your program compile. If you study the files you should be able to figure out how to do it. You may also want to modify `examples/.gitignore` to have git ignore the resulting binary file.

Some things to keep in mind if you decide to write your own programs:

- They should be written in C.

- Floating point operations cannot be used because Pintos does not save the corresponding information during process switch.
- Pintos user programs can use only those system calls which you will implement in this and the following labs.
- The necessary system call for dynamic memory allocation, e.g. with `malloc` is not implemented and it will not be implemented in these labs. Hence, you cannot use dynamic data structures inside a user program. Bear in mind that you can still use `malloc` in kernel code!

3 Demonstration

For the demonstration you are expected to:

- Be able to answer questions of similar character as the prep questions.
- Be able to discuss your solution, why you made the decisions you made.
- Show that the given example program works as expected.
- Argue for why your solution hasn't introduced any memory leaks.

4 Quick guide to `make`

Throughout this lab series, you will be using `make` to build Pintos and example programs, and later, run a test suite against your solutions. We do not have the time to cover how `make` works in any detail, and especially not how it's being used in the Pintos project. However, there are some useful things to know.

- `make` is fairly smart, and will only compile files that have actually changed. This is generally a good thing since this saves you time in the long run!
- Sometimes `make` makes mistakes, and doesn't catch that a file has been updated, and thus doesn't compile it. The easiest way to remedy this is to run `make clean`. This will delete all the built files and next time you run `make -j8` it will rebuild everything from scratch. If you start seeing weird behaviour after making changes, check to make sure that your code actually compiled without errors, if it did, try a `make clean; make -j8` to see if it fixes the issue. If you are still seeing weird behaviour, you've probably introduced a bug in Pintos, and you should debug that.
- To make Pintos compile faster, you should pass the `-j8` flag to `make`. This tells `make` to make use of more threads, 8 in fact. You can play around with the number of threads, but usually there is little point in going above the number of cores (threads) that your CPU has. You can also omit the number, and just pass `-j` which will cause `make` to not set any limit on the number of threads, which might be quite a lot. Don't be surprised if the computer fans start spinning faster.

- You can run `make` in a different directory than the one you are currently standing in by passing the `-C <path>` flag. For example: standing in `pintos/userprog`, we can have `make` compile the examples programs by running `make -j8 -C ../examples`.

If you find `make` interesting, you can read more here: https://www.gnu.org/software/make/manual/make.html or https://web.mit.edu/gnu/doc/html/make_1.html

It's not expected, or even a part of this course, that you will understand `make`, so only read if you really are interested. It will not help you in the lab series to understand `make`, it's enough to know how to make it build Pintos.