

OpenFest 2015

System Call Tracing with strace

© 2015 Michael Kerrisk
man7.org Training and Consulting
<http://man7.org/training/>
@mkerrisk mtk@man7.org

7 November 2015
Sofia, Bulgaria
@openfestbg #OpenFest2015

Outline

- 1 Preamble
- 2 Getting started
- 3 Tracing child processes
- 4 Filtering strace output
- 5 Further strace options

Outline

- 1 Preamble
- 2 Getting started
- 3 Tracing child processes
- 4 Filtering strace output
- 5 Further strace options

Who am I?

- Maintainer of Linux man-pages (since 2004)
 - Documents kernel-user-space + C library APIs
 - ~1000 manual pages
 - <http://www.kernel.org/doc/man-pages/>
- Linux involvement: API review, testing, and documentation
- “Day job”: trainer, writer, programmer



Audience

- Programmers?
- C/C++ Programmers?

What is a system call?

- Various possible answers, from different perspectives
- Answer 1: **request to kernel to perform a service**
 - Open a file
 - Execute a new program
 - Create a new process
 - Send a message to another process
- Answer 2 (programmer's perspective): "call a function"
 - `fd = open("myfile.txt", O_CREAT|O_RDWR, 0644);`
 - System call **looks like** any other function call



What is a system call?

- Answer 3: entry point providing **controlled mechanism to execute kernel code**
- User-space programs **can't** call functions inside kernel
- Syscall = one of few mechanisms by which program can ask to execute kernel code
 - Others: `/proc`, `/sys`, etc.
- Set of system calls is:
 - Operating-system specific
 - Can't run Linux binaries on another OS, and vice versa
 - Limited/strictly defined by OS
 - Linux kernel provides 400+ syscalls
 - `syscalls(2)` man page



Steps in the execution of a system call

- 1 Program calls wrapper function in C library
- 2 Wrapper function
 - Packages syscall arguments into registers
 - Puts (unique) syscall number into a register
- 3 Wrapper flips CPU to kernel mode (**user-mode** \Rightarrow **kernel-mode**)
 - Execute special machine instruction (e.g., **sysenter** on x86)
 - Main effect: CPU can now touch memory marked as accessible only in kernel mode
- 4 Kernel executes syscall handler:
 - Invokes **service routine** corresponding to syscall number
 - **Do the real work**, generate result status
 - Places return value from service routine in a register
 - Switches back to user mode, passing control back to wrapper
 - (**kernel-mode** \Rightarrow **user-mode**)
- 5 Wrapper function examines syscall return value; on error, copies error number to **errno**



Outline

- 1 Preamble
- 2 **Getting started**
- 3 Tracing child processes
- 4 Filtering strace output
- 5 Further strace options

- A tool to trace system calls made by a user-space process
 - Implemented via *ptrace(2)*
- Or: a debugging tool for tracing **complete conversation between application and kernel**
 - Application source code is not required
- Answer questions like:
 - What system calls are employed by application?
 - Which files does application touch?
 - What arguments are being passed to each system call?
 - Which system calls are failing, and why (*errno*)?

- Log information is provided in **symbolic form**
 - **System call names** are shown
 - We see **signal names** (not numbers)
 - **Strings** printed as characters (up to 32 bytes, by default)
 - **Bit-mask arguments displayed symbolically**, using corresponding bit flag names ORed together
 - **Structures** displayed with **labeled fields**
 - **errno values** displayed symbolically + matching error text
 - “large” arguments and structures are abbreviated by default

```
fstat(3, {st_dev=makedev(8, 2), st_ino=401567,
st_mode=S_IFREG|0755, st_nlink=1, st_uid=0, st_gid=0,
st_blksize=4096, st_blocks=280, st_size=142136,
st_atime=2015/02/17-17:17:25, st_mtime=2013/12/27-22:19:58,
st_ctime=2014/04/07-21:44:17}) = 0

open("/lib64/liblzma.so.5", O_RDONLY|O_CLOEXEC) = 3
```




Simple usage: tracing a command at the command line

- A very simple C program:

```
int main(int argc, char *argv[]) {
#define STR "Hello world\n"
    write(STDOUT_FILENO, STR, strlen(STR));
    exit(EXIT_SUCCESS);
}
```

- Run *strace(1)*, directing logging output (*-o*) to a file:

```
$ strace -o strace.log ./hello_world
Hello world
```

