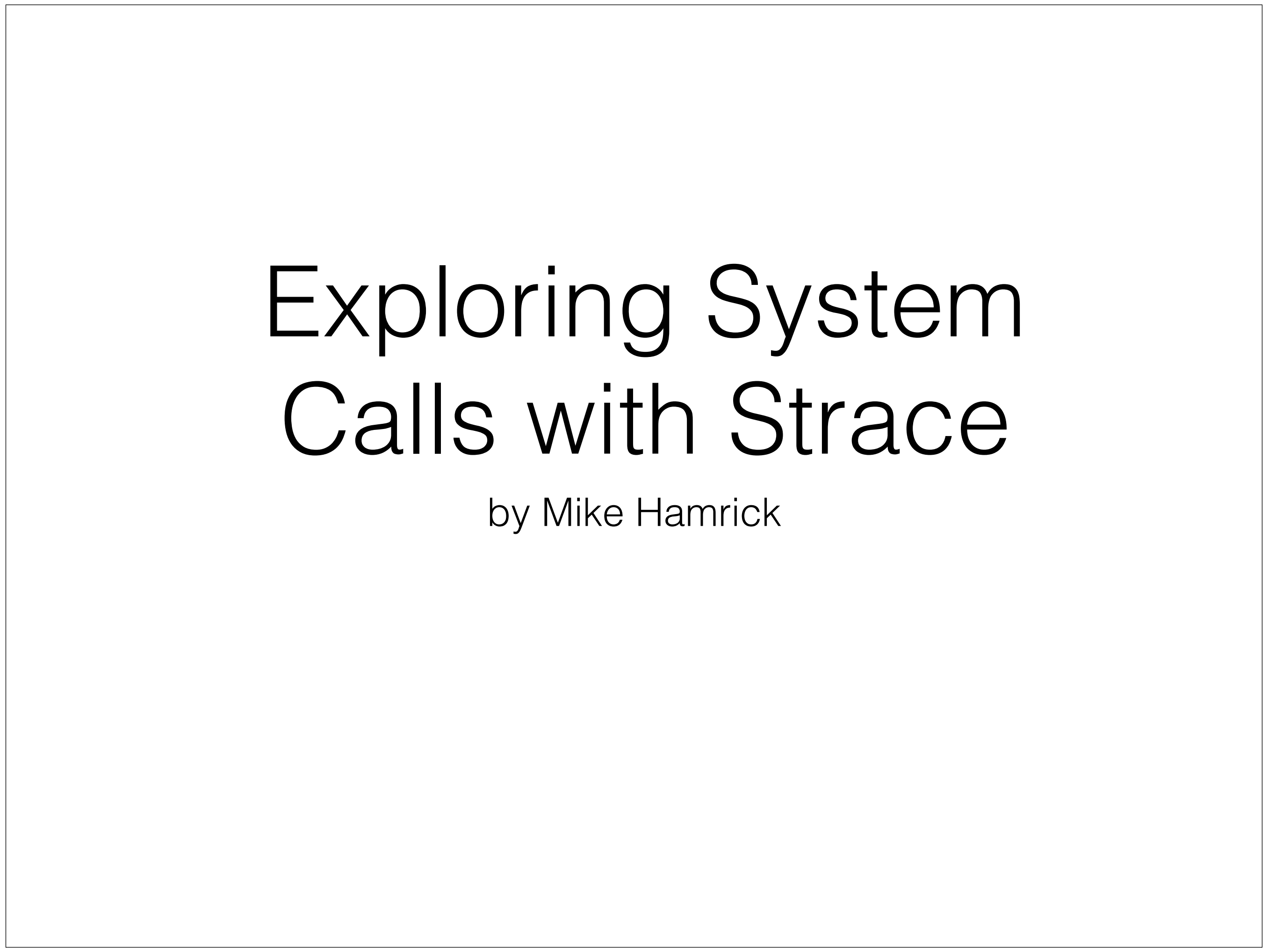
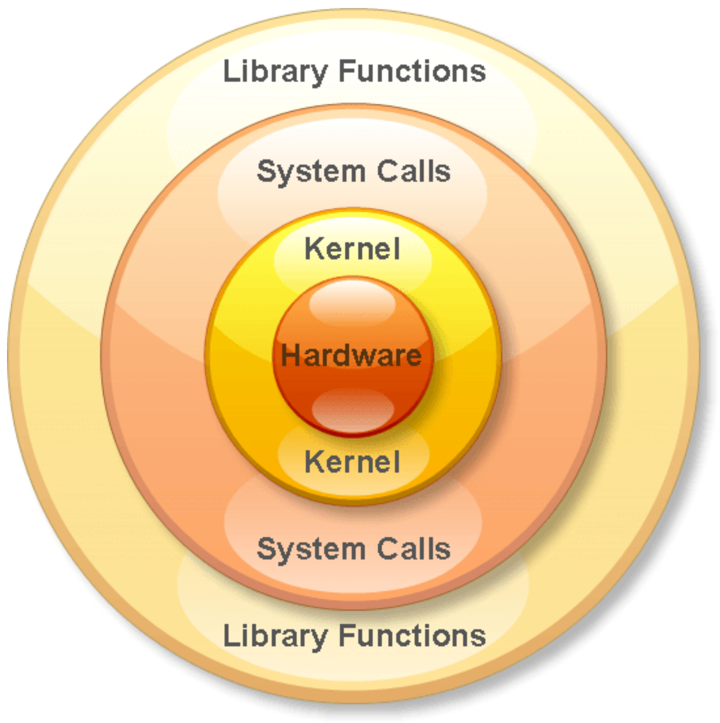


Exploring System Calls with Strace

by Mike Hamrick

I’m Mike Hamrick. In my career as a programmer, sysadmin, and DBA I’ve used strace quite a lot to learn what programs are doing under the hood. It’s often the first tool I reach for when I really want to know why programs aren’t behaving like I expect them to. With strace the opaque becomes transparent, and that transparency has helped me solve many thorny systems problems. I hope by the end of this talk you have a good idea of how strace can help you solve problems in systems that you manage.

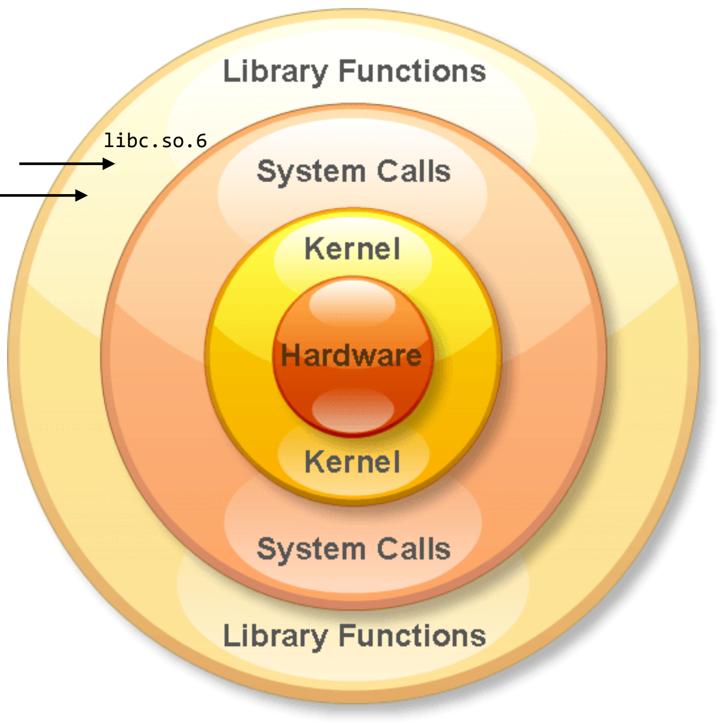
What are system calls?



Strace is a program that lets you see what system calls a program is making. What are system calls? System calls are the interface that the operating system exposes to programmers. Often times you don't call the system calls directly. You, the programmer, instead call your programming language's library functions, which in turn make system calls on your behalf.

What are system calls?

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf("Hello world!\n");
  exit(42);
}
```

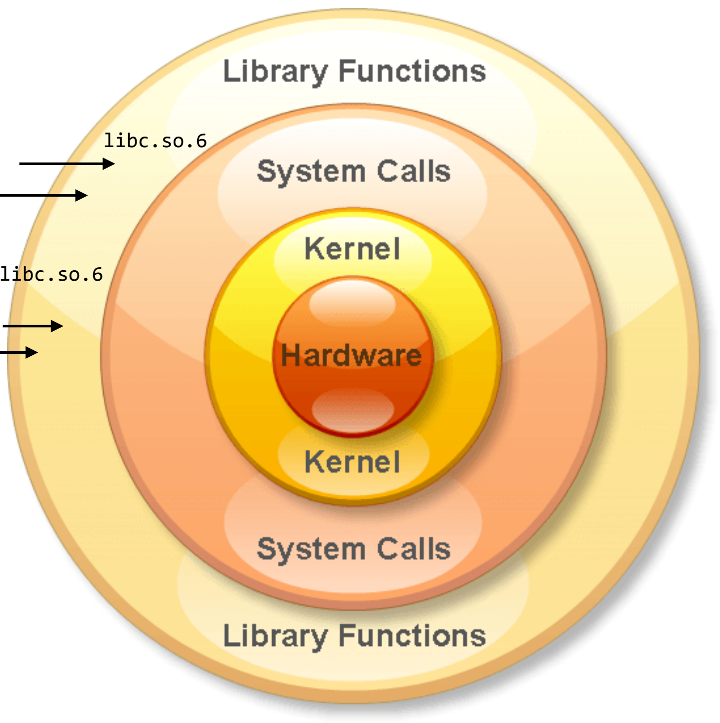


This is the canonical C ‘hello world’ program. It prints ‘hello world’ to the terminal and then exits back to the operating system with a return value of 42, well because 42. Almost all C programs use the standard C runtime library, in this case libc.so.6.

What are system calls?

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf("Hello world!\n");
  exit(42);
}
```

```
#!/usr/bin/php
<?php
  printf("Hello world!\n");
  exit(42);
?>
```



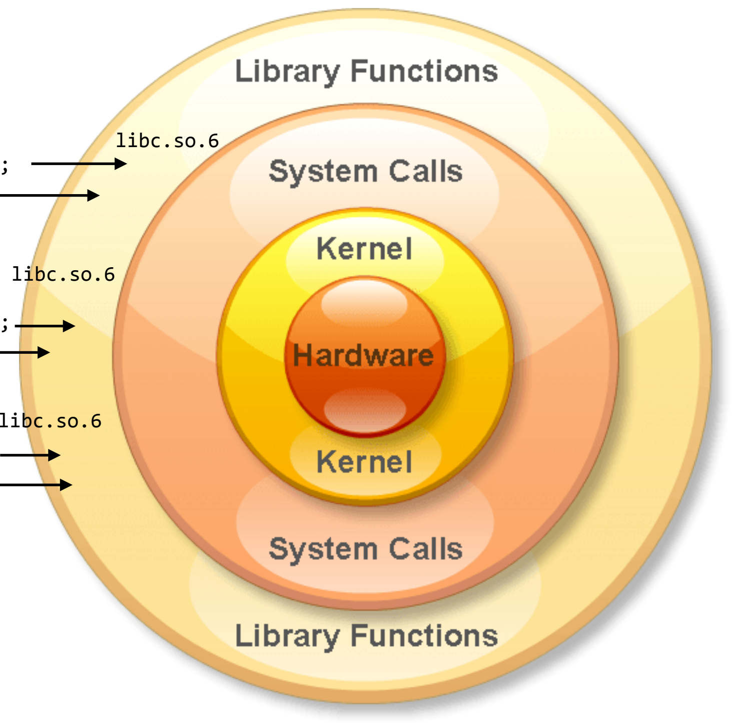
Here's a PHP program that does the same thing as the C program. Many interpreted languages like PHP have library functions that share the same name as the C library functions. That's because C is the lingua franca of UNIX. Many interpreted languages are both written in C and inspired by it.

What are system calls?

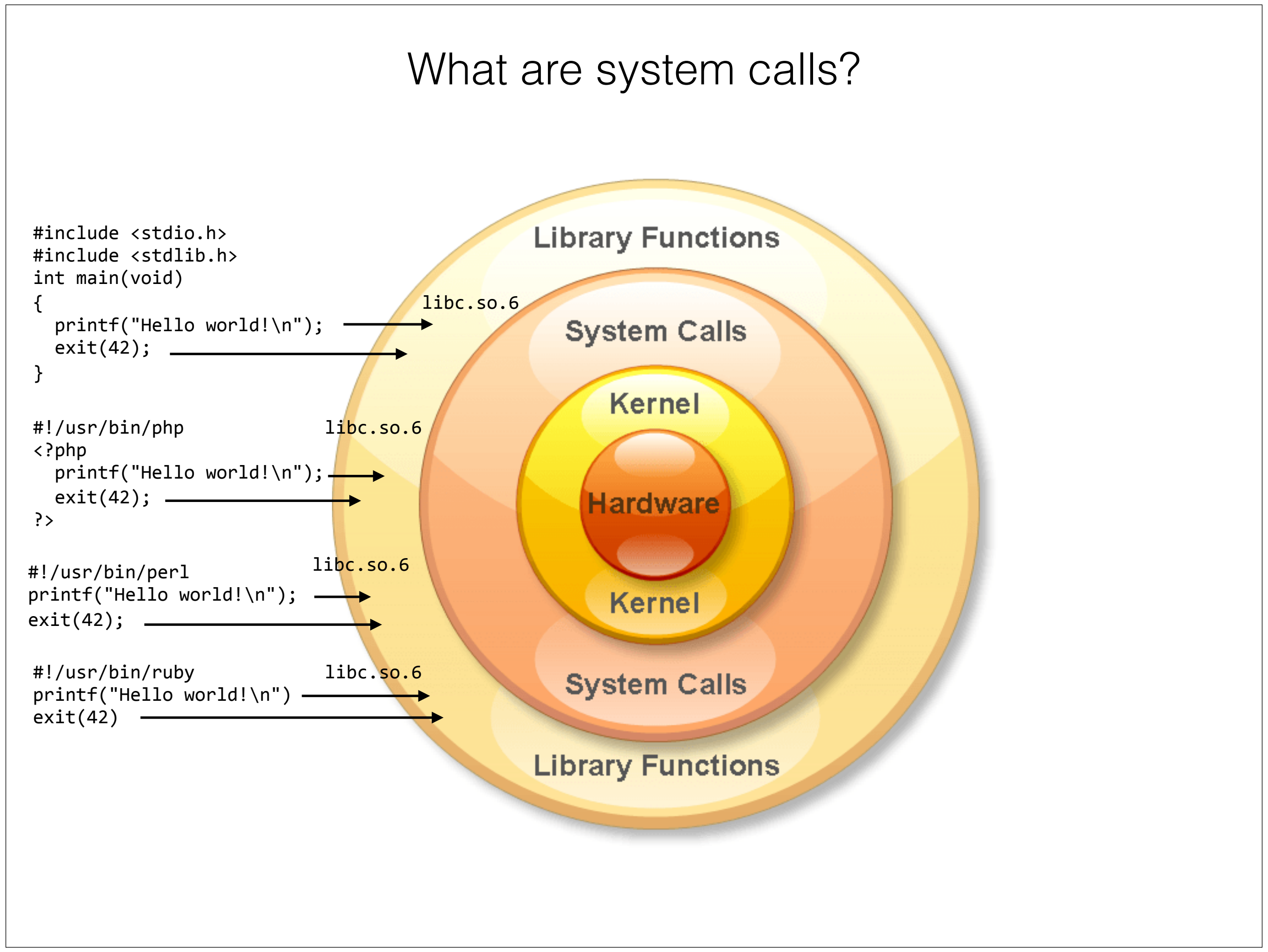
```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf("Hello world!\n");
  exit(42);
}
```

```
#!/usr/bin/php
<?php
  printf("Hello world!\n");
  exit(42);
?>
```

```
#!/usr/bin/perl
printf("Hello world!\n");
exit(42);
```



Perl is another example of an interpreted language that shares some of the same library functions as C. Perl has many, many library functions that have pretty direct mappings to the C library functions.



What are system calls?

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf("Hello world!\n");
  exit(42);
}
```

```
#!/usr/bin/php
<?php
  printf("Hello world!\n");
  exit(42);
?>
```

```
#!/usr/bin/perl
printf("Hello world!\n");
exit(42);
```

```
#!/usr/bin/ruby
printf("Hello world!\n")
exit(42)
```

Even a relatively new programming language like Ruby adopts some of the function names and methodologies of the C runtime library into its libraries.

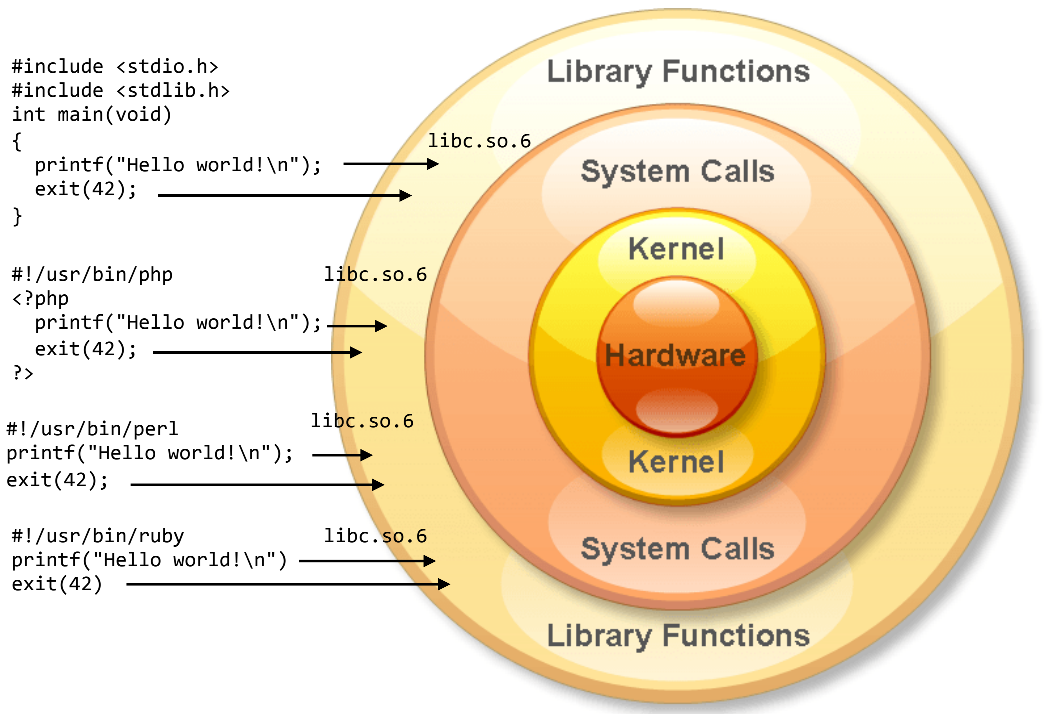
What are system calls?

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf("Hello world!\n");
  exit(42);
}
```

```
#!/usr/bin/php
<?php
  printf("Hello world!\n");
  exit(42);
?>
```

```
#!/usr/bin/perl
printf("Hello world!\n");
exit(42);
```

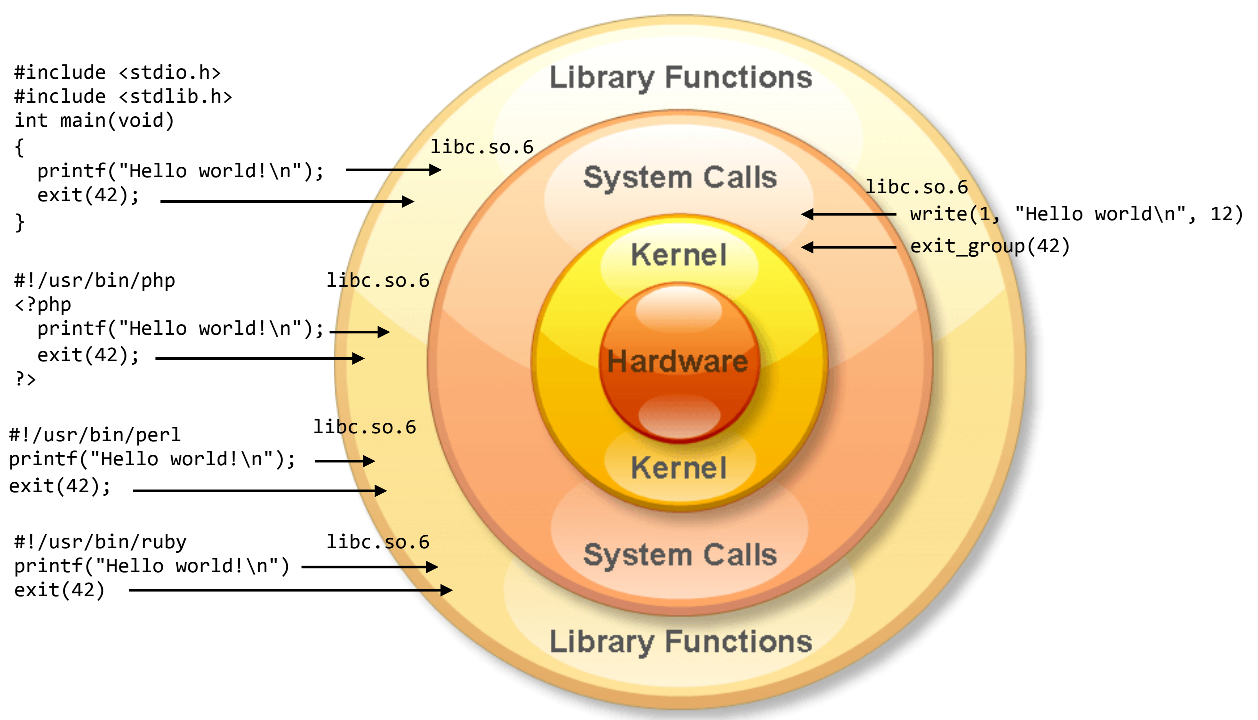
```
#!/usr/bin/ruby
printf("Hello world!\n")
exit(42)
```



```
$ for lang in {php,perl,ruby}; do ldd /usr/bin/$lang | grep libc.so; done
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd01fbe3000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5f3aac6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6851fe8000)
```

And in fact here on the bottom you can see that all three of these interpreted languages have a library dependency on libc6. It's possible that in some of these languages calls to their library functions are simply wrappers around the same functions in the C library.

