SYS2/CCCP: Interacting with the OS

For Java and Python Programmers

Dr David Griffin

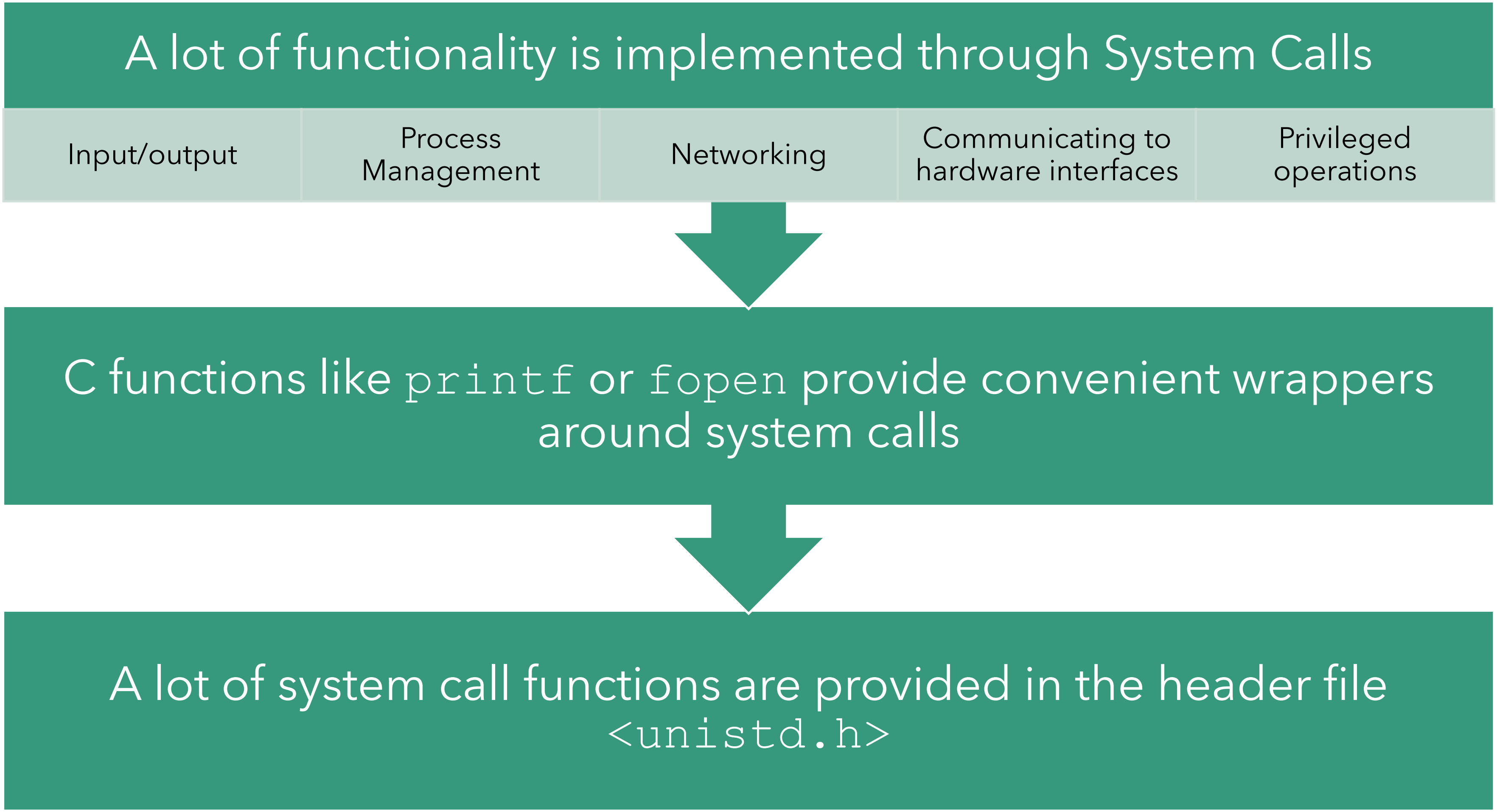
How do things get done?