- (By default, strace output goes to standard error)
-  On some systems, may first need to:

```
# echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

- Yama LSM disables *ptrace(2)* to prevent attack escalation; see man page



Simple usage: tracing a command at the command line

```
$ cat strace.log
execve("./hello_world", [ "./hello_world" ], [ /* 110 vars */ ]) = 0
...
access("/etc/ld.so.preload", R_OK)          = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=160311, ...}) = 0
mmap(NULL, 160311, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa5ecfc0000
close(3)                                     = 0
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
...
write(1, "Hello world\n", 12)              = 12
exit_group(0)                                = ?
+++ exited with 0 +++
```

- Even simple programs make lots of system calls!
 - 25 in this case (many have been edited from above output)
- Most output in this trace relates to finding and loading shared libraries
 - First call (*execve()*) was used by shell to load our program
 - Only last two system calls were made by our program



Simple usage: tracing a command at the command line

```
$ cat strace.log
execve("./hello_world", [ "./hello_world" ], [ /* 110 vars */ ]) = 0
...
access("/etc/ld.so.preload", R_OK) = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=160311, ...}) = 0
mmap(NULL, 160311, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa5ecfc0000
close(3) = 0
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
...
write(1, "Hello world\n", 12) = 12
exit_group(0) = ?
+++ exited with 0 +++
```

For each system call, we see:

- Name of system call
- Values passed in/returned via arguments
- System call return value
- Symbolic *errno* value (+ explanatory text) on syscall failures



A gotcha...

- The last call in our program was:

```
exit(EXIT_SUCCESS);
```

- But *strace* showed us:

```
exit_group(0) = ?
```

- Some detective work:

- We “know” *exit(3)* is a library function that calls *_exit(2)*
- But where did *exit_group()* come from?
- *_exit(2)* man page tells us:

```
$ man 2 _exit
```

```
...
```

C library/kernel differences

```
In glibc up to version 2.3, the _exit() wrapper function
invoked the kernel system call of the same name. Since
glibc 2.3, the wrapper function invokes exit_group(2),
in order to terminate all of the threads in a process.
```

- ⇒ may need to dig deeper to understand *strace(1)* output



Outline

- 1 Preamble
- 2 Getting started
- 3 Tracing child processes**
- 4 Filtering strace output
- 5 Further strace options

Tracing child processes

- By default, *strace* does not trace children of traced process
- *-f* option causes children to be traced
 - Each trace line is prefixed by PID
 - In a program that employs POSIX threads, each line shows kernel thread ID (*gettid()*)

Tracing child processes: strace/fork_exec.c

```
1 int main(int argc, char *argv[]) {
2     pid_t childPid;
3     char *newEnv[] = {"ONE=1", "TWO=2", NULL};
4
5     printf("PID of parent: %ld\n", (long) getpid());
6     childPid = fork();
7     if (childPid == 0) {          /* Child */
8         printf("PID of child: %ld\n", (long) getpid());
9         if (argc > 1) {
10             execve(argv[1], &argv[1], newEnv);
11             errExit("execve");
12         }
13         exit(EXIT_SUCCESS);
14     }
15     wait(NULL);                  /* Parent waits for child */
16     exit(EXIT_SUCCESS);
17 }
```

```
$ strace -f -o strace.log ./fork_exec
PID of parent: 1939
PID of child: 1940
```

Tracing child processes: `strace/fork_exec.c`

```
$ cat strace.log
1939 execve("./fork_exec", ["/fork_exec"], [/* 110 vars */]) = 0
...
1939 clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|
  CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fe484b2ea10) = 1940
1939 wait4(-1, <unfinished ...>
1940 write(1, "PID of child: 1940\n", 21) = 21
1940 exit_group(0) = ?
1940 +++ exited with 0 +++
1939 <... wait4 resumed> NULL, 0, NULL) = 1940
1939 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED,
  si_pid=1940, si_uid=1000, si_status=0, si_utime=0,
  si_stime=0} ---
1939 exit_group(0) = ?
1939 +++ exited with 0 +++
```

- Each line of trace output is prefixed with corresponding PID
- Inside glibc, `fork()` is actually a wrapper that calls `clone(2)`
- `wait()` is a wrapper that calls `wait4(2)`
- We see two lines of output for `wait4()` because call blocks and then resumes
- `strace` shows us that parent received a `SIGCHLD` signal




Outline

- 1 Preamble
- 2 Getting started
- 3 Tracing child processes
- 4 Filtering strace output**
- 5 Further strace options

Selecting system calls to be traced

- *strace -e* can be used to select system calls to be traced
 - Syntax is a little complex \Rightarrow we'll look at simple, common use cases
- *-e trace=<syscall>[,<syscall>...]*
 - Specify system call(s) that should be traced
 - Other system calls are ignored

