Userspace Digressions

GRESSIONS

Louis Solofrizzo louis@ne02ptzero.me
42 Staff pedago@42.fr

Summary: Make your own userspace init binary

Contents

Chapter I

Foreword

Chapter II

Introduction

II.1 Init System

In an Unix environment, the init binary (short for `initialization`), is the first process started during booting of the computer (Besides the kernel). It is started by the kernel using a hard-coded absolute path `/sbin/init`; a kernel error will occur if the kernel is unable to start it. By convention, the init process is usually assigned the 1 PID. This binary serves different purposes:

- Prepare the physical userspace environment
- Create main kernel interfaces
- Start vital binaries
- Initialize system consoles
- Start user-defined binaries

Theses purposes are a bit controversial in the system development community, especially the one about starting user-defined binaries. Some people think init should spawn processes directly, other think init should be kept simple and leav process spawning to its descendants (*BSD School); The reason is pretty simple: Any process that are father-less on an Unix system are attached to the 1 PID. So, if PID 1 crashes, half the system go with him. BSD folks approach on this problem is simply to keep the init binary itself very simple (about 30 lines of code) and launch every services in childs, contained in bash scripts (`/etc/rc.d`). Known init systems that follow this rule are `SysVInit` and `launchd`.
The other approach is too execute everything under PID 1, and providing three general functions:

- A system and service manager (manages both the system, as by applying various configurations, and its services)
- A software platform (serves as a basis for developing other software)
- The glue between applications and the kernel (provides various interfaces that expose functionalities provided by the kernel)

It is more of a distro maintainer-approach of the problem, since bash script inits can be pretty hard to maintain correctly on a large distribution. Known init systems that follow this rules are `Upstart` and `SystemD`.

II.2 Runlevels

A runlevel system is a mode of operation mainly used by `SysVInit`-style init system. The principle is pretty simple: A runlevel defines the state of the machine after boot. Different runlevels are typically assigned (not necessarily in any particular order) to the single-user mode, multi-user mode without network services started, multi-user mode with network services started, system shutdown, and system reboot system states.
Default Linux run-levels:

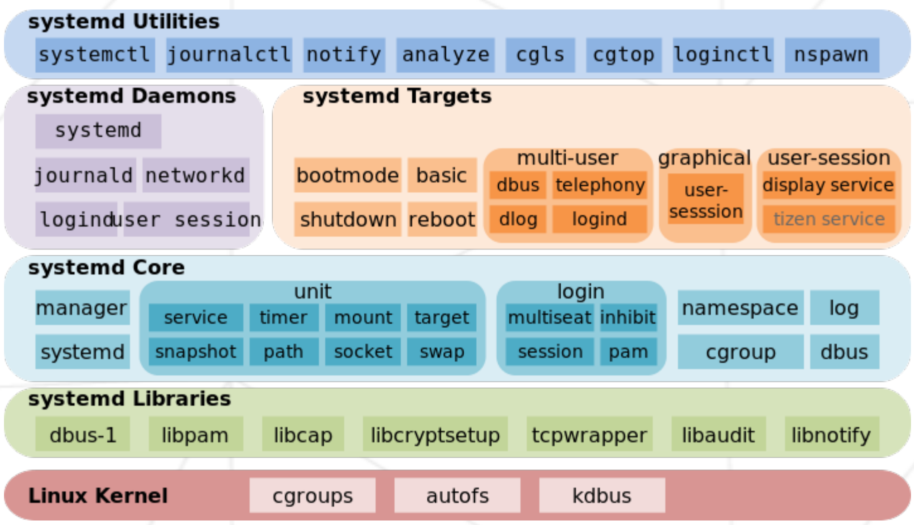
ID	Name	Description
0	Halt	Shuts down the system
1	Single-user mode	Mode for administrative tasks
2	Multi-user mode	Does not configure network interfaces
3	Multi-user with networking	Starts the system normally
4	Not used	For special purposes
5	Start the system and display manager	Same as runlevel 3 + display manager
6	Reboot	Reboots the system

II.3 The SystemD case

The design of `systemd` has ignited controversy within the free-software community. Critics regard `systemd` as overly complex and suffering from continued feature creep, arguing that its architecture violates the design principles of Unix-like operating systems. There is also concern that it forms a system of interlocked dependencies, thereby giving distribution maintainers little choice but to adopt `systemd` as more user-space software come to depend on its components. Some examples of that can be cited:

- `systemd` does UEFI bootload
- `systemd` replaces `sudo` and `su`
- `hostnamed`. There is an entire daemon dedicated to setting a computer hostname.
- There is a entire DNS server inside `systemd`
- `systemd` do log in binary format.

The PID 1 should be kept as simple as possible. There is no real reason to over-design it.



Chapter III

Goals

In this subject, you will re-code an entire init binary, with basic features and two or more advanced features of your choice. The main interests of this work are:

- Learn what's happening when you start your computer
- Discover low-level kernel configuration from userspace
- Design and implement a boot-ready program

Chapter IV

General instructions

IV.1 Language

You are free to use whatever language you want, but keep in mind that this language must be able to be executed without any standard libraries, or static linked one (When the `init` binary starts, there is no `/lib` or `/usr/lib/` mounted yet). Libraries are allowed for the purposes of parsing various files, and communication between the main daemon and the command line tool. Other than that, you are stricly limited to your language's standard library.

IV.2 Compilation

A `CMake`, a `Makefile` or a `configure` script must be turn into. This script must compile your binary, install it and required configuration files you might need.

IV.3 Run-Time

Your work must be tested against a real system, A.K.A a Linux distribution. You are free to use whatever flavor you want, or even use your own (see `ft_linux`), as long as you replace / do not use the default init system; But your work should be usable on any system that run with a Linux kernel, and it will be tested in defense.

Chapter V

Mandatory part

Your program must be able to init an entire Linux userspace, starting from PID 1. Here's all the thing it should do, in no particular order:

- It must mount the vital kernel filesystem (`/dev`, `/sys`, `/var`, `/proc`).
- It must mount the root partition, first in read only, check the partition integrity (see `fsck`) and the remount-it with write privileges.
- It must mount user-defined partitions by reading `/etc/fstab`
- It must load required kernel modules, and user-defined ones. See `udev`
- It must start and use the syslog
- It must activate swap
- It must init the Linux consoles (TTYs)
- It must configure and start user-defined network interfaces
- It must start, and watch user-defined daemons

Besides those points, you must choose two or more features from this list:

- Mounted encrypted partitions (`crypfs`)
- Early-start in RAM only filesystem (`initramfs`)
- Setup System Clock
- Start X and X environment
- Start the cron daemon
- Init system locales
- Init system hostname
- Anything that systemd does that you think is cool

You main program must be managed from a command line tool. This command line too must communicate with the main progam and must feature the following points:

- See the status of a daemon
- Start a deamon

- Stop a daemon
- Reload the configuration without stopping the program
- Enable or Disable a daemon (Do / Do not start it by default)

A basic configuration system for adding / modifying / removing daemon from the system must exists. It can be based on files, or pure command line configuration. The format is not imposed, but try to keep it clean, understandable, and please, no run-time database compilation.

Chapter VI

Bonus part

You are encouraged to implement any supplemental feature you think your project will benefit from. You will get points for it if it is correctly implemented and at least vaguely useful.

As for ideas, you can pick in the features present in the Mandatory section.

Chapter VII

Turn-in and peer-evaluation

Turn in your work using your `GiT` repository, as usual. Only the work that's in your repository will be graded during the evaluation.