1. Starting Systems Programming: Part 2: Your program and the outside world: syscalls & files

A software article by Efron Amber Licht

MAR 2025

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

This is the second of four articles on the fundamentals of systems programming. In part 1, we **wrote a bunch of programs** to act as a toolset to investigate binary files, finishing with an overview of the **ELF** (Executable and Linkable Format) file format that defines executable binaries on linux systems. In *this* article, we'll start diving in to how programs interact with the outside world. We'll answer questions like:

- How do I actually read and write files? What's a file descriptor?
- How did my program get those command-line arguments anyways? How do flags differ from positional arguments?
- What *is* the process environment? What's actually happening when I set - or read - an environment variable?
- What's the deal with unix file permissions?
- How does starting a program actually happen?
- What's a syscall and how and why do I use them?

As usual, we're going to **write a lot of programs** to answer these questions - building up to creating our own shell from scratch.

Sidenote: library functions

This article's going to do a lot more work than the last one, so in the interest of brevity, we will *reuse functions between programs* without explicitly importing or copying them. I will only reuse functions previously defined *in this article*, so a simple `ctrl+f` should find the definition you're looking for pretty quickly.

---

1. Intro: Your Program and the outside world

Programs can manipulate their own memory, but if you want to actually *do* something, you need to access the outside world, whether that's reading from a file, writing to a network socket, starting another program, or even shutting down.

In general, programs interact in the outside world in three ways, from most to least common:

1. Your program starts with a set of **command-line arguments** and a **process environment** when it starts. These are just strings that must be interpreted by the program.

2. Your program can **read**, **write**, and **execute** other data, usually represented as **files**. Files normally represent permanent storage on a physical disk, but other

common “files” are things like network sockets, pipes, and even physical devices. This is done via **system call** - assuming you have the right permissions.

3. Your program can be interrupted by a **signal** sent from the operating system or another process. Signals are a way for the operating system to tell your program that something has happened. The most common signal is `SIGINT` - “SIGnal INTerrupt” - which asks a program to quit. Hit `ctrl+c` in your shell to send a `SIGINT` to the program running in the foreground. Those signals are generated by **system calls** too.

   Aside: shared memory & memory-mapped IO

   You can also interact with the outside world by memory-mapped IO or shared memory.

   - shared memory lets processes communicate by writing to the same memory.
   - memory-mapped IO lets ‘ordinary’ reads and writes to memory trigger IO operations.

   These have performance advantages because they let you skip the syscall & associated context switch. They’re fast, but **dangerous**. They’re beyond the scope of this series, but you should know they exist.

Let’s start with command-line arguments. These are probably pretty familiar, so we’ll cover them quickly.

---

2. `args`

When your program starts, it’s handed a list of strings called **command-line arguments**. Usually, these are exactly what you typed into your shell to start the program.

Let’s **write** a program, `printargs`, that prints its command-line arguments to standard output.

```
1  // printargs.go prints its command-line arguments to
   standard output.
2  package main
3  func main() {
4      for i, arg := range os.Args {
5          fmt.Fprintf(os.Stdout, "%d: %s\n", i, arg)
6      }
7  }
```

Let’s try it out…

```
#!/usr/bin/env bash
# IN
go run ./printargs.go foo bar
# OUT
0: /tmp/go-build389224751/b001/exe/printargs
1: foo
2: bar
```

Wait, that doesn't look right. This is because `go run` *compiles* and *then* invokes the program, so the first argument is the path to the compiled binary - here, in a temporary directory. Let's separate those steps and try again...

```
#!/usr/bin/env bash
# IN
go build -o printargs ./printargs.go
./printargs foo bar
# OUT
0: ./printargs
1: foo
2: bar
```

There we go. As you can see, the command-line arguments are *the strings you provided without any interpretation*. **It's up to your program to decide what to do with them.**

Command-line arguments are just strings, but they're usually interpreted as one of two things: **flags** or **positional arguments**.

2.1. Flags

A `flag` is a command-line argument starting with a dash `-` ("short" flag) or two dashes `--` ("long" flag"). Flags are used to specify options or settings for the program. Whether short or long, flags are either a **toggle** (like `--verbose`) or have a **value**. Traditionally, you can specify the value either with a second argument (like `--output file.txt`) or with an equals sign (like `--output=file.txt`).

Most programs take a mix of flags and positional arguments. Let's use the unix standard `grep` as an example:

```
#!/usr/bin/env bash
# IN
grep -r --color=always "foo" /tmp # search for "foo" in /
tmp, recursively, with color output
```

Traditionally, flags come *before* positional arguments, but this is just a convention. Some programs allow flags *after* positional arguments, but this can be confusing.

Let's **write a program**, `parseflags` to separate flags from positional arguments. In practice, you'll rarely do this, but it's worth walking through the logic by hand at least once to internalize it.

2.2. `parseflags.go`

2.2.1. Overview

1. Get the flags by consuming CLI args until we hit a non-flag, `--`, repeated flag, or end of arguments.
    1. `--` forces all remaining arguments to be positional.
    2. Repeated flags immediately terminate flag parsing with an error.
    3. The first non-flag argument terminates flag parsing: the rest are positional.
    4. Treat `-short` and `--long` flags the same.
2. Print the flags and positional arguments to standard output.

```
 1
 2   // parseflags parses command-line flags and returns a map
     of flag names to values and a slice of positional
     arguments.
 3   // the first non-flag argument terminates flag parsing;
     pass '--' to force all remaining arguments to be
     positional.
 4   // args should NOT include the command name.
 5   // Flags are of the form -name=value, --name=value, -name
     value, or --name value: we don't
 6   // treat short or long flags differently.
 7   // It is an error to set a flag more than once.
 8   func parseflags(args []string) (flags map[string]string,
     positional []string, err error) {
 9       flags = make(map[string]string)
10
11   FLAGS:
12       for len(args) > 0 {
13           s := args[0]
14           if len(s) <= 1 { // can't possibly be a flag
15               break FLAGS
16           }
17           if s == "--" { // end of flags
18               args = args[1:] // consume "--"
19               break FLAGS
20           }
21           if s[0] != '-' { // not a flag
22               break FLAGS
23           }
24           // strip off up to two leading '-'s
25           if s[1] == '-' {
26               s = s[2:]
27           } else {
28               s = s[1:]
29           }
30
31           // it's now a potential flag. is it of the form
     -name=value?
32           // look for '='
```

```
33            for i := range s {
34                if s[i] == '=' {
35                    key, value := s[:i], s[i+1:]
36                    if _, ok := flags[key]; ok {
37                        return nil, nil, fmt.Errorf("flag -%s
    already set", key)
38                    }
39                    flags[key] = value
40                    args = args[1:] // we've consumed one arg
41                    continue FLAGS
42                }
43            }
44
45            // it's not of the form -name=value. the next arg
    is the value.
46            // is there a next arg?
47
48            if len(args) == 1 { // no. error.
49                return nil, nil, fmt.Errorf("flag -%s missing
    value", s)
50            }
51
52            key, value := s, args[1] // the next arg is the
    value
53            if _, ok := flags[key]; ok {
54                return nil, nil, fmt.Errorf("flag -%s already
    set", key)
55            }
56            flags[key] = value
57            args = args[2:] // we've consumed two args
58
59        }
60        // what's left is positional arguments.
61        return flags, args, nil
62    }
63
```

Let's give it a shot.

```
1   #!/usr/bin/env bash
2   # IN
3   go run . -name efron --animal tapir -foo bar positional --
    notflag efron
4   # OUT
5   flag name=efron
6   flag animal=tapir
7   flag foo=bar
8   positional 0=positional
9   positional 1=--notflag
10  positional 2=efron
```

Let's try out the `--` pseudo-flag to force all remaining arguments to be positional.

```
1   #!/usr/bin/env bash
```

```
# IN
go run . -name efron -- --animal tapir
# OUT
flag name=efron
positional 0=--animal
positional 1=tapir
```

If it's not a flag, it's a **positional argument**.

