

# Sysvinit 项目分析报告

李明 <limingth@gmail.com>

2013.6.22



# Contents

|  |           |
|--|-----------|
| <b>1 Sysvinit 项目工具简介</b>                           | <b>7</b>  |
| 1.1 项目背景介绍 . . . . .                               | 7         |
| 1.1.1 项目相关资源链接 . . . . .                           | 7         |
| 1.1.2 其他类似软件包 . . . . .                            | 8         |
| 1.2 项目技术架构 . . . . .                               | 8         |
| 1.2.1 核心命令 init/telinit . . . . .                  | 9         |
| 1.2.2 关机命令 halt/shutdown/poweroff/reboot . . . . . | 9         |
| 1.2.3 运行级别命令 sulogin/runlevel . . . . .            | 9         |
| 1.2.4 消息相关命令 mesg/wall/killall5/pidof . . . . .    | 9         |
| 1.2.5 日志相关命令 bootlogd/utmpdump . . . . .           | 9         |
| 1.2.6 文件系统相关命令 mountpoint/fstab-decode . . . . .   | 9         |
| <b>2 Sysvinit 项目概要分析</b>                           | <b>11</b> |
| 2.1 工具安装使用流程 . . . . .                             | 11        |
| 2.1.1 工具安装 . . . . .                               | 11        |
| 2.1.2 init 命令 . . . . .                            | 12        |
| 2.1.3 shutdown 命令 . . . . .                        | 16        |
| 2.1.4 halt 命令 . . . . .                            | 16        |
| 2.1.5 poweroff 命令 . . . . .                        | 17        |
| 2.1.6 reboot 命令 . . . . .                          | 18        |
| 2.1.7 telinit 命令 . . . . .                         | 18        |
| 2.1.8 killall5 命令 . . . . .                        | 19        |
| 2.1.9 pidof . . . . .                              | 19        |
| 2.1.10 last/lastb 命令 . . . . .                     | 19        |
| 2.1.11 mesg 命令 . . . . .                           | 20        |
| 2.1.12 mountpoint 命令 . . . . .                     | 20        |

|          |                                     |           |
|----------|-------------------------------------|-----------|
| 2.1.13   | runlevel 命令 . . . . .               | 21        |
| 2.1.14   | sulogin 命令 . . . . .                | 21        |
| 2.1.15   | wall 命令 . . . . .                   | 22        |
| 2.1.16   | bootlogd 命令 . . . . .               | 22        |
| 2.1.17   | utmpdump 命令 . . . . .               | 23        |
| 2.2      | 代码实现概要分析 . . . . .                  | 24        |
| 2.2.1    | 源码目录结构 . . . . .                    | 24        |
| 2.2.2    | Makefile 分析 . . . . .               | 26        |
| <b>3</b> | <b>Sysvinit 项目详细分析</b>              | <b>29</b> |
| 3.1      | init 命令实现代码分析 . . . . .             | 30        |
| 3.1.1    | init.c 文件中的数据分析 . . . . .           | 30        |
| 3.1.2    | init.c 中的 main 函数流程分析 . . . . .     | 31        |
| 3.1.3    | init_main 函数流程分析 . . . . .          | 31        |
| 3.1.4    | console_init 函数流程分析 . . . . .       | 33        |
| 3.1.5    | read_inittab 函数流程分析 . . . . .       | 34        |
| 3.1.6    | start_if_needed 函数流程分析 . . . . .    | 36        |
| 3.1.7    | startup 函数流程分析 . . . . .            | 37        |
| 3.1.8    | spawn 函数流程分析 . . . . .              | 37        |
| 3.1.9    | boot_transitions 函数流程分析 . . . . .   | 38        |
| 3.1.10   | @check_init_fifo() 函数流程分析 . . . . . | 39        |
| 3.1.11   | fail_check 函数流程分析 . . . . .         | 40        |
| 3.1.12   | process_signals 函数流程分析 . . . . .    | 41        |
| 3.1.13   | console_stty 函数流程分析 . . . . .       | 42        |
| 3.1.14   | fifo_new_level 函数流程分析 . . . . .     | 43        |
| 3.1.15   | re_exec 函数流程分析 . . . . .            | 43        |
| 3.1.16   | @get_init_default 函数流程分析 . . . . .  | 44        |
| 3.1.17   | telinit 函数流程分析 . . . . .            | 44        |
| 3.2      | init 进程执行流程分析汇总 . . . . .           | 44        |
| 3.2.1    | init 程序的 3 种启动执行方式 . . . . .        | 44        |
| 3.2.2    | halt 命令分析 . . . . .                 | 47        |
| <b>4</b> | <b>Sysvinit 项目安全漏洞</b>              | <b>49</b> |

|                                      |           |
|--------------------------------------|-----------|
| <b>5 Sysvinit 项目运行时调试图</b>           | <b>51</b> |
| 5.1 编译安装运行调试图 . . . . .              | 51        |
| 5.1.1 wget 下载源码包 . . . . .           | 51        |
| 5.1.2 tar 解压源码包 . . . . .            | 52        |
| 5.1.3 编译项目源码 . . . . .               | 55        |
| 5.1.4 修改 Makefile 使之能够编译通过 . . . . . | 59        |
| 5.1.5 继续编译项目源码，成功 . . . . .          | 59        |
| 5.1.6 查看生成的可执行文件 . . . . .           | 61        |
| 5.2 Linux 内核启动 init 进程 . . . . .     | 63        |
| 5.2.1 start_kernel . . . . .         | 63        |
| 5.2.2 parse_options . . . . .        | 63        |
| 5.2.3 rest_init . . . . .            | 65        |
| 5.2.4 init 函数 . . . . .              | 67        |