What are system calls?



```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf("Hello world!\n");
  exit(42);
}
```

```
#!/usr/bin/php
<?php
  printf("Hello world!\n");
  exit(42);
?>
```

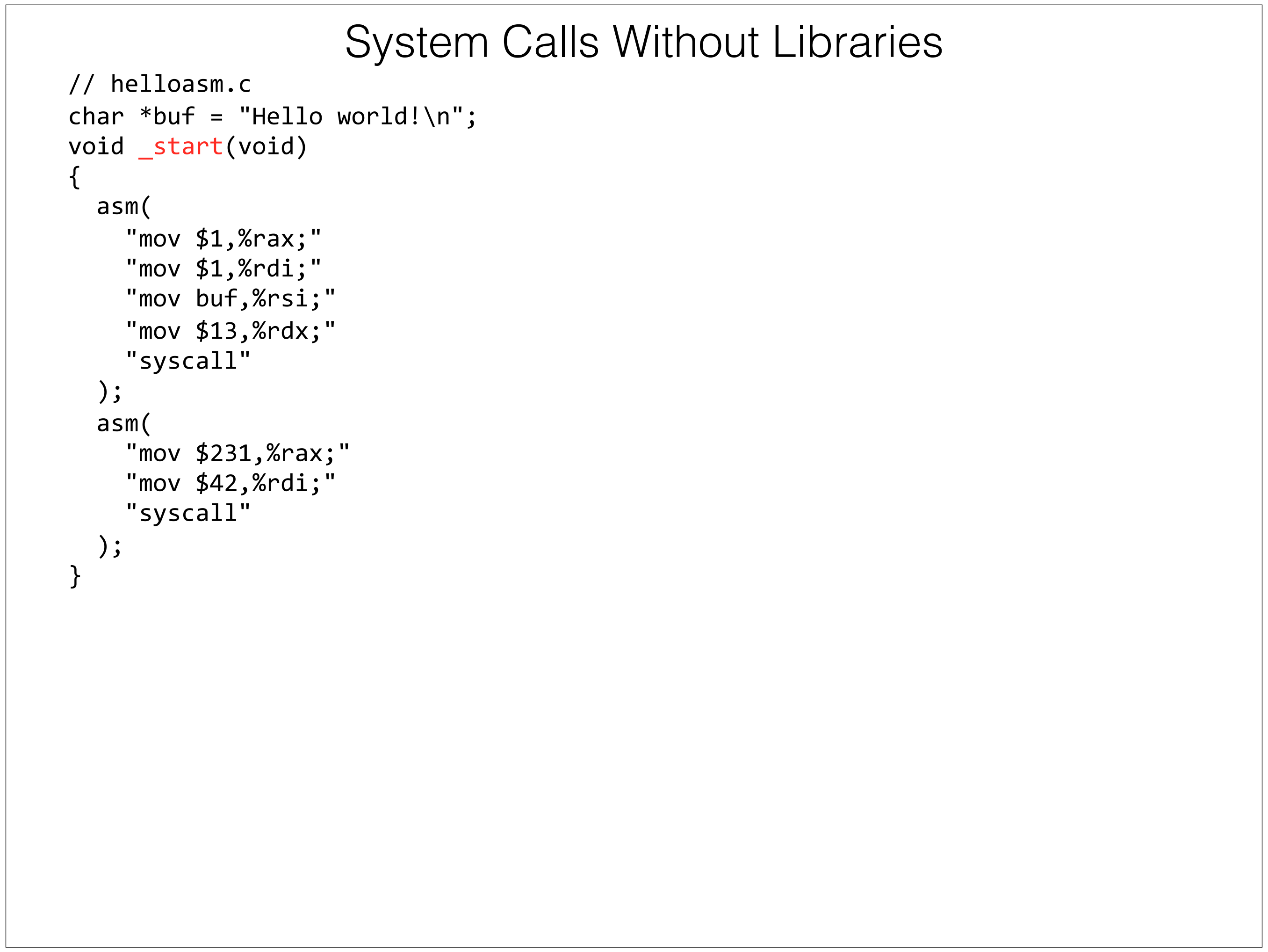
```
#!/usr/bin/perl
printf("Hello world!\n");
exit(42);
```

```
#!/usr/bin/ruby
printf("Hello world!\n")
exit(42)
```

```
write(1, "Hello world\n", 12)
exit_group(42)
```

```
$ for lang in {php,perl,ruby}; do ldd /usr/bin/$lang | grep libc.so; done
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd01fbe3000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5f3aac6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6851fe8000)
```

Now each one of the library calls in these examples eventually makes a lower-level system call do do the actual work. The printf library routine eventually calls the write() system call to ouput ‘hello world’ to the hardware — the terminal in this case. The exit function eventually maps to the exit_group() system call, which causes every thread in a program to exit.



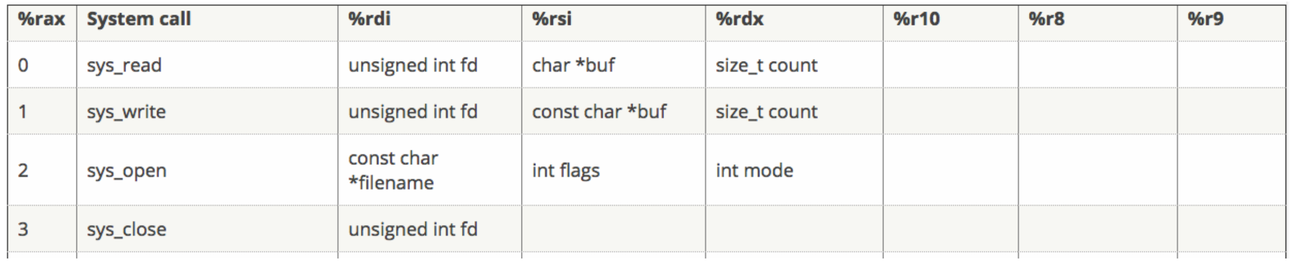
System Calls Without Libraries

```
// helloasm.c
char *buf = "Hello world!\n";
void _start(void)
{
  asm(
    "mov $1,%rax;"
    "mov $1,%rdi;"
    "mov buf,%rsi;"
    "mov $13,%rdx;"
    "syscall"
  );
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```

This example shows how you can make system calls directly without involving the C runtime library. This assembly code might look daunting at first, but don’t worry, this is easy. You might remember that in C program execution begins with the main() function, but when the operating system runs a program, looks for a _start label, which is usually provided by the C runtime library. Since we’re not using the C runtime library we use _start() and not main().

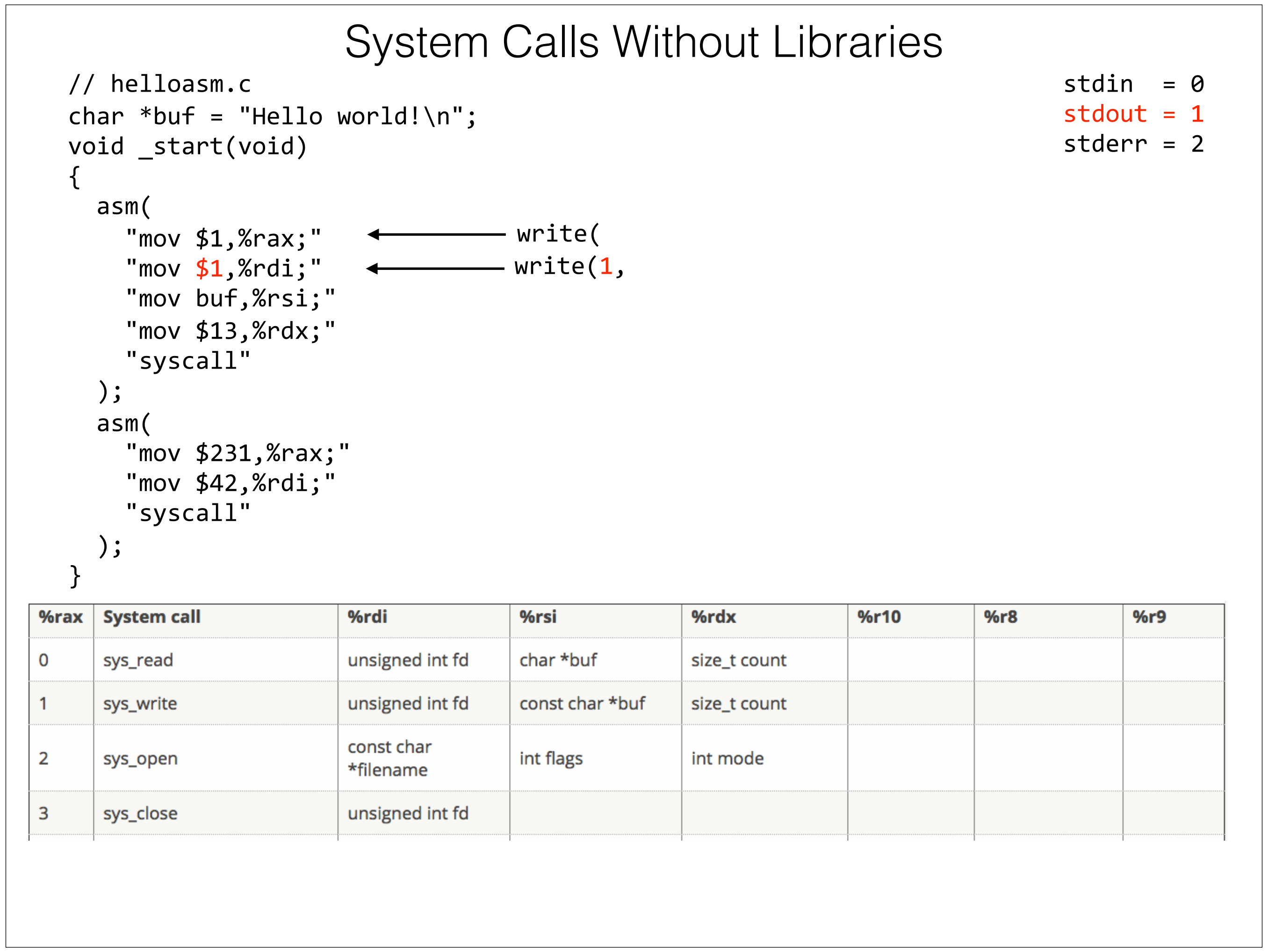
System Calls Without Libraries

```
// helloasm.c
char *buf = "Hello world!\n";
void _start(void)
{
  asm(
    "mov $1,%rax;"   ←——————— write(
    "mov $1,%rdi;"
    "mov buf,%rsi;"
    "mov $13,%rdx;"
    "syscall"
  );
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```



%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

System calls are referenced by their numbers. The first four system calls are the foundational ones, read, write, open, and close. You can do quite a lot with just these system calls. System calls are all executed by loading values into machine registers. The rax register (also called the accumulator) is where you indicate which system call you'd like to make. A system call can have up to six arguments which are placed in rdi, rsi, rdx, r8, r9, r10. It's worth noting that these are 64-bit system calls and registers we're using. In the 32 bit world this changes. Here we load 'write' (system call #1) into RAX.

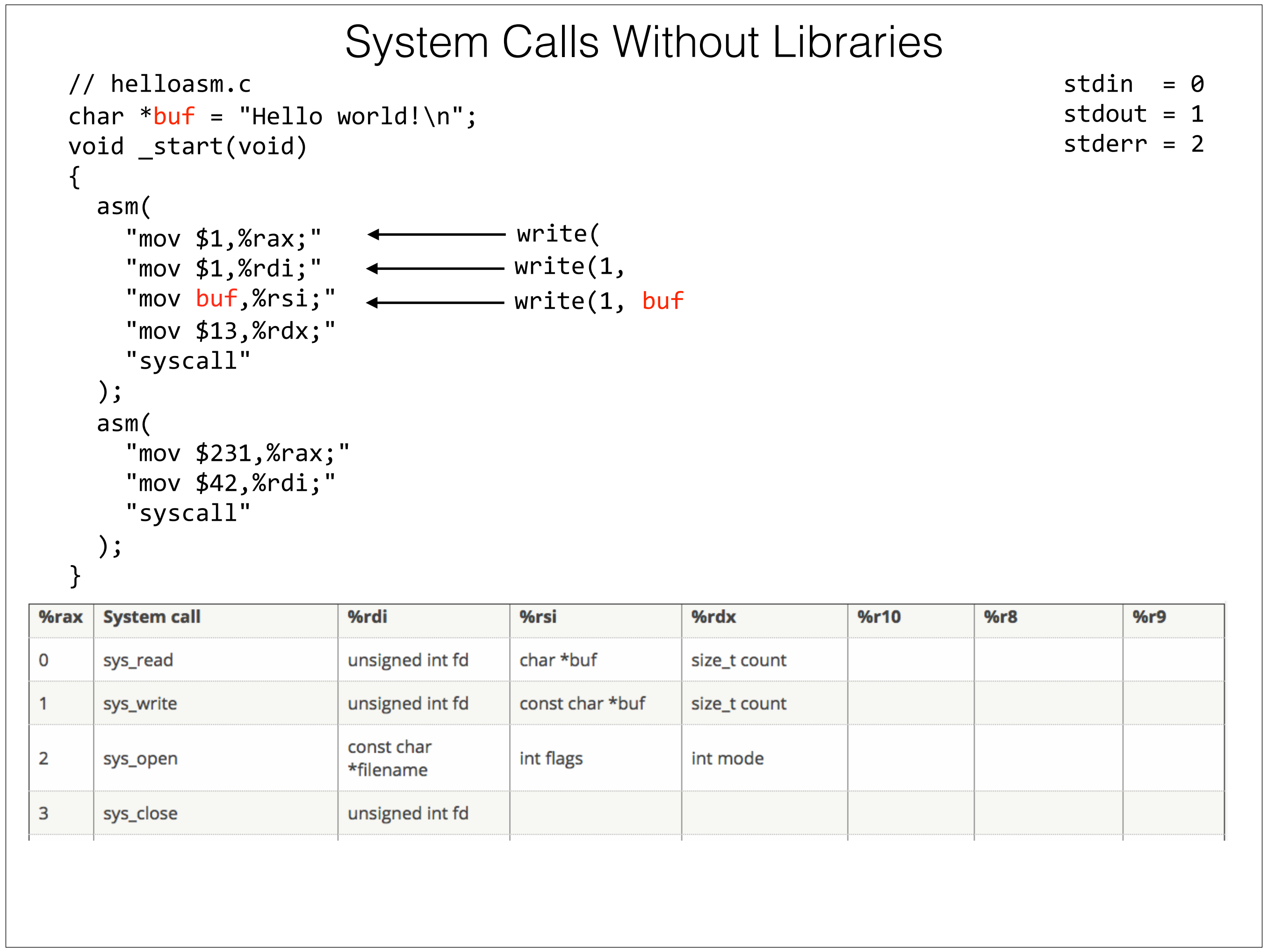


System Calls Without Libraries

```
// helloasm.c                                                          stdin  = 0
char *buf = "Hello world!\n";                                          stdout = 1
void _start(void)                                                      stderr = 2
{
  asm(
    "mov $1,%rax;"      <---------  write(
    "mov $1,%rdi;"      <---------  write(1,
    "mov buf,%rsi;"
    "mov $13,%rdx;"
    "syscall"
  );
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

The first argument to write is the file descriptor that you want to write to. UNIX uses the file abstraction to model a lot of different things. A file descriptor is a number that can refer to an actual file, a terminal, a random number generator, a printer, a joystick, a network socket, or some giant industrial robot. Traditionally the first three file descriptors (0, 1, and 2) are mapped to your programs standard input, standard output, and standard error. In our case we want to write so standard out, so we load a 1 into the RDI register.

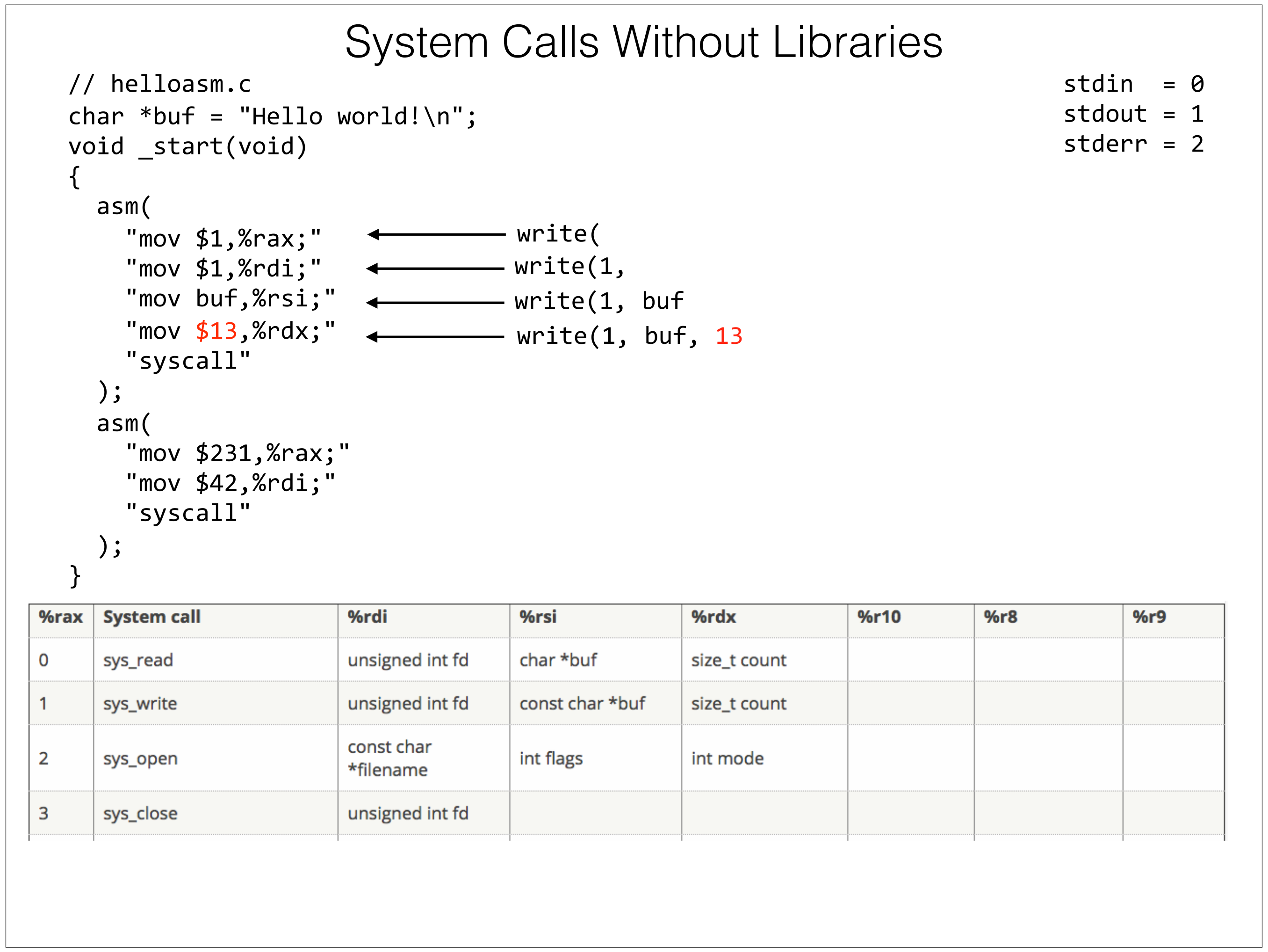


System Calls Without Libraries

```
// helloasm.c                                                              stdin  = 0
char *buf = "Hello world!\n";                                              stdout = 1
void _start(void)                                                          stderr = 2
{
  asm(
    "mov $1,%rax;"    ←———————— write(
    "mov $1,%rdi;"    ←———————— write(1,
    "mov buf,%rsi;"   ←———————— write(1, buf
    "mov $13,%rdx;"
    "syscall"
  );
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

The second argument to the write system call is a pointer to the buffer of the data that we want to write. Now at the top of our program we've declared 'buf' as a pointer to the "Hello world!" string, so now we load that pointer into the RSI register. We could have used a string literal in place of 'buf' here, but I wanted to make it explicit that we were loading a pointer into RSI.

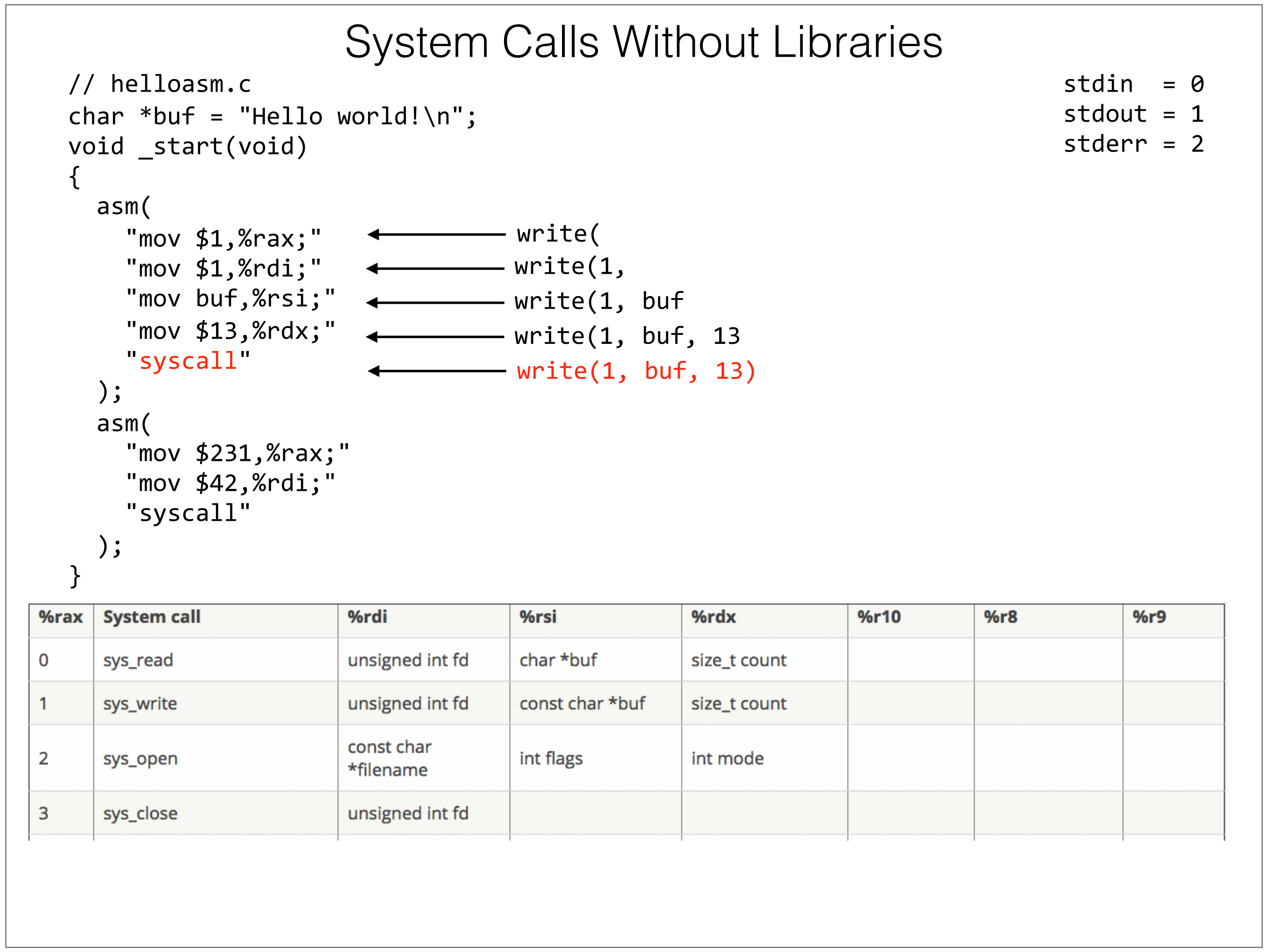


System Calls Without Libraries

```
// helloasm.c                                                     stdin  = 0
char *buf = "Hello world!\n";                                     stdout = 1
void _start(void)                                                 stderr = 2
{
  asm(
    "mov $1,%rax;"      ←———————  write(
    "mov $1,%rdi;"      ←———————  write(1,
    "mov buf,%rsi;"     ←———————  write(1, buf
    "mov $13,%rdx;"     ←———————  write(1, buf, 13
    "syscall"
  );
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

The last argument to the write() system call is the number of bytes that we're writing to the file descriptor. 'Hello' and 'world' are both 5 bytes, so that's 10, the space makes it 11, the bang makes it 12, and the newline character ends us up at 13 bytes.

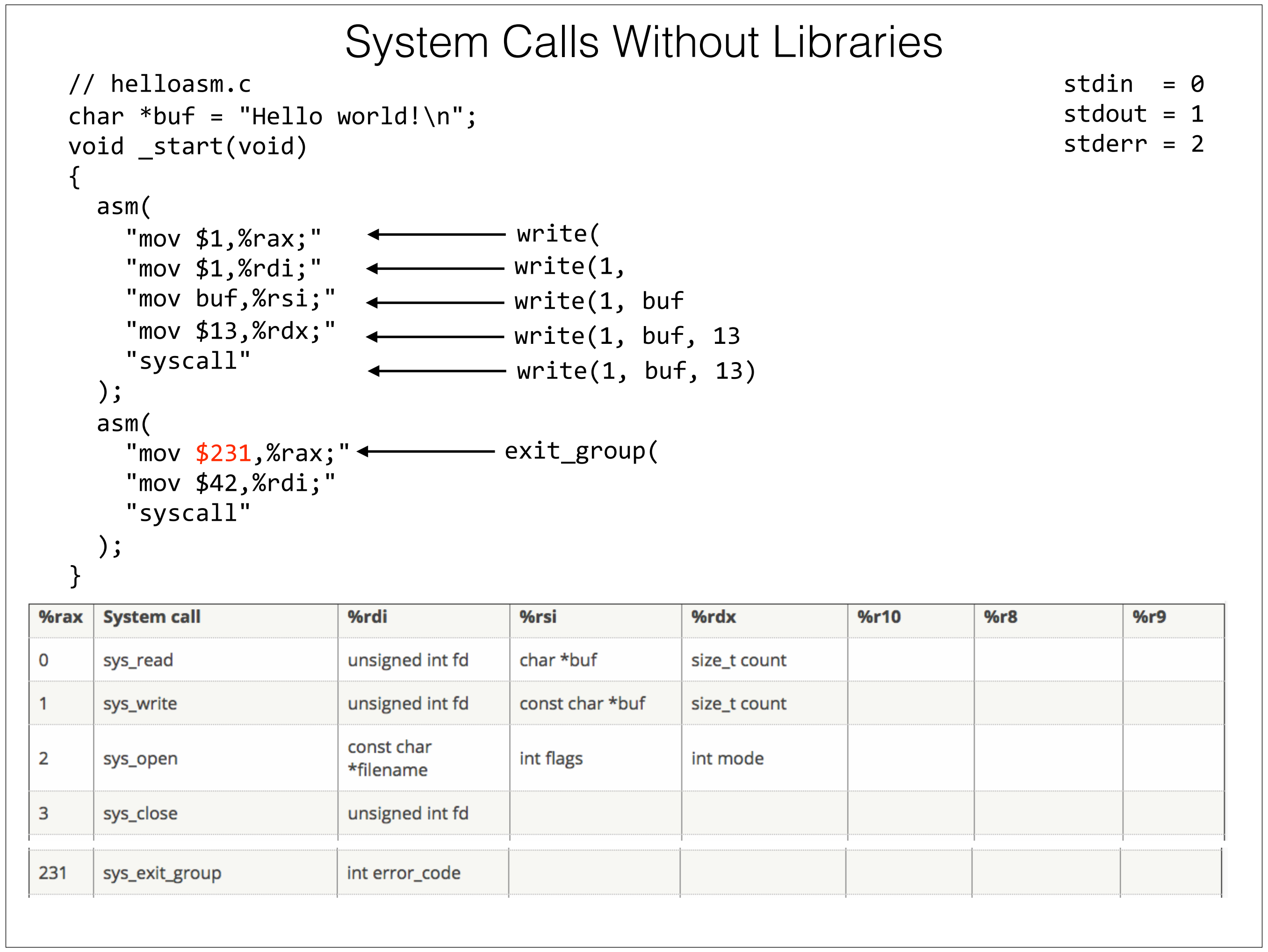


System Calls Without Libraries

```
// helloasm.c                                                  stdin  = 0
char *buf = "Hello world!\n";                                  stdout = 1
void _start(void)                                              stderr = 2
{
  asm(
    "mov $1,%rax;"    ←———— write(
    "mov $1,%rdi;"    ←———— write(1,
    "mov buf,%rsi;"   ←———— write(1, buf
    "mov $13,%rdx;"   ←———— write(1, buf, 13
    "syscall"         ←———— write(1, buf, 13)
  );
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

Now on the X86_64 CPU, which just about everything uses these days, you have a dedicated instruction called ‘syscall’ that executes the system call and takes you on the journey from userland into the kernel. In the old 32-bit days you used to call interrupt 0x80 to execute a system call after setting everything up. In the old-old days in MS-DOS you might remember INT 0x21 was the instruction you’d use to make an MS-DOS system call. The modern SYSCALL instruction is faster, and bypasses the chips interrupt handler system. The return value for the syscall is saved in RAX, and when you return the state of all your registers and stack is exactly as it was before you made the syscall. The fact that you can make a system call just by loading some values into registers and not having to fidddle with stack frames makes them really easy to call in many uncertain circumstances.

