Debugging web applications in a production environment

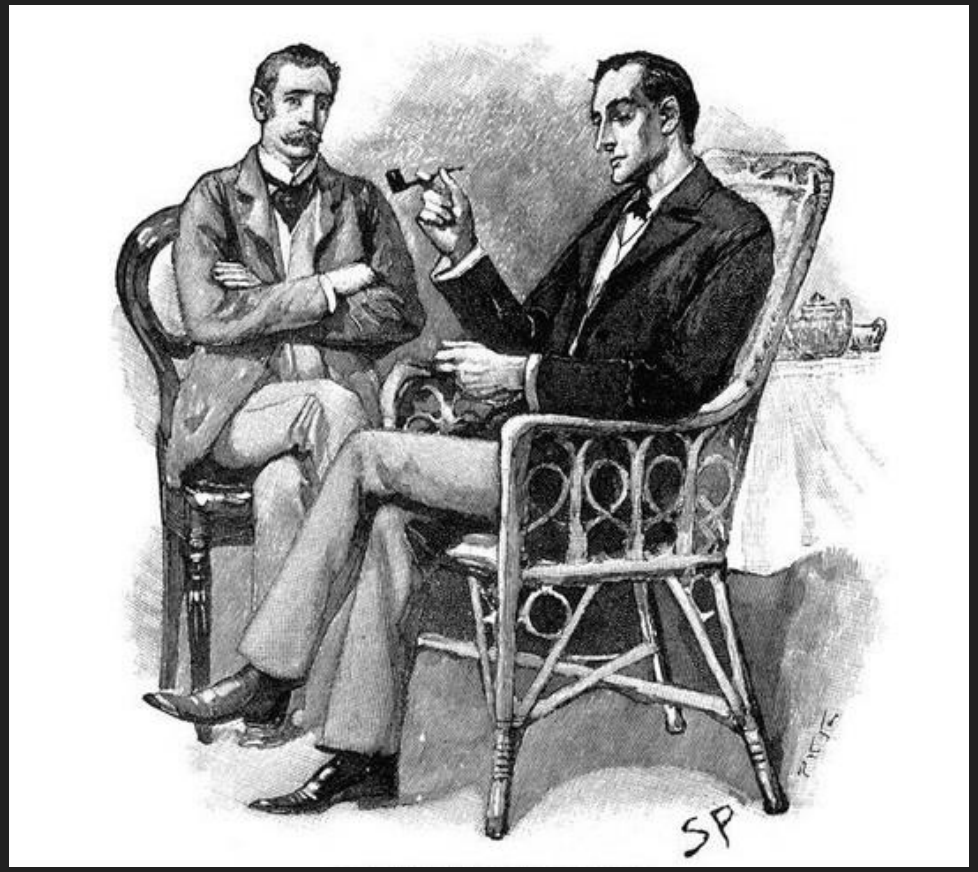
strace, ps and gdb.

Overview

- Mindset
- Production Environments
- Web applications
- Workflow
- Strace, ps and gdb
- Lessons learned
- Last production debugging session:
  - Lasted a number of days
  - Peaked at ~400 orders/second across ~90 servers
  - *Gigabytes* of data collected.

