Software Systems

Day 17 – Processes and System Calls

SQL Injection

- Head First C Chapter 9 mentions that you can pass in a string to an exec command and cause arbitrary other commands to be run.
- This is called an injection attack.
- A well-known type of injection attack can be done with SQL, a database query language used widely on the Web.
- https://www.youtube.com/watch?v=_jKylhJtPmI

Agenda

- Project 2 Preview
- Processes
- System Calls
- strace
- fork() and exec()

Project 2 Preview

- Project 2 is similar to Project 1, but you should explore an area of computer systems that you find interesting.
- As with the last project, you need to build a significant piece of software that helps you explore your topic of choice.
- The project can be in C or C++.
  - Beware: C++ has more features and a nicer standard library, but is *far* more complex to use well.
- Same assessments/rubrics as Project 1.

Project 2 Preview

- Some sample previous projects implemented/explored:
  - Bloom filters (a probabilistic data structure)
  - Reverse-engineering software (Ghidra)
  - A webcam-controlled game using OpenCV
  - A simple blockchain network
  - Most of a Game Boy emulator
  - A simple version control system
  - A digital synthesizer
  - A small, limited C compiler
  - A keylogger

Project 2 Preview

- By the end of next week, you should have picked and registered your team on Canvas
- Teams need to be 3-4 students.
- Anyone not in a team by the deadline will be randomly placed into a team.
- We'll assign project mentors after the proposal.

Processes

- Head First C only mentions them in passing, but a process is essentially a software representation of a running program.
- A process contains:
    - The program code (as a series of machine instructions)
    - Data (both static data, like constants, and dynamic data on the stack/heap)
    - Statuses of disk and network I/O operations
    - Hardware state (e.g., CPU register contents)
- A process also has a numeric identifier, and (on Linux) information made available in the `/proc` directory.

Processes

- A process is an abstraction, like many things in UNIX/Linux.
- The OS manages processes, and handles tasks like:
  - Interrupting and resuming processes to run multiple programs "at once."
  - Managing memory so each process thinks it has the whole address space.
  - Mediating access to hardware such as input devices, networking, disk, etc.