2.3. Positional Arguments

A *positional argument* is interpreted based on its position in the argument list. `cp [src] [dst]` copies **s**ou**rc**e to **dest**ination. The `src` and `dst` are positional arguments.

Usually, positional arguments are mandatory and flags are optional, but this is just convention. Some program authors prefer to use flags for everything. Try to avoid having more than a few positional arguments, especially if they share types.

Command-line arguments are passed to a program when it starts via the `execve` system call, alongside the process environment (usually just called the `env`). That's the subject of the next section.

---

3. process environment (`env`)

The operating system also provides your program with an **environment** - a list of key-value pairs, separated by =. These are used to pass configuration information to your program. Each process is expected to pass it's environment to any child processes it starts; this is called "inheriting" the environment. By convention, environment variables are in `SCREAMING_SNAKE_CASE`.

Some basic examples of environment variables are (`USER`), home directory (`HOME`), and shell (`SHELL`).

Let's **write a program** to inspect our process environment. We'll call it `printenv`.

3.1. `printenv.go`

3.1.1. Overview:

1. get the list of environment variables in form KEY=VALUE from the go runtime.
2. look up the value of each key provided by the user and print it to standard output.
3. exit with status 0 if all were found.
4. print all missing variables to standard error and exit with status 1.

printenv.py: click here

```go
// printenv.go prints the value of each environment variable given as an argument.
// it exits with status 1 if any of the variables are not found.
package main

import (
    "fmt"
    "os"
)

func main() {
    // 1. get the list of environment variables in form KEY=VALUE from the go runtime.
    env := os.Environ()
    var printed int
    keys := os.Args[1:]
    // 2. look up the value of each key provided by the user and print it to standard output.
    for _, key := range keys {
        val, ok := lookupenv(env, key)
        if ok {
            fmt.Fprintf(os.Stdout, "%s\t%s\n", key, val)
            printed++
        }
    }
    // 3. exit with status 0 if all variables were found.
    if printed == len(keys) {
        os.Exit(0) //
    }
    // 4. print all missing variables to standard error and exit with status 1.

    fmt.Fprintf(os.Stderr, "missing %d/%d environment variables\n", len(os.Args)-1-printed, len(os.Args)-1)
    for _, key := range keys {
        if _, ok := lookupenv(env, key); !ok {
            fmt.Fprintf(os.Stderr, "%s\n", key)
        }
    }
    os.Exit(1) // 4. Exit with status 0.
}

// look up an environment variable by name, returning its value and whether it was found.
// in case of duplicates, return the last value.
// environment variables are stored as "key=value" strings.
func lookupenv(env []string, key string) (string, bool) { // LIBRARY FUNCTION: first defined in printenv.go

```

```
43        /*  You may have duplicated environment variables - the
      operating system doesn't care. It's up to the receiving
      program to decide what to do with them. Usually the last
      one wins; we'll do that here.
44        */
45        for i := len(env)-1; i >= 0; i-- {
46            e := env[i]
47            if len(e) < len(key)+1 { // +1 for the '='
48                continue
49            }
50            if e[:len(key)] != key {
51                continue
52            }
53            if e[len(key)] != '=' {
54                continue
55            }
56            return e[len(key)+1:], true
57        }
58        return "", false
59    }
```

Let's test out `printenv` on some common environment variables.

```
1   #!/usr/bin/env bash
2   # IN
3   go run ./printenv.go USER HOME SHELL
```

```
1   # OUT
2   USER    efron
3   HOME    /home/efron
4   SHELL   /bin/bash
```

Lemma: Setting Eenvironment Variables in the `bash` shell

Set an environment variable for a specific run of a program by prefixing the command with `ENVVAR=VALUE`. I.e., `USER=efron go run ./printenv USER` will print `efron`.
Set an environment variable for the rest of the shell session with `export ENVVAR=VALUE`

3.1.2. Example: Setting an environment variable for the rest of the shell session

```
1   #!/usr/bin/env bash
2   # IN
3   export ANIMAL=WOOLY_TAPIR
4   go run ./printenv ANIMAL
5   # OUT
6   ANIMAL  WOOLY_TAPIR
```

3.1.3. Example: Setting an environment variable for a single run of a program

```
#!/usr/bin/env bash
export ANIMAL=BAIRDS_TAPIR # set for the rest of the shell session
ANIMAL=MALAYAN_TAPIR go run ./printenv ANIMAL # only for this run; overrides the shell session variable
```

Sidenote: `HOME` and `~`

Some programs use the tilde (~) to represent the home directory, but this is a program-by-program convenience rather than fundamental to the operating system.
Most shells - like `bash` - expand ~ to the value of the `HOME` environment variable.
If you're using another program, don't assume ~ will work: actually look up the value of `HOME`.

```
#!/usr/bin/env bash
# IN:
echo ~
printenv HOME

# OUT:
/home/efron
/home/efron
```

`printenv`: exercises

- Modify `printenv` to print all environment variables if no arguments are provided. How should you handle duplicates?
- Modify `printenv` to print the environment variables in a sorted order. How should you handle dplicates? How should you handle case?

Sidenote: Eenvironment Namespacing

There is no mechanism to prevent two programs from using the same environment variable name. It's *up to us* to avoid conflicts. Avoid short names or common names like `PATH`, `HOME`, or `NAME`. Adding a short and unlikely-to-conflict prefix is a good idea. For example, if your company is Tapir Technology and you're writing a program called "monitor", you might use `TT_MONITOR_LOG_LEVEL` instead of `LOG_LEVEL`.

OK, so environment variables are just a simple key-value mapping... but how does the program get those environment variables in the first place?

Q: How does the program obtain these values? A: The operating system provides them when the program starts via the `execve` system call.

Eenvironment variables are passed to the program when it's started via the `execve` system call. `execve(path, args, env)` **replaces** the current process with a new one with the file located at `path`, the arguments `args`, and the environment `env`. In order to talk about `execve`, first we need to know a bit about system calls and files.

---

4. Syscalls

What *is* a syscall? A system call is a "function call" to the operating system executed by a special machine instruction, creatively named `SYSCALL`. The `SYSCALL` instruction hands control back to the operating system with a *request* to do something. The operating system then either does that thing or tells you it can't, then hands control back to your program.

In general, you don't make system calls directly - system calls are architecture (`amd64`, `riscv`, `arm64`, `6502`) and operating system (`linux`, `windows`, `macos`, `freebsd`) specific.

Instead, you'll use a library function that makes the syscall for you - usually `libc`'s `syscall`. We'll be using Go's `syscall` package. We'll be using that for the rest of this article - specifically, the `syscall.Syscall` and `syscall.Syscall6` functions.

> **Lemma: syscall library functions**
>
> The syscall library functions do the following
>
> - save your registers for later
> - put the function arguments in the correct registers for the syscall
> - use the SYSCALL opcode to switch to kernel mode
> - ... operating system does its thing ... <– ACTUAL SYSTEM CALL HERE
> - return to user mode
> - restore your registers
> - return the result of the syscall
> - handle any errors
>
> **TLDR: the syscall library function makes a syscall into an ordinary function call.**

4.1. syscall basics with go's `syscall.Syscall` and `syscall.Syscall6`

- `r1, r2, errno := syscall.Syscall(syscallno, arg1, arg2, arg3)` is used for ordinary syscalls with 3 or fewer arguments
- `r1, r2, errno := syscall.Syscall6(syscallno, arg1, arg2, arg3, arg4, arg5, arg6)` is used for syscalls with 4-6 arguments.