Mindset

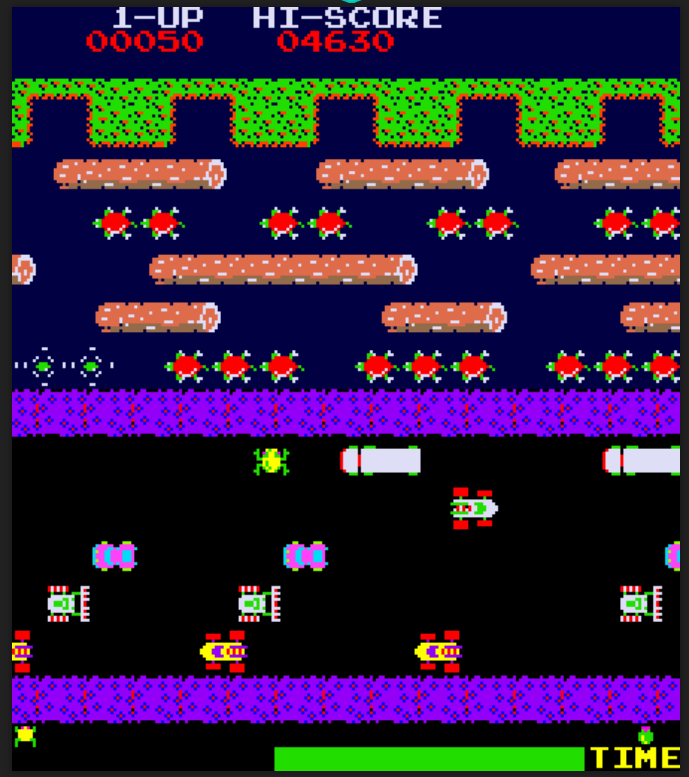
'It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.'



'Eliminate all other factors, and the one which remains must be the truth.'

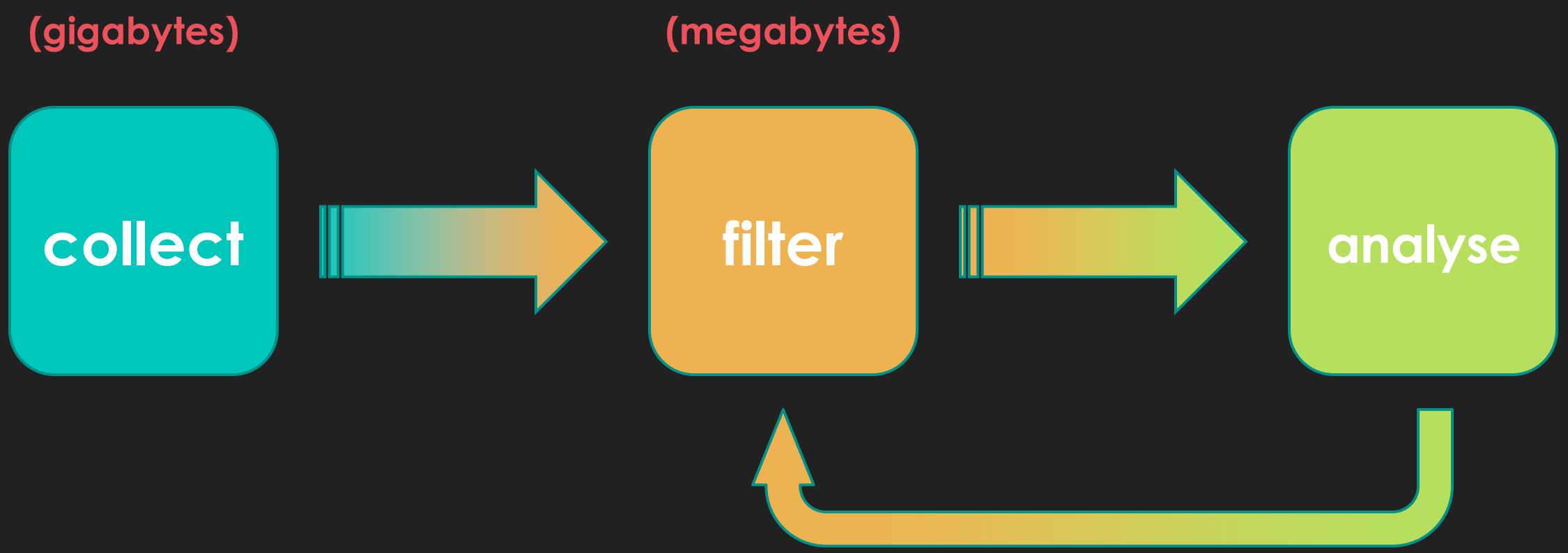
Production Environments

frogger



- Users don't stop
- Can't edit the code, no print/warn
- Can't see what they see (especially in SSL environment)
- Security concerns
- Fast!
- Data overload!