```
$ strace -o strace.log -e trace=open,close ls
```

- *-e trace=!<syscall>[,<syscall>...]*
 - **Exclude** specified system call(s) from tracing
 - Some applications do bizarre things (e.g., calling *gettimeofday()* 1000s of times/sec.)
 -  "!" needs to be quoted to avoid shell interpretation



Selecting system calls by category

- `-e trace=<syscall-category>` specifies a category of system calls to trace
- Categories include:
 - *file*: trace all system calls that take a filename as argument
 - `open()`, `stat()`, `truncate()`, `chmod()`, `setxattr()`, `link()`...
 - *desc*: trace file-descriptor-related system calls
 - `read()`, `write()`, `open()`, `close()`, `fsetxattr()`, `poll()`, `select()`, `pipe()`, `fcntl()`, `epoll_create()`, `epoll_wait()`...
 - *process*: trace process management system calls
 - `fork()`, `clone()`, `exit_group()`, `execve()`, `wait4()`, `unshare()`...
 - *network*: trace network-related system calls
 - `socket()`, `bind()`, `listen()`, `connect()`, `sendmsg()`...
 - *memory*: trace memory-mapping-related system calls
 - `mmap()`, `mprotect()`, `mlock()`...



Filtering signals

- *strace -e signal=set*

- Trace only specified set of signals
- “sig” prefix in names is optional; following are equivalent:

```
$ strace -o strace.log -e signal=sigio,int ls > /dev/null  
$ strace -o strace.log -e signal=io,int ls > /dev/null
```

- *strace -e signal=!set*

- Exclude specified signals from tracing

Filtering by pathname

- *strace -P pathname*: trace only system calls that access file at *pathname*
 - Specify multiple *-P* options to trace multiple paths
- Example:

```
$ strace -o strace.log -P /lib64/libc.so.6 ls > /dev/null
Requested path '/lib64/libc.so.6' resolved into
'/usr/lib64/libc-2.18.so'
$ cat strace.log
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\36
\2\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2093096, ...}) = 0
mmap(NULL, 3920480, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8511fa3000
mmap(0x7f8512356000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b3000)
= 0x7f8512356000
close(3) = 0
+++ exited with 0 +++
```

- *strace* noticed that the specified file was opened on FD 3, and also traced operations on that FD



Outline

- 1 Preamble
- 2 Getting started
- 3 Tracing child processes
- 4 Filtering strace output
- 5 Further strace options

Obtaining a system call summary

- `strace -c` counts time, calls, and errors for each system call and reports a summary on program exit

```
$ strace -c who > /dev/null
```

% time	seconds	usecs/call	calls	errors	syscall
21.77	0.000648	9	72		alarm
14.42	0.000429	9	48		rt_sigaction
13.34	0.000397	8	48		fcntl
8.84	0.000263	5	48		read
7.29	0.000217	13	17	2	kill
6.79	0.000202	6	33	1	stat
5.41	0.000161	5	31		mmap
4.44	0.000132	4	31	6	open
2.89	0.000086	3	29		close
2.86	0.000085	43	2		socket
2.82	0.000084	42	2	2	connect
...					
100.00	0.002976		442	13	total



- `-p PID`: **trace running process** with specified PID
 - Type *Control-C* to cease tracing
 - To **trace multiple processes**, specify `-p` multiple times
 - Can only trace processes you own
 - ⚠ tracing a process can **heavily affect performance**
 - E.g., two orders of magnitude
 - Think twice before using in a production environment
- `-p PID -f`: will **trace all threads** in specified process

Further *strace* options

- *-v*: don't abbreviate arguments (structures, etc.)
 - Output can be quite verbose...
- *-s strsize*: maximum number of bytes to display for strings
 - Default is 32 characters
 - Pathnames are always printed in full
- Various options show start time or duration of system calls
 - *-t*, *-tt*, *-ttt*, *-T*
- *-i*: print value of instruction pointer on each system call

Thanks!

mtk@man7.org @mkerrisk
Slides at <http://man7.org/conf/>

Linux/UNIX system programming training (and more)
<http://man7.org/training/>

The Linux Programming Interface, <http://man7.org/tlpi/>