# Chapter 1

## Sysvinit 项目工具简介

### 1.1 项目背景介绍

`init` 进程是 Unix 和 Linux 系统中，用来产生其他所有进程的程序。`init` 的进程号 `pid` 是 1，它是其他所有进程的祖先。关于 `init` 进程的实现方法，历史上有过两种实现方案，BSD 风格和 SysV 风格。

BSD 风格比较简单，也就是通常在很多嵌入式 Linux 系统中经常可以看到的 `/etc/rc` 脚本方式，`init` 进程负责从这个启动脚本中读取一系列需要执行的程序，依次执行直到最后启动 `getty`（基于文本模式的终端）或者 `x`（基于图形界面的窗口系统）。

Sysv 相对复杂一些，它提出了一个关于 `runlevel` 运行级别的概念，为了实现启动系统到不同的运行级别，它引入了一个 `/etc/inittab` 的启动配置文件，通过这个配置文件指定的 `initdefault` 的值（从 0-6 或者 s），可以引导系统分别进入到单用户模式，多用户模式（还可以分为有网络连接和无网络连接两种），多用户带图形界面方式和用户自定义方式等，同时也可以引导系统到关机模式和重启模式。

大部分 Linux 的发行版都是采用和 System V `init` 相兼容的方式启动，为了配置 `init` 进程的这些运行级别和功能，`sysvinit` 的作者 Miquel van Smoorenburg 开发实现了一组软件包来完成这些功能。这个软件包就是我们所要研究的 `sysvinit` 项目，其中所包含的工具在下面我们要介绍的项目技术架构中会详细阐述。

下面我们列举一些有关 `sysvinit` 项目所需要用到的资源，以便了解和查看。

#### 1.1.1 项目相关资源链接

- 项目主页  
<https://savannah.nongnu.org/projects/sysvinit/>
- 源码下载  
<http://download.savannah.gnu.org/releases/sysvinit/>
- 开发成员

- Petter Reinholdtsen
- Roger Leigh
- Dr. Werner Fink

### 1.1.2 其他类似软件包

事实上, 从 `sysvinit` 产生之后, 又出现了许多类似的项目, 其主要功能都是来帮助内核最终完成启动用户应用程序。我们列举其中一些比较常见知名的项目, 作为后继研究的参考。

- `Upstart`  
最早在 Ubuntu 6.10 中广泛采用的, 基于事件机制 `event-based` 的, 异步方式工作的 `init` 进程方案, 完全可以兼容 `sysvinit` 的脚本。现在也被很多其他操作系统发行版所采用, 例如 Google Chrome OS, HP Web OS, 中均采用了 `upstart` 来作为默认的 `init` 启动程序。
- `SystemStarter`  
在 Mac OS X v10.4 之前所采用的 BSD 风格的 `init` 进程方案。
- `launchd`  
在 Mac OS X v10.4 和之后所采用, 它会启动 `SystemStarter` 来处理 `rc.local` 脚本。
- `systemd`  
能够提供更好的服务依赖性的支持, 允许系统启动期间能够并发完成更多工作, 减小 `shell` 的开销。`systemd` 在 Fedora 15 和 openSUSE 12.1 中所采用。

## 1.2 项目技术架构

`Sysvinit` 软件包是一组 Linux 工具集, 包含控制启动, 运行和关闭所有其他程序的工具。

具体包含有如下这些命令:

- `init`
- `telinit` (链接到 `init`)
- `sulogin`
- `halt`
- `poweroff` (链接到 `halt`)
- `reboot` (链接到 `halt`)



- shutdown
- last, lastb (链接到 last)
- killall5
- pidof (链接到 killall5)
- mesg
- wall
- runlevel
- utmpdump

