TDDE47/TDDE68
Lab 2: Basic system calls

Dag Jönsson

February 7, 2024

1 Goal

In this assignment you are supposed to learn about user programs and system calls. The main goal is to clearly understand systems calls in user programs by implementing a set of system calls in Pintos.

2 Overview

This assignment covers:

- An introduction to user programs in Pintos.
- System calls.
- Layout of the user memory and the kernel memory in Pintos.
- Input/output management.

User Programs

An operating system must be able to run user programs, each with their own memory space. Starting from this assignment you will be using real user land programs and run them under Pintos. Here are the features and limitations of Pintos user programs:

- They should be written in C.
- Floating point operations cannot be used because Pintos does not save the corresponding information during process switch.
- Multithreaded processes are not supported, therefore we will use the words thread and process interchangeably (although it might not be the case for other operating systems).

- 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 terms of these labs. Hence, you cannot use dynamic data structures inside a user program. (Bear in mind that you will still use `malloc` in kernel code!)
- A user program needs to be copied to simulated disk used by Pintos.

System Calls

The communication between the user program and the kernel is done by system calls. System calls can be seen as special functions, called from the user program and performed by the kernel. Usually computers use interrupts to accomplish that switch from user code to system code, and so does the x86-machine.

When the programmer wants to invoke a system call in a user program, he or she calls one of the functions defined in `lib/user/syscall.h`. Those functions are implemented in `lib/user/syscall.c` and do nothing except placing function's arguments with the respective system call number on the stack and raising an internal (software) interrupt (0x30). The raised interrupt makes the processor temporarily stop the user program, change from the user to the system mode and jump to the interrupt handler `syscall_handler` defined in the kernel (`userprog/syscall.c`).

The interrupt handler is the entrance to the kernel. All communications with the kernel via system calls must go through it. The interrupt handler must then determine what system call it is and handle it properly. Look into `threads/interrupt.[h|c]` and clearly understand the main structures and main functions of the interrupt handler in Pintos. Pay attention especially on `intr_frame` structure and stack pointer into it. You will need to use this stack pointer in order to access system call arguments.

Note, that there are two different pairs of files with the same names: `lib/user/syscall.[h|c]` and `userprog/syscall.[h|c]`.

The first pair is visible from the user program side and is merely a wrapper to raise an interrupt, while the second one is having the real implementation of the system calls. Currently, the interrupt handler contains no useful code and forces the calling program to exit.

In this assignment you will write the OS code to implement a number of system calls as well as some simple user programs to test your implementation. You can find names of system calls in Pintos by looking into `lib/syscall-nr.h`.

Examples of system calls are:

- create - creates a file.
- open - opens a file.

- close - closes a file.
- read - reads from a file or the console (the keyboard).
- write - writes to a file or the console (the monitor).
- halt - halts the processor.
- sleep - pause the execution of a process for given time.
- exit - terminates a program and deallocates resources occupied by the program, for example, closes all files opened by the program.
- seek - sets position in a file for read and write system calls.
- tell - returns the position in a file.
- filesize - returns the file size.
- remove - removes a file from the file system.

The following are also system calls, which you will continue implementing in the following labs where you will have to handle several processes in memory at the same time.

- exec - Loads a program into memory and executes it in its own thread or process.
- wait - Waits for a child process to exit and returns its exit status.

3 Preparation

3.1 System Calls

Get familiar with the code in `userprog/syscall.[h|c]`, `threads/interrupt.[h|c]`, `lib/syscall-nr.h`, and `lib/user/syscall.[h|c]` files. The latter files contain some assembler code, which just puts arguments and the respective system call number to registers and raises the interrupt 0x30. You should get the clear image of the system call architecture in Pintos after studying those files and reading the documentation:

- **Interrupt Handling**
- **System Call Details**
- **Accessing User Memory Section**

Preparatory question 0:
(a) What is the idea behind system calls?

(b) Why cannot the code of the system calls be available simply as a library to user processes? (Tip: you may look at an article in Wikipedia for the answer).

