Fundamentals

Linux Process Execution

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 is launched.

A normal ELF automatically calls __libc_start_main() in libc, which in turn calls the program's main() function.

Your code is running!

Now what?

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 reads its arguments and environment.

```
int main(int argc, void **argv, void **envp);
```

Your process's entire input from the outside world, at launch, comprises of:

- the loaded objects (binaries and libraries)
- command-line arguments in argv
- "environment" in envp

Of course, processes need to keep interacting with the outside world.

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.

Using library functions

The binary's *import symbols* have to be resolved using the libraries' *export symbols*.

In the past, this was an on-demand process and carried great peril.

In modern times, this is all done when the binary is loaded, and is much safer.

We'll explore this further in the future.

Interacting with the environment

Almost all programs have to interact with the outside world!

This is primarily done via *system calls* (`man syscalls`). Each system call is well-documented in section 2 of the man pages (i.e., `man 2 open`).

We can trace process system calls using `strace`.

System Calls

System calls have very well-defined interfaces that very rarely change.

There are over 300 system calls in Linux. Here are some examples:

`int open(const char *pathname, int flags)` - returns a file new file descriptor of the open file (also shows up in /proc/self/fd!)
`ssize_t read(int fd, void *buf, size_t count)` - reads data from the file descriptor
`ssize_t write(int fd, void *buf, size_t count)` - writes data to the file descriptor
`pid_t fork()` - forks off an *identical* child process. Returns 0 if you're the child and the PID of the child if you're the parent.
`int execve(const char *filename, char **argv, char **envp)` - *replaces* your process.
`pid_t wait(int *wstatus)` - wait child termination, return its PID, write its status into *wstatus.
`long syscall(long syscall, ...)` - invoke specified syscall.

Typical signal combinations:

- fork, execve, wait (think: a shell)
- open, read, write (cat)

Signals

System calls are a way for a process to call into the OS. What about the other way around?

Enter: signals. Relevant system calls:

`sighandler_t signal(int signum, sighandler_t handler)` - register a signal handler
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` - more modern way of registering a signal handler
`int kill(pid_t pid, int sig)` - send a signal to a process.

Signals pause process execution and invoke the handler.

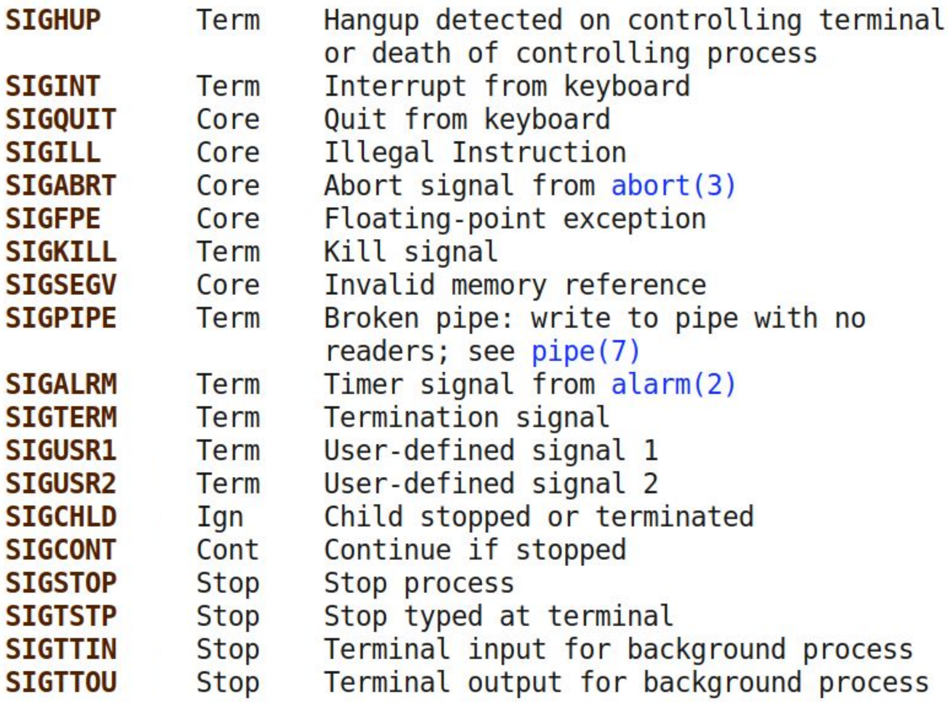
Handlers are functions that take one argument: the signal number.

Without a handler for a signal, the default action is used (often, kill).

SIGKILL (signal 9) and SIGSTOP (signal 19) cannot be handled.

Signals

Full list in section 7 of man (`man 7 signal`) and `kill -l`. Common signals:



```
SIGHUP      Term    Hangup detected on controlling terminal
                    or death of controlling process
SIGINT      Term    Interrupt from keyboard
SIGQUIT     Core    Quit from keyboard
SIGILL      Core    Illegal Instruction
SIGABRT     Core    Abort signal from abort(3)
SIGFPE      Core    Floating-point exception
SIGKILL     Term    Kill signal
SIGSEGV     Core    Invalid memory reference
SIGPIPE     Term    Broken pipe: write to pipe with no
                    readers; see pipe(7)
SIGALRM     Term    Timer signal from alarm(2)
SIGTERM     Term    Termination signal
SIGUSR1     Term    User-defined signal 1
SIGUSR2     Term    User-defined signal 2
SIGCHLD     Ign     Child stopped or terminated
SIGCONT     Cont    Continue if stopped
SIGSTOP     Stop    Stop process
SIGTSTP     Stop    Stop typed at terminal
SIGTTIN     Stop    Terminal input for background process
SIGTTOU     Stop    Terminal output for background process
```

Shared memory

Another way of interacting with the outside world is by sharing memory with other processes.

Requires system calls to establish, but once established, communication happens without system calls.

Easy way: use a shared memory-mapped file in /dev/shm.

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.**

Process termination

Processes terminate by one of two ways:

1. Receiving an unhandled signal.
2. Calling the exit() system call: `int exit(int status)`

All processes must be "reaped":

- after termination, they will remain in a zombie state until they are wait()ed on by their parent.
- When this happens, their exit code will be returned to the parent, and the process will be freed.
- If their parent dies without wait()ing on them, they are re-parented to PID 1 and will stay there until they're cleaned up.

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.