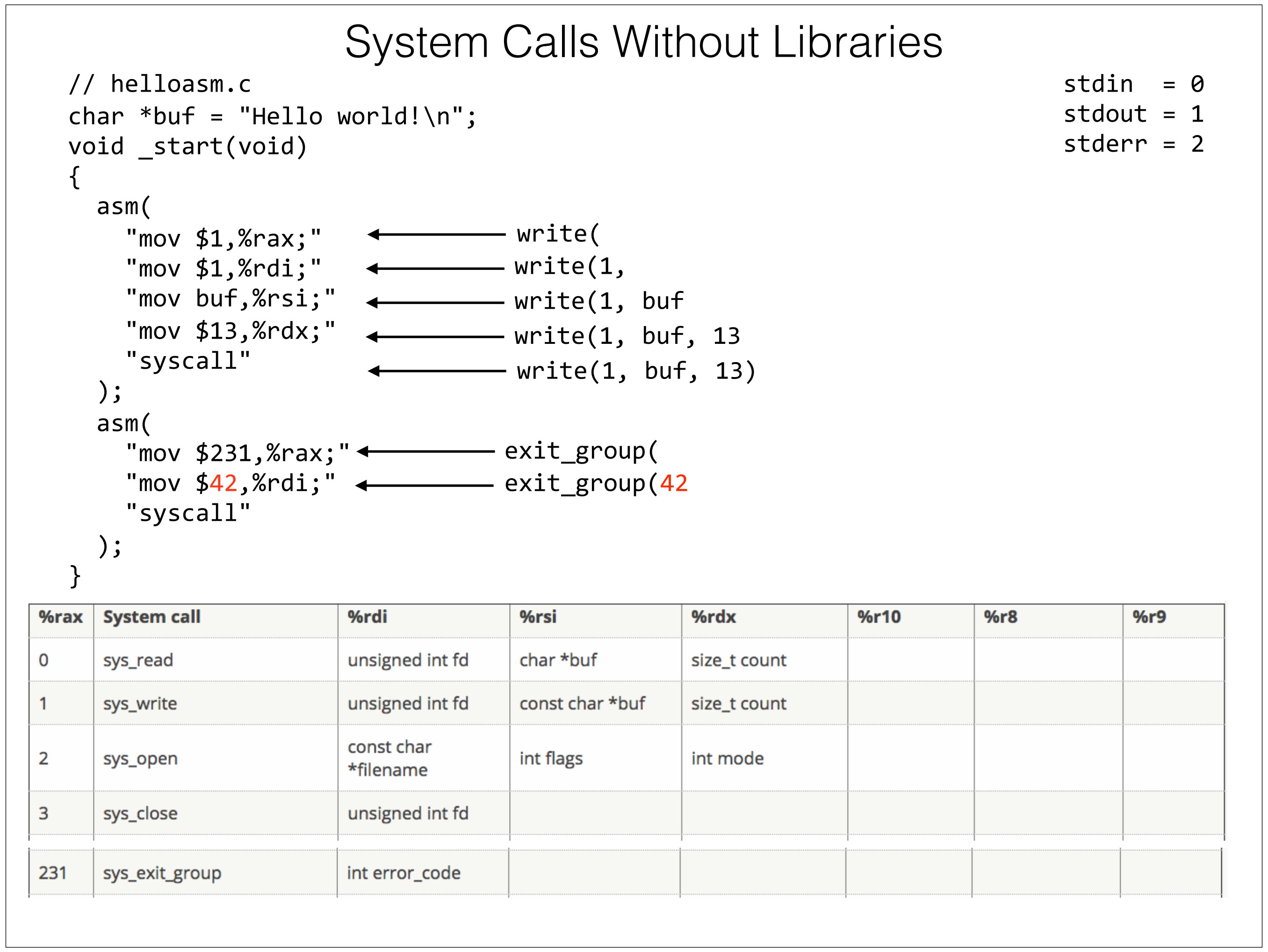


System Calls Without Libraries

```
// helloasm.c                                                          stdin  = 0
char *buf = "Hello world!\n";                                          stdout = 1
void _start(void)                                                      stderr = 2
{
  asm(
    "mov $1,%rax;"     <----------  write(
    "mov $1,%rdi;"     <----------  write(1,
    "mov buf,%rsi;"    <----------  write(1, buf
    "mov $13,%rdx;"    <----------  write(1, buf, 13
    "syscall"          <----------  write(1, buf, 13)
  );
  asm(
    "mov $231,%rax;"  <----------  exit_group(
    "mov $42,%rdi;"
    "syscall"
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
231	sys_exit_group	int error_code					

Now that we've outputted our 'Hello World' string it's time to exit. We're going to call the exit_group system call, which is #231, which we load into the RAX register. There is also just a plain exit() system call (60) but modern systems often use the thread aware exit_group syscall instead.

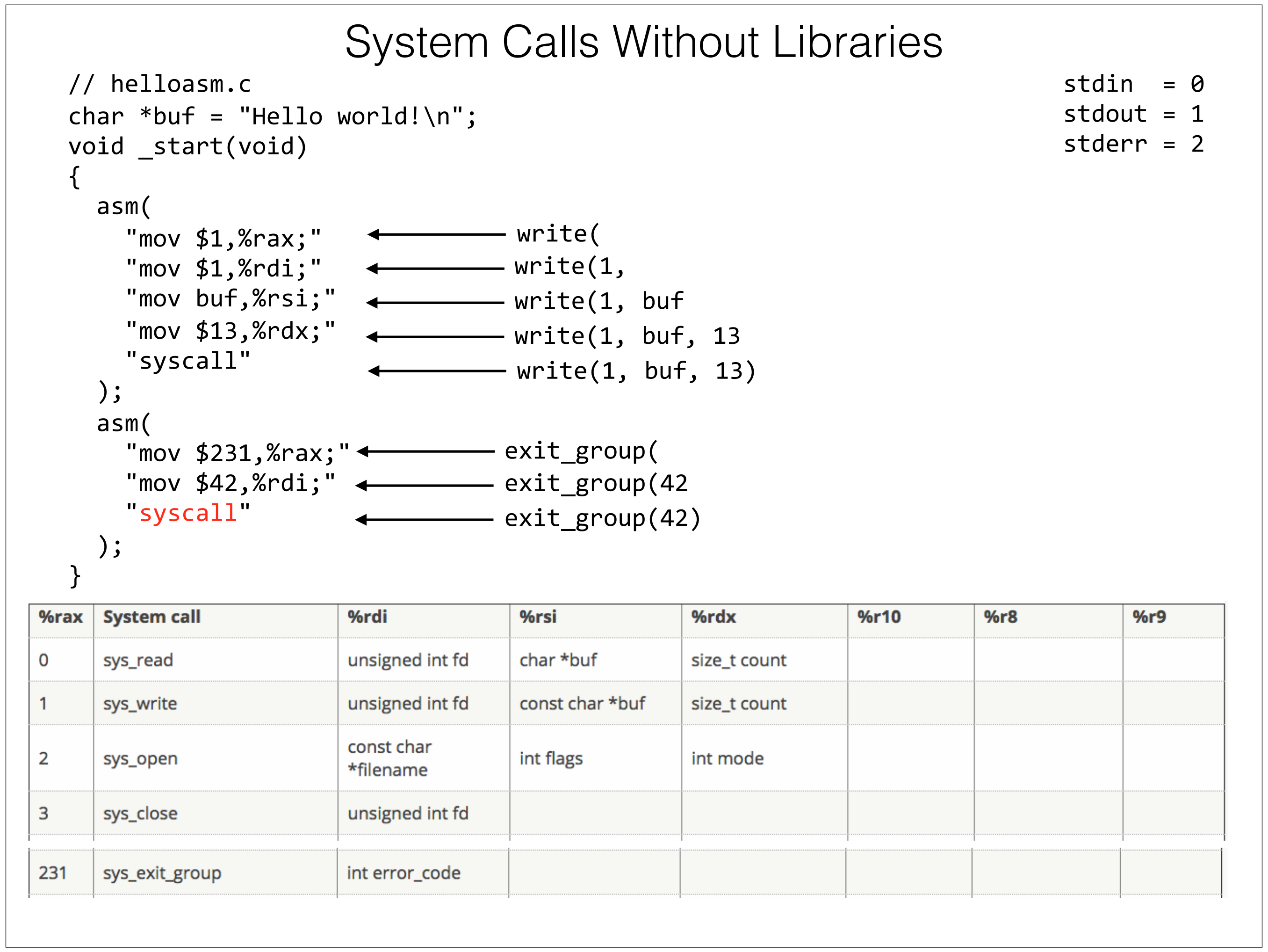


System Calls Without Libraries

```
// helloasm.c                                                  stdin  = 0
char *buf = "Hello world!\n";                                  stdout = 1
void _start(void)                                              stderr = 2
{
  asm(
    "mov $1,%rax;"    <--------- write(
    "mov $1,%rdi;"    <--------- write(1,
    "mov buf,%rsi;"   <--------- write(1, buf
    "mov $13,%rdx;"   <--------- write(1, buf, 13
    "syscall"         <--------- write(1, buf, 13)
  );
  asm(
    "mov $231,%rax;" <--------- exit_group(
    "mov $42,%rdi;"  <--------- exit_group(42
    "syscall"
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
231	sys_exit_group	int error_code					

The exit_group syscall takes a single argument, the error code to return back to the operating system. We load our magic number into the RDI register here.



System Calls Without Libraries

```
// helloasm.c                                                    stdin  = 0
char *buf = "Hello world!\n";                                    stdout = 1
void _start(void)                                                stderr = 2
{
  asm(
    "mov $1,%rax;"      ←──────── write(
    "mov $1,%rdi;"      ←──────── write(1,
    "mov buf,%rsi;"     ←──────── write(1, buf
    "mov $13,%rdx;"     ←──────── write(1, buf, 13
    "syscall"           ←──────── write(1, buf, 13)
  );
  asm(
    "mov $231,%rax;"    ←──────── exit_group(
    "mov $42,%rdi;"     ←──────── exit_group(42
    "syscall"           ←──────── exit_group(42)
  );
}
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
231	sys_exit_group	int error_code					

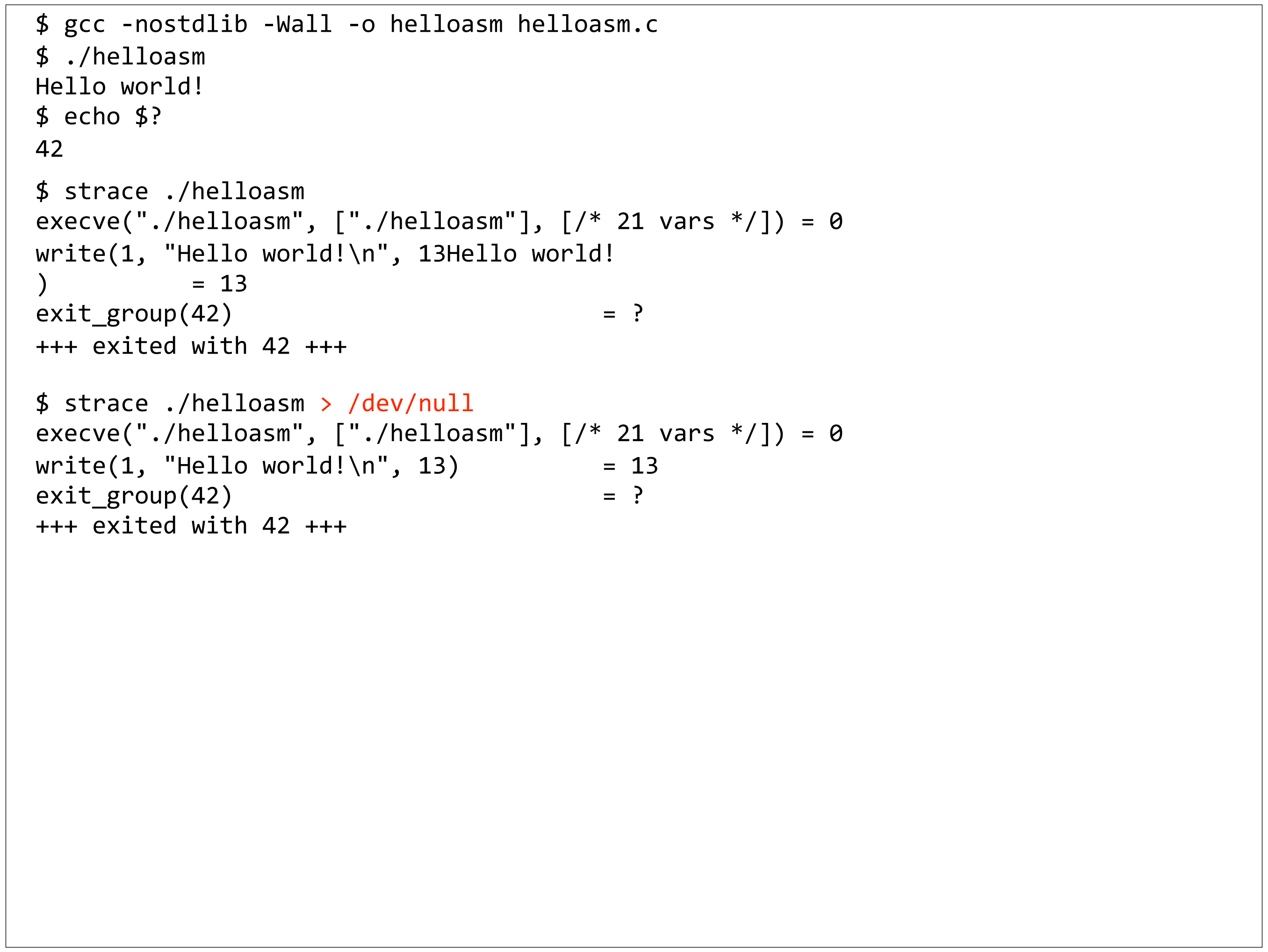
Finally we execute the last instruction our program makes in userland. This system call does not return control to the program, it exits back to the OS.

```
$ gcc -nostdlib -Wall -o helloasm helloasm.c
$ ./helloasm
Hello world!
$ echo $?
42
```

So now we compile and run our helloasm program. We pass gcc the -nostdlib argument to tell it that this is a stand-alone program and to not link with or initialize the standard library. Normally the C library initializes itself in the _start function and then calls the program's main() function to hand control over to you. Because we pass gcc the -nostdlib argument the operating system will call our _start function straight away when it loads our program.

```
$ gcc -nostdlib -Wall -o helloasm helloasm.c
$ ./helloasm
Hello world!
$ echo $?
42

$ strace ./helloasm
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

So now we’re going to run strace for the first time. The easiest way to run strace is just to pass it a program to run, like we do here. By default strace will write its output to STDERR. That line in red looks a little bit funny because the output of the helloasm program is being run together with the output from strace.

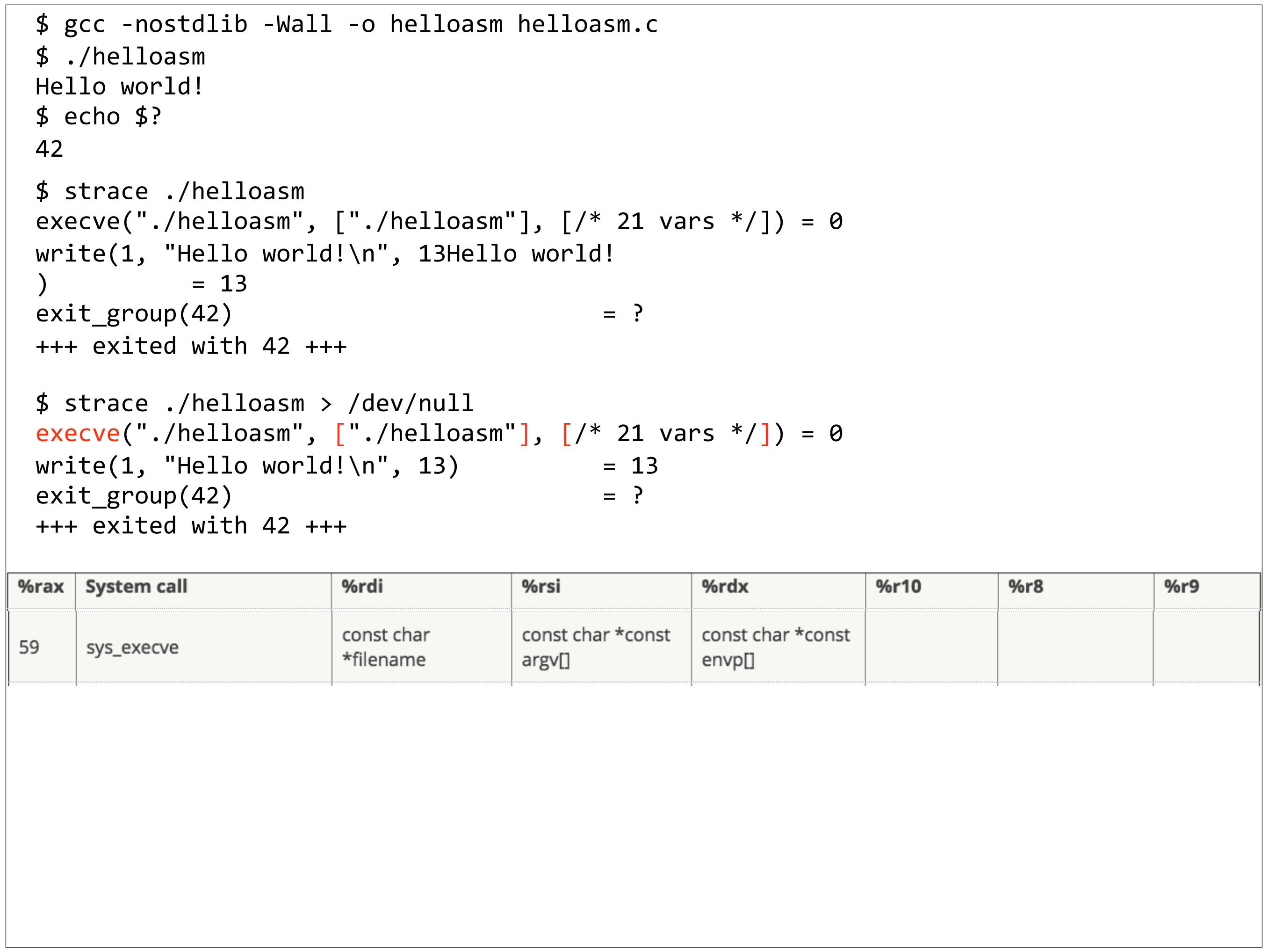


```
$ gcc -nostdlib -Wall -o helloasm helloasm.c
$ ./helloasm
Hello world!
$ echo $?
42

$ strace ./helloasm
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++

$ strace ./helloasm > /dev/null
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

We can clean this up a bit by piping the ouptut of helloasm to /dev/null, so we only see strace's ouput. Strace is doing what its supposed to do, it's showing you every system call a program makes with its arguments and return codes. You'll notice that the two system calls our program made are listed here along with their arguments, but there is an additional one listed here that we didn't make, execve, which I'll return to in a moment.