Work flow



Nature of web apps

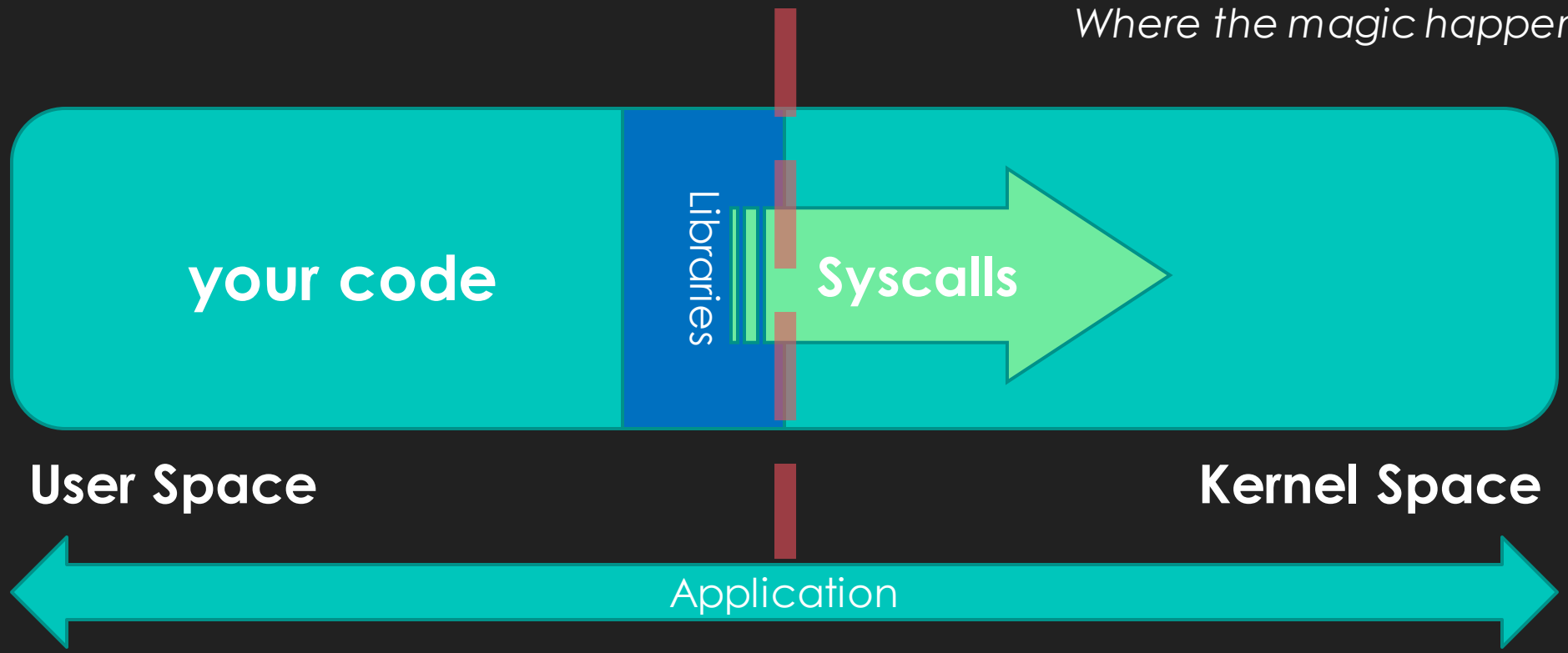
Dr. Watson, it's a well known fact that 87.36% of web applications are I/O bound

Sherlock Holmes

Making sausage.

- You take stuff in various sources
- You output a web page
- System calls are your I/O interface.

