TDDB68
Lesson 1

Felipe Boeira

Contents

Administration

Webreg:

- Register there! If you are a returning student, there is a group there for that.

General information about the labs

- The labs are based on Pintos: an educational OS (developed at Stanford University)
- Pintos is written in C and well documented
- The labs are about adding functionality to Pintos
- Pintos is around 7 500 lines of code (LOC)
- Linux Kernel has around 25 million LOC
- How many LOC have your programs had so far?

General information about the labs

- The complicated part of the labs is understanding what you should do: you do not need to write a lot of code
- In other words, try to read a lot!
- If you are unsure about C programming, take some time to review it. A proper understanding of C will save you a lot of time debugging!
- You **must** work on **non-scheduled** time as well! These labs tend to be time-consuming

General information about the labs

- If you pass the labs on time by **March 15**, you earn **4 bonus points** on the exam (~20% towards passing it)
- This offer is only available to new students
- The final deadline is **March 29**
- The procedure of handing in the labs…

Lab 0

- Understanding of lists
- Pintos setup
- Debugging
- Demo at the end of the class

Lab 1

- The first real lab
- Single user process
- Implement a number of system calls:
  - Reading from and writing to the console
  - Creating, reading from and writing to files
  - Exit a process and halt the machine
- Tends to take some time since you have to familiarise yourself with Pintos

Lab 2

- Multiple user processes
- Avoid busy waiting in sleep
- Sleep delays execution of the calling process by a given value
- Synchronisation is now necessary
- This lab tends to take the least amount of time

Lab 3

- Multiple user processes
- Implement the system call Exec
    - Exec: Let processes start execute programs in a child process
- Create parent-child relationship
- This lab together with the next two tend to take **a lot of time** to finish!

Lab 4

- Programs cannot have arguments yet - fix it!
- Setup the user space program stack with arguments according to the x86 convention
- This lab requires careful understanding of memory layout and pointer arithmetic

Lab 5

- Multiple user processes
- Implement the system call Wait
  - Wait: Let processes wait for their children to finish executing
- Use or extend the parent-child relationship you have already created

Lab 6

- If several processes write to the same file, they will overwrite each other's content arbitrarily
- Make sure that no order of system call, or internal calls, leads to an invalid state (open, close, write, read, and so on)
- Synchronise the file system! (readers writers algorithm, and more)
- This lab tends to take about as much time as lab 1

Pointer Arithmetic

Let **char *p** be some pointer. These are some common confusions:

- **p + 40** and **((uint32_t *) p) + 10** points to the same address. Addition takes the size of the pointer type into account
- ***(p + 40)** reads 1 byte, and ***((uint32_t *) p + 10)** reads 4 bytes! How many bytes are read depends on the type of the pointer!
- ***((uint32_t *) p + 10)** and **((uint32_t *) p)[10]** are equivalent! The latter is just syntactic sugar for the former

A closer look at Lab 1

- Only one user process at a time - no concurrency!
- Suppose a user process wants to open a file, then it:

Already implemented!

1. Calls the syscall function `int open(const char *file)`;
2. The function `open` puts the arguments on the stack, together with an interrupt no. and a syscall no.;
3. Produces an interrupt to switch to kernel mode;
4. The interrupt handler then looks at the interrupt no. and delegates it to the appropriate subhandler, in this case, the syscall handler;

A closer look at Lab 1

This is the assignment!

- The syscall handler then (in kernel mode):
    1. Reads the syscall no. to decide the type of syscall (write, read, open, close, and so on).
    2. Based on the type, the handler reads the correct number of arguments from the stack, and performs the syscall.
- For example, the handler does not get the arguments to this syscall directly: `bool create(const char *file, unsigned size)`
  To get them you have to read them from the stack: f->esp
- Note that the actual string is NOT on the stack. The stack only has a pointer to the first character of the string.
- To return the result of the syscall, set this register: f->eax