`syscallno` ('syscall number') is the number of the syscall you want to make, and the `arg` arguments are the arguments to that syscall - think of each syscallno as a function in a library, and the rest as the arguments *to* that function. Sometimes the `syscallno` is called a "trap" or "interrupt" for historical reasons.

Filesystem operations like `write` and `read` are some of the most common syscalls.

`write(fd, buf, count)` writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd. It returns the number of bytes written.

> **Lemma: File Descriptors**
>
> System calls don't operate on file "objects" - objects aren't real. Your machine only knows about registers and memory. Instead, they use integers called "file descriptors" (usually just `fd`) to open files. The operating system maintains a table of open files for each process. The file descriptor is an index into that table. These file descriptors do not necessarily correspond to the file's position in the filesystem. The operating system automatically opens three file descriptors for you when your program starts:
>
> - `STDIN` ("standard input") is fd 0.
> - `STDOUT` ("standard output") is fd 1.
> - `STDERR` ("standard error") is fd 2

To warm up, let's wrap the `write` syscall in a Go function:

4.1.1. Wrapping the `write` syscall

```
1  func gowrite(fd int, buf []byte) (int, error)
   {  // library function
2      """write the contents of buf to the file descriptor
   fd"""
3      n, _, errno := syscall.Syscall(
4          syscall.SYS_WRITE, // which syscall?
5          uintptr(fd),       // write to standard output
6
   uintptr(unsafe.Pointer(&buf[0])), // where to write from
7          uintptr(len(buf)), // how many bytes to write
8      )
```

```
 9       return int(n), errno // errno implements the error
    interface
10  }
```

> **Lemma: Pointers, `unsafe.Pointer`, and `uintptr`**
>
> Inside the machine, there's no such thing as a "pointer" - there's just registers and memory. A pointer is just a number that "points" to a certain address in memory. If we want to tell the operating system where to read or write data, all we can do is pass it a number that represents a memory address and hope that it interprets it correctly.
> We can do so in Go by converting a pointer to a number using `uintptr` (an **u**nsigned **int**eger large enough to hold a **p**oin**t**e**r**) by way of `unsafe.Pointer`.

Just like last time, let's write a program that writes "hello, world!" to standard output... but let's do it using syscalls rather than Go's `fmt` package.

4.2. syscallhelloworld.go

4.2.1. Overview

- Get a pointer to the sequence of bytes that make up the string `"hello, world!\n"`.
- Use the `write` syscall to write those bytes to standard output.
- Use the `exit` syscall to exit the program on success or failure.

`hello-world-syscall.py`: click here

```
 1  // syscallhelloworld.go writes "hello, world!" to standard
    output using the write syscall.
 2  package main
 3  import (
 4      "syscall"
 5      "unsafe"
 6  )
 7  func main() {
 8      var b = []byte("hello, world!\n")
 9      n, _, errno := syscall.Syscall(
10          syscall.SYS_WRITE, // which syscall?
11          uintptr(fd),       // write to standard output
12
    uintptr(unsafe.Pointer(&buf[0])), // where to write from
13          uintptr(len(buf)), // how many bytes to write
14      )
15      if errno != 0 {
16          fatalf("write: %v\n", errno)
```

```
17          }
18          if n != len(buf) {
19              fatalf("write: wrote %d bytes, expected %d\n", n,
    len(buf))
20          }
21      }
22
23      // fatalf writes a formatted string to standard error and
    exits with status 1. LIBRARY
24      func fatalf(format string, args ...interface{}) {
25          buf := []byte(fmt.Sprintf(format, args...))
26          syscall.Syscall(syscall.SYS_WRITE, STDERR,
    uintptr(unsafe.Pointer(&buf[0])), uintptr(len(buf))) // no
    point in checking the error here; we're about to exit.
27          syscall.Syscall(syscall.SYS_EXIT, 1, 0,
    0) // exit with status 1
28      }
```

Let's try it out...

```
1   #!/usr/bin/env bash
2   # IN
3   go run ./hello-world-syscall.go
4
5   # OUT
6   hello, world!
```

It works! **Every time you read or write a file, this is happening under the hood.**

Let's explore files a bit more. Last time, we wrote `cat` to concatenate files. Let's do the same thing again with syscalls.

Let's **write a program**, `syscallcat`, to read from a file and write to standard output using syscalls.

4.3. `syscallcat.go`

4.3.1. `syscallcat`: Overview

1. Open the file specified by the first argument w/ `SYS_OPEN`.
2. Read chunks into memory with `SYS_READ`.
3. Write those chunks to standard output with `SYS_WRITEE`.
4. Flush the output buffer to disk with `SYS_FSYNC`. (more on this when it comes up)
5. Close the file with `SYS_CLOSE`.
6. Exit with `SYS_EXIT`.

The following table summarizes the system calls used in `syscallcat.go`:

4.3.2. `syscallcat`: system calls used

name	number	arguments	description
close	3	fd	close the file descriptor `fd`
exit	60	status	exit the program with status `status`
fsync	74	fd	flush the file descriptor `fd` to disk
open	2	path, flags, mode	open a file at `path` with `mode` permissions and behavior specified by `flags`
read	0	fd, buf, count	read `count` bytes from file descriptor `fd` into `buf`
write	1	fd, buf, count	write `count` bytes starting at `buf` to file descriptor `fd`

`syscallcat.py`: click here

```go
// syscallcat.go opens the file specified by the first argument and writes its contents to standard output using raw syscalls.
package main

import (
    "fmt"
    "os"
    "syscall"
    "unsafe"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "usage: %s <file>\n", os.Args[0])
        os.Exit(1)
    }

    path := []byte(os.Args[1])   // convert the string to a byte array so we can point to it
    path = append(path, 0)           // null-terminate the string
    ptr := unsafe.Pointer(&path[0]) // point to the first byte of the array
    const MODE = syscall.O_RDONLY   // open the file for reading only
    const FLAGS = 0                 // we don't need any


    // 1. Open the file specified by the first argument w/ `SYS_OPEN`.

    fileDescriptor, _, err := syscall.Syscall(
        syscall.SYS_OPEN,
        uintptr(unsafe.Pointer(ptr)),
        MODE,
```

```
30              FLAGS,
31          )
32          if err != 0 {
33              fatalf("open: %v\n", err)
34          }
35
36
37          // 2. Read chunks into memory with `SYS_READ`.
38          // we've now opened the file. let's read from it and copy the data to standard output.
39          var buf [1024]byte // 1KB to read into
40
41      READ:
42          for {
43              ptr := &buf[0] // point to the first byte of the buffer
44              n, _, readErr := syscall.Syscall(
45                  syscall.SYS_READ,              // which syscall?
46                  fileDescriptor,                // tell it to read from the file we opened
47                  uintptr(unsafe.Pointer(ptr)), // where to write the data?
48                  uintptr(len(buf)),             // how many bytes to read?
49              )
50
51              // we'll check the error in a second - we may have read some data even if there was an error.
52
53
54              // 3. Write those chunks to standard output with `SYS_WRITE`.
55
56              // standard output just another file descriptor: it's automatically opened for us when the program starts. it's always file descriptor 1.
57              const FD_STDOUT = 1
58
59              // we want to write all the data we read to standard output.
60              // writes are not guaranteed to write all the data you ask for in one go. among other things,
61              // signals like SIGPIPE or SIGINT can interrupt them (more on that later).
62              // we need to keep writing until we've written all the data we read.
63              // functions like io.Copy usually do this for you.
64              for offset := uintptr(0); offset < n; {
65                  ptr := &buf[offset] // point to the first byte we need to write
66                  m, _, writeErr := syscall.Syscall(
67                      syscall.SYS_WRITE,
68                      FD_STDOUT,
```

```
                    uintptr(unsafe.Pointer(ptr)),
                    n,
                )
                if m == n {
                    continue READ
                }
                if writeErr != 0 {
                    fatalf("write: %v\n", writeErr)
                }
                offset += m
            }

            if readErr != 0 {
                fatalf("read: %v\n", readErr)
            }

            if n == 0 { // we've read all the data; exit the loop
                break READ
            }
        }

        // we've now written all the data we read from standard input to standard output... or have we?
        // it's usually pretty inefficient to do lots of small writes to permanent storage, so operating systems maintain a buffer of data to write to disk when it's convenient.
        // we can force the operating system to write that buffer to disk with the fsync syscall.
        // fsync(fd) writes the buffer for file descriptor fd to disk, blocking until it's done.
        // the similarly-named sync() does this for _all_ open files; it's usually better to be specific.
        _, _, _ = syscall.Syscall(syscall.SYS_FSYNC, FD_STDOUT, 0, 0) // no error checking here. we're about to exit anyway.

        // 5. Close the file with `SYS_CLOSE`.
        syscall.Syscall(syscall.SYS_CLOSE, fileDescriptor, 0, 0)

        // 6. Exit with `SYS_EXIT`.
        syscall.Syscall(syscall.SYS_EXIT, 0, 0, 0)
    }
```