- It doesn’t make sense for a programming language to implement all functionality.
- For example, C doesn’t know how to write a file.
  - The C code might be compiled on Windows or Linux, which have different rules.
  - Other programming languages exist, and these will also want to write files. Why should the code be duplicated?
- C doesn’t even know how to print things to a terminal.

Operating Systems

- Instead, Operating Systems provide basic functionality.
  - This can be shared across all programs on the system.
  - Programs ‘ask’ the operating system to do things for them.
  - Programming languages provide functions in libraries to bridge between what the operating system provides and what a programmer expects.
- This basic functionality is provided through **System Calls**.

System Calls in C on Linux



A lot of functionality is implemented through System Calls

Input/output	Process Management	Networking	Communicating to hardware interfaces	Privileged operations

C functions like `printf` or `fopen` provide convenient wrappers around system calls

A lot of system call functions are provided in the header file `<unistd.h>`

Using System Calls Natively

If you need to interact with something that's a system call but not a native function, GCC provides the `syscall` function in `<unistd.h>`

As well as a bunch of useful defines in `<sys/syscall.h>`

- Although if you try to open this file, you'll see it's more of an index to a bunch of other files…

So there's nothing to stop you from using `syscall(SYS_getpid);` to get the current running process ID.

- Other than it being more clunky than `get_pid();`

Seeing what System Calls are being Made

Linux has the `strace` utility

`strace` lets you run a command and see all the system calls it makes

```
strace hello_world # see the
system calls hello_world makes
in its printf call
```

`strace` is also useful for debugging any rogue system calls

Especially if you're going to try deploying to another operating system, that maybe doesn't implement all system calls on Linux

Using strace

One problem with strace – it prints its output to stderr

If you print print output in your program, then it’s going to conflict with strace’s output

Use pipes to disentangle the output

```
strace my_program 2> strace_log.txt # redirect straces output on stderr to strace_log.txt
```

```
strace my_program > /dev/null # throw away output from your program, keep straces output
```

Using strace

- Strace will give you a lot of output
- Don’t be intimidated! Most of it will be things to do with setting up the program.
- Instead, look for something you recognise. Maybe the input to a function.

strace_log.txt (beginning)

```
execve("./t", ["./t"], 0x7ffcfb723fb0 /* 51 vars */) = 0

brk(NULL)                       = 0x55e283e2d000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffca52a1d70) = -1 EINVAL (Invalid
argument)

mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2ab261b000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=77597, ...},
AT_EMPTY_PATH) = 0

mmap(NULL, 77597, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7f2ab2608000
```

Using strace

- Strace will give you a lot of output
- Don’t be intimidated! Most of it will be things to do with setting up the program.
- Instead, look for something you recognise. Maybe the input to a function.

strace_log.txt (end)

```
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...},
AT_EMPTY_PATH) = 0

getrandom("\x4c\xa0\x1e\x65\xae\x84\x12\x4e", 8, GRND_NONBLOCK)
= 8

brk(NULL)                 = 0x55e283e2d000

brk(0x55e283e4e000)         = 0x55e283e4e000

write(1, "Hello World\n", 12)      = 12  <- Hey! I recognise this bit!

exit_group(0)               = ?

+++ exited with 0 +++
```