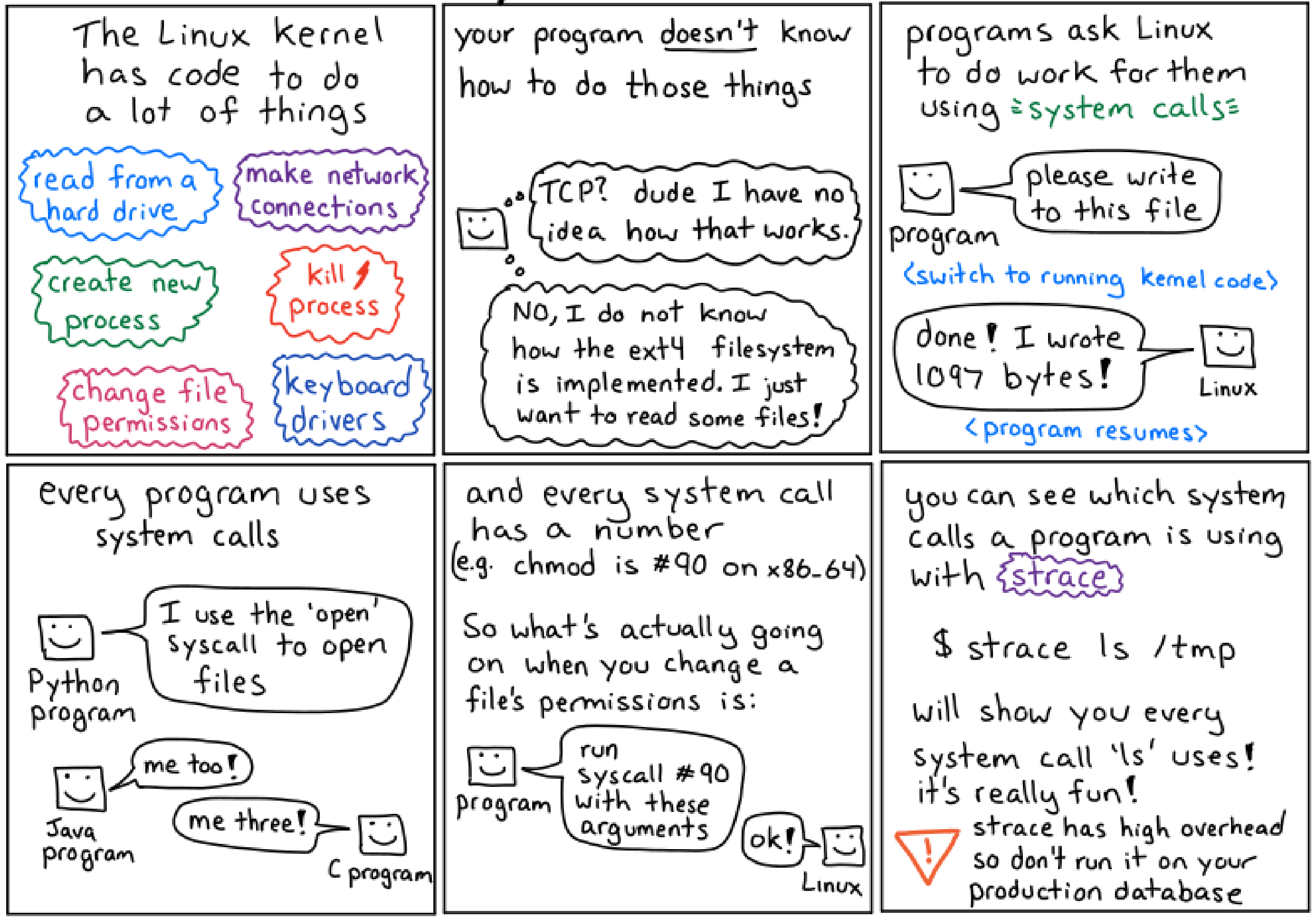
Processes

- Exercise:
    - Use `ps -e` to see the processes running on your machine, along with their identifiers.
    - What is process `1`?
    - What interesting processes are running that you weren't aware of?
    - Use `lsof -p <PID>` to see the files that the OS has opened for a given process.
    - Are there interesting files that a process has open? (Big programs like browsers and IDEs can be interesting.)

System Calls

- The OS mediates requests from processes for things like memory, hardware, etc.
- The part of the OS that manages the hardware is called the kernel.
- Programs make these requests through system calls, functions used to talk to the kernel.

system calls



The Linux kernel has code to do a lot of things

read from a hard drive
make network connections
create new process
kill process
change file permissions
keyboard drivers

your program doesn't know how to do those things

TCP? dude I have no idea how that works.
NO, I do not know how the ext4 filesystem is implemented. I just want to read some files!

programs ask Linux to do work for them using system calls

please write to this file
program
<switch to running kernel code>
done! I wrote 1097 bytes!
Linux
<program resumes>

every program uses system calls

I use the 'open' syscall to open files
Python program
me too!
Java program
me three!
C program

and every system call has a number (e.g. chmod is #90 on x86_64)

So what's actually going on when you change a file's permissions is:

run syscall #90 with these arguments
program
ok!
Linux

you can see which system calls a program is using with strace

```
$ strace ls /tmp
```

will show you every system call 'ls' uses! it's really fun!

⚠ strace has high overhead so don't run it on your production database

JULIA EVANS
@børk

System Calls

- As a note, Head First C calls `system()` a system call, but it's not – it's a function in the C standard library.
- It spawns a shell, runs your command there, and prints the output.
- In general, if you want to see information on a system call, you can look at the relevant manpage: `man 2 <name>`
  - Section 2 of the manual is where system calls are documented.
  - You can see the sections of the manual with `man man`.