Let's try it out...

```
#!/usr/bin/env bash
# IN
echo "hello, world!" > hello.txt
syscallcat hello.txt
# OUT
hello, world!
```

4.3.3. `syscallcat` exercises:

- Modify `syscallcat` to read from standard input if no file is specified. HINT: standard input is file descriptor 0.
- Modify `syscallcat` to read from multiple files and concatenate them to standard output, actually implementing the `cat` tool.

`syscallhello.go` and `syscallcat.go` gave us some idea of how programs interact with the outside world - but how do programs start in the first place? What is *actually happening* when you click an icon or type `grep foobar` into your shell?

A system call, of course: `execve`, which starts executing a new program - assuming you have the proper permissions.

But wait, who's "you"? And what are those "permissions"? We'll quickly talk about ownership and access control and then get back to systems calls and `execve`.

---

5. Ownership and Access Control

All modern operating systems are *multi-user*, *multi-program* operating systems - the files and programs of many users can share resources like the CPU, memory, and disk. In a system like this, it's important to control who can touch what so that ordinary users can't damage the system or each other - **access control**.

> This is a complicated subject and I'm going to gloss over a lot here in order to give you a quick & dirty overview: please don't treat it as gospel.

The original form of access control - and still the most common - are **file permissions**.

> **Sidenote: the genealogy of modern operating systems**
>
> All modern operating systems descend from one of two sources.
>
> - System V Unix (1983) was the origin of Linux, BSD, and through BSD, MacOS.
> - Windows NT (1993) is the origin of modern Windows.

File permissions let you control who can **read**, **write**, and **execute** a file. Conveniently, these map directly to the syscalls we just covered: `read`, `write`, and `execve`. One of reasons these are *system calls* rather than 'normal' functions is because the operating system needs to check permissions before allowing the operation to proceed.

File permissions divide the world into three categories: the **user** who owns the file, a single **group** who also has access, and **everyone else**. For each category, you can set permissions to **read**, **write**, and **execute** individually.

These are usually represented either as a symbolic string in the form `rwxr-xr--`, where `r` is read, `w` is write, and `x` is execute, in the order `user`, `group`, `other`. You can also represent these as an **octal** number (like `0755`). The wikipedia article has a pretty good explanation of the octal and symbolic representations, so I won't go into it here.

Let's give a couple examples:

Symbolic	Octal	Description
`rwxrwxrwx`	`0777`	Everyone can read, write, and execute
`---------`	`0000`	No one can do anything
`rwx------`	`0700`	The user can read, write, and execute: no one else can do anything
`rwxr-xr--`	`0754`	The user can read, write, and execute; the group can read and execute; other people can only read.

5.0.1. Exercises: File Permissions

- **write a program**, `printperms`, that prints the permissions of a file in symbolic form.
- **write a program**, `chmod`, that changes the permissions of a file to the permissions specified on the command line. It should take both octal and symbolic permissions.
- Modify `chmod` to allow you to *modify* permissions for the user, group, and other categories using `+` and `-` operators. That is, `chmod o+r file` should add read permissions for others, and `chmod g-w file` should remove write permissions for the group.
- Modify `chmod` to allow you to set permissions for *all* categories at once using `a` for "all". That is, `chmod a-r file` should remove read permissions for all categories.

5.1. Users

Unix maintains a list of users, each with a unique numeric id called a UID (**U**ser **ID**) and a human-readable name. Each file and process have a user who 'owns' them. The `getuid` syscall returns the UID of the current user. **Usernames are a human-readable convenience; the operating system uses the UID.** The `id` command prints the UID and GID of the current user.

5.1.1. Exercises: Users

- **Write a program**, `getusername`, that prints the username of the specified UID. If no UID is specified, it should print the username of the current user - use the `getuid` syscall to get the current user's UID.
    - **hint**: the `/etc/passwd` file contains a list of users and their UIDs. Use the `id` command to find your own username and UID and use that to figure out

how to parse the file.
    - **hint** - no need to parse the *whole* file.

Each user *must* be part of at least one group - their "primary" group - and may be part of more. That's the subject of the next section.

5.2. Groups

Just like users, Unix maintains a list of groups, each with a unique numeric id - here called a GID rather than UID. A **group** contains zero or more users. You can give file permissions to a *group* of users rather than a single user - for example, allowing students to read assignments but not write them. Each group has a unique numeric id called a GID (**G**roup **ID**) and a human-readable name. The `getgid` syscall returns the *primary* GID of the user who called it. The `getegid` syscall returns the *effective* GID of the user who called it - that is, the GID of the group that you are currently using to access a file.

5.2.1. Exercises: Groups

- **Write a program**, `getgroupname`, that prints the username corresponding to the specified GID. If no GID is specified, it should print the group name of the current user - use the `getgid` syscall to get the current user's GID.
    - **hint**: the `/etc/group` file contains a list of groups and their GIDs.
- Add a `-e` flag to `getgroupname` that prints the *effective* GID of the current user instead of the primary GID.
- Experiment with `chmod` and `chown` to see how `getgroupname` behaves. You may need to create a new user and/or group to see the effects.

When a program starts, it inherits the user and group of the process that started it. *How* those processes start is the subject of the next section - `execve`.

---

6. Starting a Program with `execve`

`execve` (**exec**ute with arg **v**ector and **e**nvironment) is the system call that starts a new program by **replacing** the current process with the new one. That's the subject ofThe new process maintains the file descriptors of the old process, but everything else is new.

Let's **write a program** that serves as the core of a shell: one that listens to standard input, echoes it's arguments to stderr, then runs the program specified. For now, we'll limit this to running programs specified by an `absolute path` - that is, a path that starts with a `/`.

6.0.1. Lemma: C arrays and `**byte`

Like C strings, arrays are just pointers to memory, terminated via nulls (`0`). In go terms, if a string is `*byte`, an *array* of strings is `**byte`.