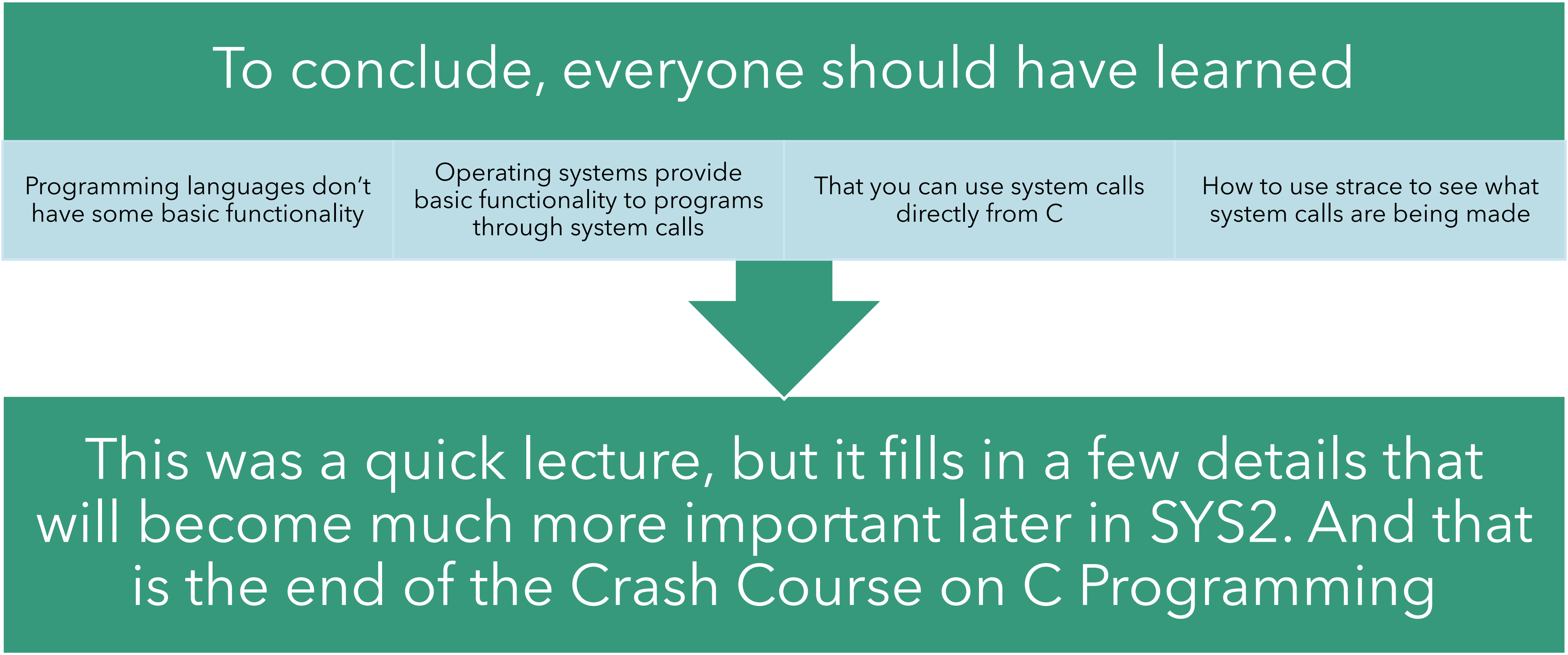
System Call Manual

- Once you have something you recognise, you can use Section 2 of the manual to look up system calls
  - `man 2 write`
- Just like with library calls, this will tell you exactly what headers to include and how to use the system call directly

strace_log.txt (end)

```
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...},
AT_EMPTY_PATH) = 0

getrandom("\x4c\xa0\x1e\x65\xae\x84\x12\x4e", 8, GRND_NONBLOCK)
= 8

brk(NULL)                    = 0x55e283e2d000

brk(0x55e283e4e000)          = 0x55e283e4e000

write(1, "Hello World\n", 12)      = 12  <- Hey! I recognise this bit!

exit_group(0)                = ?

+++ exited with 0 +++
```

Conclusions



To conclude, everyone should have learned

- Programming languages don’t have some basic functionality
- Operating systems provide basic functionality to programs through system calls
- That you can use system calls directly from C
- How to use strace to see what system calls are being made

This was a quick lecture, but it fills in a few details that will become much more important later in SYS2. And that is the end of the Crash Course on C Programming

Assignment

This lectures assignment has you look at the `strace` utility and `hello_world`

Go back and run `hello_world` under `strace` to see how `printf` works

`strace` will give you a lot of output, but you should find that `printf` is writing to a file – a special one, called standard output (`stdout`)

You can then modify your `hello_world` to work without using `printf`, by using the output of `strace` and the manual `man` to work out how to write to the standard output directly using system calls.

It is possible (but not recommended) to only use `syscall` to do this.