Fundamentals

Linux Process Loading

Yan Shoshitaishvili
Arizona State University

/bin/cat

cat /flag

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

cat /flag

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

cat /flag

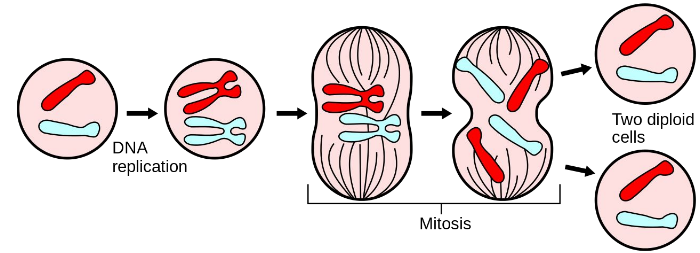
1. **A process is created.**
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

Portrait of a process

Every Linux process has:

- state (running, waiting, stopped, zombie)
- priority (and other scheduling information)
- parent, siblings, children
- shared resources (files, pipes, sockets)
- virtual memory space
- security context
  - effective uid and gid
  - saved uid and gid
  - capabilities

But where do processes come from?



In Linux, processes propagate by mitosis!

`fork` and (more recently) `clone` are *system calls* that create a nearly exact copy of the calling process: a *parent* and a *child*.

Later, the child process usually uses the `execve` syscall to *replace* itself with another process.

Example:

- you type `/bin/cat` in bash
- bash `fork`s itself into the old parent process and the child process
- the child process `execve`s `/bin/cat`, becoming `/bin/cat`

cat /flag

1. A process is created.
2. **Cat is loaded.**
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

Can we load?

Before anything is loaded, the kernel checks for executable permissions.

If a file is not executable, `execve` will fail.

What to load?

To figure out what to load, the Linux kernel reads the beginning of the file (i.e., `/bin/cat`), and makes a decision:

1. If the file starts with `#!`, the kernel extracts the interpreter from the rest of that line and executes this interpreter with the original file as an argument.
2. If the file matches a format in `/proc/sys/fs/binfmt_misc`, the kernel executes the interpreter specified for that format with the original file as an argument.
3. If the file is a dynamically-linked ELF, the kernel reads the interpreter/loader defined in the ELF, loads the interpreter and the original file, and lets the interpreter take control.
4. If the file is a statically-linked ELF, the kernel will load it.
5. Other legacy file formats are checked for.

These can be recursive!

Further reading: https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-4.html

Dynamically linked ELFs: the interpreter

Process loading is done by the ELF interpreter specified in the binary.

```
$ readelf -a /bin/cat | grep interpret
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Colloquially known as "the loader".

Can be overridden: `/lib64/ld-linux-x86-64.so.2 /bin/cat /flag`
Or changed permanently: `patchelf --set-intepreter`

Further reading: https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-4.html

Dynamically linked ELFs: the loading process

1. The program and its interpreter are loaded by the kernel.
2. The interpreter locates the libraries.
   a. LD_PRELOAD environment variable, and anything in /etc/ld.so.preload
   b. LD_LIBRARY_PATH environment variable (can be set in the shell)
   c. DT_RUNPATH or DT_RPATH specified in the binary file (both can be modified with patchelf)
   d. system-wide configuration (/etc/ld.so.conf)
   e. /lib and /usr/lib
3. The interpreter loads the libraries.
   a. these libraries can depend on other libraries, causing more to be loaded
   b. relocations updated

Further reading: https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-4.html

Where is all this getting loaded to?

Each Linux process has a *virtual memory space*. It contains:

- the binary
- the libraries
- the "heap" (for dynamically allocated memory)
- the "stack" (for function local variables)
- any memory specifically mapped by the program
- some helper regions
- kernel code in the "upper half" of memory (above 0x8000000000000000 on 64-bit architectures), inaccessible to the process

Virtual memory is dedicated to your process.
Physical memory is shared among the whole system.

You can see this whole space by looking at `/proc/self/maps`

Good resource: https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4

The Standard C Library

libc.so is linked by almost every process.

Provides functionality you take for granted:

- printf()
- scanf()
- socket()
- atoi()
- malloc()
- free()

... and a lot of other crazy stuff!

- xdr_keystatus()

The loading process (for statically linked binaries)

1. The binary is loaded.

cat /flag

1. A process is created.
2. Cat is loaded.
3. **Cat is initialized.**
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

Cat is initialized.

Every ELF binary can specify *constructors*, which are functions that run before the program is actually launched.

For example, depending on the version, libc can initialize memory regions for dynamic allocations (malloc/free) when the program launches.

You can specify your own!

```
__attribute__((constructor)) void haha()
{
	puts("Hello world!");
}
```

Demo: LD_PRELOAD and constructors.

cat /flag

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.