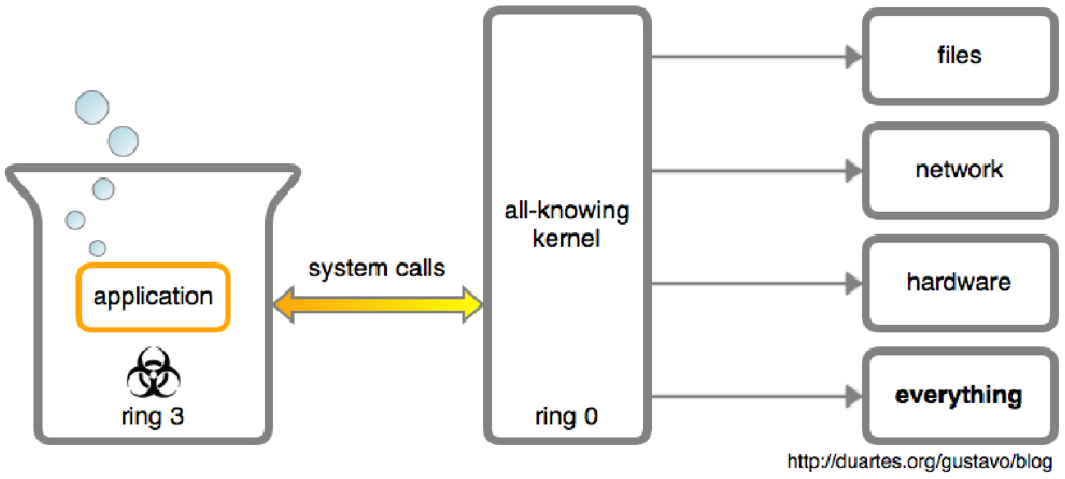
System Calls Make the World Go Round

Gustavo Duarte, Nov 6th, 2014

I hate to break it to you, but a user application is a helpless brain in a vat:

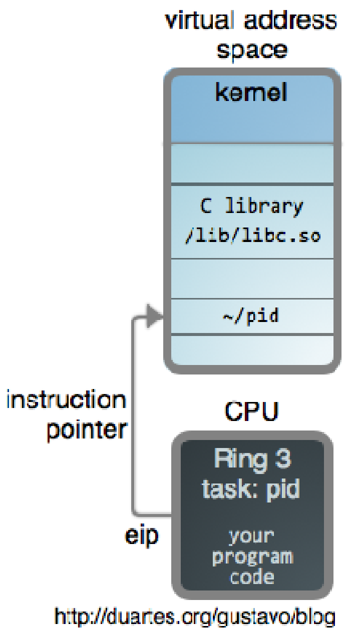


Every interaction with the outside world is mediated by the kernel through **system calls**. If an app saves a file, writes to the terminal, or opens a TCP connection, the kernel is involved. Apps are regarded as highly suspicious: at best a bug-ridden mess, at worst the malicious brain of an evil genius.

These system calls are function calls from an app into the kernel. They use a specific mechanism for safety reasons, but really you're just calling the kernel's API. The term "system call" can refer to a specific function offered by the kernel (*e.g.*, the `open()` system call) or to the calling mechanism. You can also say **syscall** for short.

This post looks at system calls, how they differ from calls to a library, and tools to poke at this OS/app interface. A solid understanding of what happens *within an app* versus what happens through the OS can turn an impossible-to-fix problem into a quick, fun puzzle.

So here's a running program, a *user process*:



It has a private virtual address space, its very own memory sandbox. The vat, if you will. In its address space, the program's binary file plus the libraries it uses are all memory mapped. Part of the address space maps the kernel itself.