6.1. syscallexec.go

6.1.1. Overview

1. Check that the user has provided an argument and that it's an absolute path.
2. Convert the go-style command-line arguments and environment (`[]string`) to the C-style null-terminated arrays of null-terminated strings (`**byte`).
3. Call `execve` via the `SYS_EXECVE` syscall to start the new program.

```
1   // syscallexec.go runs another program specified by
    absolute path using the execve system call. it uses its
    first command-line argument as the path to the program to
    run and the rest of the command-line argument as the name
    of
2   // the program to run.
3   // it should have exactly the same effect as just running
    the program directly.
4   package main
5
6   import (
7       "fmt"
8       "os"
9       "syscall"
10      "unsafe"
11  )
12
13  func main() {
14      //1. Check that the user has provided an argument and
    that it's an absolute path.
15      if len(os.Args) < 2 {
16          fatalf("usage: %s <command> [args...]\n",
    os.Args[0])
17      }
18      goargs :=
    os.Args[1:] // the first item is the command, the rest are
    arguments
19
20      // we'll cover the process PATH later. for now, let's
    protect our users from themselves.
21      if len(os.Args[1]) == 0 || os.Args[1][0] != '/' {
22          fatalf("error: %s is not an absolute path\n",
    os.Args[1])
23      }
24      exec(goargs, os.Environ()) // we'll talk about the
    environment in just a bit.
```

```
25  }
26
27
28  // execute a program, replacing the current process. on
    success, this never returns,
29  // so err is always a non-zero syscall.Errno.
30  func exec(args, env []string) error { // LIBRARY
31      // we need to convert the command and arguments to a
    slice of pointers to null-terminated strings to pass to
    execve.
32      // we need a null-terminated array of pointers to null-
    terminated strings.
33
34      cargs := make([]unsafe.Pointer, len(args)+1) // +1 for
    null terminator
35      for i := range args {
36          cargs[i] = cstr(args[i])
37      }
38
39      cenv := make([]unsafe.Pointer, len(env)+1) // +1 for
    null terminator
40      for i := range env {
41          cenv[i] = cstr(env[i])
42      }
43
44      path := cstr(args[0])
45
46      // 3. Call `execve` via the `SYS_EXECVE` syscall to
    start the new program.
47      _, _, err := syscall.Syscall(
48          syscall.SYS_EXECVE,
49          uintptr(path),                      // path to the
    program to run as null-terminated string
50          uintptr(unsafe.Pointer(&cargs[0])), // pointer to
    pointer to byte.
51          uintptr(unsafe.Pointer(&cenv[0])),  // pointer to
    pointer to byte.
52      )
53      return err
54  }
55
56  // 2. Convert the go-style command-line arguments and
    environment (`[]string`) to the C-style null-terminated
    arrays of null-terminated strings (`**byte`).
57  // cstr converts a Go string to a null-terminated C string.
58  // this allocates memory.
59  func cstr(s string) unsafe.Pointer { // LIBRARY
60      b := make([]byte, len(s)+1)
61      copy(b, s) // copy the string into the buffer. the
    leftover byte will be the null terminator.
62      return unsafe.Pointer(&b[0])
63  }
64
```

Let's try it out by calling `echo` - or more specifically, `/bin/echo` - with a few arguments.

```
#!/usr/bin/env bash
# write the program to a file at the absolute path /bin/syscallexec
go build -o /bin/syscallexec ./syscallexec.go

# use it to run /bin/echo
/bin/syscallexec /bin/echo hello, world!

# call our program recursively
/bin/syscallexec /bin/syscallexec /bin/echo hello, hello, world!
```

```
# OUT
hello, world!
hello, hello, world!mj,jkm,
```

While this works, it's not particularly useful. Some limitations include:

- We can only run a single program exactly once, since `execve` *replaces* the current process. We can solve this by **forking** the process first - more on this soon.
- We don't have any control over our child program. We can do this via signals and pipes - again, more soon.
- We inherit the parent environment without modifications.
  > **exercise**: Modify `syscallexec` to take a new flag `-e` that sets the environment for the child process. You need to be able to replace multiple environment variables.
- We have to know the absolute path to the program we want to run. We can solve this by looking up the program in the `PATH` environment variable - this is called **command resolution**.

6.1.2. Lemma: the `PATH` environment variable and command resolution

Commands are resolved by looking them up in your `PATH` environment variable. Path is a colon-separated `:` list of directories to search for programs. First match wins.
If `PATH=/bin:/usr/bin:/usr/local/bin`, it contains the following directories:

- `/bin`
- `/usr/bin`
- `/usr/local/bin`

If I enter `syscallexec` in my shell, the shell will search for `syscallexec` in each of those directories in order, stopping at the first match. In this case, since `/bin/syscallexec` exists, it will run that program. We say that `syscallexec` **resolves** to `/bin/syscallexec`.

6.1.3. Example: Command Resolution in the `bash` shell

```
> # IN
> # note: no /bin/ prefix
> syscallexec echo hello, world!
> ```
>
> ```bash
> # OUT
> hello, world!
> ```

In the next section, we'll **write a program** to resolve commands ourselves - that is, find out `which` program we're running.

---

7. command resolution with `whiche.go`

`whiche` - pronounced "witchy" - finds the absolute path to a command using the `PATH` environment variable like a shell would.

7.1. overview

1. get the environment from the go runtime.
2. resolve the PATH environment variable to a list of directories via string manipulation.
3. Use the SYS_STAT system call to check if the file exists in each directory.
4.
if no match is found, print an error message to standard error and exit 1.

[whiche.py: click here](https://gitlab.com/efronlicht/blog/-/blob/64b5bd1c71896796fd486c24d9e36aec688522ff/articles/startingsystems/cmd/pythonports/whiche.py)

```go
// whiche.go ("witch-e") finds the absolute path to a command using the PATH environment variable like a shell would.
package main

import (
    "fmt"
    "os"
    "syscall"
    "unsafe"
)
```
```

```
38
39  func main() {
40      if len(os.Args) != 2 {
41          fmt.Fprintf(os.Stderr, "usage: %s
    <command> [args...]\n", os.Args[0])
42          os.Exit(1)
43      }
44      // 1. get the environment from the operating
    system.
45      env := os.Environ()
46      // 2. resolve the PATH environment variable to
    a list of directories via string manipulation.
47      var path []string
48      {
49          // this whole block is just doing
    `strings.Split(rawpath, ":")`.
50          // it's good to practice ordinary string
    manipulation every now and then.
51          var start int
52          rawpath := getenv(env, "PATH")
53          for i := range rawpath {
54
55              if rawpath[i] != ':' {
56                  continue
57              }
58              if start == i {
59                  continue // skip empty strings.
60              }
61              path = append(path, rawpath[start:i])
62              start = i + 1
63          }
64          if start < len(path) {
65              path = append(path, rawpath[start:])
66          }
67      }
68
69      // 3. find the first match in the PATH
    directories; print it to standard output and exit
    0.
70      for _, dir := range path {
71          if path, err := exists(dir + "/" +
    os.Args[1]); err == nil && path {
72              // found it. print & exit.
73              fmt.Println(dir + "/" + os.Args[1])
74              os.Exit(0)
75          }
76      }
77      // 4. if no match is found, print an error
    message to standard error and exit 1.
78
79      fmt.Fprintf(os.Stderr, "%s: command not
    found\n", os.Args[1])
80      os.Exit(1)
81  }
```

```
 82
 83  // getenv retrieves the value of the environment
     variable named by the key, or an empty string if
     it's not set.
 84  func getenv(environ []string, key string) string {
 85      key +=
     "=" // environment variables are stored as
     "key=value"
 86      n := len(key)
 87      for i := range environ {
 88          if len(environ[i]) < len(key) {
 89              continue
 90          }
 91          // KEY=VALUE
 92          if environ[i][:n] == key { // KEY=
 93              return environ[i][n:] // VALUE
 94          }
 95
 96      }
 97      return ""
 98  }
 99
100  // check if a file exists using stat (https://
     linux.die.net/man/2/stat)
101  // (true, nil) if it does.
102  // (false, nil) if it doesn't.
103  // (false, error) if there was an error.
104  func exists(path string) (bool, error) {
105      p := cstr(path)
106      var statbuf [144]byte // we'll worry about
     this later.
107      // the STAT system call fills in a stat
     structure with information about the file,
108      // returning 0 on success and -1 on error.
109      // errno will be set to the error code if it
     fails.
110      _, _, err :=
     syscall.Syscall(syscall.SYS_STAT, uintptr(p),
     uintptr(unsafe.Pointer(&statbuf)), 0)
111      switch err {
112      case 0: // success!
113          return true, nil
114      // the opaquely named ENOENT means Error NO
     ENTry; aka, "file not found".
115      case syscall.ENOENT: // file doesn't exist.
116          return false, nil
117      default: // some other error.
118          return false, err
119      }
120  }
121
122  // cstr converts a Go string to a null-terminated
     C string.
123  // this allocates memory.
```

```
124   func cstr(s string) unsafe.Pointer {
125       b := make([]byte, len(s)+1)
126       copy(b,
      s) // copy the string into the buffer. the
      leftover byte will be the null terminator.
127       return unsafe.Pointer(&b[0])
128   }
```