Preparatory question 1:
Pintos uses one interrupt (0x30) for all system calls, so there is only one interrupt handler syscall_handler to be called for different system calls.

(a) How it is possible to distinguish in syscall_handler which system call it is?

(b) Where are the arguments of a system call stored (if there are any) and how can you access them?

Memory Issues and Argument Passing
Read about Pintos virtual memory layout in the corresponding section of the Pintos documentation.

Preparatory question 2:
(a) Where are the user-mode stack and the kernel-mode stack of a process located?

(b) How can you address the user-mode stack in the kernel code, particularly, in an interrupt handler?

(c) What is the reason of having two stacks instead of one?

Preparatory question 3:
When a user program executes a system call like `open()`, an address to a string containing the file name is provided as an argument.
(a) In which memory is this string stored, and (b) how can we access it in the kernel code?

(c) Specify the situations when accessing the data via that pointer can lead to problems.

(d) What can be done about it?

3.2 Making user programs

You already have a number of simple user programs in `examples`. You can compile them by issuing `make -j` in that directory. Modify `examples/Makefile` whenever you wish to compile your own user programs (you will find instructions inside the `Makefile`), it's also a good idea to modify the `examples/.gitignore` to ignore the resulting binary. Write all your test programs in the `examples` directory.

Simulated Disk

Before running user programs first you have to create a simulated disk, format it and copy user programs there.

Go to `userprog` and run `make -j`. Then, go to `userprog/build` and issue the command:

```
pintos-mkdisk filesys.dsk --filesys-size=2
```

This will create a file `filesys.dsk` with a 2MB simulated disk in the directory. Format the disk with the command:

```
pintos -- -f -q
```

Copy Pintos user programs to the simulated disk with the command: (Remember to copy already compiled programs (binaries), not the source code files.)

```
pintos -p programname -a newname -- -q
```

Most probably you will copy user program from the `examples` directory. Then the command will look like this:

```
pintos -p ../../examples/programname -a programname -- -q
```

If you need to copy a file from the simulated disk, use the command:

```
pintos -g filename -- -q
```

or

```
pintos -g filename -a newname -- -q
```

As you see, the only difference is in the switch: -p is used to put files to the disk and -g to get a file from the disk. If you need to run a user program that has been already copied:

```
pintos -- run programname
```

Furthermore you can also list files with `ls`, remove files with `rm` and print contents of a file with `cat`. Several of these commands can be run on the same line, e.g.:

```
pintos -- ls rm a rm b ls
```

will list files, delete the files a and b and then list files again.

3.3 File System

The current distribution contains a very simple but complete file system. Get acquainted with file system interface (which is available only in the kernel code!) in the `filesys/filesys.[h|c]` files. You do not have to modify that code in this lab, so it is enough if you have a look at the available functions and read their documentation. The same concerns the files `filesys/file.[h|c]`. It is good to look into other files in the filesys directory, although it is not strictly necessary to complete this lab.

Read about file system limitations in the corresponding section of the Pintos documentation. Although the access to files are not synchronized in the current implementation of the file system, you should not worry about it at this point.

You can read more about the filesystem at the **3.1.2 Using the File System** section.
Preparatory question 4:
When a file is opened, a file id (also know as file descriptor) is returned to the user program, which is used to refer to a specific opened file when doing file operations. Explain how to generate file identifiers and map them to `struct file` pointers that are created in the kernel.

4 Assignment

The assignment is to implement the following system calls:
(Suggested order of implementation is the order of this list)

`void sleep(int millis)`
Makes the current process sleep for `millis` milliseconds. You can find some useful functions in `devices/timer.h`. This system call will be used later in the lab series.
Note: This system call does not currently exist in userspace, so you will also have to modify `lib/user/syscall.[h|c]` and `lib/syscall-nr.h` to make it possible for user programs to call `sleep()`.

`void halt (void)`
Shuts down the whole system. Use `shutdown_power_off()` for that (declared in `devices/shutdown.h`). Do not use this system call to terminate your user program!

`bool create (const char *file, unsigned initial_size)`
Creates a new file called file initially `initial_size` bytes in size. Returns true if successful, false otherwise.