这些命令多达 13 条，我们可以把它们按功能和用途进行归类。

### 1.2.1 核心命令 `init/telinit`

`init` 是所有命令中的核心，也是该项目最重要的输出成果，提供一个可以和 Kernel 进行对接的 `init` 程序，实现支持不同运行级别的启动方式。

### 1.2.2 关机命令 `halt/shutdown/poweroff/reboot`

这 4 条命令都是和系统关机有关，名字很接近。

### 1.2.3 运行级别命令 `sulogin/runlevel`

### 1.2.4 消息相关命令 `mesg/wall/killall5/pidof`

### 1.2.5 日志相关命令 `bootlogd/utmpdump`

### 1.2.6 文件系统相关命令 `mountpoint/fstab-decode`

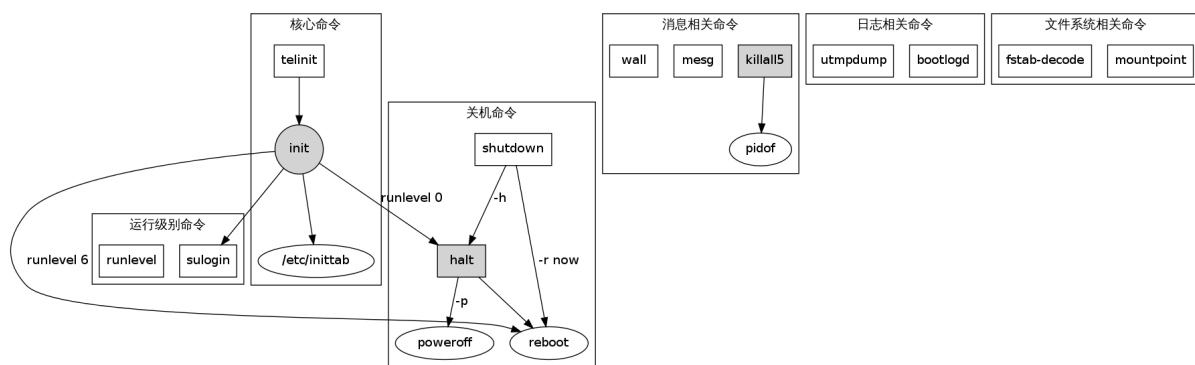


Figure 1.1: sysvinit 系统层次结构图

## Chapter 2

# Sysvinit 项目概要分析

## 2.1 工具安装使用流程

### 2.1.1 工具安装

```
$ find sbin/ bin/ usr/bin/ | xargs ls -l
-rwxr-xr-x 1 akaedu akaedu 7708 Jun 23 17:20 bin/mountpoint
lrwxrwxrwx 1 akaedu akaedu 14 Jun 23 17:20 bin/pidof -> /sbin/killall5
-rwxr-xr-x 1 akaedu akaedu 18162 Jun 23 17:20 sbin/bootlogd
-rwxr-xr-x 1 akaedu akaedu 7402 Jun 23 17:20 sbin/fstab-decode
-rwxr-xr-x 1 akaedu akaedu 17625 Jun 23 17:20 sbin/halt
-rwxr-xr-x 1 akaedu akaedu 42121 Jun 23 17:20 sbin/init
-rwxr-xr-x 1 akaedu akaedu 22259 Jun 23 17:20 sbin/killall5
lrwxrwxrwx 1 akaedu akaedu 4 Jun 23 17:20 sbin/poweroff -> halt
lrwxrwxrwx 1 akaedu akaedu 4 Jun 23 17:20 sbin/reboot -> halt
-rwxr-xr-x 1 akaedu akaedu 7368 Jun 23 17:20 sbin/runlevel
-rwxr-xr-x 1 akaedu akaedu 27547 Jun 23 17:20 sbin/shutdown
-rwxr-xr-x 1 akaedu akaedu 17677 Jun 23 17:20 sbin/sulogin
lrwxrwxrwx 1 akaedu akaedu 4 Jun 23 17:20 sbin/telinit -> init
-rwxr-xr-x 1 akaedu akaedu 22117 Jun 23 17:20 usr/bin/last
lrwxrwxrwx 1 akaedu akaedu 4 Jun 23 17:20 usr/bin/lastb -> last
-rwxr-xr-x 1 akaedu akaedu 7730 Jun 23 17:20 usr/bin/mesg
-rwxr-xr-x 1 akaedu akaedu 12638 Jun 23 17:20 usr/bin/utmpdump
-rwxr-xr-x 1 akaedu akaedu 13243 Jun 23 17:20 usr/bin/wall
```