```
$ gcc -nostdlib -Wall -o helloasm helloasm.c
$ ./helloasm
Hello world!
$ echo $?
42

$ strace ./helloasm
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++

$ strace ./helloasm > /dev/null
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

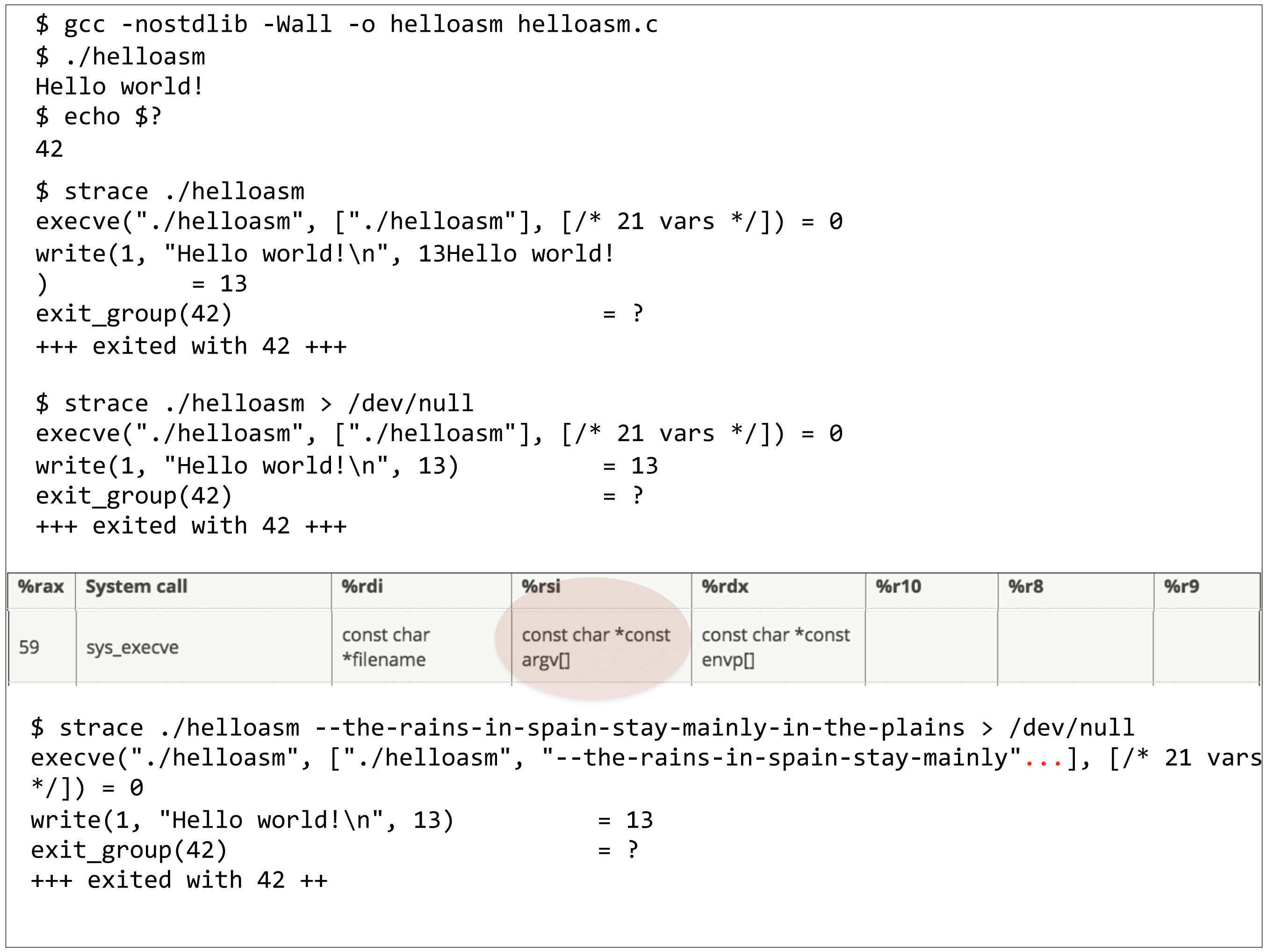
System calls return values to the program to let the programmer know if the system call was successful or not. In our helloasm program, the number 13 would be stored in the RAX register to indicate that the write system call successfully wrote 13 bytes to STDOUT. Now we didn't do anything with this value in our program, because what are we going to do if we failed to write our 'Hello world!' string to the terminal? Would we write another string to the terminal saying we were unable to write 'Hello world!' to the terminal'? But this is extremely handy for us admins trying to debug something. We can see the return codes from system calls that the programmer might be ignoring completely. Earlier I said the exit_group function never returns, so its return code is shown with a question mark.



```
$ gcc -nostdlib -Wall -o helloasm helloasm.c
$ ./helloasm
Hello world!
$ echo $?
42

$ strace ./helloasm
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13Hello world!
)         = 13
exit_group(42)                          = ?
+++ exited with 42 +++

$ strace ./helloasm > /dev/null
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			

Ok, so now let’s talk about the execve system call. This system call loads a new program into memory on top of your existing program and then runs it. This system call is technically made by strace, and not our program, but it shows exactly how strace invoked us. The first argument to execve is the name of the program to execute, the second and third arguments are an array of strings, denoted in the strace output by square brackets. The second argument to execve (the arg vector) tells helloasm how it was invoked, and if any arguments were passed to it. We didn’t provide any arguments to helloasm so only the name of the program is passed, this is how helloasm can learn its program name if it wants to.

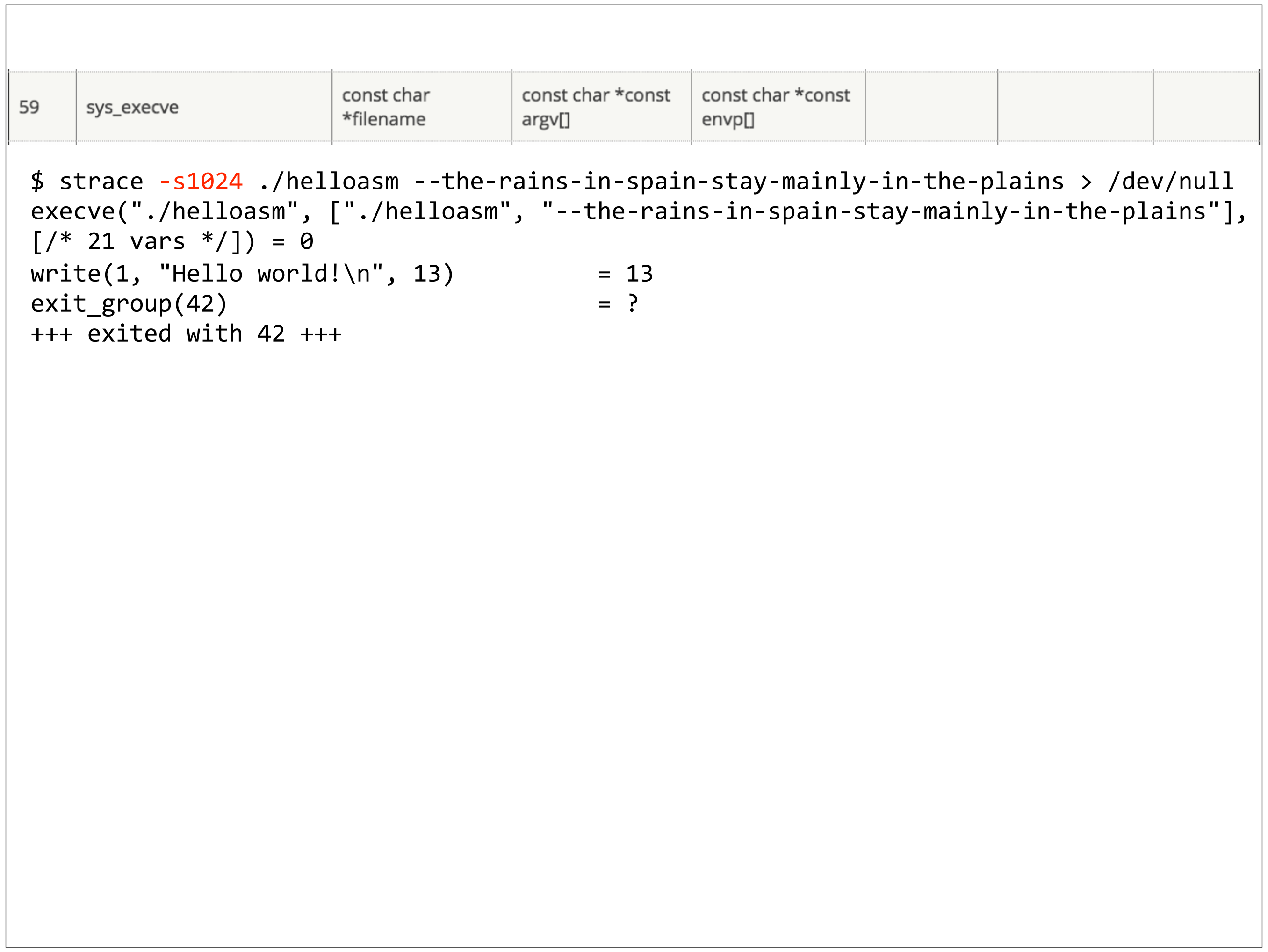


```
$ gcc -nostdlib -Wall -o helloasm helloasm.c
$ ./helloasm
Hello world!
$ echo $?
42

$ strace ./helloasm
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++

$ strace ./helloasm > /dev/null
execve("./helloasm", ["./helloasm"], [/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			

```
$ strace ./helloasm --the-rains-in-spain-stay-mainly-in-the-plains > /dev/null
execve("./helloasm", ["./helloasm", "--the-rains-in-spain-stay-mainly"...], [/* 21 vars
*/]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 ++
```

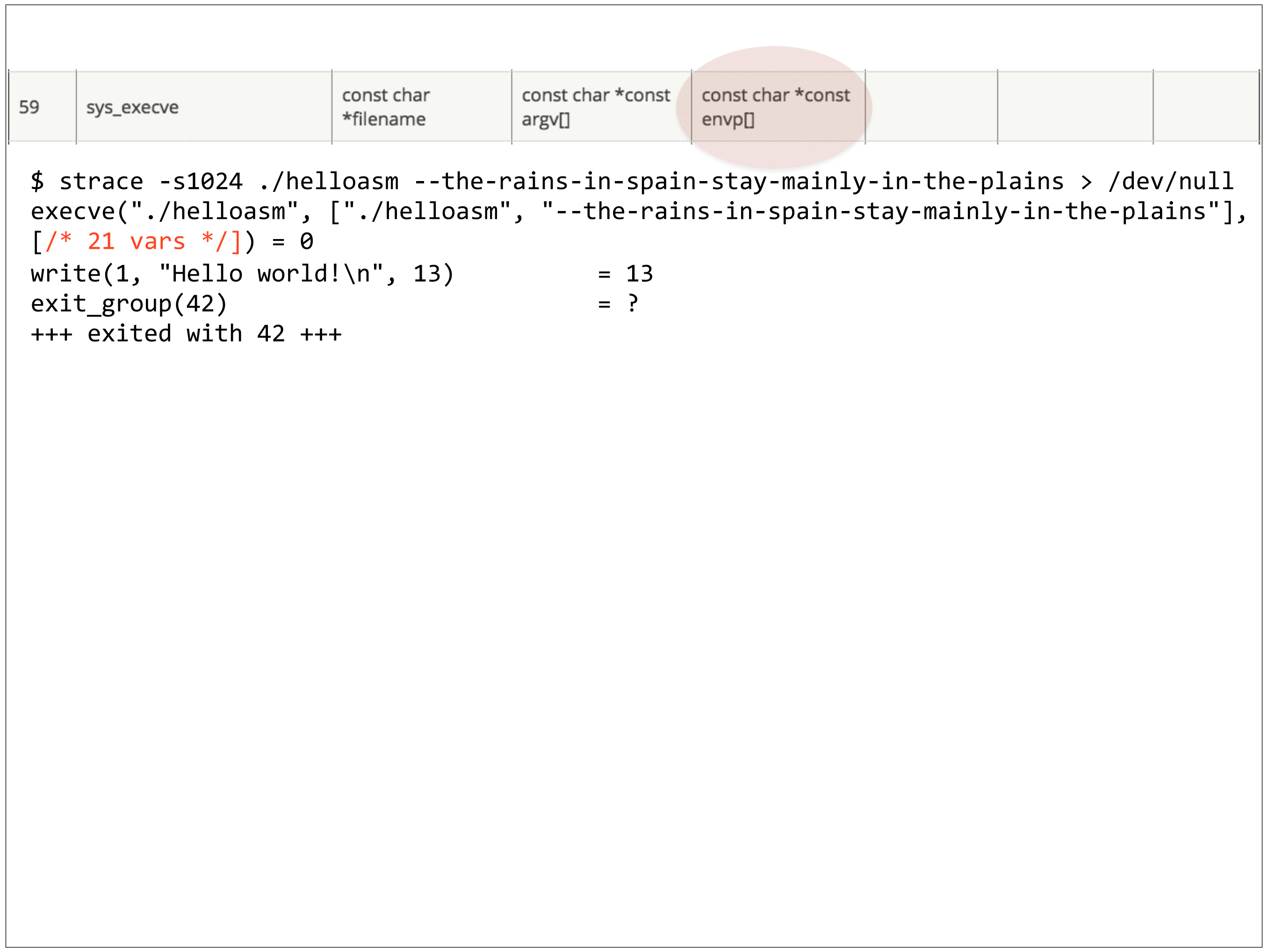
Let’s say if we told strace to trace helloasm with some arguments. Like this really long one here. You can see now that the second argument to execve passes along our giant argument string in the helloasm’s argument vector. Notice the three dots after ‘mainly’. By default strace will only show you up to 32 characters of any string that appears in system call arguments. The three dots let you know there is more that strace is not showing you. Strace will often bombard you with a ton of information about what a program is doing, so it tries to sumarize things a bit for you as to not be too overwhelming.



59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			

```
$ strace -s1024 ./helloasm --the-rains-in-spain-stay-mainly-in-the-plains > /dev/null
execve("./helloasm", ["./helloasm", "--the-rains-in-spain-stay-mainly-in-the-plains"],
[/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

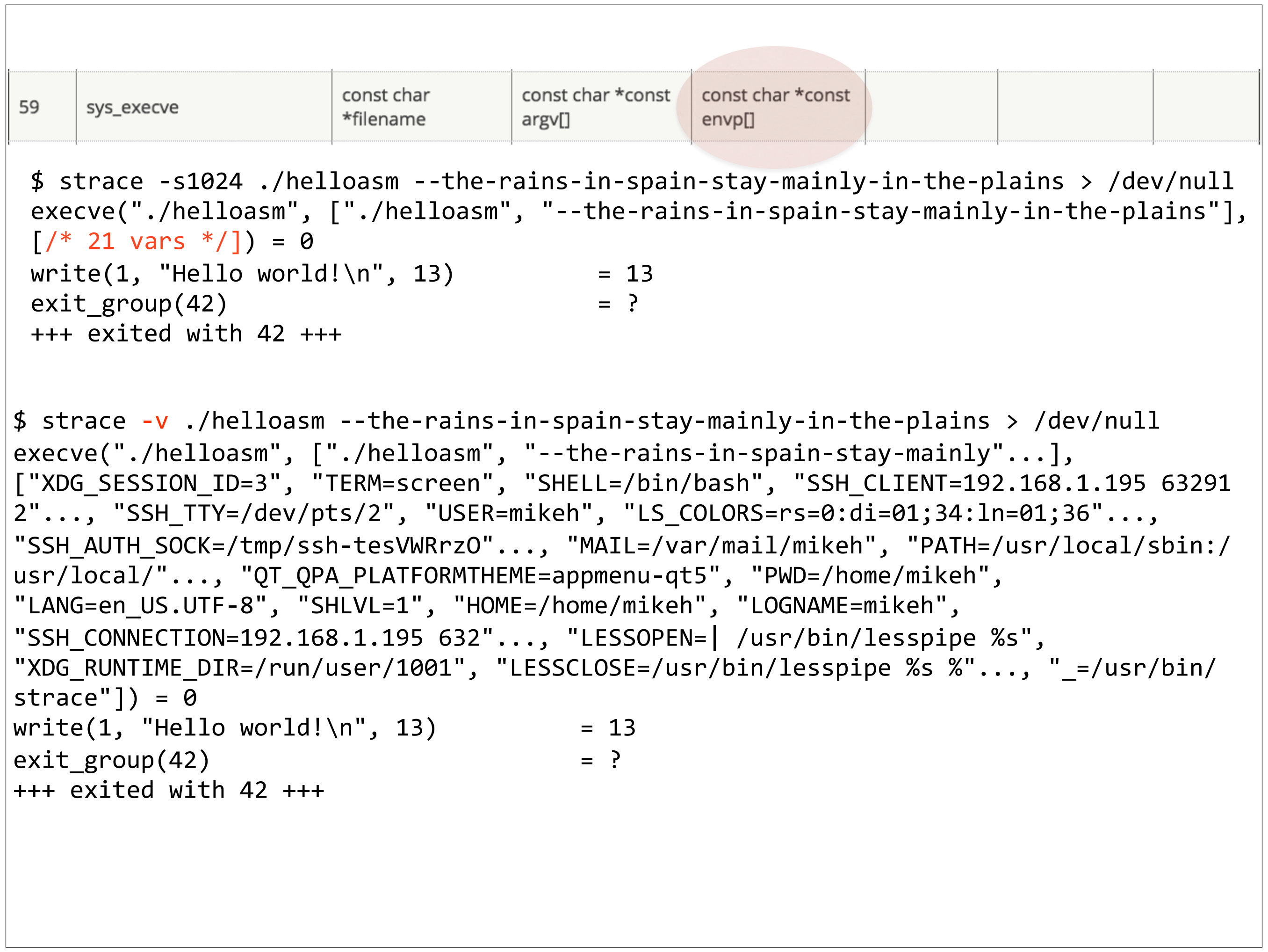
Sometimes you need to see more than 32 characters of a string to make sense of it. I often like to see several kilobytes of string data when I strace. When you're reading data from a socket or a file it's nice to have more context, and of course more things you can grep for. The -s option to strace lets you set a new limit for displayed string length. Now that we've set it to 1024 we can see the whole string in the argument, and there are no trailing dots to tell you it's truncated.



59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			

```
$ strace -s1024 ./helloasm --the-rains-in-spain-stay-mainly-in-the-plains > /dev/null
execve("./helloasm", ["./helloasm", "--the-rains-in-spain-stay-mainly-in-the-plains"],
[/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

Finally we come to the third argument of execve, which is an array of strings that represent all the environment variables to send to the new program that you're executing. Here strace knows that you probably don't want to see all of these, so it helpfully sumarizes for you that there are 21 environment variables that are being passed to helloasm, without actually showing them to you. If you are debugging an environment variable program though you might want to see them all.



59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			

```
$ strace -s1024 ./helloasm --the-rains-in-spain-stay-mainly-in-the-plains > /dev/null
execve("./helloasm", ["./helloasm", "--the-rains-in-spain-stay-mainly-in-the-plains"],
[/* 21 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

```
$ strace -v ./helloasm --the-rains-in-spain-stay-mainly-in-the-plains > /dev/null
execve("./helloasm", ["./helloasm", "--the-rains-in-spain-stay-mainly"...],
["XDG_SESSION_ID=3", "TERM=screen", "SHELL=/bin/bash", "SSH_CLIENT=192.168.1.195 63291
2"..., "SSH_TTY=/dev/pts/2", "USER=mikeh", "LS_COLORS=rs=0:di=01;34:ln=01;36"...,
"SSH_AUTH_SOCK=/tmp/ssh-tesVWRrzO"..., "MAIL=/var/mail/mikeh", "PATH=/usr/local/sbin:/
usr/local/"..., "QT_QPA_PLATFORMTHEME=appmenu-qt5", "PWD=/home/mikeh",
"LANG=en_US.UTF-8", "SHLVL=1", "HOME=/home/mikeh", "LOGNAME=mikeh",
"SSH_CONNECTION=192.168.1.195 632"..., "LESSOPEN=| /usr/bin/lesspipe %s",
"XDG_RUNTIME_DIR=/run/user/1001", "LESSCLOSE=/usr/bin/lesspipe %s %"..., "_=/usr/bin/
strace"]) = 0
write(1, "Hello world!\n", 13)          = 13
exit_group(42)                          = ?
+++ exited with 42 +++
```

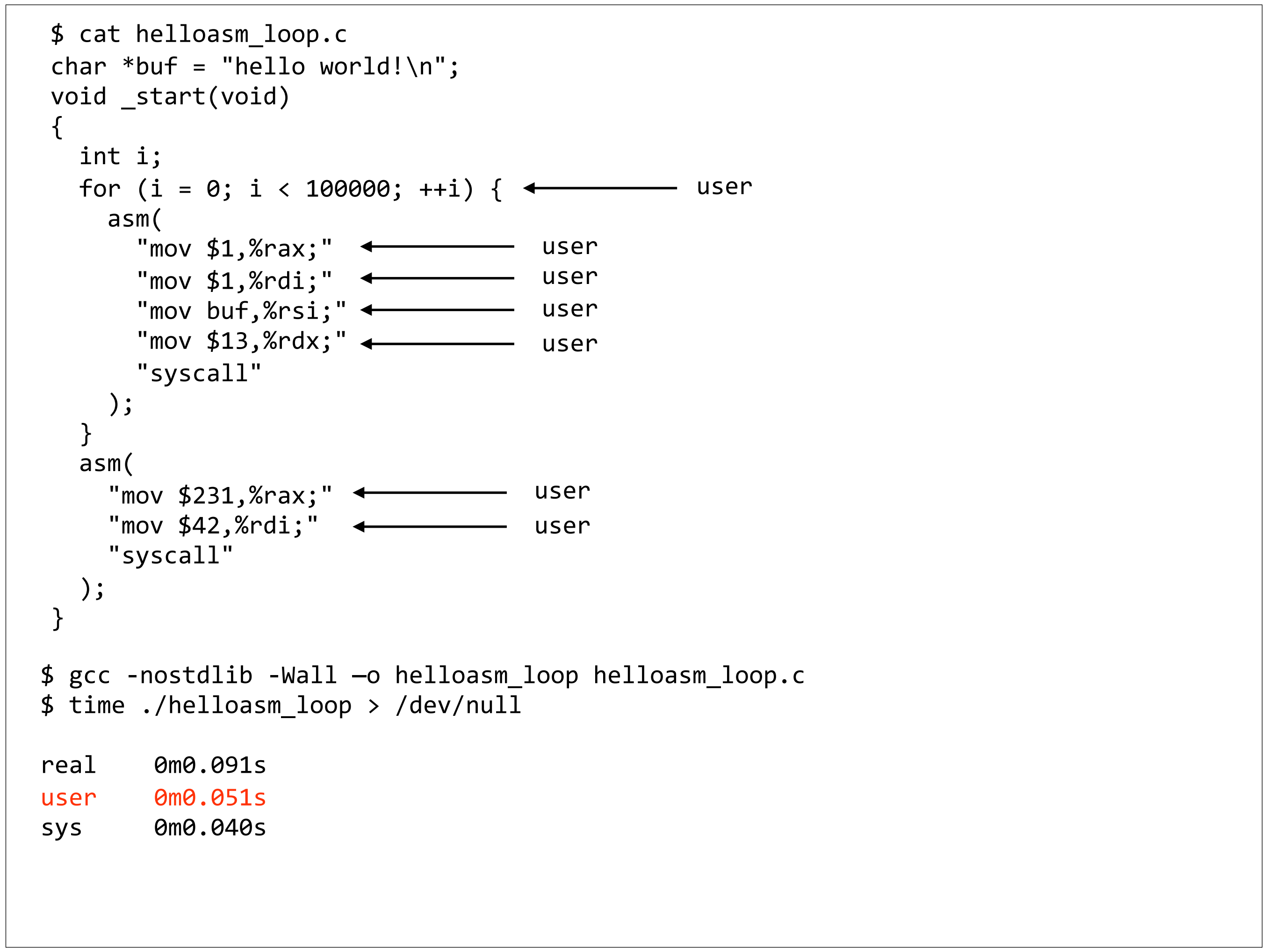
If you do for some reason want to see all of them, you can pass strace the -v option, which will tell strace not to do handy summaries of common system call arguments for you, but actually show you everything.

```
$ cat helloasm_loop.c
char *buf = "hello world!\n";
void _start(void)
{
  int i;
  for (i = 0; i < 100000; ++i) {
    asm(
      "mov $1,%rax;"
      "mov $1,%rdi;"
      "mov buf,%rsi;"
      "mov $13,%rdx;"
      "syscall"
    );
  }
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}
```

Ok, let’s modify the helloasm program just a bit so it prints ‘Hello world’ one hundred thousand times before exiting.

```
$ cat helloasm_loop.c
char *buf = "hello world!\n";
void _start(void)
{
  int i;
  for (i = 0; i < 100000; ++i) {
    asm(
      "mov $1,%rax;"
      "mov $1,%rdi;"
      "mov buf,%rsi;"
      "mov $13,%rdx;"
      "syscall"
    );
  }
  asm(
    "mov $231,%rax;"
    "mov $42,%rdi;"
    "syscall"
  );
}

$ gcc -nostdlib -Wall –o helloasm_loop helloasm_loop.c
$ time ./helloasm_loop > /dev/null

real    0m0.091s
user    0m0.051s
sys     0m0.040s
```

We compile it, and then run it using the time program which gives us some statistics on how long it took. We redirect STDOUT to /dev/null because we don't want to measure how long it takes to make 100k writes to the terminal. Now the time command breaks down the time your program spent running into three buckets. The real time, that is how much wall clock time has elapsed, the user time which is how much time your program spent in userland, and then sys time, which is how long your program spent in system calls, or inside the OS kernel.