System calls



System calls

Common System calls

- open()
- read()
- write()
- execve()
- brk()
- time()
- mmap()
- ioctl()
- connect()

More Info:

- System Calls are described in section 2 of ‘man’
  - E.g man 2 mmap2
- System calls are C functions and will be of the format function(arg1, arg2, .. argN) → value

Strace

strace intro

- Allow you to see which syscalls an application is executing along with arguments and return values
- Two Modes
  1. Run the application with strace
  2. Attach to a running application (with –p.) **Needs root access!**
- Can be found in your Linux distribution's package manager (apt-get, yum, etc.)
- Generates a LOT of data.
  - 116 lines of output for a simple "ls"
  - Filter, filter, filter, grep.

strace sample

Strace options

- -c : show summary table
- -f : follow child processes
- -T : show time spent in system cals
- -t, -tt, -ttt : show start time, with milis or in epoch time
- -e trace=[set name] : trace only a certain set of calls.
- -e read=fd1, fd2, etc write=fd1,f2 : show ascii and hex dump of data read/written
- In order to use ssql.pl please use: -Ttt –f
- More options and examples listed in the man page

What to look for

- Long running system calls
  - Slow network activity
  - Bad hardware
  - Bad Cable!
- Infinite/Long Running Loops (repeated syscalls doing the same thing.)
- Unnecessary I/O:
  - Possible cache candidates
  - Reading large, unexpected datasets (e.g. an SQL query gone wrong.)
- Customer's view.

Ssql.pl

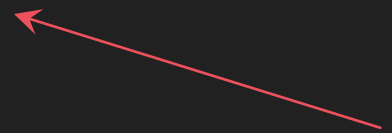
- Download from https://github.com/jonphilpott/stracetools
- Very helpful for Filter and Analyze stages:

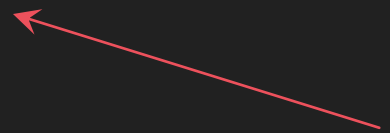
```
sqlite> select * from strace order by dur desc limit 5 ;
1177|17061|22:57:45|136849|read|read(0, "l", 1)|1|2.165879
1398|17061|22:57:48|172280|read|read(0, "\r", 1)|1|0.910384
1474|17061|22:57:49|96344|read|read(0, "\4", 1)|1|0.666353
1197|17061|22:57:47|704137|read|read(0, "\r", 1)|1|0.428623
1192|17061|22:57:47|473728|read|read(0, "&", 1)|1|0.229005
```

ps

Regular ps output

```
jon@jon-craptop:~/stracetools$ ps -eal|head
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -   917 poll_s ?        00:00:01 init
1 S     0     2     0  0  80   0 -     0 kthrea ?        00:00:00 kthreadd
1 S     0     3     2  0  80   0 -     0 run_ks ?        00:00:35 ksoftirqd/0
1 S     0     6     2  0 -40   - -     0 cpu_st ?        00:00:00 migration/0
1 S     0     7     2  0 -40   - -     0 watchd ?        00:00:07 watchdog/0
1 S     0     8     2  0 -40   - -     0 cpu_st ?        00:00:00 migration/1
1 S     0    10     2  0  80   0 -     0 run_ks ?        00:00:13 ksoftirqd/1
1 S     0    11     2  0 -40   - -     0 watchd ?        00:00:06 watchdog/1
1 S     0    12     2  0 -40   - -     0 cpu_st ?        00:00:00 migration/2
```