所有工具编译之后可执行文件都生成在 `src` 源码目录下，同时，这些命名的帮助文件在 `man` 目录下。

```
$ ls -l
total 108
-rw-r--r-- 1 akaedu akaedu 2847 Jun 23 11:13 bootlogd.8
-rw-r--r-- 1 akaedu akaedu 1971 Jun 23 11:13 bootlogd.8.todo
-rw-r--r-- 1 akaedu akaedu 1444 Jun 23 11:13 fstab-decode.8
-rw-r--r-- 1 akaedu akaedu 3957 Jun 23 11:13 halt.8
-rw-r--r-- 1 akaedu akaedu 12124 Jun 23 11:13 init.8
-rw-r--r-- 1 akaedu akaedu 2428 Jun 23 11:13 initscript.5
-rw-r--r-- 1 akaedu akaedu 8290 Jun 23 11:13 inittab.5
-rw-r--r-- 1 akaedu akaedu 1866 Jun 23 11:13 killall5.8
-rw-r--r-- 1 akaedu akaedu 4242 Jun 23 11:13 last.1
-rw-r--r-- 1 akaedu akaedu 16 Jun 23 11:13 lastb.1
-rw-r--r-- 1 akaedu akaedu 1867 Jun 23 11:13 mesg.1
-rw-r--r-- 1 akaedu akaedu 1886 Jun 23 11:13 mountpoint.1
-rw-r--r-- 1 akaedu akaedu 3230 Jun 23 11:13 pidof.8
-rw-r--r-- 1 akaedu akaedu 16 Jun 23 11:13 poweroff.8
-rw-r--r-- 1 akaedu akaedu 16 Jun 23 11:13 reboot.8
-rw-r--r-- 1 akaedu akaedu 1872 Jun 23 11:13 runlevel.8
-rw-r--r-- 1 akaedu akaedu 8017 Jun 23 11:13 shutdown.8
-rw-r--r-- 1 akaedu akaedu 3309 Jun 23 11:13 sulogin.8
-rw-r--r-- 1 akaedu akaedu 16 Jun 23 11:13 telinit.8
-rw-r--r-- 1 akaedu akaedu 1949 Jun 23 11:13 utmpdump.1
-rw-r--r-- 1 akaedu akaedu 1960 Jun 23 11:13 wall.1
```

通过使用 `man` 命令，加上 `-l` 参数，例如 `man -l init.8` 我们可以了解到这些命令的用法。

- 注意

我们这里没有直接使用例如 `man init` 这样的命令，而是改用 `man -l init.8`，这是因为前者是查看当前系统的帮助，而当前系统是 `ubuntu 12.04` 已经改用 `upstart` 作为 `init` 进程。后者才是针对 `sysvinit` 工具中的可执行文件配套的帮助信息。

下面我们针对这些命令的帮助信息，来给出每个命令的具体用法，在测试案例报告中，我们会详细说明每个命令如何使用。

## 2.1.2 `init` 命令

### `init` 命令说明

`init` 进程是所有进程的父进程。它的主要任务就是从 `/etc/inittab` 文件中读取命令行，从而创建出一系列后继进程。`init` 进程本身是被 `Kernel` 所启动，

Kernel 将控制权交给它之后，用它来负责启动所有其他的进程。inittab 文件中通常有关于登录接口的定义，就是在每个终端产生 getty，使用户可以进行登录。

## 命令格式

```
/sbin/init [ -a ] [ -s ] [ -b ] [ -z xxx ] [ 0123456Ss ]
```

## 运行级别

运行级别是 Linux 操作系统的一个软件配置，用它来决定启动哪些程序集来运行。系统启动时，可以根据/etc/inittab 文件的配置，进入不同的运行级别。每个运行级别可以设置启动不同的程序。

启动的每个程序都是 init 的进程的子进程，运行级别有 8 个，分别是 0-6,s 或 S。运行级别 0,1 和 6 是系统保留的。

- 运行级别 0 用来关闭系统，
- 运行级别 1 先关闭所有用户进程和服务，然后进入单用户模式。
- 运行级别 6 用来重启系统。
- 运行级别 S 和 s，会直接进入单用户模式。
  - 这种模式下不再需要/etc/inittab 文件。
  - /sbin/sulogin 会在/dev/console 上被启动。
  - 运行级别 S 和 s 的功能是相同的。

## 启动过程

在 kernel 启动的最后阶段，会调用 init。init 会查找/etc/inittab 文件内容，进入指定的运行级别。其中 initdefault 代表着系统默认要进入的运行级别，如果用户指定了，就会进入到 initdefault 代表的那个运行级别。如果用户没有指定，则系统启动时，会通过 console 来要求用户输入一个运行级别。

当启动一个新进程时，init 会先检查/etc/initscript 文件是否存在。如果存在，则使用这个脚本来启动那个进程。

## 选项

- -s, S, single  
进入单用户模式。

- 1-5  
启动进入的运行级别。
- -b, emergency  
直接进入单用户 shell，不运行任何其他启动脚本。
- -a, auto  
如果指定该参数，init 会将 AUTOBOOT 环境变量设置为 yes。
- -z xxx  
-z 后面的参数将被忽略。可以使用这种方法将命令行加长一点，这样可以增加在堆栈中占用的空间。
- init 0 这条命令也可以用来关机。

### **/etc/inittab 文件范例**

```
id:1:initdefault:
rc::bootwait:/etc/rc
1:1:respawn:/etc/getty 9600 tty1
2:1:respawn:/etc/getty 9600 tty2
3:1:respawn:/etc/getty 9600 tty3
4:1:respawn:/etc/getty 9600 tty4
```

#### **id**

inittab 文档中条目的唯一标识，限于 1-4 个字符。

#### **runlevels**

列出发生指定动作的运行级，可以是单个的数字，也可以是连续的多个数字，例如 2345 表示在多个运行级别下都需要执行。

#### **action**

描述要发生的动作，常用的有 respawn, wait, boot, once, bootwait, off, initdefault, ctrlaltdel, sysinit 等。具体含义如下：