```
$ cat helloasm_loop.c
char *buf = "hello world!\n";
void _start(void)
{
  int i;
  for (i = 0; i < 100000; ++i) {  ←———————  user
    asm(
      "mov $1,%rax;"  ←———————  user
      "mov $1,%rdi;"  ←———————  user
      "mov buf,%rsi;" ←———————  user
      "mov $13,%rdx;" ←———————  user
      "syscall"
    );
  }
  asm(
    "mov $231,%rax;"  ←———————  user
    "mov $42,%rdi;"   ←———————  user
    "syscall"
  );
}

$ gcc -nostdlib -Wall –o helloasm_loop helloasm_loop.c
$ time ./helloasm_loop > /dev/null

real    0m0.091s
user    0m0.051s
sys     0m0.040s
```

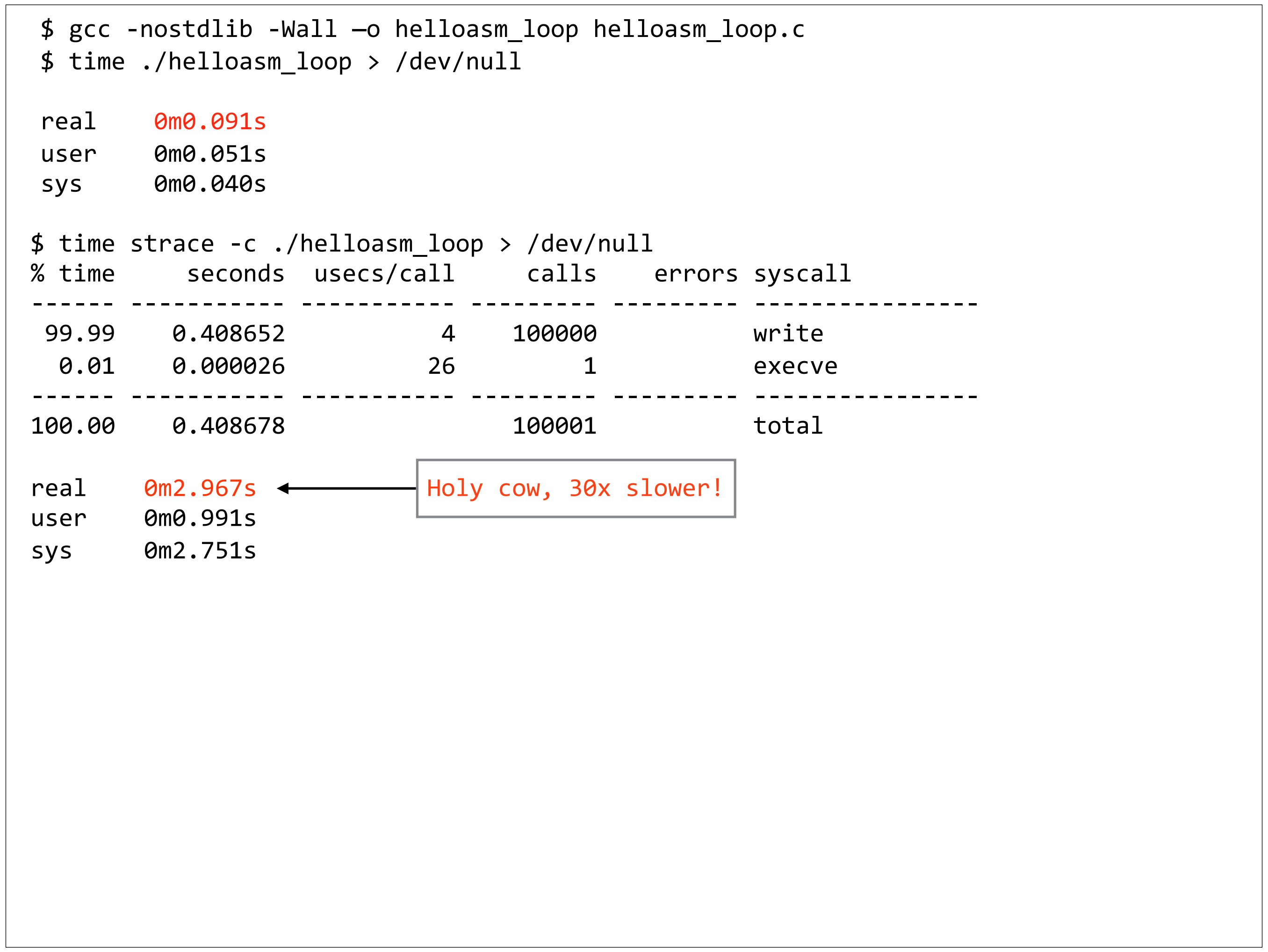
This program took under 1 tenth of a second to complete. More than 50% of that time was spent in userland, or non-kernel, non-privileged execution. Incrementing and testing the loop counter, and moving data into registers are all user CPU cycles. As a side note, some of you might notice some inefficiencies in the write() loop, as we needlessly load the same values into registers on every loop iteration that are preserved in between system calls, but optimization is another subject.

```
 $ cat helloasm_loop.c
 char *buf = "hello world!\n";
 void _start(void)
 {
   int i;
   for (i = 0; i < 100000; ++i) {
     asm(
       "mov $1,%rax;"
       "mov $1,%rdi;"
       "mov buf,%rsi;"
       "mov $13,%rdx;"
       "syscall"                   sys
     );
   }
   asm(
     "mov $231,%rax;"
     "mov $42,%rdi;"
     "syscall"                   sys
   );
 }

$ gcc -nostdlib -Wall —o helloasm_loop helloasm_loop.c
$ time ./helloasm_loop > /dev/null

real    0m0.091s
user    0m0.051s
sys     0m0.040s
```

The syscall executions themselves are what make up the 40ms or so of system time as reported by time.

```
 $ gcc -nostdlib -Wall —o helloasm_loop helloasm_loop.c
 $ time ./helloasm_loop > /dev/null

 real    0m0.091s
 user    0m0.051s
 sys     0m0.040s

$ time strace -c ./helloasm_loop > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.99    0.408652           4    100000           write
  0.01    0.000026          26         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.408678                100001           total

real    0m2.967s
user    0m0.991s
sys     0m2.751s
```

Next I wanted to show you the -c option to strace which prints out a nice summray of the system calls that the program makes along with some timing information. This is handy when you’re trying to get a high-level idea of what the program does without getting too mired in the details.

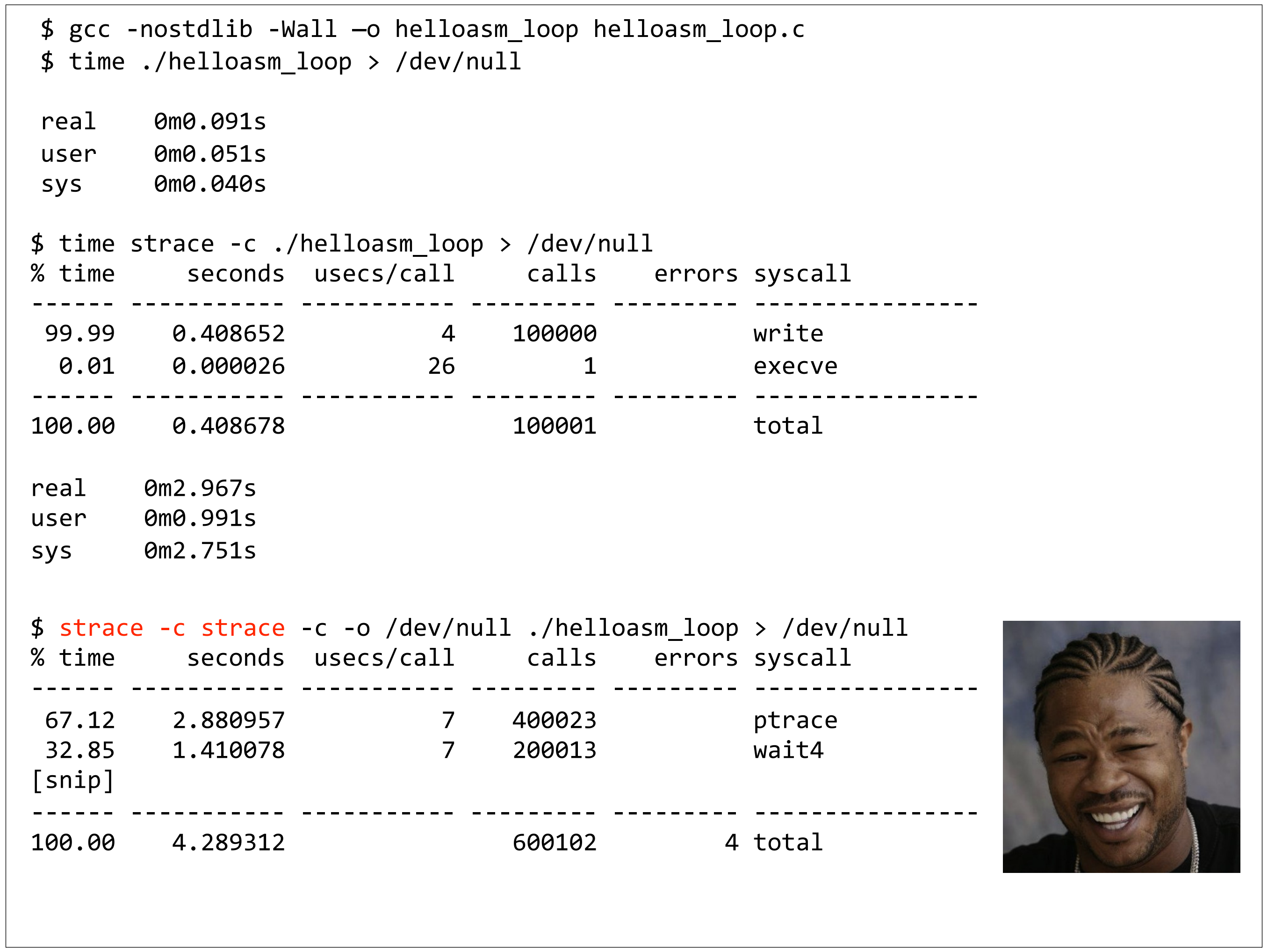


```
 $ gcc -nostdlib -Wall —o helloasm_loop helloasm_loop.c
 $ time ./helloasm_loop > /dev/null

 real    0m0.091s
 user    0m0.051s
 sys     0m0.040s

$ time strace -c ./helloasm_loop > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.99    0.408652           4    100000           write
  0.01    0.000026          26         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.408678                100001           total

real    0m2.967s ◄───────── Holy cow, 30x slower!
user    0m0.991s
sys     0m2.751s
```

I love strace, and it's extremely handy for debugging systems, but it does come at a high performance cost. You want to be very careful if you're using it to debug a production system. You don't want to attach strace to a running service and then go to lunch.

```
 $ gcc -nostdlib -Wall –o helloasm_loop helloasm_loop.c
 $ time ./helloasm_loop > /dev/null

 real    0m0.091s
 user    0m0.051s
 sys     0m0.040s

$ time strace -c ./helloasm_loop > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.99    0.408652           4    100000           write
  0.01    0.000026          26         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.408678                100001           total

real    0m2.967s
user    0m0.991s
sys     0m2.751s


$ strace -c strace -c -o /dev/null ./helloasm_loop > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 67.12    2.880957           7    400023           ptrace
 32.85    1.410078           7    200013           wait4
[snip]
------ ----------- ----------- --------- --------- ----------------
100.00    4.289312                600102         4 total
```

It’s worth noting, that for fun you can have strace trace strace. This gives you an idea of where strace is spending its time. This also demonstrates the -o option to strace, which tells it where to write its output instead of STDERR. We use this on the second copy of strace we run so that we only see the trace of strace, and not the trace output of helloasm. Now helloasm only makes 100k system calls, but tracing those requires 600k syscalls from strace, which is a 6x amplification. The ptrace() and wait4() system calls are also more expensive than the write() calls being traced. I’m pretty sure there is a joke to be made here somewhere.



```
 $ gcc -nostdlib -Wall —o helloasm_loop helloasm_loop.c
 $ time ./helloasm_loop > /dev/null

 real    0m0.091s
 user    0m0.051s
 sys     0m0.040s

$ time strace -c ./helloasm_loop > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.99    0.408652           4    100000           write
  0.01    0.000026          26         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.408678                100001           total

real    0m2.967s
user    0m0.991s
sys     0m2.751s


$ strace -c strace -c -o /dev/null ./helloasm_loop > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 67.12    2.880957           7    400023           ptrace
 32.85    1.410078           7    200013           wait4
[snip]
------ ----------- ----------- --------- --------- ----------------
100.00    4.289312                600102         4 total
```

Haha.

```
$ file ./helloasm
./helloasm: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), statically linked,
BuildID[sha1]=dd79b0166017038503af9ca84026a5c37fedc509, not stripped
```

So if you run ‘file’ on helloasm, you can see that it’s statically linked. It does not use the C runtime library, or any other library for that matter.

```
$ file ./helloasm
./helloasm: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), statically linked,
BuildID[sha1]=dd79b0166017038503af9ca84026a5c37fedc509, not stripped

$ ls -lh ./helloasm
-rwxrwxr-x 1 mikeh mikeh 5.3K Nov 10 14:06 ./helloasm
```

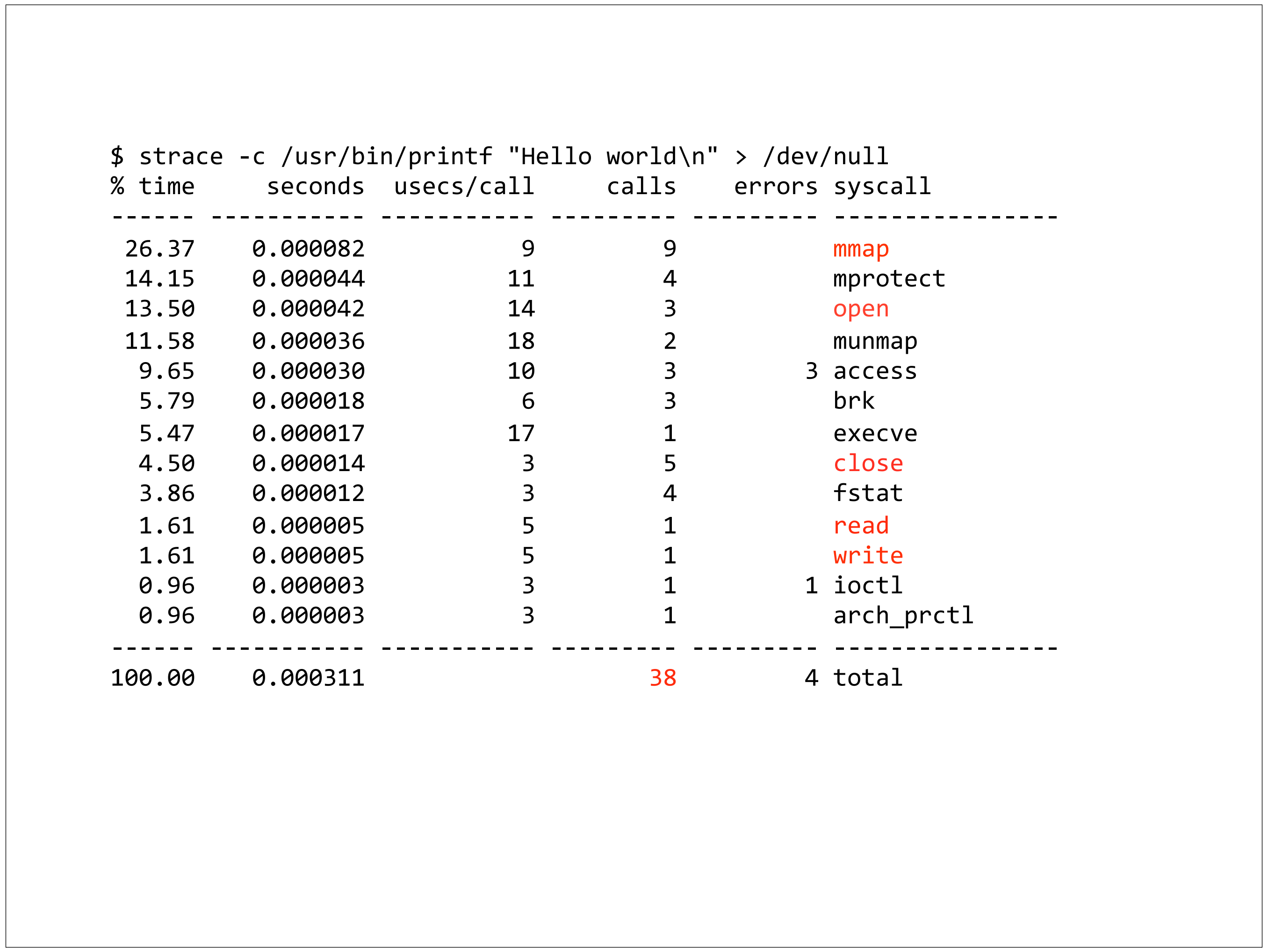
And as far as executables go, it’s pretty small, clocking in at just over 5k. That’s because it doesn’t have any of the boiler plate code inside it that enables dynamically loading code from external libraries.

```
$ file ./helloasm
./helloasm: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), statically linked,
BuildID[sha1]=dd79b0166017038503af9ca84026a5c37fedc509, not stripped

$ ls -lh ./helloasm
-rwxrwxr-x 1 mikeh mikeh 5.3K Nov 10 14:06 ./helloasm

$ ldd ./helloasm
        not a dynamic executable
```

And if you run ldd on it, which shows you which libraries a program requires, you can verify that it needs no libraries. You can take this helloasm binary and run it on any 64-bit linux system out there without worrying about library dependencies or compatibility issues.

```
$ file ./helloasm
./helloasm: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), statically linked,
BuildID[sha1]=dd79b0166017038503af9ca84026a5c37fedc509, not stripped

$ ls -lh ./helloasm
-rwxrwxr-x 1 mikeh mikeh 5.3K Nov 10 14:06 ./helloasm

$ ldd ./helloasm
        not a dynamic executable


$ file /usr/bin/printf
/usr/bin/printf: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=8b00e1432e9743b6df71197b2d4c668537dae159, stripped

$ ls -lh /usr/bin/printf
-rwxr-xr-x 1 root root 51K Mar 10  2016 /usr/bin/printf

$ ldd /usr/bin/printf
        linux-vdso.so.1 =>  (0x00007fff549e0000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb24f815000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fb24fbf6000)
```

Compare this to /usr/bin/printf, which is dynamicaly linked, and around 10x larger in size and links against the standard C library. It’s larger because it contains a bunch of code that initializes the C runtime library, as well as code that knows how to dynamically run code that lives external libraries.



```
$ strace -c /usr/bin/printf "Hello world\n" > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 26.37    0.000082           9         9           mmap
 14.15    0.000044          11         4           mprotect
 13.50    0.000042          14         3           open
 11.58    0.000036          18         2           munmap
  9.65    0.000030          10         3         3 access
  5.79    0.000018           6         3           brk
  5.47    0.000017          17         1           execve
  4.50    0.000014           3         5           close
  3.86    0.000012           3         4           fstat
  1.61    0.000005           5         1           read
  1.61    0.000005           5         1           write
  0.96    0.000003           3         1         1 ioctl
  0.96    0.000003           3         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000311                    38         4 total
```

Now if we run the strace system call summary on /usr/bin/printf with a “Hello World” argument, we can see that it makes a 38 system calls which is 35 more than our helloasm program, which does essentially the same thing. Let’s dig into what some of these system calls are doing. I’ve highlighted 5 system calls (mmap, open, close, read, and write) for us to look at.

```
$ strace -y -emmap,open,close,read,write /usr/bin/printf "Hello world\n" > /dev/null
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a4000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 105800, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f450788a000
close(3</etc/ld.so.cache>)              = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>
\0\1\0\0\0P \2\0\0\0\0\0"..., 832) = 832
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-
linux-gnu/libc-2.19.so>, 0) = 0x7f45072bf000
mmap(0x7f4507679000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3</lib/x86_64-linux-gnu/libc-2.19.so>, 0x1ba000) = 0x7f4507679000
mmap(0x7f450767f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f450767f000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507887000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 7216688, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) =
0x7f4506bdd000
close(3</usr/lib/locale/locale-archive>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a3000
write(1</dev/null>, "Hello world\n", 12) = 12
close(1</dev/null>)                     = 0
close(2</dev/pts/2>)                    = 0
+++ exited with 0 +++
```

So now we run strace with the -e option, to specify just the 5 system calls we want to trace. The other ones will be ignored.

```
$ strace -y -emmap,open,close,read,write /usr/bin/printf "Hello world\n" > /dev/null
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a4000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 105800, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f450788a000
close(3</etc/ld.so.cache>)              = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>
\0\1\0\0\0P \2\0\0\0\0\0"..., 832) = 832
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-
linux-gnu/libc-2.19.so>, 0) = 0x7f45072bf000
mmap(0x7f4507679000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3</lib/x86_64-linux-gnu/libc-2.19.so>, 0x1ba000) = 0x7f4507679000
mmap(0x7f450767f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f450767f000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507887000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 7216688, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) =
0x7f4506bdd000
close(3</usr/lib/locale/locale-archive>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a3000
write(1</dev/null>, "Hello world\n", 12) = 12
close(1</dev/null>)                     = 0
close(2</dev/pts/2>)                    = 0
+++ exited with 0 +++
```

Let’s look at these mmap calls. The mmap system call is used to map files into memory so your program can read a file like it would a block of memory. The second to last argument is the file descriptor to map into memory. Notice on these calls to mmap we’re using MMAP_ANONYNOUS and a -1 for the file descriptor. This tells mmap to simply allocate some memory, and not actually map a file into memory. The second argument is the size of the memory you want mapped. And in these examples they are 8k and 4k chunks..

```
$ strace -y -emmap,open,close,read,write /usr/bin/printf "Hello world\n" > /dev/null
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a4000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 105800, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f450788a000
close(3</etc/ld.so.cache>)              = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>
\0\1\0\0\0P \2\0\0\0\0\0"..., 832) = 832
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-
linux-gnu/libc-2.19.so>, 0) = 0x7f45072bf000
mmap(0x7f4507679000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3</lib/x86_64-linux-gnu/libc-2.19.so>, 0x1ba000) = 0x7f4507679000
mmap(0x7f450767f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f450767f000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507887000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 7216688, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) =
0x7f4506bdd000
close(3</usr/lib/locale/locale-archive>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a3000
write(1</dev/null>, "Hello world\n", 12) = 12
close(1</dev/null>)                     = 0
close(2</dev/pts/2>)                    = 0
+++ exited with 0 +++
```

This next block of system calls that we're looking at opens up the /etc/ld.so.cache file and maps it into memory. One thing to highlight here is that I'm using the -y option to strace which tries to map file descriptors to the files they represent. This makes it easier for you to figure out which file descriptor goes with which file. You can see the open call for ld.so.cache returns 3 (the first file descriptor available after STDIN, STDOUT, and STDERR) but it's nice to not have to keep track of that. The ld.so.cache file contains information about where on the filesystem dynamic libraries live. On Linux, the 'ldconfig' program updates this file.