System Calls

- So what does a typical program need to do?
- Probably load some libraries (stdio.h or stdlib.h), open some files, etc.
- Can we find out what system calls a program is making?

Tracing System Calls with strace

- Generally, you can view all of the system calls a program is making with the strace utility.
- This is Linux only; macOS may allow dtruss/dtrace or similar programs, but they may require turning off SIP.
  - Doing these operations in a Linux VM may help.
- Talk on strace: https://www.youtube.com/watch?v=4pEHfGKB-OE
- Slides (in Markdown): https://bit.ly/3zdLFXK
- System Calls zine: https://wizardzines.com/comics/syscalls/

Tracing System Calls with strace

- Exercise:
  - Run strace on a few programs (examples could be ls, echo, which, git, or small programs you've written in C).
    - Is the number of syscalls more or less than you expect?
    - What interesting syscalls does the program make? (Look some up in their manpages.)
  - Does strace produce the same output each time? Why or why not?
  - In the class sessions folder for today, there is a file called switchboard.c. Call system there with a short command. Does the output in strace match what you get running the command directly?
  - Call `echo $0` in the `system()` call. What shell is being spawned to execute your command?

Some Problems with system()

- The system() function is simple but limited:
  - The command goes directly to your default shell.
  - You have to use your OS's current configuration.
  - You can't pass input or extract output.
- We'll talk about the input/output next time.
- But how can you run a command with more options?

The exec() System Call

- We talk about exec(), but it's actually a family of functions: execl, execlp, execle, execv, execvp, execve, and execvpe.
- Only execve is actually a syscall (check the manpage).
- With this family of functions, you can specify:
    - How command-line arguments are passed
    - Whether you search along your PATH environment variable
    - What environment variables you pass to the program

The exec() System Call

- A quick example:
  `execlp("echo", "echo", "Hello world", NULL);`
- `"echo"` needs to be passed twice – once for the command itself, and once for the list of arguments.
  - By convention, the first argument is always the name of the program.
- exec returns –1 if something went wrong, and sets `errno`.
- You can print out the specific error like this:
  `fprintf(stderr, "exec error: %s\n", strerror(errno));`

The exec() System Call

- Exercise:
  - Look at the spoon.c program in today's exercise folder.
  - What do you expect to see printed?
  - Now run the program. Does it match what you thought? Why or why not?

The exec() System Call

- Be aware that exec() completely replaces the current process image.
- That means the program, memory, and registers are all replaced.
- It also means that if an exec() call is successful, it never returns to the original program, because that process is gone.
- So how do we run more than one exec() call in a program?

The fork() System Call

- The fork() system call is often used in conjunction with exec().
- It makes a copy of the current process image, in a new process with its own process ID.
- The original process is called the parent, and the new one is called the child.
- Weirdly, that means that fork returns twice (with type `pid_t`):
  - In the parent, it returns the process ID of the child.
  - In the child, it returns 0.
  - If there's an error, it returns –1 instead.

The fork() System Call

- This means that you can handle a fork with a switch, like this:

```
pid_t pid = fork();
switch(pid) {
  case -1:
    // Handle error.
  case 0:
    // Child process does its thing.
  default:
    // Parent process does its thing.
}
```

The fork() System Call

- So how is fork() used with exec()?
  - Make the child process call exec(), which replaces just the child process.
  - The parent process is left intact.
  - The parent can then fork more child processes, and/or communicate with the children.
- Communication can be used for I/O (which we'll see next time).

The fork() System Call

- Exercise:
  - Implement fork.c so it immediately forks. In the parent, print the PID of the child, and in the child, print "Hello world!". No need to handle errors.
  - Implement greet.c in a similar way, but use exec(). In the child, run `echo Hello`, and in the parent, run `echo Goodbye`. What happens?
  - Modify greet.c to run `echo Goodbye` in another child process instead of in the parent. Is the output different, and why or why not?