#### **\* respawn**

该进程只要终止就立即重新启动（如 getty）。

#### **\* wait**

只要进入指定的运行级就启动本进程，并且 init 等待该进程的结束。

- \* `once`  
只要进入指定的运行级就启动一次本进程。
- \* `boot`  
在系统引导期间执行本进程。 `runlevels` 域被忽略。
- \* `bootwait`  
在系统引导期间执行本进程。并且 `init` 等待该进程的结束 (如 `/etc/rc`)。 `runlevels` 域被忽略。
- \* `off`  
什么也不做。
- \* `ondemand`  
在进入 `ondemand` 运行级时才会执行标记为 `ondemand` 的那些进程。无论如何，实际上没有改变运行级 (`ondemand` 运行级就是 ``a'` , ``b'` , 和 ``c'` )。
- \* `initdefault`  
`initdefault` 条目给出系统引导完成后进入的运行级，假如不存在这样的条目，`init` 就会在控制台询问要进入的运行级。 `process` 域被忽略。
- \* `sysinit`  
系统引导期间执行此进程。本进程会在 `boot` 或 `bootwait` 条目之前得到执行。 `runlevels` 域被忽略。
- \* `ctrlaltdel`  
在 `init` 收到 `SIGINT` 信号时执行此进程。这意味着有人在控制台按下了 `CTRL-ALT-DEL` 组合键，典型地，可能是想执行类似 `shutdown` 然后进入单用户模式或重新引导机器。
- \* `kbrequest`  
本进程在 `init` 收到一个从控制台键盘产生的特别组合按键信号时执行。

## **process**

要执行的程序或者脚本的名称，常见的有 `getty`, `/etc/init.d/rcS`, `/etc/rc.d/rc.sysinit`, `/etc/rc.d/rc`, `/bin/sh`, `/bin/umount` 等。

- 参 考 资 料: <http://www.linuxsky.org/doc/newbie/200706/62.html>    <http://www.2cto.com/os/201108/98426.html>

### 2.1.3 shutdown 命令

#### shutdown 命令说明

`shutdown` 以一种安全的方式终止系统，所有正在登录的用户都会收到系统将要终止的通知，并且不准新的登录。

#### 命令格式

```
/sbin/shutdown [-akrhPHfFnc] [-t sec] time [warning message]
```

#### 参数选项

- `-h`  
将系统关机，在某种程度上功能与 `halt` 命令相当。
- `-k`  
只是送出信息给所有用户，但并不会真正关机。
- `-n`  
不调用 `init` 程序关机，而是由 `shutdown` 自己进行（一般关机程序是由 `shutdown` 调用 `init` 来实现关机动作），使用此参数将加快关机速度，但是不建议用户使用此种关机方式。
- `-r`  
`shutdown` 之后重新启动系统。
- `-f`  
送出警告信息和关机信号之间要延迟多少秒。警告信息将提醒用户保存当前进行的工作

### 2.1.4 halt 命令

#### halt 命令说明

`halt` 用来停止系统。正常情况下等效于 `shutdown` 加上 `-h` 参数（当前系统运行级别是 0 时除外）。它将告诉内核去中止系统，并在系统正在关闭的过程中将日志记录到 `/var/log/wtmp` 文件里。

#### 命令格式



```
/sbin/halt [-n] [-w] [-d] [-f] [-i] [-p] [-h]
```

## 主要选项

- `-n`  
reboot 或者 halt 之前，不同步 (sync) 数据。
- `-w`  
仅仅往 `/var/log/wtmp` 里写一个记录，并不实际做 reboot 或者 halt 操作。
- `-f`  
强制 halt 或者 reboot，不等其他程序退出或者服务停止就重新启动系统。这样会造成数据丢失，建议一般不要这样做。
- `-i`  
halt 或 reboot 前，关闭所有网络接口。
- `-h`  
halt 或 poweroff 前，使系统中所有的硬件处于等待状态。
- `-p`  
在系统 halt 同时，做 poweroff 操作。即停止系统同时关闭电源。

## 2.1.5 poweroff 命令

poweroff 告诉内核中止系统并且关闭系统 (参见 halt)

## 命令格式

```
poweroff [OPTION]...
```

## 主要选项

- `-f, --force` 强制关机
- `-p, --poweroff`  
等价于 `halt -p`
- `-w, --wtmp-only` 仅仅往 `/var/log/wtmp` 里写一个记录，并不实际做 reboot 或者 halt 操作。