```
$ strace -y -emmap,open,close,read,write /usr/bin/printf "Hello world\n" > /dev/null
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a4000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 105800, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f450788a000
close(3</etc/ld.so.cache>)              = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>
\0\1\0\0\0P \2\0\0\0\0\0"..., 832) = 832
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-
linux-gnu/libc-2.19.so>, 0) = 0x7f45072bf000
mmap(0x7f4507679000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3</lib/x86_64-linux-gnu/libc-2.19.so>, 0x1ba000) = 0x7f4507679000
mmap(0x7f450767f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f450767f000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507887000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 7216688, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) =
0x7f4506bdd000
close(3</usr/lib/locale/locale-archive>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a3000
write(1</dev/null>, "Hello world\n", 12) = 12
close(1</dev/null>)                     = 0
close(2</dev/pts/2>)                    = 0
+++ exited with 0 +++
```

This next section has the system calls that the dynamic linking code uses to load the libc library into memory. It opens the libary, reads the ELF header and some other data, and then maps some of the library's code into memory. You can tell the first mmap is code because of the PROT_EXEC flag which tells mmap that these map data pages may be executed. The second mmap maps some of the C library's data into memory, and the final mmap just allocates more anonymous memory.

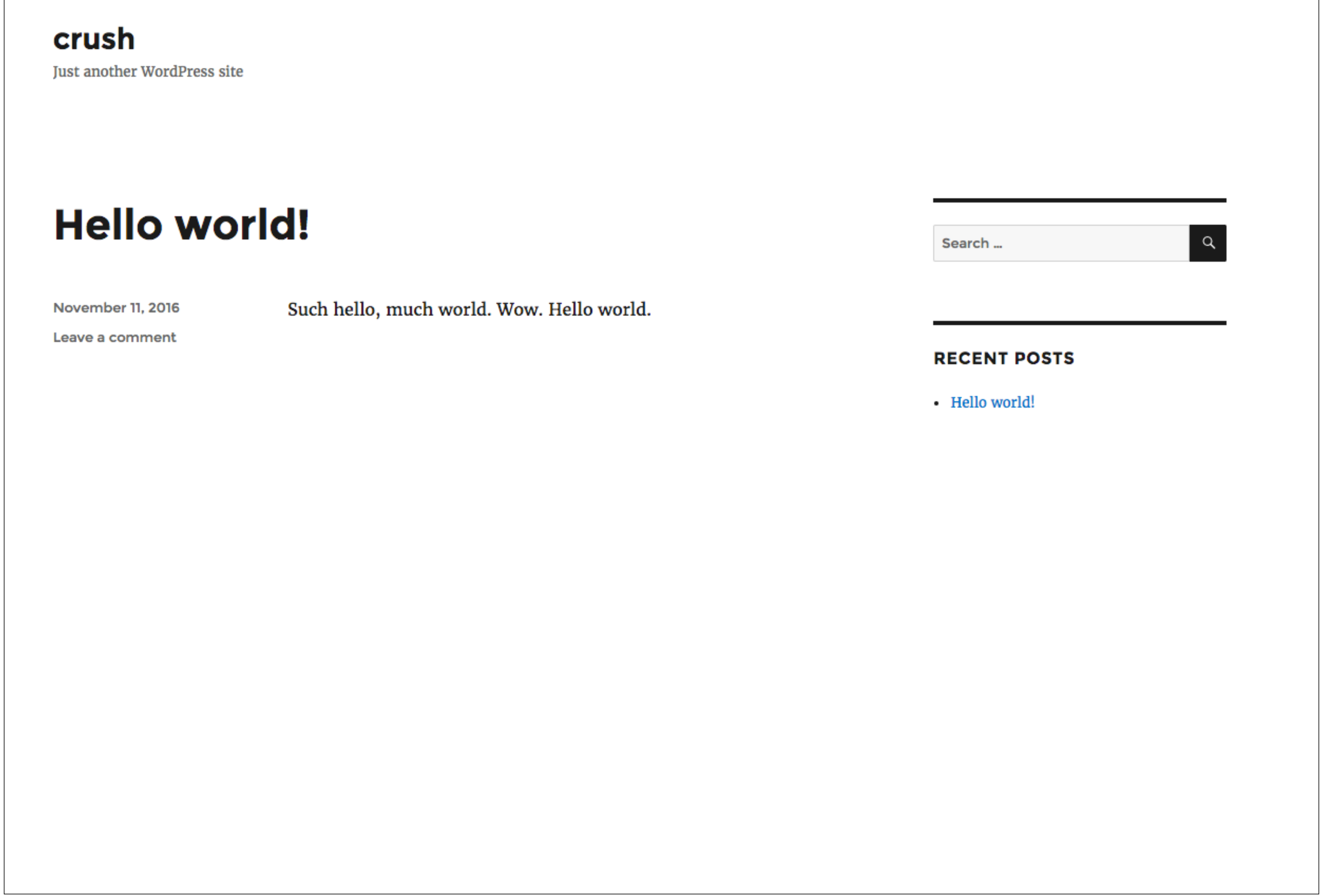
```
$ strace -y -emmap,open,close,read,write /usr/bin/printf "Hello world\n" > /dev/null
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a4000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 105800, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f450788a000
close(3</etc/ld.so.cache>)             = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>
\0\1\0\0\0P \2\0\0\0\0\0"..., 832) = 832
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-
linux-gnu/libc-2.19.so>, 0) = 0x7f45072bf000
mmap(0x7f4507679000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3</lib/x86_64-linux-gnu/libc-2.19.so>, 0x1ba000) = 0x7f4507679000
mmap(0x7f450767f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f450767f000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507887000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 7216688, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) =
0x7f4506bdd000
close(3</usr/lib/locale/locale-archive>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a3000
write(1</dev/null>, "Hello world\n", 12) = 12
close(1</dev/null>)                    = 0
close(2</dev/pts/2>)                   = 0
+++ exited with 0 +++
```

This group of system calls mmaps all 7MB of the local-archive file into memory. This file is used by the C runtime library for so it can print locale speicfic strings for things like the days of the week. Some distributions of linux have locale-archive files that are hundreds of megabytes in size, which would be accounted for in your programs ‘RSS’ or resident set size, which led to some confusion.

```
$ strace -y -emmap,open,close,read,write /usr/bin/printf "Hello world\n" > /dev/null
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a4000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 105800, PROT_READ, MAP_PRIVATE, 3</etc/ld.so.cache>, 0) = 0x7f450788a000
close(3</etc/ld.so.cache>)              = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3</lib/x86_64-linux-gnu/libc-2.19.so>, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>
\0\1\0\0\0P \2\0\0\0\0\0"..., 832) = 832
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib/x86_64-
linux-gnu/libc-2.19.so>, 0) = 0x7f45072bf000
mmap(0x7f4507679000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3</lib/x86_64-linux-gnu/libc-2.19.so>, 0x1ba000) = 0x7f4507679000
mmap(0x7f450767f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f450767f000
close(3</lib/x86_64-linux-gnu/libc-2.19.so>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f4507887000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 7216688, PROT_READ, MAP_PRIVATE, 3</usr/lib/locale/locale-archive>, 0) =
0x7f4506bdd000
close(3</usr/lib/locale/locale-archive>) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f45078a3000
write(1</dev/null>, "Hello world\n", 12) = 12
close(1</dev/null>)                    = 0
close(2</dev/pts/2>)                   = 0
+++ exited with 0 +++
```

And finally we have the actual system calls that do the work we asked the program to do. We write “Hello World” to our STDOUT, which you can see thanks to the -y option has been mapped to /dev/null, then we clean up and close our remaining open file descriptors.

The World's Most Boring WordPress Site


crush

Just another WordPress site

Hello world!

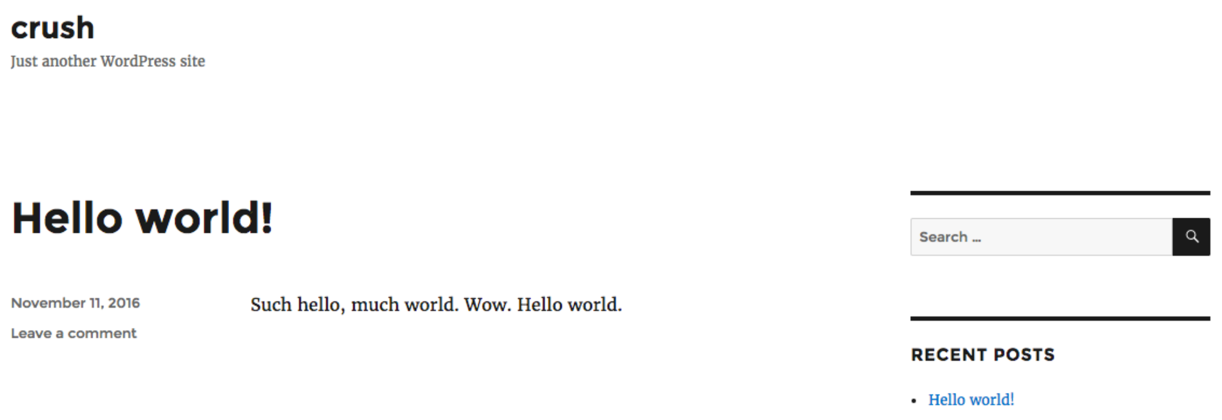
November 11, 2016

Leave a comment

Such hello, much world. Wow. Hello world.

Next I'm going to show you how to use strace to trace a running program, like a webserver. Here I have the world's most boring word press site, which I've set up for this talk.

The World’s Most Boring WordPress Site


crush

Just another WordPress site

Hello world!

November 11, 2016

Leave a comment

Such hello, much world. Wow. Hello world.

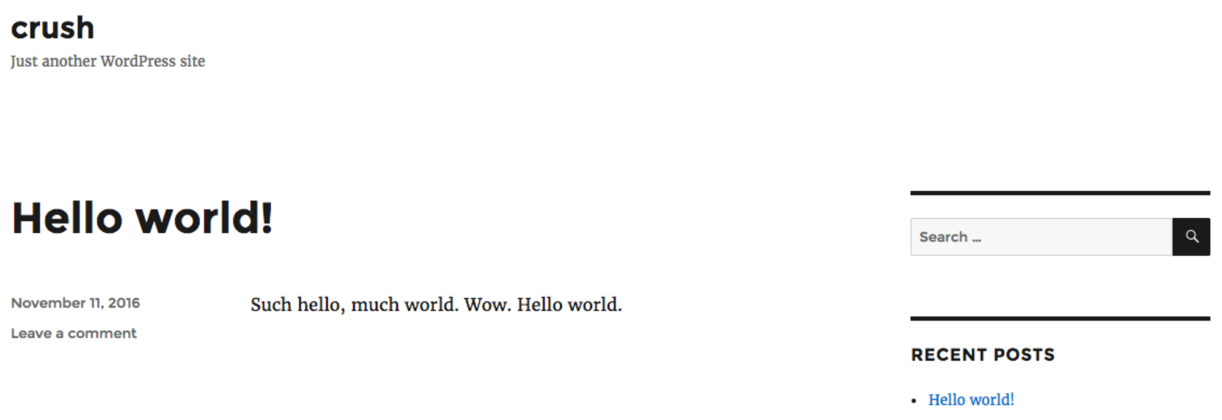
RECENT POSTS

- Hello world!

```
# curl -s http://localhost/wordpress/ | grep Wow
                <p>Such hello, much world. Wow. Hello world.</p>
```

For this example we need to do our work logged into one of the hosts that serves up this website, and we need to be root. So if we run curl against localhost/wordpress you can see that we’re talking to the webserver and we get the front page of the wordpress site.

The World's Most Boring WordPress Site


crush

Just another WordPress site

Hello world!

November 11, 2016

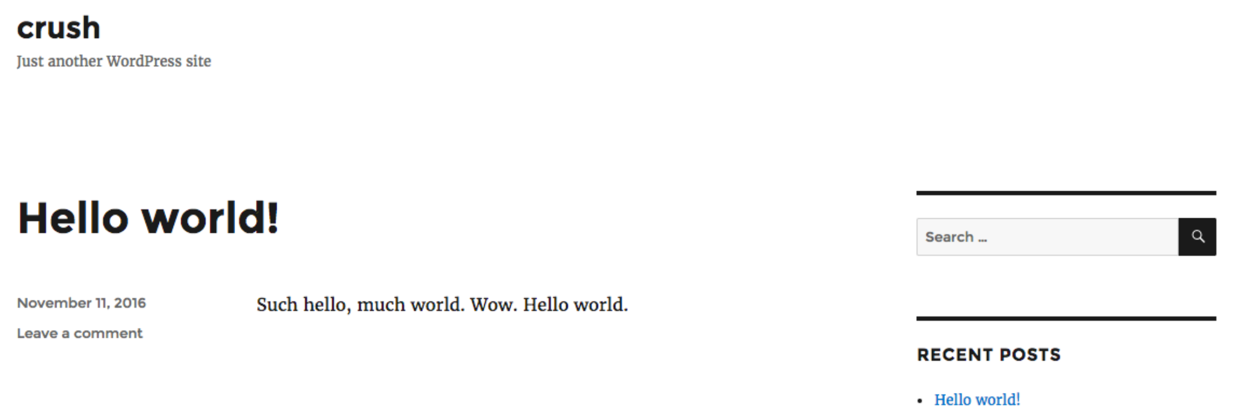
Leave a comment

Such hello, much world. Wow. Hello world.

```
# curl -s http://localhost/wordpress/ | grep Wow
                <p>Such hello, much world. Wow. Hello world.</p>

# ps auxw | grep nginx..worker | grep -v grep
www-data 13711  0.0  0.0  86524  2716 ?        S    19:36   0:00 nginx: worker process
www-data 13712  0.0  0.0  86180  1788 ?        S    19:36   0:00 nginx: worker process
www-data 13713  0.0  0.0  86180  1788 ?        S    19:36   0:00 nginx: worker process
www-data 13714  0.0  0.0  86180  1788 ?        S    19:36   0:00 nginx: worker process
```

If we run ps and grep for nginx workers we can see that for this site we have the default four workers configured, each of them running as a stand alone process.

The World’s Most Boring WordPress Site



crush

Just another WordPress site

Hello world!

November 11, 2016

Leave a comment

Such hello, much world. Wow. Hello world.

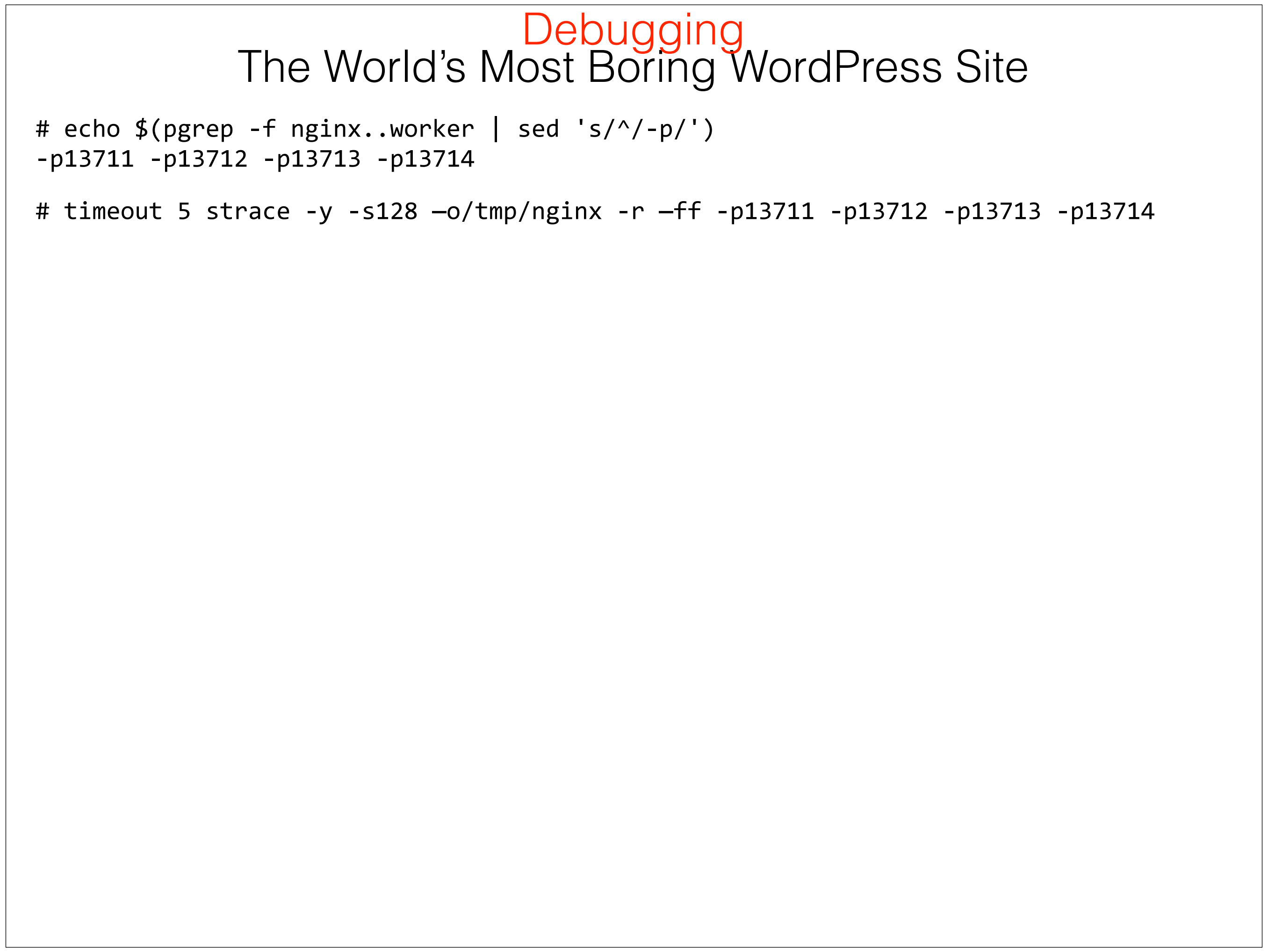
Search …

RECENT POSTS

- Hello world!

```
# curl -s http://localhost/wordpress/ | grep Wow
                <p>Such hello, much world. Wow. Hello world.</p>

# ps auxw | grep nginx..worker | grep -v grep
www-data 13711  0.0  0.0  86524  2716 ?        S    19:36   0:00 nginx: worker process
www-data 13712  0.0  0.0  86180  1788 ?        S    19:36   0:00 nginx: worker process
www-data 13713  0.0  0.0  86180  1788 ?        S    19:36   0:00 nginx: worker process
www-data 13714  0.0  0.0  86180  1788 ?        S    19:36   0:00 nginx: worker process

# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714
```

We can use pgrep in conjunction with sed in order to get the pids of these worker processes. I add the -p to each one of these pids using sed, but I’ll get to that later.

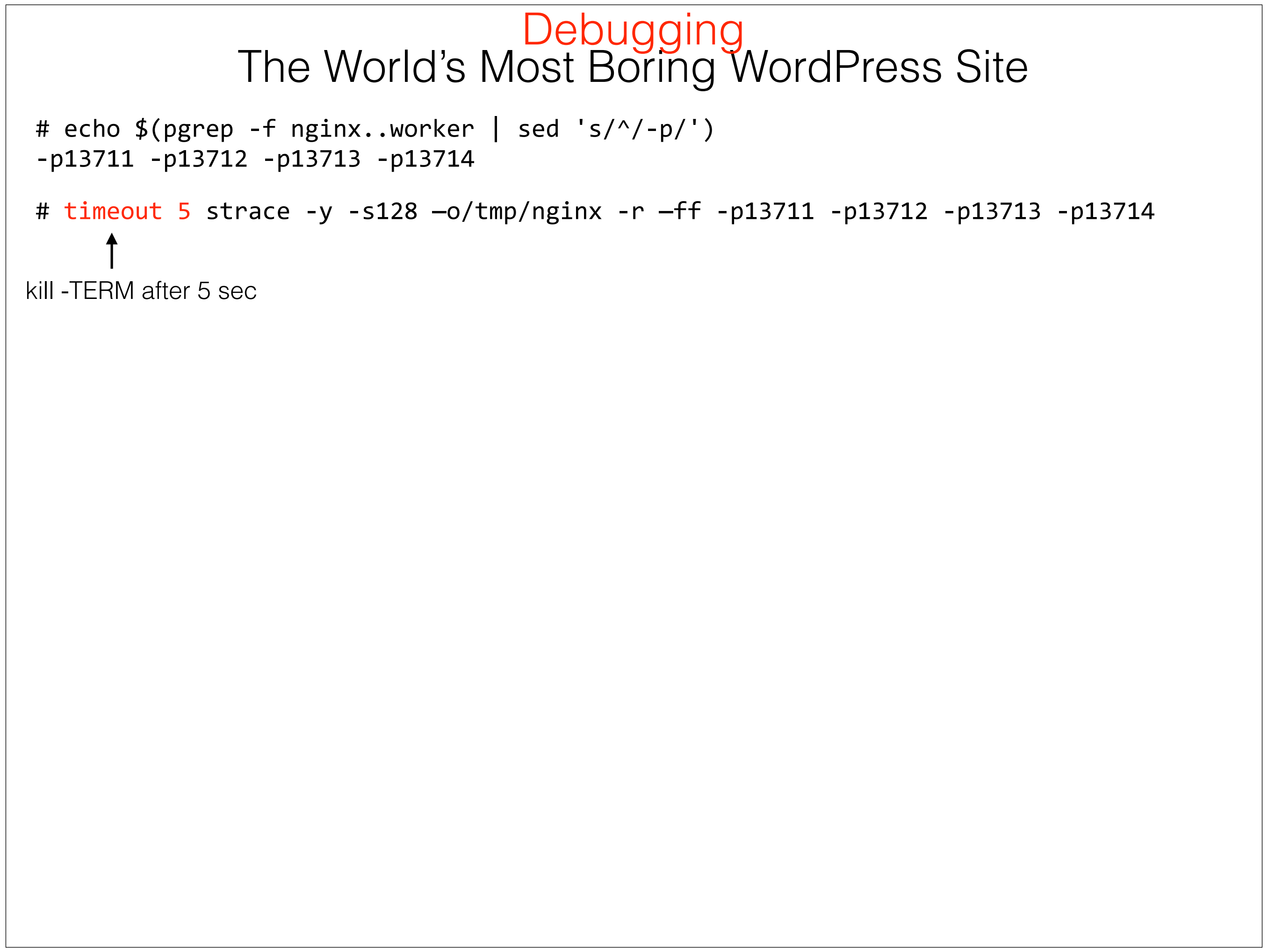


Debugging

The World's Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

Here is my strace command line that we're going to use to figure out what is happening with the nginx workers.

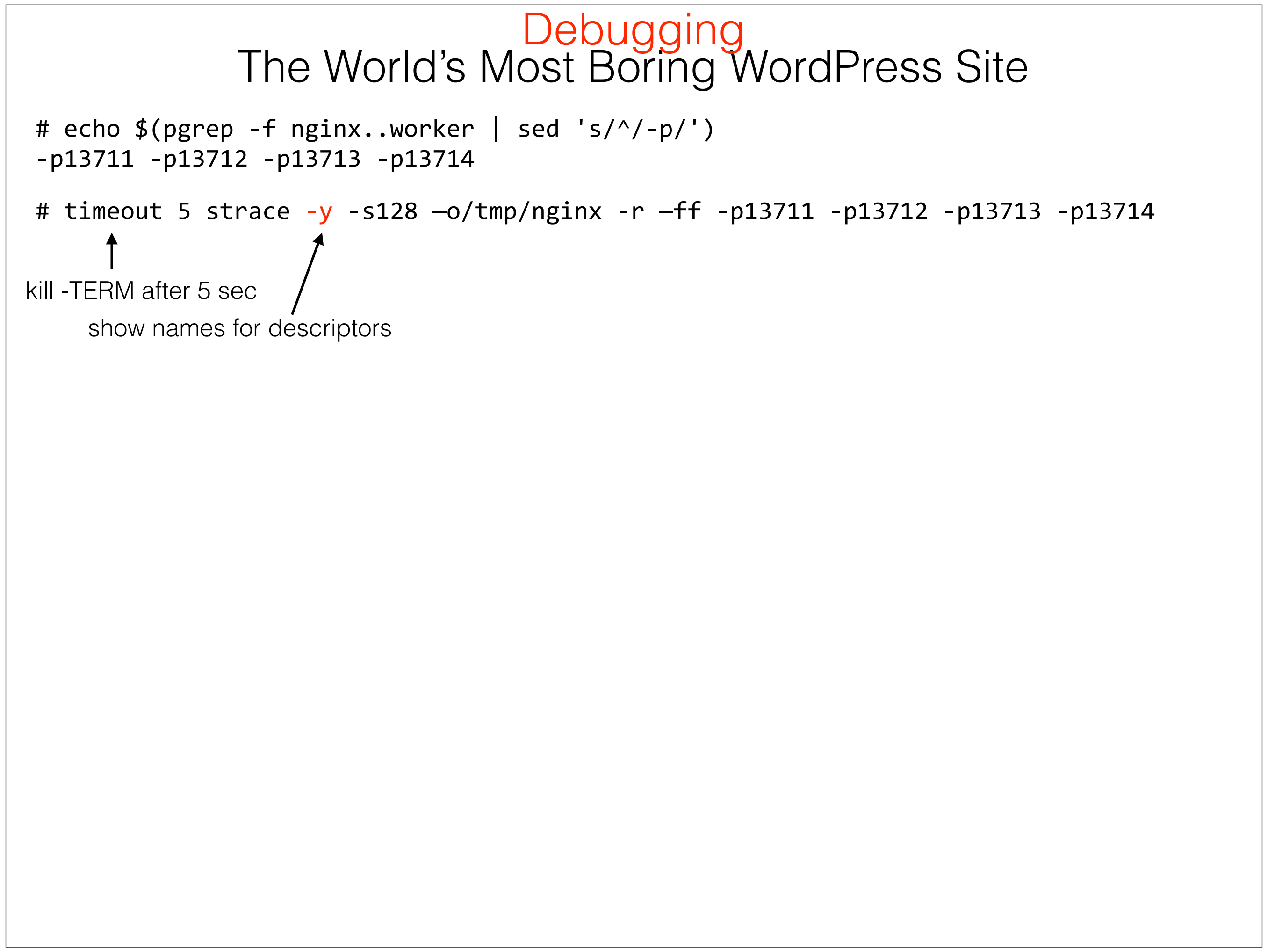


Debugging
The World’s Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec

This first part of the command uses the UNIX timeout program, which will send a TERM signal to the program you ask it to run after 5 seconds. We don’t want to run strace for too long, remember the 30x performance impact is can have.



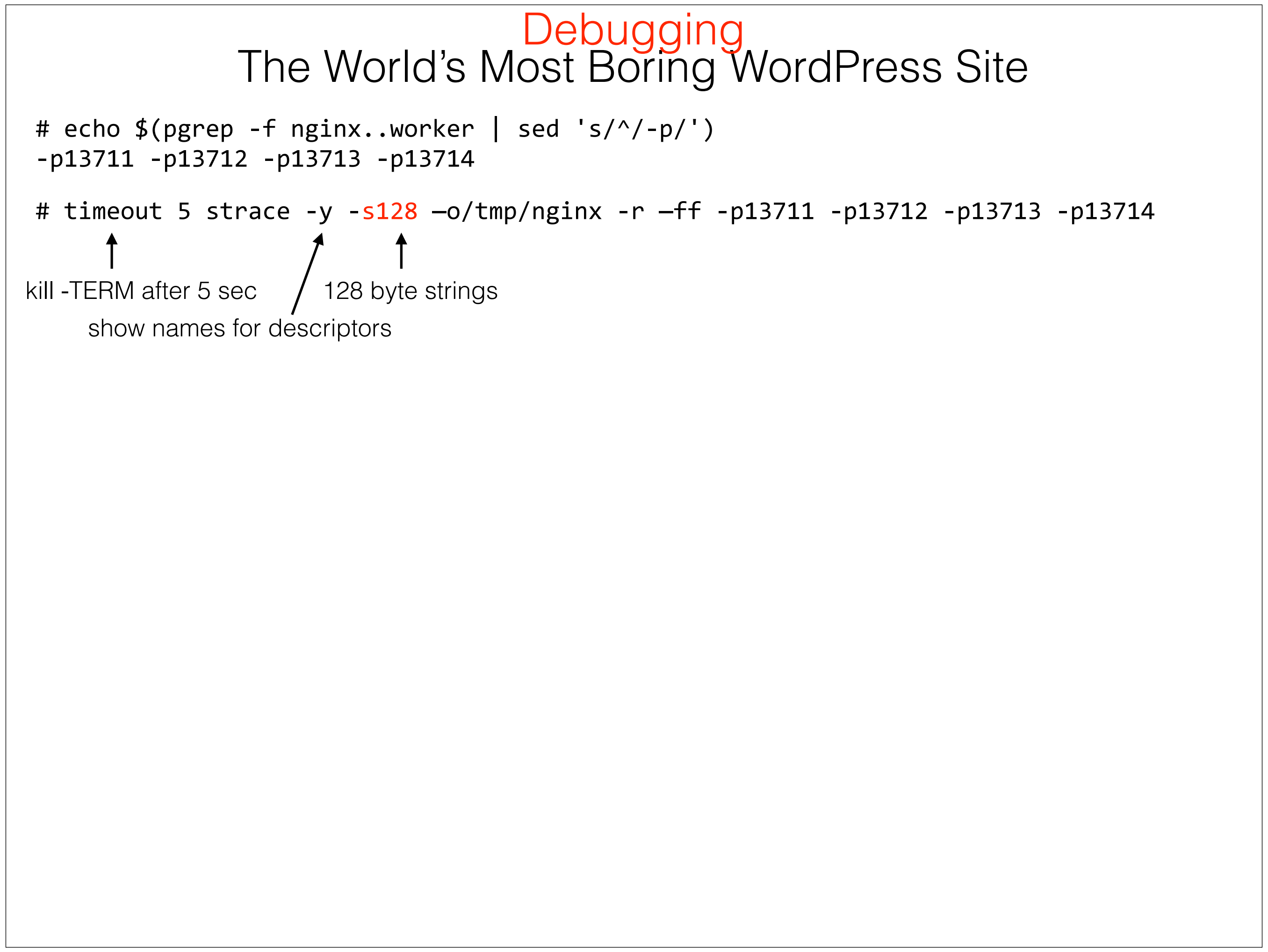
Debugging
The World's Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec

show names for descriptors

This next argument we talked about earlier, it shows filenames associated with descriptors, which is pretty handy.



Debugging
The World’s Most Boring WordPress Site

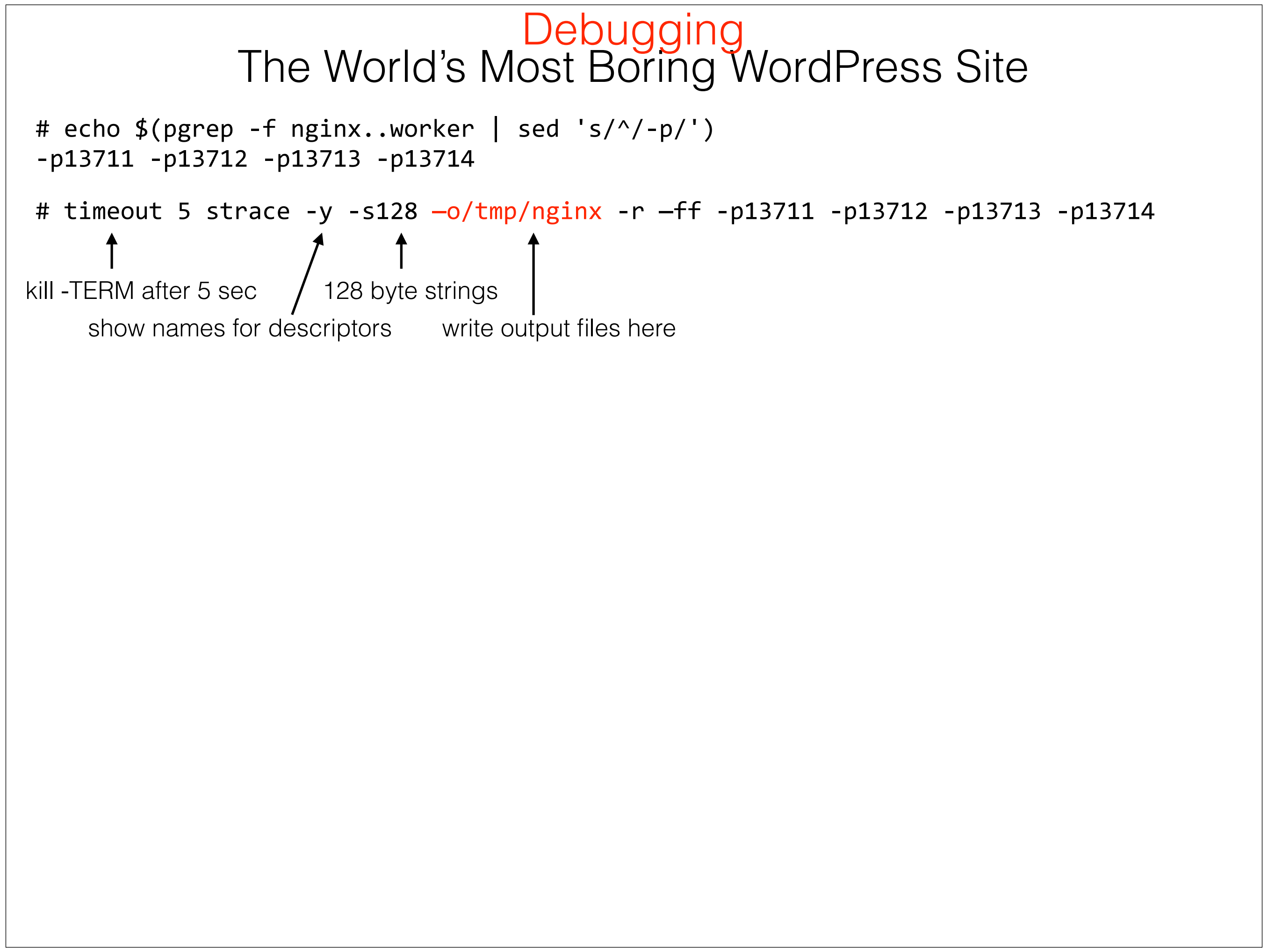
```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec