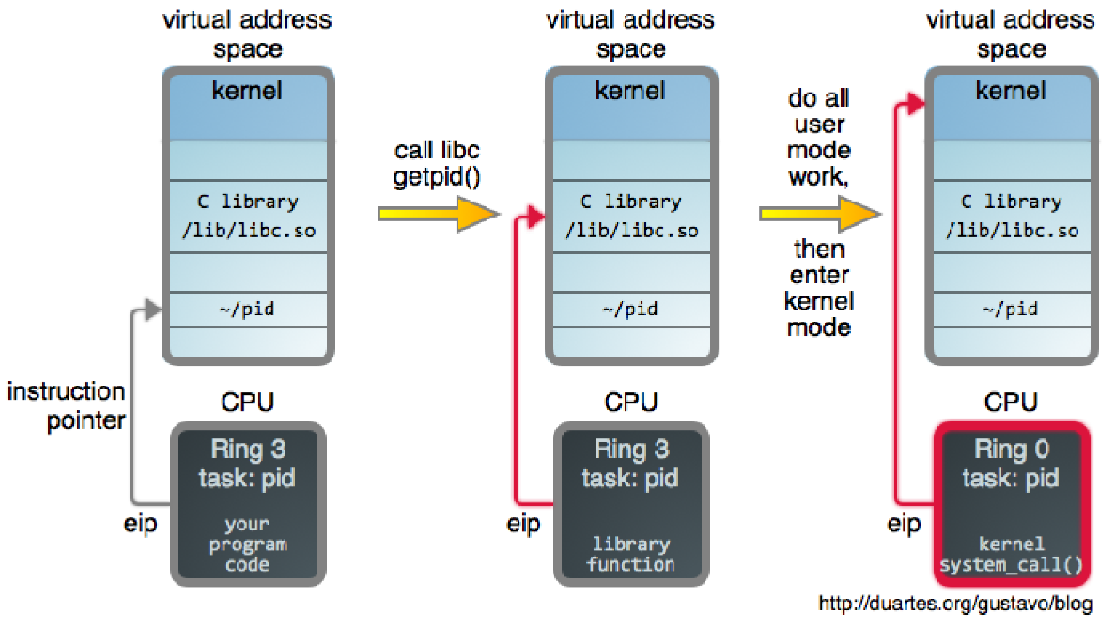
Below is the code for our program, `pid`, which simply retrieves its process id via getpid(2):

pid (pid.c)download

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t p = getpid();
    printf("%d\n", p);
}
```

In Linux, a process isn't born knowing its PID. It must ask the kernel, so this requires a system call:



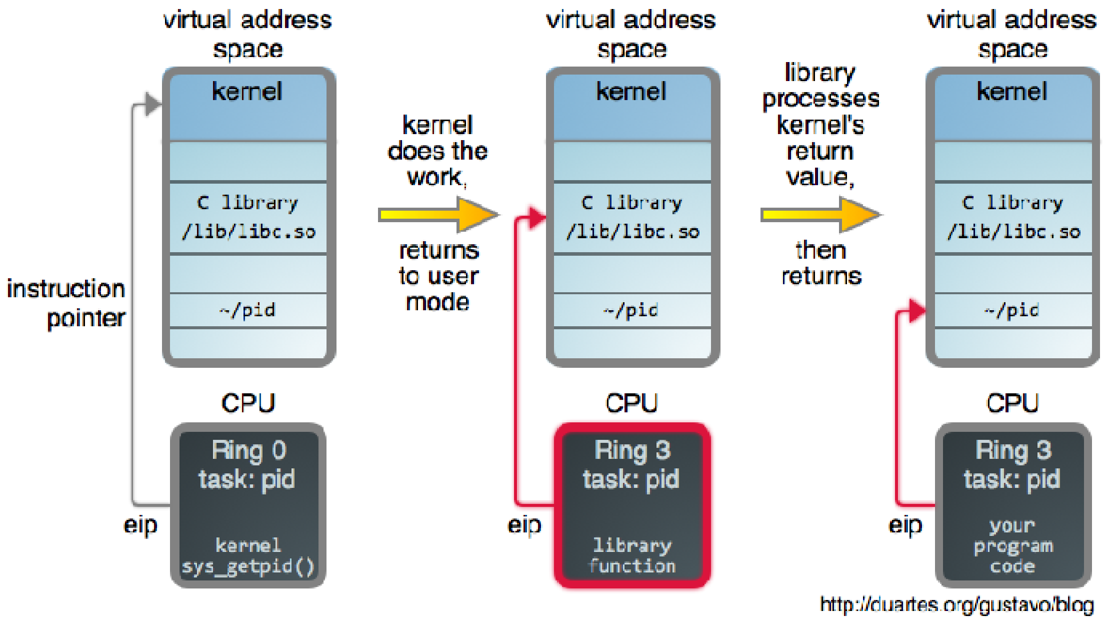
It all starts with a call to the C library's getpid(), which is a *wrapper* for the system call. When you call functions like `open(2)`, `read(2)`, and friends, you're calling these wrappers. This is true for many languages where the native methods ultimately end up in libc.

Wrappers offer convenience atop the bare-bones OS API, helping keep the kernel lean. Lines of code is where bugs live, and *all kernel code* runs in privileged mode, where mistakes can be disastrous. Anything that can be done in user mode should be done in user mode. Let the libraries offer friendly methods and fancy argument processing a la `printf(3)`.

Compared to web APIs, this is analogous to building the simplest possible HTTP interface to a service and then offering language-specific libraries with helper methods. Or maybe some caching, which is what libc's `getpid()` does: when first called it actually performs a system call, but the PID is then cached to avoid the syscall overhead in subsequent invocations.

Once the wrapper has done its initial work it's time to jump into ~~hyperspace~~ the kernel. The mechanics of this transition vary by processor architecture. In Intel processors, arguments and the syscall number are loaded into registers, then an instruction is executed to put the CPU inprivileged mode and immediately transfer control to a global syscall entry point within the kernel. If you're interested in details, David Drysdale has two great articles in LWN (first,second).

The kernel then uses the syscall number as an index into sys_call_table, an array of function pointers to each syscall implementation. Here, sys_getpid is called:



In Linux, syscall implementations are mostly arch-independent C functions, sometimes trivial, insulated from the syscall mechanism by the kernel's excellent design. They are regular code working on general data structures. Well, apart from being *completely paranoid* about argument validation.

Once their work is done they `return` normally, and the arch-specific code takes care of transitioning back into user mode where the wrapper does some post processing. In our example, getpid(2) now caches the PID returned by the kernel. Other wrappers might set the global `errno` variable if the kernel returns an error. Small things to let you know GNU cares.

If you want to be raw, glibc offers the syscall(2) function, which makes a system call without a wrapper. You can also do so yourself in assembly. There's nothing magical or privileged about a C library.

This syscall design has far-reaching consequences. Let's start with the incredibly useful strace(1), a tool you can use to spy on system calls made by Linux processes (in Macs, see dtruss(1m) and the amazing dtrace; in Windows, see sysinternals). Here's strace on `pid`:

```
~/code/x86-os$ strace ./pid

execve("./pid", ["./pid"], [/* 20 vars */]) = 0
brk(0)                                  = 0x9aa0000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7767000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=18056, ...}) = 0
mmap2(NULL, 18056, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7762000
close(3)                                = 0

[...snip...]

getpid()                                = 14678
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7766000
write(1, "14678\n", 614678
)                   = 6
exit_group(6)                           = ?
```

Each line of output shows a system call, its arguments, and a return value. If you put `getpid(2)`in a loop running 1000 times, you would still have only one `getpid()` syscall because of the PID caching. We can also see that `printf(3)` calls `write(2)` after formatting the output string.

`strace` can start a new process and also attach to an already running one. You can learn a lot by looking at the syscalls made by different programs. For example, what does the `sshd`daemon do all day?

```
~/code/x86-os$ ps ax | grep sshd
12218 ?        Ss     0:00 /usr/sbin/sshd -D

~/code/x86-os$ sudo strace -p 12218
Process 12218 attached - interrupt to quit
select(7, [3 4], NULL, NULL, NULL

[
 ... nothing happens ...
 No fun, it's just waiting for a connection using select(2)
 If we wait long enough, we might see new keys being generated and so on, but
 let's attach again, tell strace to follow forks (-f), and connect via SSH
]

~/code/x86-os$ sudo strace -p 12218 -f

[lots of calls happen during an SSH login, only a few shown]