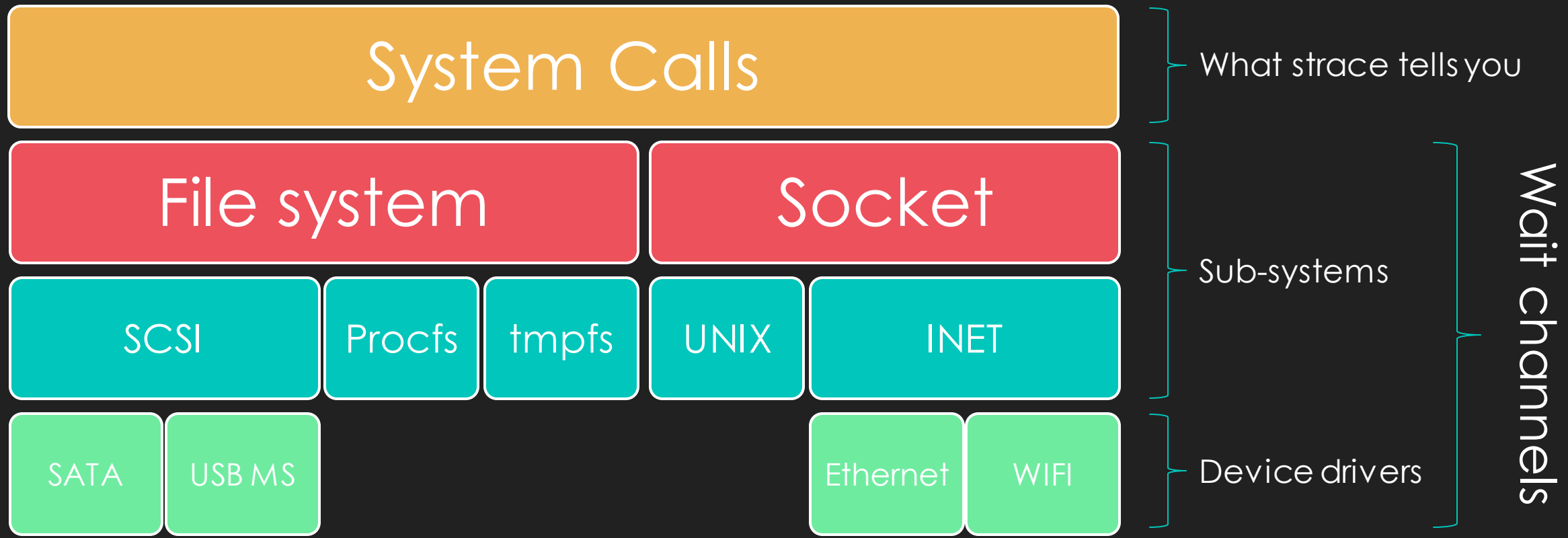




Process states

STATE	DESCRIPTION
D	**UNINTERRUPTIBLE SLEEP (I/O) –ARGH!!!!**
R	Running/Runnable
S	Interruptible sleep
T	Stopped
W	Paging (not in 2.6 kernels)
X	Dead (never see this)
Z	Zombie

Wait Channels



wtf Channels?

- Tell you where *exactly* your process is stuck in kernel land.
- Two places to find out what the wait channel means
  1. The sensible name of the function, e.g.: `scsi_error_handler`
  2. The kernel source and grep.

procmon.pl

- Download from https://github.com/jonphilpott/stracetools
- Allows you collect process information over time
- Can output CSV or into a Sqlite3 database.

```
sqlite> select * from ps where vmswap > 0 limit 5;
3|1|1365654779|init|S|0|0|3668|3672|1892|712|136|184|2532|104|1|poll_schedule_timeout
51|799|1365654779|upstart|S|0|0|2848|2848|288|176|136|132|2332|56|1|poll_schedule_timeout
54|908|1365654779|dbus|S|102|105|6172|6212|3560|3096|136|424|2404|172|1|poll_schedule_timeout
55|920|1365654779|modem|S|0|0|7224|7288|2076|400|136|388|5988|368|1|poll_schedule_timeout
56|925|1365654779|bluetoothd|S|0|0|4744|4744|1196|180|136|840|3456|156|1|poll_schedule_timeout
```