128 byte strings

show names for descriptors

This lets us look at 128 bytes of strings that are used in system calls, this gives us a bit of context so we can figure out what’s going on.

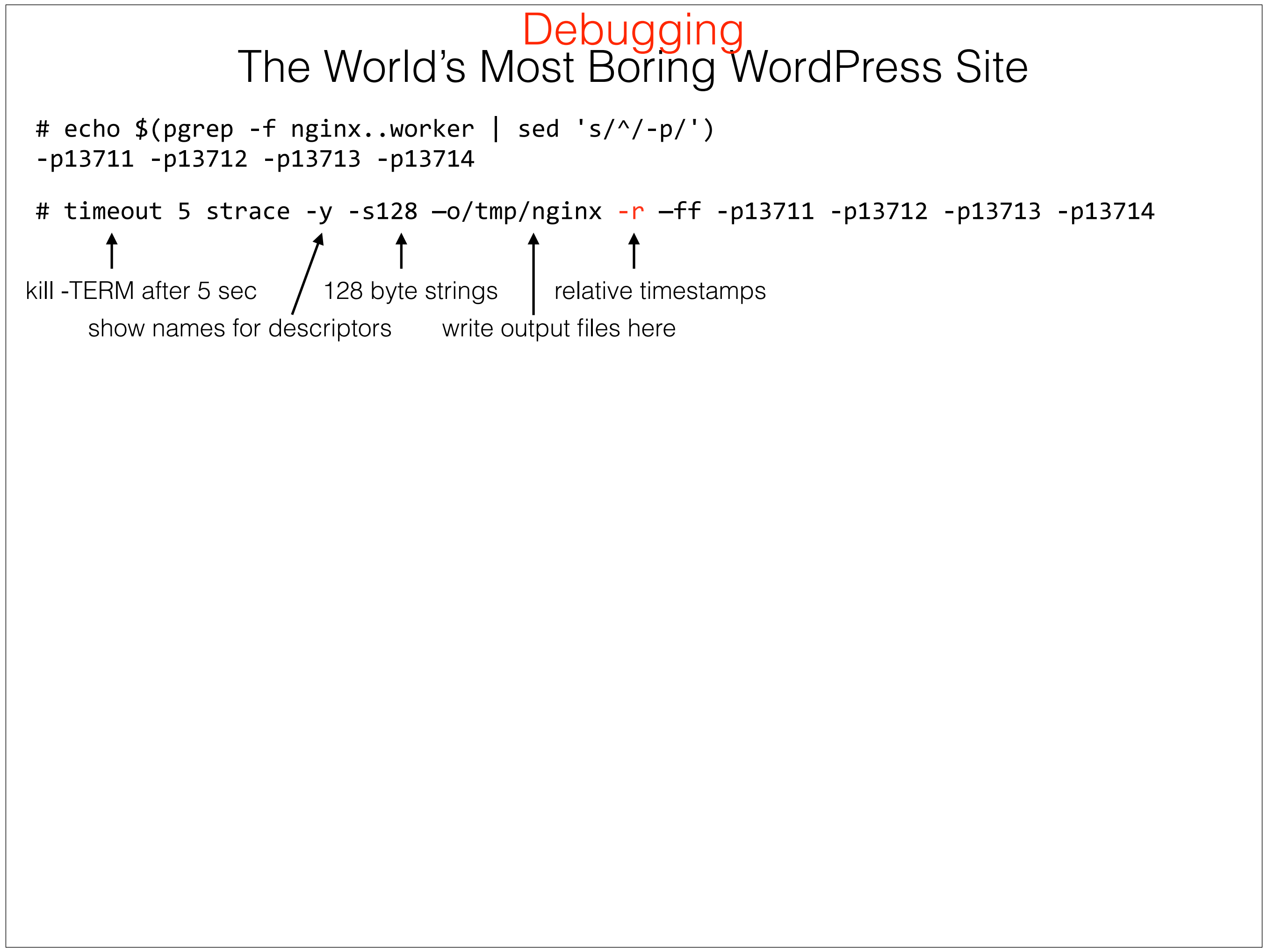


Debugging
The World’s Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec
show names for descriptors
128 byte strings
write output files here

The -o option we discussed earlier, it tells strace where to write its output files. We don’t want them going to STDERR in this case.



Debugging
The World’s Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec

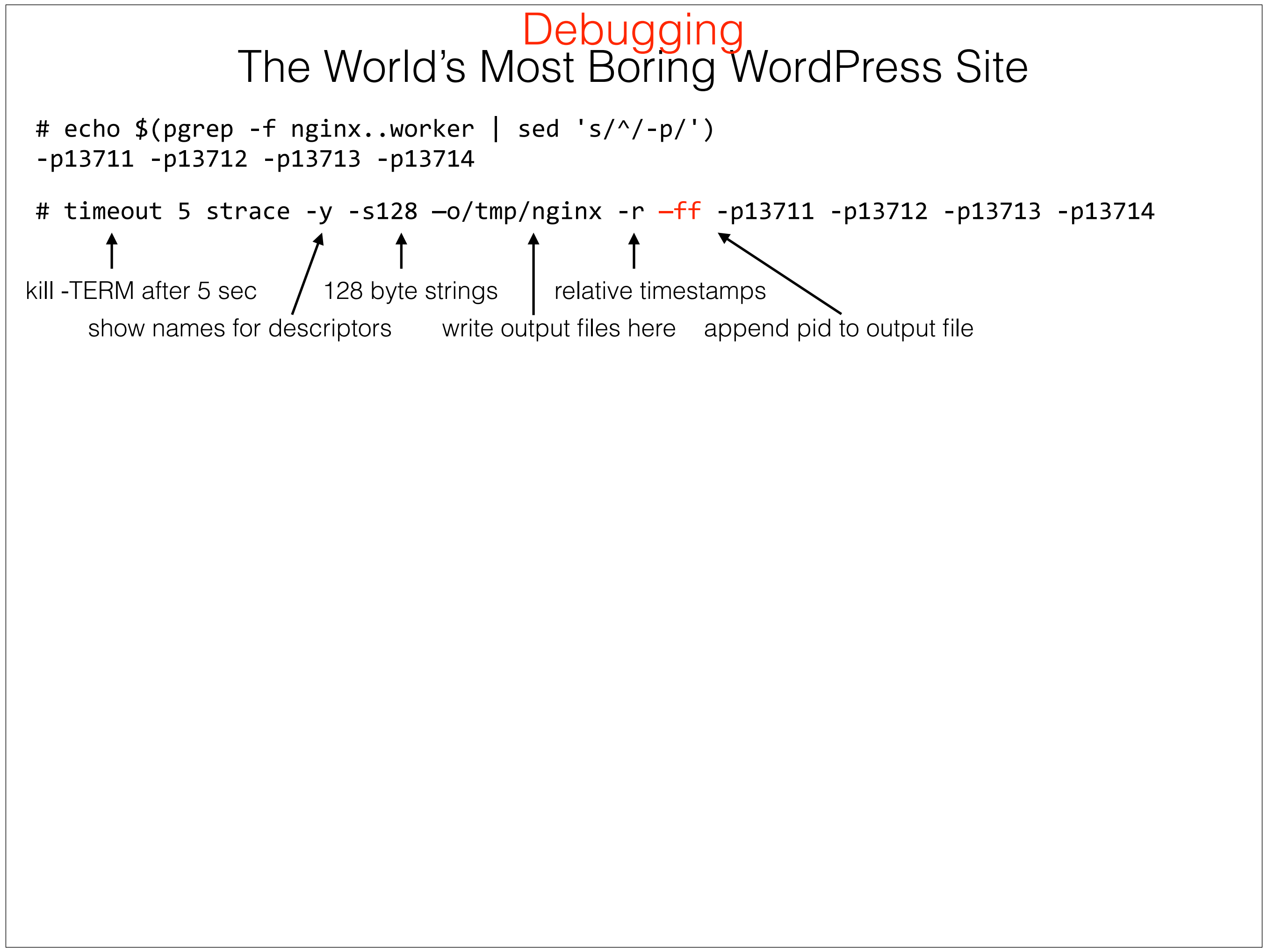
128 byte strings

relative timestamps

show names for descriptors

write output files here

The -r option is new. This tells strace to generate relative timestamps for each system call, so you can get an idea of how long each system call took. This is helpful for figuring out if certain system calls are really slowing down our application.



Debugging

The World’s Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec

show names for descriptors

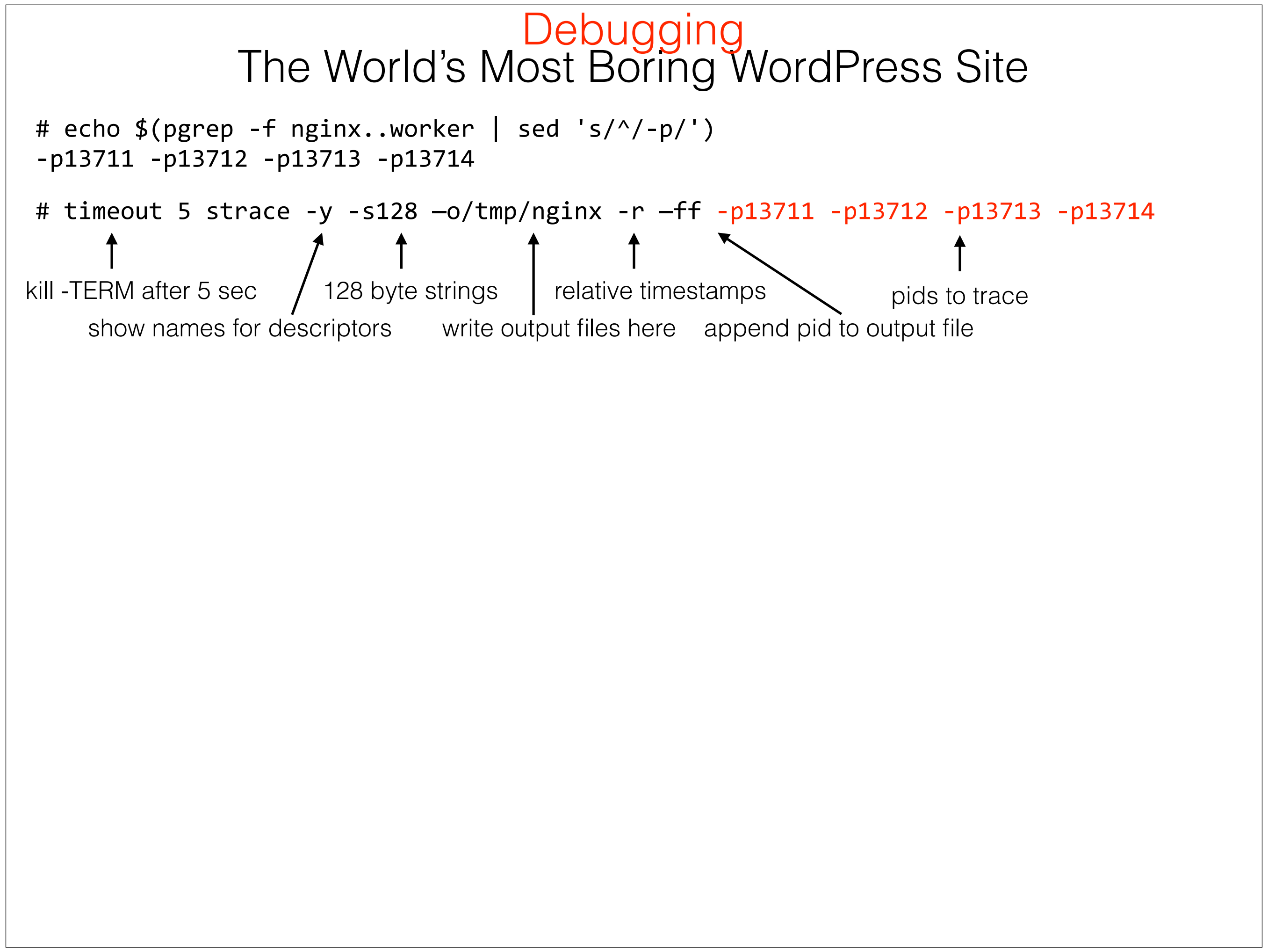
128 byte strings

write output files here

relative timestamps

append pid to output file

The ‘ff’ option tells strace to append the pid of the traced process to the output file. So for example if we’re tracing worker process 13711, the filename will be /tmp/nginx.13711.



Debugging

The World’s Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 —o/tmp/nginx -r —ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec

show names for descriptors

128 byte strings

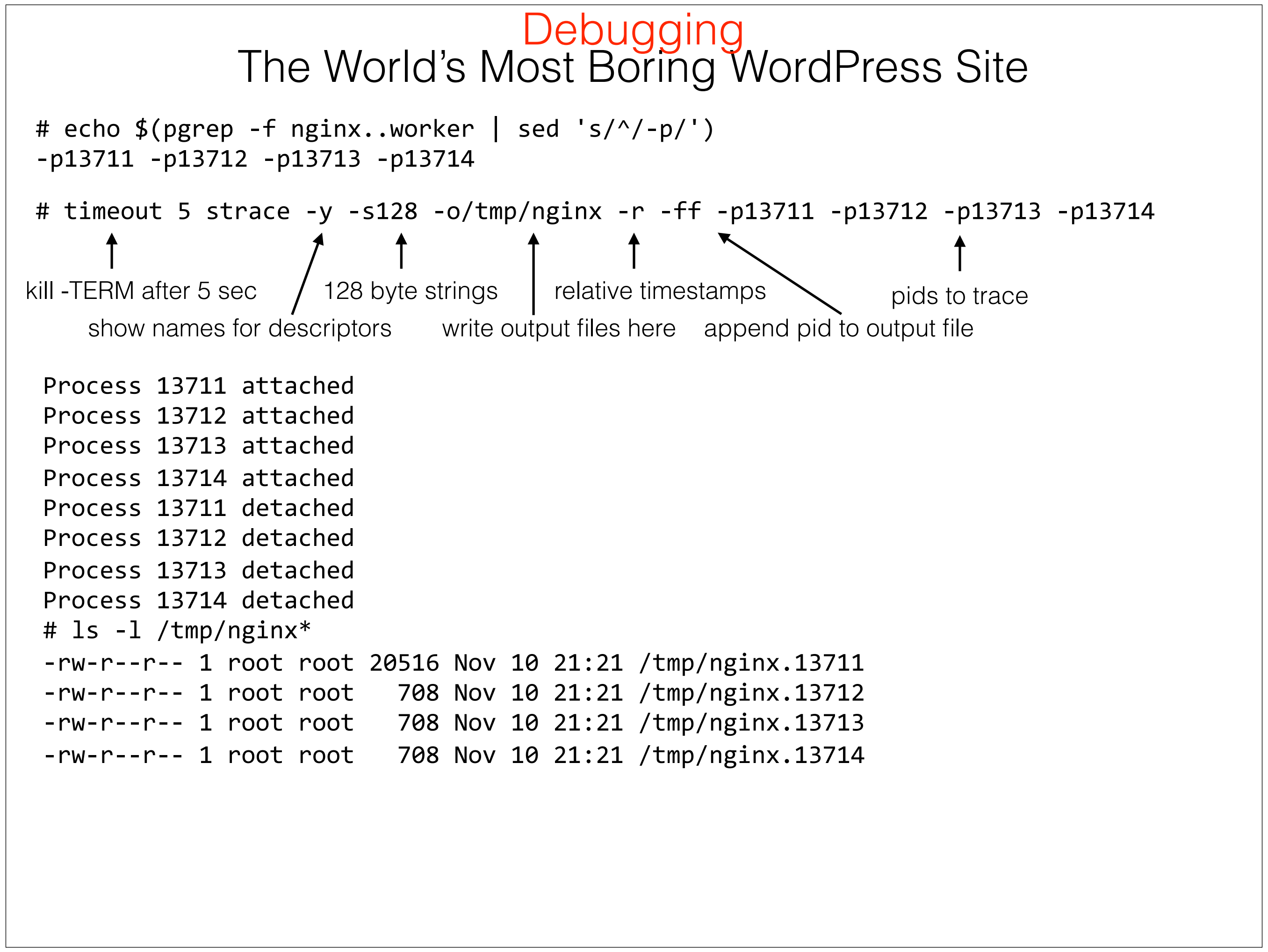
write output files here

relative timestamps

append pid to output file

pids to trace

The final arguments here we geneated using our pgrep/sed magic. This tells strace that we want to trace the process ids of the four nginx worker processes.