Let's try it out...

```
1   #!/usr/bin/env bash
2   # IN
3   go run ./whiche.go echo
4
5   # OUT
6   /usr/bin/echo
```

`whiche`**: exercises**

- Add a flag `-a` to `whiche` that prints *all* matches in the `PATH` environment variable in order of resolution, separated by newlines.
- **Write a program**, `run`, that combines the functionality of `whiche` and `syscallexec` to run a program specified by name.

One last topic to cover before we bring everything together - **signals**.

---

8. signals

Sometimes you need to reach in to a program and tell it about something that's happened outside of its control, like a broken network pipe (`SIGPIPE`), a user interrupt (`SIGINT`), or a request to shut down (`SIGTERM`). These are called signals.

The single most common signal is `SIGSEGV` - "segmentation fault" - which is sent by the operating system when your program tries to access memory it shouldn't, usually by dereferencing a null pointer.

Let's **write a program** that crashes with a segmentation fault to see it in action.

8.1. segfault.go

```
1   // https://go.dev/play/p/0l8t2y_aQ92
2   package main
3
4   func main() {
5       var nullptr *int
6       _ = *nullptr
7   }
```

```
#!/usr/bin/env bash
# IN
go run ./segfault.go
```

```
# OUT
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0
pc=0x468f22]

goroutine 1 [running]:
main.main()
/home/efron/scratch/segfault.go:5 +0x2
```

These signals originate via the `kill` system call. `kill(pid, signal)` sends the signal `signal` to the process with the process ID `pid`. The name is misleading - while *many* signals kill a process, most are used for other purposes.

7.3.2. Lemma: Common signals

The following table summarizes the most common signals you'll encounter.

SIGNAL	DESCRIPTION	EXAMPLE	NOTE
SIGINT	interrupt from the keyboard; usually signals "start shutting down at your > convenience"	`ctrl+c` on a hanging command-line program	
SIGPOLL	I/O is ready to read	waiting for network data	
SIGTERM	start shutting down *now*	`kill`	can be caught or ignored
SIGKILL		`kill -9`	cannot be caught or ignored
SIGSEGV	memory access violation	see above	crashes the program

Let's **write a program** to send signals to other programs.

8.2. sendsignal.go

7.3.3.1. Overview

1. Parse the PID and signal as integers from the command line.

2. Send the signal to the process with the given PID via the `kill` system call.

[`sendsignal.py`: click here]

```go
package main

import (
    "fmt"
    "os"
    "strconv"
    "syscall"
)

// sendsignal.go sends a signal to a process by PID.
// usage: sendsignal <pid> <signal>
func main() {
    if len(os.Args) != 3 {
        fatal(fmt.Errorf("usage: %s <pid> <signal>", os.Args[0]))
    }
    pid, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fatal(err)
    }
    signal, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fatal(err)
    }
    _, _, errno := syscall.Syscall(syscall.SYS_KILL, uintptr(pid), uintptr(signal), 0)
    if errno != 0 {
        fatal(errno)
    }
}

func fatal(err error) {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

```

If we want to test this out, we'll need

- a process that lasts long enough to send a signal to
- that process's PID

8.3. Lemma: PID

> The PID (process Identifier) uniquely identifies a process on a system. Each process knows its own PID via the `getpid` system call, and it can find the

PID of it's *parent* (the process that started it) via the `getppid` system call.
All processes form a tree with the `init` process - process 1 - at the root.

Let's **write a program** that prints it's PID to standard output and then waits for a signal. We'll call it `killme`.

8.4. killme.go

8.4.1. Overview

1. use the `getpid` system call to get the PID of the current process.
2. print the PID to standard output once.
3. print 'still alive' to standard error every 15 seconds.
   Program

`killme.py`: click here

```
 1  package main
 2  func main() {
 3      // 1. use the `getpid` system call to get the PID of
    the current process.
 4      pid, _, _ := syscall.Syscall(syscall.SYS_GETPID, 0, 0,
    0)
 5      // 2. print the PID to standard output once.
 6      fmt.Println(pid)
 7      // 3. print 'still alive' to standard error every 15s.
 8      for ; ; time.Sleep(15*time.Second) {
 9          fmt.Fprintln(os.Stderr, "still alive")
10      }
11  }
```

Let's test it out. We'll start `killme` as a background process, wait for one print of 'still alive', then use `sendsignal` to send `SIGKILL` (9 on linux) to the process.

```
1  #!/usr/bin/env bash
2  # IN
3  go run ./killme.go &
```

```
1  # OUT
2  8656 # this number will be different on your machine.
3  still alive
```

```
1  #!/usr/bin/env bash
2  # IN
3  PID=8656 # from above
4  SIGNAL=9 # SIGKILL
5  go run ./sendsignal.go $PID $SIGNAL
```

```
1  # OUT
2  signal: killed
```

The `killme` process should have stopped printing 'still alive' to standard error. If you run `sendsignal` again, you'll get an error:

```
#!/usr/bin/env bash
# IN
go run ./sendsignal.go 8656 9
```

```
# OUT
no such process
```

Most signals terminate the process they're sent to unless specifically caught by the program. Your language will provide some way to set this up by wrapping the `sigaction` system call - in Go, that's the `os/signal` package.

Let's **write a program**, `killmeslowly`, that exits only after it's received 5 `SIGINT` signals. (Remember, `ctrl+c` sends a `SIGINT` to the currently running program in your shell.) We'll use the `os/signal` package to catch the signals.

8.5. killmeslowly.go

there is no python equivalent for this program.

8.5.1. Overview

1. Print our `PID` to standard output.
2. Register a signal handler for `SIGINT`.
3. Count down from 5 to 1 on each `SIGINT` on stderr.
4. Exit on the fifth and final `SIGINT` with status 0.

8.5.2. Program

```
// killmeslowly runs until it receives 5 SIGINT signals.
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
)

func main() {
    // 1. Print our `PID` to standard output.

    fmt.Println(os.Getpid())

    //  2. Register a signal handler for `SIGINT`.