## 2.1.6 reboot 命令

`reboot` 告诉内核重启系统（参见 `halt`）

### 命令格式

```
reboot [OPTION]...
```

### 主要选项

## 2.1.7 telinit 命令

`telinit` 告诉 `init` 该进入哪个运行级。

执行 `telinit` 时，`telinit` 函数仍然通过向 `init fifo` 写入命令的方式通知 `init` 执行相应的操作。

### 命令格式

```
telinit [-t sec] [0123456sSqabcUu]
```

### 参数说明

- `0,1,2,3,4,5,6` 将运行级别切换到指定的运行级别。
- `a,b,c` 只运行那些 `/etc/inittab` 文件中运行级别是 `a, b` 或 `c` 的记录。
- `Q,q` 通知 `init` 重新检测 `/etc/inittab` 文件。
- `S,s` 将运行级别切换到单用户模式下。
- `U,u` 自动重启（保留状态），此操作不会对文件 `/etc/inittab` 进行重新检测。执行此操作时，运行级别必须处在 `Ss12345` 之一，否则，该请求将被忽略。
- `-t sec` 告诉 `init` 两次发送 `SIGTERM` 和 `SIGKILL` 信号的时间间隔。默认值是 5 秒

### 2.1.8 killall5 命令

killall5 命令发送一个信号到所有进程，但那些在它自己设定级别的进程将不会被这个运行的脚本所中断。killall5 就是 SystemV 的 killall 命令。向除自己的会话 (session) 进程之外的其它进程发出信号，所以不能杀死当前使用的 shell。

#### 命令格式

```
killall5 -signalnumber [-o omitpid[,omitpid..]] [-o omitpid[,omit -pid..]..]
```

#### 主要选项

- -o omitpid 可以忽略的进程 pid 号

### 2.1.9 pidof

pidof 命令可以报告给定程序的进程识别号 (pid)，输出到标准输出设备。这个命令其实是指向 killall5 的一个软链接。

#### 命令格式

```
pidof [-s] [-c] [-n] [-x] [-o omitpid[,omitpid..]] [-o omitpid[,omit -pid..]..] program [program..]
```

#### 主要选项

- -s 表示只返回 1 个 pid
- -o omitpid 表示告诉 pidof 表示忽略后面给定的 pid，可以使用多个 -o。

### 2.1.10 last/lastb 命令

last 命令给出哪一个用户最后一次登录（或退出登录），它回溯/var/log/wtmp 文件（或者 -f 选项指定的文件），显示自从这个文件建立以来，所有用户的登录情况。

lastb 显示所有失败登录企图，并记录在/var/log/btmp。

## 命令格式

```
last [-R] [-num] [-n num] [-adFiowx] [-f file] [-t YYYYMMDDHHMMSS] [name...] [tty...]
```

## 主要选项

- `-num` (`-n num`) 指定 `last` 要显示多少行。
- `-R` 不显示主机名列。
- `-a` 在最后一列显示主机名 (和下一个选项合用时很有用)
- `-d` 对于非本地的登录, Linux 不仅保存远程主机名而且保存 IP 地址。这个选项可以将 IP 地址转换为主机名。
- `-i` 这个选项类似于显示远程主机 IP 地址的 `-d` 选项, 只不过它用数字和点符号显示 IP 地址。
- `-o` 读取一个旧格式的 `wtmp` 文件 (用 `linux-libc5` 应用程序写入的)。
- `-x` 显示系统关机记录和运行级别改变的日志。

### 2.1.11 mesg 命令

该命令的作用是, 控制是否允许在当前终端上显示出其它用户对当前用户终端发送的消息。

## 命令格式

```
mesg [y|n]
```

## 主要选项

- `y` 允许消息传到当前终端
- `n` 不允许消息传到当前终端

### 2.1.12 mountpoint 命令

`mountpoint` 检查给定的目录是否是一个挂载点

## 命令格式

```
/bin/mountpoint [-q] [-d] /path/to/directory  
/bin/mountpoint -x /dev/device
```

## 主要选项

```
-q      Be quiet - don't print anything.  
-d      Print major/minor device number of the filesystem on stdout.  
-x      Print major/minor device number of the blockdevice on stdout.
```

### 2.1.13 runlevel 命令

`runlevel` 命令读取系统的登录记录文件（一般是 `/var/run/utmp`）把以前和当前的系统运行级输出到标准输出设备。

## 命令格式

```
runlevel [utmp]
```

## 主要选项

```
utmp    The name of the utmp file to read.
```

### 2.1.14 sulogin 命令

`sulogin` 命令允许 `root` 登录，它通常情况下是在系统在单用户模式下运行时，由 `init` 所派生。

## 命令格式

```
sulogin [ -e ] [ -p ] [ -t SECONDS ] [ TTY ]
```

## 主要选项

无

### 2.1.15 wall 命令

#### wall 命令说明

`wall` 命令用来向所有用户的终端发送一条信息。发送的信息可以作为参数在命令行给出，也可在执行 `wall` 命令后，从终端中输入。使用终端输入信息时，按 `Ctrl-D` 结束输入。`wall` 的信息长度的限制是 20 行。

只有超级用户有权限，给所有用户的终端发送消息。

## 命令格式

```
wall [-n] [ message ]
```

- 用法  
usage: wall [message]
- 举例  
wall ``hello msg''

### 2.1.16 bootlogd 命令

`bootlogd` 命令把启动信息记录到一个日志文件。

## 命令格式

```
/sbin/bootlogd [-c] [-d] [-r] [-s] [-v] [ -l logfile ] [ -p pidfile ]
```

## 主要选项

- d Do not fork and run in the background.
- c Attempt to write to the logfile even if it does not yet exist. Without this option, bootlogd will wait for the logfile to appear before attempting to write to it. This behavior prevents bootlogd from creating logfiles under mount points.
- r If there is an existing logfile called logfile rename it to log - file~ unless logfile~ already exists.
- s Ensure that the data is written to the file after each line by calling fdatasync(3). This will slow down a fsck(8) process running in parallel.
- v Show version.
- l logfile  
Log to this logfile. The default is /var/log/boot.