Debugging

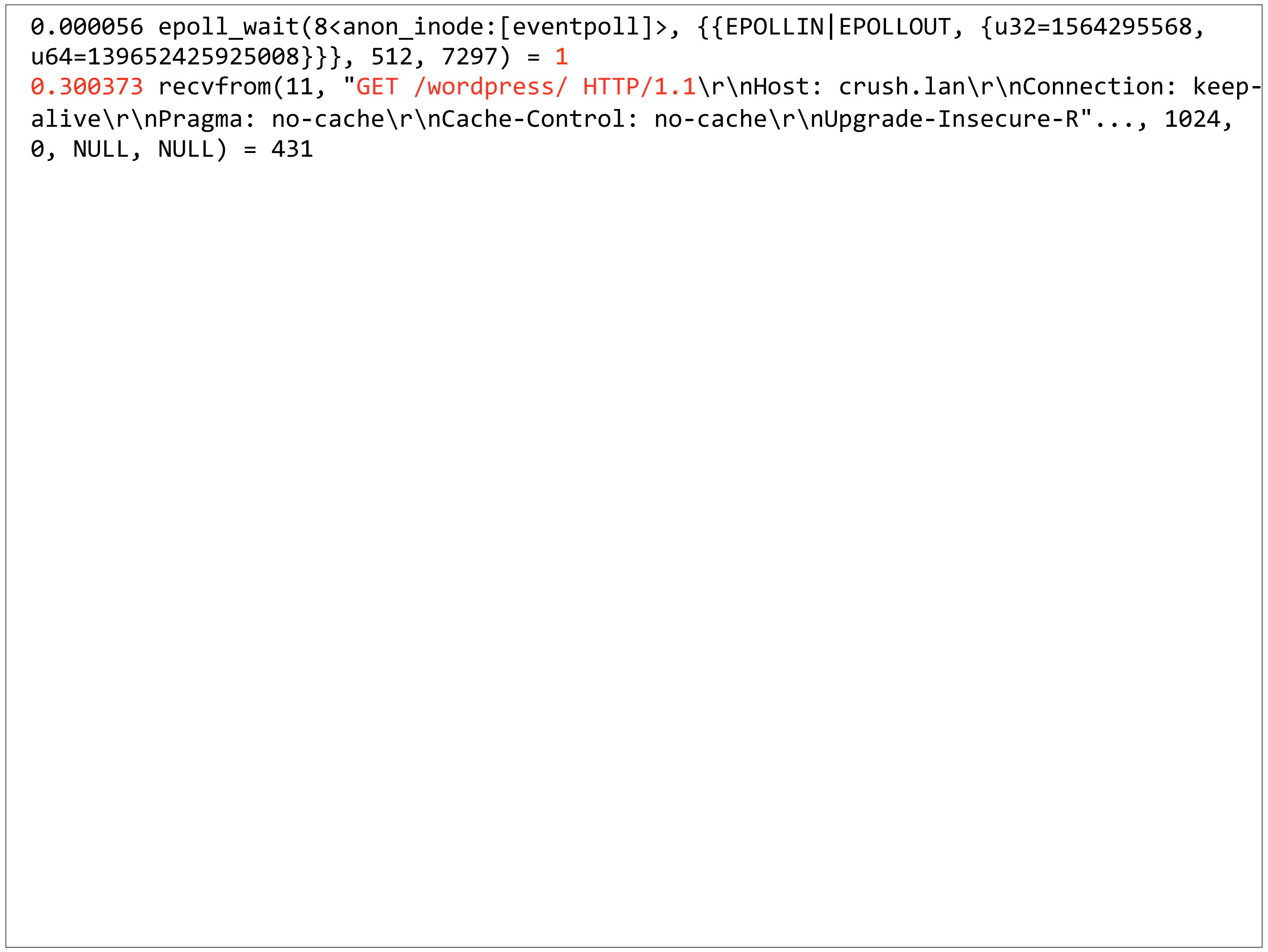
The World’s Most Boring WordPress Site

```
# echo $(pgrep -f nginx..worker | sed 's/^/-p/')
-p13711 -p13712 -p13713 -p13714

# timeout 5 strace -y -s128 -o/tmp/nginx -r -ff -p13711 -p13712 -p13713 -p13714
```

kill -TERM after 5 sec
show names for descriptors
128 byte strings
write output files here
relative timestamps
append pid to output file
pids to trace

```
Process 13711 attached
Process 13712 attached
Process 13713 attached
Process 13714 attached
Process 13711 detached
Process 13712 detached
Process 13713 detached
Process 13714 detached
# ls -l /tmp/nginx*
-rw-r--r-- 1 root root 20516 Nov 10 21:21 /tmp/nginx.13711
-rw-r--r-- 1 root root   708 Nov 10 21:21 /tmp/nginx.13712
-rw-r--r-- 1 root root   708 Nov 10 21:21 /tmp/nginx.13713
-rw-r--r-- 1 root root   708 Nov 10 21:21 /tmp/nginx.13714
```

No we run strace, and we see some output telling us that strace has attached its hooks into the processes, and after 5 seconds it will detach from these processes after the ‘timeout’ program sends it a SIGTERM. If we look in /tmp after strace finishes, we can see that strace wrote its output where we expected it to. Only one of these files has a lot of information in it, pid 13711 is the one that must have processed a request during the 5 seconds we were tracing.



```
0.000056 epoll_wait(8<anon_inode:[eventpoll]>, {{EPOLLIN|EPOLLOUT, {u32=1564295568,
u64=139652425925008}}}, 512, 7297) = 1
0.300373 recvfrom(11, "GET /wordpress/ HTTP/1.1\r\nHost: crush.lan\r\nConnection: keep-
alive\r\nPragma: no-cache\r\nCache-Control: no-cache\r\nUpgrade-Insecure-R"..., 1024,
0, NULL, NULL) = 431
```

Let’s have a look at some of the system calls that the nginx worker made. The first epoll_wait call is the worker waiting around for something to happen. It doesn’t wait long though, 300ms later it learns that one of its file descriptors (#11) is ready to be read from. It reads a 431 byte request from a browser that requests an HTTP GET on the /wordpress/ location.

```
0.000056 epoll_wait(8<anon_inode:[eventpoll]>, {{EPOLLIN|EPOLLOUT, {u32=1564295568,
u64=139652425925008}}}, 512, 7297) = 1
0.300373 recvfrom(11, "GET /wordpress/ HTTP/1.1\r\nHost: crush.lan\r\nConnection: keep-
alive\r\nPragma: no-cache\r\nCache-Control: no-cache\r\nUpgrade-Insecure-R"..., 1024,
0, NULL, NULL) = 431
0.000305 stat("/var/www/html/wordpress/index.php", {st_mode=S_IFREG|0644,
st_size=418, ...}) = 0
```

It runs the stat() system call to get some information about the index.php script.

```
0.000056 epoll_wait(8<anon_inode:[eventpoll]>, {{EPOLLIN|EPOLLOUT, {u32=1564295568,
u64=139652425925008}}}, 512, 7297) = 1
0.300373 recvfrom(11, "GET /wordpress/ HTTP/1.1\r\nHost: crush.lan\r\nConnection: keep-
alive\r\nPragma: no-cache\r\nCache-Control: no-cache\r\nUpgrade-Insecure-R"..., 1024,
0, NULL, NULL) = 431
0.000305 stat("/var/www/html/wordpress/index.php", {st_mode=S_IFREG|0644,
st_size=418, ...}) = 0
0.000214 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 12
```

It then does something somewhat unxpected, it creates a new socket, and gets back file descriptor #12.

```
0.000056 epoll_wait(8<anon_inode:[eventpoll]>, {{EPOLLIN|EPOLLOUT, {u32=1564295568,
u64=139652425925008}}}, 512, 7297) = 1
0.300373 recvfrom(11, "GET /wordpress/ HTTP/1.1\r\nHost: crush.lan\r\nConnection: keep-
alive\r\nPragma: no-cache\r\nCache-Control: no-cache\r\nUpgrade-Insecure-R"..., 1024,
0, NULL, NULL) = 431
0.000305 stat("/var/www/html/wordpress/index.php", {st_mode=S_IFREG|0644,
st_size=418, ...}) = 0
0.000214 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 12
0.000161 connect(12, {sa_family=AF_INET, sin_port=htons(9000),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS
```

After it gets the handle back from the socket, it calls connect() to connect that socket to port 9000 on localhost. Notice those curly braces in the strace output. Strace knows that the second argument to connect is a pointer to a sockaddr data structure. Strace is being extremely helpful here, by deferencing that pointer and breaking out the various structure members (like sa_family, sa_port, etc) and showing them to you.

```
0.000056 epoll_wait(8<anon_inode:[eventpoll]>, {{EPOLLIN|EPOLLOUT, {u32=1564295568,
u64=139652425925008}}}, 512, 7297) = 1
0.300373 recvfrom(11, "GET /wordpress/ HTTP/1.1\r\nHost: crush.lan\r\nConnection: keep-
alive\r\nPragma: no-cache\r\nCache-Control: no-cache\r\nUpgrade-Insecure-R"..., 1024,
0, NULL, NULL) = 431
0.000305 stat("/var/www/html/wordpress/index.php", {st_mode=S_IFREG|0644,
st_size=418, ...}) = 0
0.000214 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 12
0.000161 connect(12, {sa_family=AF_INET, sin_port=htons(9000),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS
0.038520 recvfrom(12, "\1\6\0\1\37\370\0\0X-Powered-By: PHP/5.5.9-1ubuntu4.20\r
\nContent-Type: text/html; charset=UTF-8\r\nLink: <http://crush/wordpress/?
rest_route=/"..., 4096, 0, NULL, NULL) = 4096
```

The webserver reads 4k of information from this socket, with the recvfrom() system call from our socket on file descriptor #12 call that judging from the ‘Powered-By’ header appear to be from PHP.

```
0.000056 epoll_wait(8<anon_inode:[eventpoll]>, {{EPOLLIN|EPOLLOUT, {u32=1564295568,
u64=139652425925008}}}, 512, 7297) = 1
0.300373 recvfrom(11, "GET /wordpress/ HTTP/1.1\r\nHost: crush.lan\r\nConnection: keep-
alive\r\nPragma: no-cache\r\nCache-Control: no-cache\r\nUpgrade-Insecure-R"..., 1024,
0, NULL, NULL) = 431
0.000305 stat("/var/www/html/wordpress/index.php", {st_mode=S_IFREG|0644,
st_size=418, ...}) = 0
0.000214 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 12
0.000161 connect(12, {sa_family=AF_INET, sin_port=htons(9000),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS
0.038520 recvfrom(12, "\1\6\0\1\37\370\0\0X-Powered-By: PHP/5.5.9-1ubuntu4.20\r
\nContent-Type: text/html; charset=UTF-8\r\nLink: <http://crush/wordpress/?
rest_route=/"..., 4096, 0, NULL, NULL) = 4096
0.000133 close(12<socket:[713910]>) = 0
0.000211 writev(11<socket:[713795]>, [{"HTTP/1.1 200 OK\r\nServer: nginx/1.4.6
(Ubuntu)\r\nDate: Fri, 11 Nov 2016 05:21:18 GMT\r\nConten
t-Type: text/html; charset=UTF-8\r\nTran"..., 311}, {"ec2\r\n", 5},
{"\37\213\10\0\0\0\0\0\0\3", 10}, {"\265Zyo\333\310\331\377\333\376\24c\6
\265\244\206\207(\371\224E\31\251\223\305\273\305n\33\304\t
\212\"\16\26\0249\22\351\360*I\331\326\353\370\273\367\367\314\f/\371\314&]d-
\36\317<\363\334\327p\272\363\366\237g\37\377\375\376\35\v\3128\232mO\351\207En
\262t4\236\30\237\3165\346EnQ8Z\222\32\227\205F\20\334\365g\333[\32
3\230\227.\363\0027/x\351h\237>\376b\34\341\265"..., 3768}, {"\r\n0\r\n\r\n",
7}], 5) = 4101
```

A bit later is closes the socket, and then writes an HTTP response back to the web browser. What’s happening here is that nginx is farming out the request for index.php to something listening on port 9000, and then sending what it gets back from that port back to the web browser. It’s essentially acting like a middle man.

```
# lsof -i :9000
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
php5-fpm 1071     root    7u  IPv4  10048      0t0  TCP localhost:9000 (LISTEN)
php5-fpm 1104 www-data    0u  IPv4  10048      0t0  TCP localhost:9000 (LISTEN)
php5-fpm 1105 www-data    0u  IPv4  10048      0t0  TCP localhost:9000 (LISTEN)
```

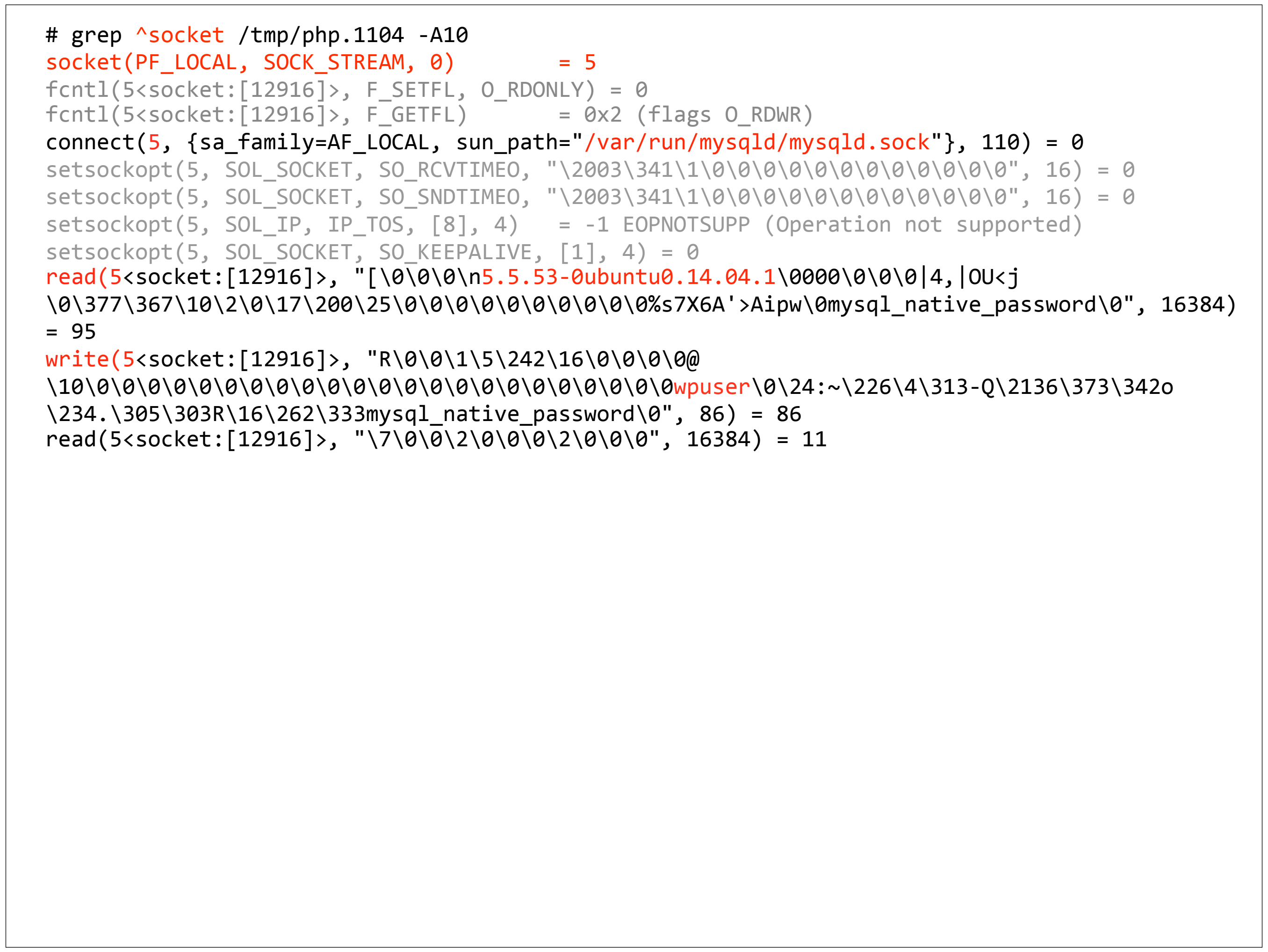
Sure enough, that middle man listening on port 9000 is the php5-fpm worker. It listens for requests on port 9000 to execute PHP code, runs the code, and then hands hte results back to the requestor.

```
# lsof -i :9000
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
php5-fpm 1071     root    7u  IPv4  10048      0t0  TCP localhost:9000 (LISTEN)
php5-fpm 1104 www-data    0u  IPv4  10048      0t0  TCP localhost:9000 (LISTEN)
php5-fpm 1105 www-data    0u  IPv4  10048      0t0  TCP localhost:9000 (LISTEN)

# timeout 5 strace -y -s4096 -o /tmp/php -ff  $(pgrep php | sed 's/^/-p/') &

 Process 1071 attached
 Process 1104 attached
 Process 1105 attached

 # curl -s http://localhost/wordpress/ | grep Wow
                 <p>Such hello, much world. Wow. Hello world.</p>

 Process 1071 detached
 Process 1104 detached
 Process 1105 detached


 # ls -l /tmp/php*
 -rw-r--r-- 1 root root  1645 Nov 11 12:44 /tmp/php.1071
 -rw-r--r-- 1 root root    26 Nov 11 12:44 /tmp/php.1104
 -rw-r--r-- 1 root root 70811 Nov 11 12:44 /tmp/php.1105
```

So now we run strace against php5-fpm instead of nginx, and while it's running we run curl to generate a request to nginx, which in turn will generate a request to PHP. Based on the file size, it looks like the php5-fpm worker on pid 1105 handled the request. Let's see if we can determine using strace where PHP grabs that snarky string from. 'Such hello, much world. Wow.' Notice I now ask strace to show me 4k worth of strings.

```
write(5<socket:[12916]>, "0\0\0\0\3SELECT wp_posts.* FROM wp_posts WHERE ID IN (4)", 52) = 52
read(5<socket:[12916]>, "\1\0\0\1\0273\0\0\2\3def\twordpress\10wp_posts\10wp_posts\2ID\2ID\f?
\0\24\0\0\0\10#B\0\0\0E\0\0\3\3def\twordpress\10wp_posts\10wp_posts\vpost_author\vpost_author
\f?\0\24\0\0\0\10)@\0\0\0A\0\0\4\3def\twordpress\10wp_posts\10wp_posts\tpost_date\tpost_date
\f?\0\23\0\0\0\f\201@\0\0\0I\0\0\5\3def\twordpress\10wp_posts\10wp_posts\rpost_date_gmt
\rpost_date_gmt\f?\0\23\0\0\0\f\201\0\0\0\0G\0\0\6\3def\twordpress\10wp_posts\10wp_posts
\fpost_content\fpost_content\f\340\0\377\377\377\377\374\21\20\0\0\0C\0\0\7\3def\twordpress
\10wp_posts\10wp_posts\npost_title\npost_title\f\340\0\374\377\3\0\374\21\20\0\0\0G
\0\0\10\3def\twordpress\10wp_posts\10wp_posts\fpost_excerpt\fpost_excerpt\f
\340\0\374\377\3\0\374\21\20\0\0\0E\0\0\t\3def\twordpress\10wp_posts\10wp_posts\vpost_status
\vpost_status\f\340\0P\0\0\0\375\1@\0\0\0K\0\0\n\3def\twordpress\10wp_posts\10wp_posts
\16comment_status\16comment_status\f\340\0P\0\0\0\375\1\0\0\0\0E\0\0\v\3def\twordpress
\10wp_posts\10wp_posts\vping_status\vping_status\f\340\0P\0\0\0\375\1\0\0\0\0I\0\0\f\3def
\twordpress\10wp_posts\10wp_posts\rpost_password\rpost_password\f\340\0P\0\0\0\375\1\0\0\0\0A
\0\0\r\3def\twordpress\10wp_posts\10wp_posts\tpost_name\tpost_name\f\340\0 \3\0\0\375\t@
\0\0\0=\0\0\16\3def\twordpress\10wp_posts\10wp_posts\7to_ping\7to_ping\f
\340\0\374\377\3\0\374\21\20\0\0\0;\0\0\17\3def\twordpress\10wp_posts\10wp_posts\6pinged
\6pinged\f\340\0\374\377\3\0\374\21\20\0\0\0I\0\0\20\3def\twordpress\10wp_posts\10wp_posts
\rpost_modified\rpost_modified\f?\0\23\0\0\0\f\201\0\0\0\0Q\0\0\21\3def\twordpress\10wp_posts
\10wp_posts\21post_modified_gmt\21post_modified_gmt\f?\0\23\0\0\0\f\201\0\0\0\0Y\0\0\22\3def
\twordpress\10wp_posts\10wp_posts\25post_content_filtered\25post_content_filtered\f
\340\0\377\377\377\377\374\21\20\0\0\0E\0\0\23\3def\twordpress\10wp_posts\10wp_posts
\vpost_parent\vpost_parent\f?\0\24\0\0\0\10)@\0\0\0007\0\0\24\3def\twordpress\10wp_posts
\10wp_posts\4guid\4guid\f\340\0\374\3\0\0\375\1\0\0\0\0C\0\0\25\3def\twordpress\10wp_posts
\10wp_posts\nmenu_order\nmenu_order\f?\0\v\0\0\0\3\1\0\0\0\0A\0\0\26\3def\twordpress
\10wp_posts\10wp_posts\tpost_type\tpost_type\f\340\0P\0\0\0\375\t@\0\0\0K\0\0\27\3def
\twordpress\10wp_posts\10wp_posts\16post_mime_type\16post_mime_type\f
\340\0\220\1\0\0\375\1\0\0\0\0I\0\0\30\3def\twordpress\10wp_posts\10wp_posts\rcomment_count
\rcomment_count\f?
\0\24\0\0\0\10\1\0\0\0\0\5\0\0\31\376\0\0\2\0\354\0\0\32\0014\0011\0232016-11-11
03:19:11\0232016-11-11 03:19:11=Such hello, much world. Wow. Hello world.\r\n\r\n&nbsp;\r\n\r
\n&nbsp;\fHello world!\0\7publish\4open\4open\0\rhello-world-2\0\0\0232016-11-11
03:56:33\0232016-11-11 03:56:33\0\0010\33http://crush/wordpress/?p=4\0010\4post
\0\0010\5\0\0\33\376\0\0\2\0", 16384) = 1940
```

So by grepping for the string in the strace output I found where it comes from — a read from a socket. If we look at the write to the same socket just above the read, you can see php sending out a MySQL query to file descriptor #5 and then receiving the results, by reading from the file descriptor. The string is kept in the MySQL wp_posts table.



```
# grep ^socket /tmp/php.1104 -A10
socket(PF_LOCAL, SOCK_STREAM, 0)        = 5
fcntl(5<socket:[12916]>, F_SETFL, O_RDONLY) = 0
fcntl(5<socket:[12916]>, F_GETFL)       = 0x2 (flags O_RDWR)
connect(5, {sa_family=AF_LOCAL, sun_path="/var/run/mysqld/mysqld.sock"}, 110) = 0
setsockopt(5, SOL_SOCKET, SO_RCVTIMEO, "\2003\341\1\0\0\0\0\0\0\0\0\0\0\0\0", 16) = 0
setsockopt(5, SOL_SOCKET, SO_SNDTIMEO, "\2003\341\1\0\0\0\0\0\0\0\0\0\0\0\0", 16) = 0
setsockopt(5, SOL_IP, IP_TOS, [8], 4)   = -1 EOPNOTSUPP (Operation not supported)
setsockopt(5, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
read(5<socket:[12916]>, "[\0\0\0\n5.5.53-0ubuntu0.14.04.1\0000\0\0\0|4,|OU<j
\0\377\367\10\2\0\17\200\25\0\0\0\0\0\0\0\0\0\0%s7X6A'>Aipw\0mysql_native_password\0", 16384)
= 95
write(5<socket:[12916]>, "R\0\0\1\5\242\16\0\0\0\0@
\10\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0wpuser\0\24:~\226\4\313-Q\2136\373\342o
\234.\305\303R\16\262\333mysql_native_password\0", 86) = 86
read(5<socket:[12916]>, "\7\0\0\2\0\0\0\2\0\0\0", 16384) = 11
```

So in the last slide we saw that file descriptor #5 was the socket used for the MySQL connection. Let's trace where that connection was made. I grepped the strace output file for the socket system call and found it right away. That connect call connects socket 5 to the mysqld local UNIX domain socket, which tells us the DB is on the same host as the webserver. You can also see it read the MySQL banner (which includes the version number) from the socket, and it then writes the username and an encoded password to the socket to authenticate.