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.
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?

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.
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.
 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?

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?

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?

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 output ‘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.
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 a 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 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.
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 program's standard input, standard output, and standard error. In our case we want to write to standard out, so we load a 1 into the RDI register.
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.
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.
// helloasm.c
char *buf = "Hello world!
";
void _start(void)
{
    asm(
        "mov $1,%rax;"  __write(1, 0, buf, 13)  
        "mov $1,%rdi;"  __write(1, 0, buf, 13)  
        "mov buf,%rsi;"  __write(1, buf)  
        "mov $13,%rdx;"  __write(1, buf, 13)  
        "syscall"         __write(1, buf, 13)  
    )
    asm(
        "mov $231,%rax;"  __write(2, 0, buf, 13)  
        "mov $42,%rdi;"  __write(2, 0, buf, 13)  
        "syscall"         __write(2, 0, buf, 13)  
    )
}

<table>
<thead>
<tr>
<th>%rax</th>
<th>System call</th>
<th>%rdi</th>
<th>%rsi</th>
<th>%rdx</th>
<th>%r10</th>
<th>%r8</th>
<th>%r9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sys_read</td>
<td>signed int fd</td>
<td>char *buf</td>
<td>size_t count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>sys_write</td>
<td>unsigned int fd</td>
<td>const char *buf</td>
<td>size_t count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sys_open</td>
<td>const char *filename</td>
<td>int flags</td>
<td>int mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sys_close</td>
<td>unsigned int fd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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"
    );
}

<table>
<thead>
<tr>
<th>%rax</th>
<th>System call</th>
<th>%rdi</th>
<th>%rsi</th>
<th>%rdx</th>
<th>%r10</th>
<th>%r8</th>
<th>%r9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sys_read</td>
<td>unsigned int fd</td>
<td>char *buf</td>
<td>size_t count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>sys_write</td>
<td>unsigned int fd</td>
<td>const char *buf</td>
<td>size_t count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sys_open</td>
<td>const char *filename</td>
<td>int flags</td>
<td>int mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sys_close</td>
<td>unsigned int fd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>231</td>
<td>sys_exit_group</td>
<td>Int error_code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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

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

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.
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.
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.
We can clean this up a bit by piping the output of helloasm to /dev/null, so we only see strace's output. Strace is doing what it's 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", 13) = 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.
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.
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 summarize things a bit for you as to not be too overwhelming.
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.
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 summarizes 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.
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.
Ok, let's modify the helloasm program just a bit so it prints 'Hello world' one hundred thousand times before exiting.
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.
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.
The syscall executions themselves are what make up the 40ms or so of system time as reported by `time`.
Next I wanted to show you the \texttt{-c} option to \texttt{strace} which prints out a nice summary 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.
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.
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.
Haha.

```bash
$ 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                    1  000001  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

--- -------- -----------------  ------  -------- ----------------------
100.00  4.289312                    600102  4  total
```

Haha.
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.
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.
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 (0x00007fb24f8f6000)

Compare this to /usr/bin/printf, which is dynamically 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.
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.
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.
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_ANONYMOUS` 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..
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.
This next section has the system calls that the dynamic linking code uses to load the libc library into memory. It opens the library, 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.
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 specific 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 program's 'RSS' or resident set size, which led to some confusion.
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.
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.
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.
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.
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.
Here is my strace command line that we're going to use to figure out what is happening with the nginx workers.

```bash
# 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
```
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.
This next argument we talked about earlier, it shows filenames associated with descriptors, which is pretty handy.
The World’s Most Boring WordPress Site

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

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

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.
The World's Most Boring WordPress Site

Debugging

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

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

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.
# 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

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.
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.
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.
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.
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.
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 unexpected, it creates a new socket, and gets back file descriptor #12.
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.
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.
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.
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 the results back to the requestor.
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.
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.
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.