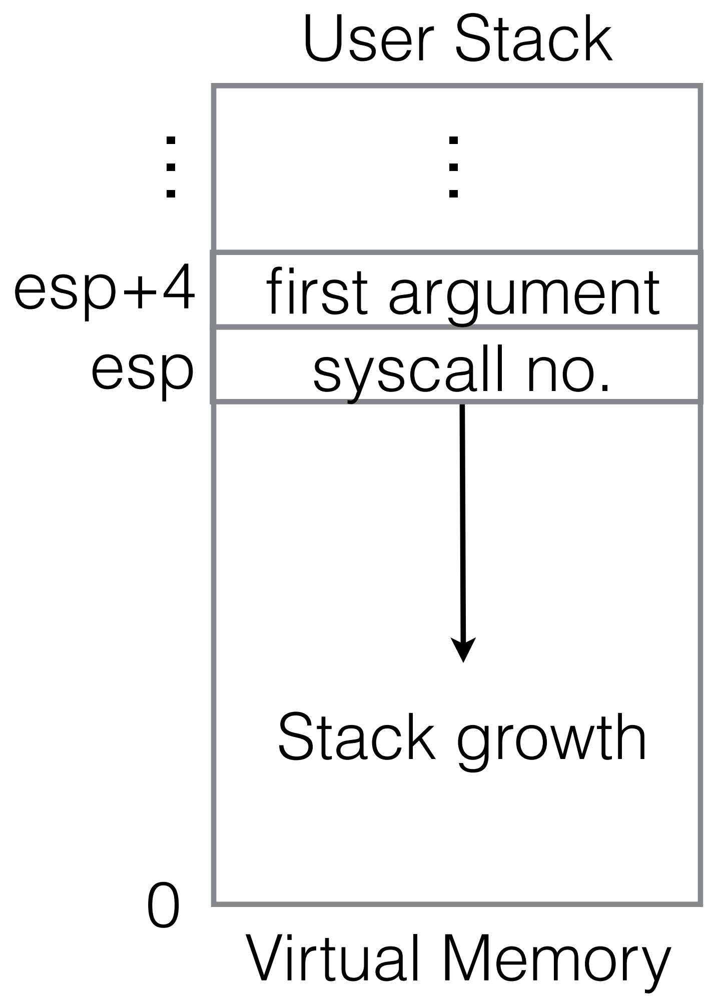
A closer look at Lab 1

Files to study:

- pintos/src/lib/user/syscall.[h|c] - The syscall wrapper
- userprog/syscall.[h|c] - Implement syscalls here!
- threads/thread.[h|c] - Implement something here!
- threads/interrupt.[h|c] - Important structures
- lib/syscall-nr.h - Syscall numbers
- filesys/filesys.[h|c] - Pintos file system

A closer look at Lab 1

- Currently, the syscall handler only prints a message stating that it was called
- The handler must do the things that we discussed earlier
- f->esp is the stack of the calling process
- The syscall number is at the top, then the arguments
- Every syscall has its own syscall number: use it to decide the number of arguments



A closer look at Lab 1

File descriptors (FD)

- A FD is a non-negative integer that represents abstract input/output resources
- Input/output resources are, for example, files, consoles, network sockets, and so on
- The user processes only knows about FDs, and the OS knows what concrete resource it represents
- In the labs, FD 0 and 1 are reserved for the console

A closer look at Lab 1

What to think about when implementing:

- **create** - Create a file. Return true if a file was created, false otherwise. Hint: Use already implemented functions.
- **open** - Open a file. Return a FD to the user process. How do we decide on a FD and how do we map it to an opened file? Every process has its *own* collection of opened files. FD values 0 and 1 are reserved for the console. Hint: Modify the struct thread to contain every opened file.
- **close** - Close the file associated with the given FD. Disassociate the FD with the file. Hint: Use already implemented functions.
- **exit** - Kill the process. Deallocate all of its resources (eg. files). We revisit this syscall in Lab 5. Hint: Free resources in `thread_exit`.

A closer look at Lab 1

What to think about when implementing:

- **read** - Read the file associated with the given FD. The user process gives a buffer (a piece of memory) in which the read bytes are written to. Return the number of read bytes. Hint: Use already implemented functions. Use `input_getc` to read from the console.

- **write** - Write to the file associated with the given FD. The user process gives a buffer with the content that should be written. Return the number of written bytes. Hint: Use already implemented functions. Use `putbuf` to write to the console (check `lib/kernel/stdio.h` and `lib/kernel/console.c`).

- **halt** - Shutdown the machine (halts the processor). Hint: Use already implemented functions.

A closer look at Lab 1

What to think about when implementing all of them:

- Every user process should be able to have at least 128 files opened at the same time
- It is *dangerous* to assume that the arguments are valid! We will revisit this topic.
- Special cases: What happens if the arguments of the syscall are invalid? Such as NULL pointers, invalid buffer size, FDs with no associated file, and if the process has opened too many files

A closer look at Lab 1

Frequently Asked Questions:

- Use the function `thread_current()` to get the thread of the calling process

- The function `filesys_open(char *)` opens a file, and the function `file_close(file *)` closes it

- The function `init_thread(...)` initialises every thread, whilst the function `thread_init(...)` initialises the thread module (once, when Pintos starts up). If you need to do some initialisation for every thread, modify the former function (at the end of it)

A closer look at Lab 1

- Run `lab1test2` to test your code. It will:
  - Create files
  - Open files
  - Read and write from the console
  - Try to use bad FDs
- Gotchas:
  - Remove all the files on the virtual hard drive before running it again:
    ```
    pintos --qemu -- rm test0 rm test1 rm test2
    ```
- Passing `lab1test2` does NOT mean that you have finished the lab!
  You must ensure that there are no special cases

System call overview

- Number of system calls to implement: 13 system calls
- Linux: Around 320 system calls
- Windows: More than 1000 system calls

Debugging

- Read Appendix E: Debugging tools in the documentation
- `ASSERT(p != NULL)`
- If you get "Kernel Panic", then try the `backtrace` tool
- `free` sets bytes to `0xcc`: If you see bytes with this value, then something likely freed the memory
- Use versioning, commit frequently! Rollback if necessary!

Debugging

Backtrace example:

- If you were to get this:
  ```
  Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
  0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
  ```
- Then type this:
  ```
  backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
  0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
  ```
- To get this:
  ```
  0xc0106eff: debug_panic (lib/debug.c:86)
  0xc01102fb: file_seek (filesys/file.c:405)
  0xc010dc22: seek (userprog/syscall.c:744)
  0xc010cf67: syscall_handler (userprog/syscall.c:444)
  0xc0102319: intr_handler (threads/interrupt.c:334)
  0xc010325a: intr_entry (threads/intr-stubs.S:38)
  ...
  ```

Questions?