### 2.1.17 utmpdump 命令

utmpdump 命令以一种用户友好的格式向标准输出设备显示/var/run/utmp 文件的内容。

## 命令格式

```
utmpdump [-froh] filename
```

## 主要选项

```
-f      output appended data as the file grows.

-r      reverse. Write back edited login information into utmp or wtmp
        files.

-o      use old libc5 format.

-h      usage information.
```

## 2.2 代码实现概要分析

我们目前所要分析的源码压缩包为 2.88 版本的，这个版本的发布时间是 26-Mar-2010。

### 2.2.1 源码目录结构

```
$ make distclean
make -C src distclean
make[1]: Entering directory `/home/akaedu/Github/sysvinit/sysvinit-2.88dsf/src'
rm -f *.o *.bak
rm -f mountpoint init halt shutdown runlevel killall5 fstab-decode sulogin bootlogd last
make[1]: Leaving directory `/home/akaedu/Github/sysvinit/sysvinit-2.88dsf/src'
$ make clean
$ tree
```

```
.
├── contrib
│   ├── alexander.viro
│   ├── notify-pam-dead.patch
│   ├── start-stop-daemon.c
│   ├── start-stop-daemon.README
│   ├── TODO
│   └── zefram-patches
├── COPYING
├── COPYRIGHT
├── doc
│   ├── bootlogd.README
│   ├── Changelog
│   └── Install
```



- ├── Propaganda
- ├── sysvinit-2.86.lsm
- ├── Makefile
- ├── man
  - ├── bootlogd.8
  - ├── bootlogd.8.todo
  - ├── fstab-decode.8
  - ├── halt.8
  - ├── init.8
  - ├── initscript.5
  - ├── inittab.5
  - ├── killall5.8
  - ├── last.1
  - ├── lastb.1
  - ├── mesg.1
  - ├── mountpoint.1
  - ├── pidof.8
  - ├── poweroff.8
  - ├── reboot.8
  - ├── runlevel.8
  - ├── shutdown.8
  - ├── sulogin.8
  - ├── telinit.8
  - ├── utmpdump.1
  - └── wall.1
- ├── obsolete
  - ├── bootlogd.init
  - ├── powerd.8
  - ├── powerd.c
  - ├── powerd.cfg
  - ├── powerd.README
  - ├── README.RIGHT.NOW
  - └── utmpdump.c.OLD
- ├── README
- └── src
  - ├── a.out
  - ├── bootlogd.c
  - ├── dowall.c
  - ├── fstab-decode.c
  - ├── halt.c
  - └── hddown.c

```
|— ifdown.c
|— init.c
|— init.h
|— initreq.h
|— initscript.sample
|— killall5.c
|— last.c
|— Makefile
|— mesg.c
|— mountpoint.c
|— oldutmp.h
|— paths.h
|— reboot.h
|— runlevel.c
|— set.h
|— shutdown.c
|— sulogin.c
|— utmp.c
|— utmpdump.c
|— wall.c
```

5 directories, 69 files

### 2.2.2 Makefile 分析

```
93 init:          LDLIBS += $(INITLIBS) $(STATIC)
94 init:          init.o init_utm.o
95
96 halt:          halt.o ifdown.o hddown.o utmp.o reboot.h
97
98 last:          last.o oldutmp.h
99
100 mesg:          mesg.o
101
102 mountpoint:    mountpoint.o
103
104 utmpdump:      utmpdump.o
105
106 runlevel:      runlevel.o
107
```

```
108 sulogin:      LDLIBS += $(SULOGINLIBS) $(STATIC)
109 sulogin:      sulogin.o
110
111 wall:          dwall.o wall.o
112
113 shutdown:      dwall.o shutdown.o utmp.o reboot.h
114
115 bootlogd:      LDLIBS += -lutil
116 bootlogd:      bootlogd.o
```

以上是生成可执行文件的 `Makefile` 关键片段。从这里可以大致看出，每个可执行文件的生成，需要依赖于哪些目标文件，也就是由哪些源码文件生成的。例如 `init` 程序，是由 `init.c` `init_utm.c` 这 2 个文件生成，如果我们要研究 `init` 程序的源码，就需要读懂这 2 个源码文件。