```

```
18      ch := make(chan os.Signal, 5)      // always make a
    buffer of at least 1 so you don't drop signals
19      signal.Notify(ch, syscall.SIGINT) // forward SIGTERM
    and SIGINT to ch
20
21      // 3. Count down from 5 to 1 on each `SIGINT` on
    stderr.
22      fmt.Fprintf(os.Stderr, "waiting for SIGINT\n")
23      remaining := 5
24      for range ch {
25          remaining--
26          if remaining == 0 {
27              fmt.Fprintf(os.Stderr, "exit\n")
28              os.Exit(0)
29          }
30          fmt.Fprintf(os.Stderr, "got SIGINT: %d more to
    exit\n", remaining)
31      }
32  }
```

Let's test it out. Since `ctrl+c` sends a `SIGINT` to the currently running program in your shell, we don't need to fire up any background processes.

```
1  #!/usr/bin/env bash
2  # IN
3  go run ./killmeslowly.go
```

```
 1  # OUT
 2  43555
 3  waiting for SIGINT
 4  # ctrl+c
 5  got SIGINT: 4 more to exit
 6  # ctrl+c
 7  got SIGINT: 3 more to exit
 8  # ctrl+c
 9  got SIGINT: 2 more to exit
10  # ctrl+c
11  got SIGINT: 1 more to exit
12  # ctrl+c
13  exit
```

That's the basics of signals. We've only scratched the surface here, but it should be enough to orient you. Make sure to read the docs for both your language - like go's os/signal package - and your operating system - like the man page for signal

We've covered a lot of ground so far. Let's see if we can combine all of our new knowledge to write something useful and systems-programming-y: an interactive command-line interpreter (aka, a shell).

At it's core, a shell (or command-line interpreter)

- reads a line from standard input
- interprets it as a command

- runs the command

We now know how to do all of these things using linux system calls. In our final section, let's **write a program** that does just that - `syscallshell`.

---

9. bringing it all together with `syscallshell`

`syscallshell` is a simple shell that runs subcommands entered by the user.

9.1. Overview

1. Read a line at a time from standard input.
2. Split the line into a command and arguments (don't worry about quotes or escaping for now).
3. Use the PATH environment variable to resolve the command to an absolute path.
4. Fork a new process
    1. in the child, use `execve` to start the new program.
    2. in the parent, wait for the child to finish.
5. Repeat until we get a signal from the operating system to quit via `SIGINT` (ctrl+c). Print a message and exit with status 0.

    there is no python equivalent for this program. ran out of time.

9.2. `syscallshell.go`

```
 1  // syscallshell.go implements a simple shell that runs
    subcommands entered by the user. it uses the PATH
    environment variable to find the commands to run.
 2  // the shell reads a line at a time from standard input,
    splits it into arguments, and runs the command.
 3  // example usage:
 4  //
 5  //  echo -e "echo hello\necho goodbye" | go run
    syscallshell.go
 6  //  hello
 7  package main
 8
 9  import (
10      "bufio"
11      "errors"
12      "fmt"
13      "os"
14      "strings"
15      "syscall"
16      "unsafe"
17  )
18
```

```
19  func main() {
20      // 1. read a line at a time from standard input.
21      for scanner := bufio.NewScanner(os.Stdin);
    scanner.Scan(); { // scan a line at a time.
22          line := scanner.Text()
23          line = strings.TrimSpace(line)
24          if line == "" {
25              continue
26          }
27          // 2. Split the line into a command and arguments
    (don't worry about quotes or escaping for now).
28          args := strings.Fields(line) // split on
    whitespace.
29
30          // the first argument is the command to run, as
    usual. now you get why it's like that ;).
31
32          // 3. Use the PATH environment variable to resolve
    the command to an absolute path.
33          path, err := whiche(os.Environ(), args[0])
34          if err != nil {
35              fmt.Fprintf(os.Stderr, "%q: %v\n", args[0],
    err)
36              continue
37          }
38
39          name := path
40
41          // replace the command with the resolved path so
    call() has to do less work.
42          args[0] = path
43          // 4. Fork a new process; in the child, use
    `execve` to start the new program.
44
45          status, err := call(args, os.Environ())
46          if err != nil {
47              fmt.Fprintf(os.Stderr, "%s: %v\n", name, err)
48          }
49          if status != 0 {
50              fmt.Fprintf(os.Stderr, "%s: exit status
    %d\n", name, status)
51          }
52      }
53  }
54
55  // call runs a command like a shell would.
56  // it
57  //   - resolves the PATH environment variable to find the
    command (no syscalls, just string manipulation)
58  //   - forks a new process using FORK
59  //   - CHILD executes the command in the child process
    (syscall.EXECVE)
60  //   - PARENT waits for the child process to finish in the
    parent process.
```

```
61  //   - PARENT returns the exit status of the child
    process.
62  func call(args []string, env []string) (status int, err
    error) { // LIBRARY: first defined in syscallshell.go
63      { // bounds checks
64          if len(args) == 0 {
65              return 0, errors.New("no command")
66          }
67          if args[0] == "" {
68              return 0, errors.New("empty command")
69          }
70          if args[0][0] != '/' {
71              return 0, errors.New("command must be an
    absolute path; try using lookupPath")
72          }
73      }
74      // 4. Fork a new process
75      // spawn a new process.
76      // we'll know if we're the parent or the child based
    on the return value of FORK.
77      // the CHILD gets a 0.
78      // the parent either gets PID > 0 (child's PID) or a
    negative number (error).
79      pid, _, errno := syscall.Syscall(syscall.SYS_FORK, 0,
    0, 0) // PID: *P*process *ID*entifier
80      if errno != 0 {
81          return status, fmt.Errorf("syscall: fork: %v",
    err)
82      }
83
84      // there are 3 cases:
85      // - we spawn a new process, it calls EXECVE, and
    succeeds (exit status 0)
86      // - we spawn a new process, it calls EXECVE, and
    fails (exit status 1)
87      // - we spawn a new process, it fails to call EXECVE
    (bad path? weird permissions? etc.)
88
89      // we want to find about about the third case. we can
    re-use the exit status to signal that the exec failed.
90      const STATUS_FAILED_EXEC = 0xB01D // magic number to
    signal an exec error. picked at random.
91
92      // 4.1: in the child, use `execve` to start the new
    program.
93      if isChild := pid == 0; isChild { // we're the child.
94          err := exec(args, env)
95          // WARNING: might be tempting to return the error
    here - but the parent process will never see it,
96          // since we're the child. instead, we'll print the
    error and exit.
97          // let's use our magic number to signal that we
    couldn't exec the command.
```

```
 98            fmt.Fprintf(os.Stderr, "syscall: execve: %v\n",
    err)
 99            os.Exit(STATUS_FAILED_EXEC)
100        }
101
102        // 4.2: in the parent, wait for the child to finish.
103        { // we're the parent. wait for the child to finish.
104            // the WAIT system call waits for a child process
    to finish or for a signal, whichever comes first.
105            // returning the PID of the child and its exit
    status.
106            // if the child hasn't finished yet, it will block
    until it does.
107            // if the child has already finished, it will
    return immediately.
108            var waitstatus uint32
109            pid := syscall.Syscall(syscall.SYS_WAIT4, pid,
    uintptr(unsafe.Pointer(&waitstatus)), 0)
110            fmt.Fprintf(os.Stderr, "pid %d exited with status
    %d\n", pid, waitstatus)
111
112            // the exit status is in the upper 8 bits of the
    status.
113            // the lower 8 bits are the signal that killed the
    process, if any.
114            status = int(waitstatus >> 8)
115            if status == STATUS_FAILED_EXEC {
116                return status, errors.New("execve failed")
117            }
118            // signal handling could also happen here; we'll
    leave it out for now.
119            return status, nil
120        }
121    }
122
123    // execute a program, replacing the current process. on
    success, this never returns,
124    // so err is always a non-zero syscall.Errno.
125    func exec(args, env []string) error { // LIBRARY: first
    defined in syscallexec.go
126        // we need to convert the command and arguments to a
    slice of pointers to null-terminated strings to pass to
    execve.
127        // we need a null-terminated array of pointers to
    null-terminated strings.
128
129        cargs := make([]unsafe.Pointer, len(args)
    +1) // +1 for null terminator
130        for i := range args {
131            cargs[i] = cstr(args[i])
132        }
133
134        cenv := make([]unsafe.Pointer, len(env)+1) // +1 for
    null terminator
```

```
135         for i := range env {
136             cenv[i] = cstr(env[i])
137         }
138 
139         path := cstr(args[0])
140 
141         _, _, err := syscall.Syscall(
142             syscall.SYS_EXECVE,
143 
    uintptr(path),                      // path to the program
    to run as null-terminated string
144             uintptr(unsafe.Pointer(&cargs[0])), // arguments
    as null-terminated array pointer to null-terminated
    strings
145 
    uintptr(unsafe.Pointer(&cenv[0])),  // environment as
    null-terminated array pointer to null-terminated strings
146         )
147         return err
148     }
149 
150     // whiche resolves the path to a command using the PATH
    environment variable like a shell would.
151     // it returns the absolute path to the command, or an
    error if the command couldn't be found.
152     // suppose our path is "/bin:/usr/bin:/usr/local/bin" and
    the command is "ls".
153     // we'll look for "/bin/ls", "/usr/bin/ls", and "/usr/
    local/bin/ls", in that order.
154     // the first one that exists is returned.
155     // absolute paths are returned as-is.
156     func whiche(env []string, command string) (string, error)
    {
157         // 4 cases
158         switch {
159         case strings.Contains(command, ".."):
160             return "", errors.New("no weird relative paths,
    please") // we'll handle this later.
161         case command == "":
162             return "", errors.New("empty path")
163         case command[0] == '/': // already absolute. nothing
    to do.
164             return command, nil
165         case command[0] == '.': // relative path to the
    current working directory.
166             // working directories are somewhat complicated;
    different shells have different rules.
167             // we'll skip this for now and rely on the
    operating system to handle it.
168             wd, _ := os.Getwd()
169             return lookupPath(command, wd)
170         default: // look up the command in the PATH
    environment variable.
171 
```

```
172             // we'll resolve the PATH environment variable to
    find the command.
173             // we know from our last program that environment
    variables are handed to a program as an array of null-
    terminated strings
174             // by the EXECVE syscall. the go runtime converted
    those to go-style strings for us before go's main()
    function was called;
175             // we'll grab them with os.Environ() and convert
    them back to C-style strings.
176             // find the value of the PATH environment
    variable; it's a ':'-separated list of directories, like /
    bin:/usr/bin:/usr/local/bin
177             pathEnv := getenv(env, "PATH")
178
179             // resolve the PATH environment variable into a
    list of directories... PATH=/bin:/usr/bin:/usr/local/bin
    -> ["/bin", "/usr/bin", "/usr/local/bin"]
180             dirs := strings.Split(pathEnv, ":") // this
    doesn't handle certain kinds of quoting & escaping, but
    it's good enough for now.
181             // find the command in the PATH directories.
182             // e.g. if the command is "ls", we'll look for "/
    bin/ls", "/usr/bin/ls", and "/usr/local/bin/ls".
183             return lookupPath(command, dirs...)
184         }
185     }
186
187     // getenv retrieves the value of the environment variable
    named by the key, or an empty string if it's not set.
188     func getenv(environ []string, key string) string {
189         key += "=" // environment variables are stored as
    "key=value"
190         n := len(key)
191         for i := range environ {
192             if len(environ[i]) < len(key) {
193                 continue
194             }
195             // KEY=VALUE
196             if environ[i][:n] == key { // KEY=
197                 return environ[i][n:] // VALUE
198             }
199
200         }
201         return ""
202     }
203
204     // lookupPath searches for a file in a list of
    directories.
205     // usually these are the directories in the $PATH
    environment variable.
206     // use resolvePath to get that list.
207     func lookupPath(name string, dirs ...string) (string,
    error) {
```

```
208         // different shells have different rules for relative
    paths.
209         // to keep things simple, we'll just say "no".
210         if strings.Contains(name, "..") {
211             return "", fmt.Errorf("no weird relative paths,
    please: %q", name)
212         }
213         for i, dir := range dirs {
214             path := dir + "/" + name
215             if ok, err := exists(path); ok {
216                 return path, nil
217             } else if err != nil {
218                 return "",
    fmt.Errorf("lookupPath: stat in dir #%d: %q: %w", i,
    path, err)
219             }
220         }
221         return "", errors.New("not found in PATH")
222     }
223
224     // check if a file exists using stat (https://
    linux.die.net/man/2/stat)
225     // (true, nil) if it does.
226     // (false, nil) if it doesn't.
227     // (false, error) if there was an error.
228     func exists(path string) (bool, error) {
229         p := cstr(path)
230         var statbuf [144]byte // we'll worry about this later.
231         // the STAT system call fills in a stat structure with
    information about the file,
232         // returning 0 on success and -1 on error.
233         // errno will be set to the error code if it fails.
234         _, _, err := syscall.Syscall(syscall.SYS_STAT,
    uintptr(p), uintptr(unsafe.Pointer(&statbuf)), 0)
235         switch err {
236         case 0: // success!
237             return true, nil
238         // the opaquely named ENOENT means Error NO ENTry;
    aka, "file not found".
239         case syscall.ENOENT: // file doesn't exist.
240             return false, nil
241         default: // some other error.
242             return false, err
243         }
244     }
245
246     // cstr converts a Go string to a null-terminated C
    string.
247     // this allocates memory.
248     func cstr(s string) unsafe.Pointer {
249         b := make([]byte, len(s)+1)
250         copy(b, s) // copy the string into the buffer. the
    leftover byte will be the null terminator.
251         return unsafe.Pointer(&b[0])
```

```
252   }
```