[pid 14692] read(3, "-----BEGIN RSA PRIVATE KEY-----\n"..., 1024) = 1024
[pid 14692] open("/usr/share/ssh/blacklist.RSA-2048", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such
file or directory)
[pid 14692] open("/etc/ssh/blacklist.RSA-2048", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or
directory)
[pid 14692] open("/etc/ssh/ssh_host_dsa_key", O_RDONLY|O_LARGEFILE) = 3
[pid 14692] open("/etc/protocols", O_RDONLY|O_CLOEXEC) = 4
[pid 14692] read(4, "# Internet (IP) protocols\n#\n# Up"..., 4096) = 2933
[pid 14692] open("/etc/hosts.allow", O_RDONLY) = 4
[pid 14692] open("/lib/i386-linux-gnu/libnss_dns.so.2", O_RDONLY|O_CLOEXEC) = 4
[pid 14692] stat64("/etc/pam.d", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[pid 14692] open("/etc/pam.d/common-password", O_RDONLY|O_LARGEFILE) = 8
[pid 14692] open("/etc/pam.d/other", O_RDONLY|O_LARGEFILE) = 4
```

SSH is a large chunk to bite off, but it gives a feel for strace usage. Being able to see which files an app opens can be useful ("where the hell is this config coming from?"). If you have a process that appears stuck, you can strace it and see what it might be doing via system calls. When some app is quitting unexpectedly without a proper error message, check if a syscall failure explains it. You can also use filters, time each call, and so so:

```
~/code/x86-os$ strace -T -e trace=recv curl -silent www.google.com. > /dev/null

recv(3, "HTTP/1.1 200 OK\r\nDate: Wed, 05 N"..., 16384, 0) = 4164 <0.000007>
recv(3, "fl a{color:#36c}a:visited{color:"..., 16384, 0) = 2776 <0.000005>
recv(3, "adient(top,#4d90fe,#4787ed);filt"..., 16384, 0) = 4164 <0.000007>
recv(3, "gbar.up.spd(b,d,1,!0);break;case"..., 16384, 0) = 2776 <0.000006>
recv(3, "$),a.i.G(!0)),window.gbar.up.sl("..., 16384, 0) = 1388 <0.000004>
recv(3, "margin:0;padding:5px 8px 0 6px;v"..., 16384, 0) = 1388 <0.000007>
recv(3, "){window.setTimeout(function(){v"..., 16384, 0) = 1484 <0.000006>
```

I encourage you to explore these tools in your OS. Using them well is like having a super power.

But enough useful stuff, let's go back to design. We've seen that a userland app is trapped in its virtual address space running in ring 3 (unprivileged). In general, tasks that involve only computation and memory accesses do *not* require syscalls. For example, C library functions likestrlen(3) and memcpy(3) have nothing to do with the kernel. Those happen within the app.

The man page sections for a C library function (the 2 and 3 in parenthesis) also offer clues. Section 2 is used for system call wrappers, while section 3 contains other C library functions. However, as we saw with `printf(3)`, a library function might ultimately make one or more syscalls.

If you're curious, here are full syscall listings for Linux (also Filippo's list) and Windows. They have ~310 and ~460 system calls, respectively. It's fun to look at those because, in a way, they represent *all that software can do* on a modern computer. Plus, you might find gems to help with things like interprocess communication and performance. This is an area where "Those who do not understand Unix are condemned to reinvent it, poorly."

Many syscalls perform tasks that take eons compared to CPU cycles, for example reading from a hard drive. In those situations the calling process is often *put to sleep* until the underlying work is completed. Because CPUs are so fast, your average program is **I/O bound** and spends most of its life sleeping, waiting on syscalls. By contrast, if you strace a program busy with a computational task, you often see no syscalls being invoked. In such a case, top(1) would show intense CPU usage.

The overhead involved in a system call can be a problem. For example, SSDs are so fast that general OS overhead can be more expensive than the I/O operation itself. Programs doing large numbers of reads and writes can also have OS overhead as their bottleneck. Vectored I/O can help some. So can memory mapped files, which allow a program to read and write from disk using only memory access. Analogous mappings exist for things like video card memory. Eventually, the economics of cloud computing might lead us to kernels that eliminate or minimize user/kernel mode switches.

Finally, syscalls have interesting security implications. One is that no matter how obfuscated a binary, you can still examine its behavior by looking at the system calls it makes. This can be used to detect malware, for example. We can also record profiles of a known program's syscall usage and alert on deviations, or perhaps whitelist specific syscalls for programs so that exploiting vulnerabilities becomes harder. We have a ton of research in this area, a number of tools, but not a killer solution yet.

And that's it for system calls. I'm sorry for the length of this post, I hope it was helpful. More (and shorter) next week, RSS and Twitter. Also, last night I made a promise to the universe. This post is dedicated to the glorious Clube Atlético Mineiro.