`int open (const char *file)`
Opens the file called `file`. Returns a non negative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened. File descriptors numbered 0 and 1 are reserved for the console: fd 0 (`STDIN_FILENO`) is standard input, fd 1 (`STDOUT_FILENO`) is standard output. You do not have to open the standard input and output before using them.

Each process has an independent set of file descriptors. A user program should be able to have up to 128 files open at the same time. File descriptors are not inherited by child processes. Note that Pintos does not manage file descriptors yet, you need to implement this feature yourself.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to close and they do not share a file position.

If you add code outside the `userprog` directory, protect it with the following (to make lab 3 work later on):

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

`void close (int fd)`
Closes file descriptor fd, freeing up any memory allocated in the kernel.

`bool remove (const char *file_name)`
Removes the file with the name `file_name`. Returns true if successful, false otherwise.

`void seek (int fd, unsigned position)`
Sets the current position in the open file fd to position. If the position exceeds the file size, it should be set to the end of file.

`unsigned tell (int fd)`
Returns the current position in the open file fd.

`int filesize (int fd)`
Returns the file size of the open file fd.

`int write (int fd, const void *buffer, unsigned size)`
Writes size bytes from buffer to the open file fd. Returns the number of bytes actually written or -1 if the file could not be written.

Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written or -1 if no bytes could be written at all.
Fd 1 writes to standard out, which can be done with the `putbuf()` function (declared in `lib/kernel/stdio.h`) .

`int read (int fd, void *buffer, unsigned size)`
Reads size bytes from the file open as 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).
Fd 0 reads from the keyboard using `input_getc()` (defined in `devices/input.h`). The user should be able to see what characters have been entered.

`void exit (int status)`
Terminates the current user program, returning status to the kernel (you don't have to worry about status for Lab 2, it will be covered in the following labs). Conventionally, a status of 0 indicates success and nonzero values indicate errors.
Remember to free all the resources (e.g., closing the files of the process) that will be not needed anymore. You may free the resources within the `process_exit()` function as Pintos will call it (through the use of `thread_exit()`) to kill the thread in some cases (e.g., the thread attempted to access invalid memory).

This system call will be improved in the following labs.

Tip: The system call name and the arguments you can get from the stack with a stack pointer. Some pointer arithmetics will be useful to go up and down in the stack.

In all of the above you need to check that the user is giving the system reasonable values. If the values aren't reasonable, and the call is expected to return something, return the equivalent of "fail". Undefined behavior is not acceptable.

The system call handler function will be used a lot in this and the following assignments so you should think about structuring your code in an organized and clear way to make reading it and performing future expansion easier.
You may need to implement some additional functions and data structures that are not specified here, and which you must think of yourself. This is part of the

assignment.

4.1 What you do not need think about (yet)

For now, you may assume that only one user program can run at a time (indeed, the `exec` system call is not implemented in this lab). Therefore, synchronization of any thread-unsafe code can be delayed until the next labs.

Another simplification you may do in this laboration is to assume that all pointers passed to the kernel from the user program are valid, i.e. do not worry about page faults.

Do not worry, we will fix these two limitations of your operating system in the next lab.

5 Testing

An example test program for testing your system calls is `examples/lab2test.c`.

Often when you encounter a bug in your operating system, the best way to analyze it is to make a minimal user program which activates this bug. So you will probably also need to make your own user programs. You can check in the examples directory how other programs are created.

Note: Not all syscalls are tested in the given example test. `seek, tell, remove, filesize, sleep` are not tested, and you should write test yourself to make sure that those work as intended. You are of course free to extend the given example file.

6 Helpful Information

Code directory: `userprog`, `lib/`, `lib/kernel`
Textbook chapters:

- Chapter 2.3: System Calls
- Chapter 9.3: Paging

Documentation:
Pintos documentation related to Project 2

(Always remember that the TDDE47/TDDE68 lab instructions have higher precedence)

7 Acknowledgement

Parts of this document contains material from the TDIU16 course at LiU, previous lab instructions found on the course web page, or from previous lab instructions written by Felipe Boeira.