gdb

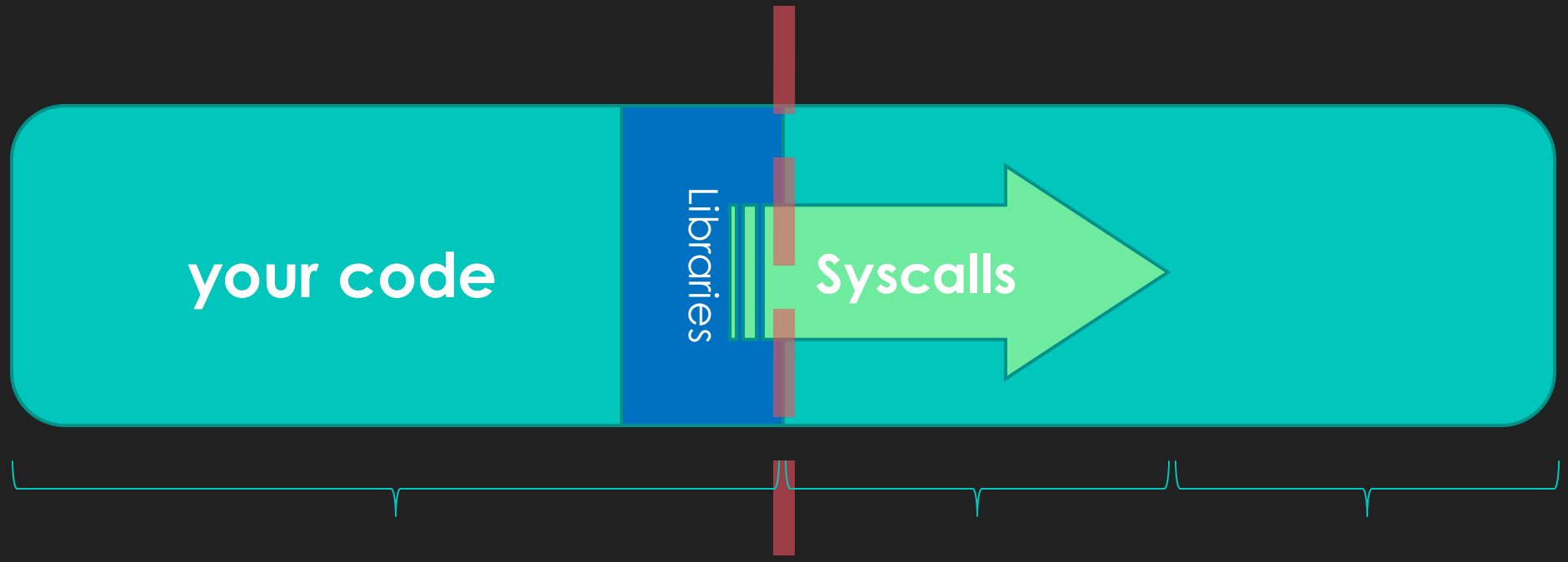
The GNU debugger is the only tool I use.

Sherlock Holmes

gdb – The GNU debugger

- Like strace; has two modes
  - Execute the application within gdb
  - Attach to an existing process
- The ultimate tool
  - And the hardest to use
- The last resort:
  - Pauses processes

Where gdb sits



gdb simple tools

- Simple techniques to gather information
  1. Controlling execution
  2. The Stack
  3. Examining and altering data
  4. Breakpoints
  5. Macros, logging and .gdbinit
  6. Executing code.

C pointer syntax? Oh hell, no.

Dr. Watson

gdb commands

- continue : Continue Execution
- finish : execute until the end of the current stack frame
- backtrace : display stack trace
- up/down # : move up and down the stack
- x/16wx $esp : Display 16 words of data at the top of the stack
- x/s *(char **)($esp + *n*4)* : display nth string argument
- Info registers
- *(int **)($esp + *n* * 4) = 1234; : set value to 1234
- set $ret_value = function(args, …) : execute C function return value into $ret_value
- break *function* : set breakpoint on function
- catch syscall *systemcallname*
- continue : continue execution
- Finish : run current functions until completion
- Condition n [conditional] : set breakpoint condition

gdb macros

```
define xna
   x/s *(char **)($esp + $arg0 * 4)
end
```

Executing perl code

- Ancient Perl (5.8.4 + No Symbols!)

```
define run_perl
 set $sv = Perl_eval_pv($arg0, 1)
 printf "ret: %s\n", **$sv
end
```

- Modern Perl (Needs Symbols ☹)

```
define run_perl
  set $sv =
Perl_eval_pv(PL_curinterp, $arg0, 1)
  printf “%s”, *(char **)((void
*)($sv + 12)
end
```

Play with it

- Take some time and tinker
- Dig
- Break it
- … but not in prod.

Questions?

- Email: jonathan.philpott@ticketmaster.com
- Presentation and tools downloadable from: https://github.com/jonphilpott/stracetools