9.3. `syscallshell`: exercises

- Modify `syscallshell` to handle `SIGINT` (ctrl+c). If a child process is running, `SIGINT` should be sent to the child process instead of the shell itself. If no child process is running, the shell should exit.
- Modify `syscallshell` to interpret environment variables in the form `$VAR`. E.g., "$HOME" should be replaced with the value of the `HOME` environment variable.
- Modify `syscallshell` to respect bash-style double quotes and backslash escapes. For example, `cat "some file.txt"` should print the contents of `some file.txt` rather than trying to concatenate the contents of `"some` and `file.txt"`.
- Modify `syscallshell` to allow the redirection `>` and append `>>` operators to redirect standard output to a file. For example, `echo hello > file.txt` should write `hello` to `file.txt`, and `echo world >> file.txt` should append `world` to `file.txt`.

Let's try an interactive session with our simple shell before we finish up.

```
1  #!/usr/bin/env bash
2  # START PROGRAM
3  go run syscallshell.go
```

```
1  # INTERACTIVE SESSION
2  > echo hello, world!
3  hello, world!
4  > echo goodbye, world!
5  goodbye, world!
6  > thisisnotaprogram
7  "thisisnotaprogram": not found in PATH
```

We now have a nearly-fully-functional user environment, built (nearly) from scratch. Pretty cool. If you want a tool to fix a problem, *you can **write the program** to do it*.

10. Conclusion

We've blitzed through the fundamentals of operating systems and I/O today. Any one of these topics - program startup, signals, permissions, system calls, I/O - could be the subject of an article in it's own right, but the material here should be enough to orient you.

That's all for now. I'm still sketching out the next two articles, but I plan to take us even deeper into low-level programming, exploring how programs *actually run* and what resources they use. We'll look at memory performance, and even touch a tiny bit of assembly.

As a personal note, this is my longest and most complex article to date, rendering out at nearly 40 pages, and bringing me to over 300 pages & 100,000 words in total. That's a whole book!

Remember, **programmers write programs**.

MORE ARTICLES