## Chapter 3

# Sysvinit 项目详细分析

在分析源码之前，我们可以先了解一下所有源代码文件的行数，以便我们对分析的工作量和重点有所认识。

```
$ ls *.c | xargs wc -l | sort -n
  53 runlevel.c
  86 fstab-decode.c
 109 ifdown.c
 122 wall.c
 124 mesg.c
 128 mountpoint.c
 253 dowall.c
 264 utmp.c
 302 utmpdump.c
 315 halt.c
 568 hddown.c
 607 sulogin.c
 690 bootlogd.c
 762 shutdown.c
 928 last.c
1104 killall5.c
2898 init.c
9313 total
```

可以看出，全部源码的代码行数合计约 9313 行，其中代码量最多的一个源文件是 `init.c` 程序，也就是我们要分析的核心程序，这个程序的代码行已经接近 3000 行。除了这个文件之外，最多的代码行文件就是 `killall5.c` 只有约 1000 行。

## 3.1 init 命令实现代码分析

### 3.1.1 init.c 文件中的数据分析

```
00106
00107 CHILD *family = NULL;          /* The linked list of all entries */
00108 CHILD *newFamily = NULL;        /* The list after inittab re-read */
00109
00110 CHILD ch_emerg = {              /* Emergency shell */
00111     WAITING, 0, 0, 0, 0,
00112     "~ ",
00113     "S",
00114     3,
00115     "/sbin/sulogin",
00116     NULL,
00117     NULL
00118 };
00119
00120 char runlevel = 'S';            /* The current run level */
00121 char thislevel = 'S';          /* The current runlevel */
00122 char prevlevel = 'N';          /* Previous runlevel */
00123 int dfl_level = 0;             /* Default runlevel */
00124 sig_atomic_t got_cont = 0;     /* Set if we received the SIGCONT signal */
00125 sig_atomic_t got_signals;      /* Set if we received a signal. */
00126 int emerg_shell = 0;           /* Start emergency shell? */
00127 int wrote_wtmp_reboot = 1;     /* Set when we wrote the reboot record */
00128 int wrote_utmp_reboot = 1;     /* Set when we wrote the reboot record */
00129 int wrote_wtmp_rlevel = 1;     /* Set when we wrote the runlevel record */
00130 int wrote_utmp_rlevel = 1;     /* Set when we wrote the runlevel record */
00131 int sltime = 5;               /* Sleep time between TERM and KILL */
00132 char *argv0;                  /* First arguments; show up in ps listing */
00133 int maxproclen;               /* Maximal length of argv[0] with \0 */
00134 struct utmp utproto;          /* Only used for sizeof(utproto.ut_id) */
00135 char *console_dev;            /* Console device. */
00136 int pipe_fd = -1;             /* /dev/initctl */
00137 int did_boot = 0;             /* Did we already do BOOT* stuff? */
00138 int main(int, char **);
00139
00140 /*      Used by re-exec part */
00141 int reload = 0;               /* Should we do initialization stuff? */
00142 char *myname="/sbin/init";    /* What should we exec */
00143 int oops_error;               /* Used by some of the re-exec code. */
00144 const char *Signature = "12567362"; /* Signature for re-exec fd */
```

### 3.1.2 init.c 中的 main 函数流程分析

我们从 init.c 中 main 函数的执行逻辑开始分析。在 main 函数中主要负责完成以下工作：

1. 获取 argv[0] 参数，用以判断用户执行了 init 还是 telinit，因为 telinit 是指向 init 程序的软链接。
2. 检查当前执行用户的权限，必须是 superuser，否则直接退出。
3. 通过 getpid() 获取当前执行进程的 pid，判断是否为 1（1 表示是通过内核调用执行的第一个进程，而不是通过用户来执行 init 程序启动的进程）。(同时从源码中可以看出，init 程序也支持用 -i 或者 --init 参数来表示当前要求执行的是 init 进程。不过这个方式在 man -l init.8 的 man page 中没有明确提供此信息)
4. 如果不是要求执行 init 进程，则转交控制权给 telinit(p, argc, argv) 函数进行处理。在后面介绍 telinit 函数的地方，我们再对此做详细说明。
5. 如果是要求执行 init 进程，还需要接着进行检查是否是属于 re-exec，也就是重新执行，而不是首次执行。判断思路是通过读取 STATE\_PIPE，看是否收到一个 Signature = "12567362" 的字符串来确定。如果是重新执行，则将 reload 全局变量置为 1。re-exec 和首次执行最大的区别是没有对/etc/inittab 进行解析，在后面我们会再次提到，为保持思路直接和简单，我们在这里不展开，直奔 init 进程中最关键的代码。
6. 如果是属于 init 进程的首次执行，则需要对 argv[] 的参数进行相应处理，简单说来，就是把 -s single 或者 0123456789 这样的数字，转换为 dfl\_level 变量，这个变量代表的就是默认的运行级别。
7. 如果宏定义了 WITH\_SELINUX，则会通过调用 is\_selinux\_enabled() 判断是否系统使能了 SELINUX，如果是，则在通过调用 selinux\_init\_load\_policy 来加载策略，最后通过 execv 来再执行 init。
8. 在进行一系列判断检测之后，通过传递 argv[0] -> argv0 这个全局变量，最终调用了 init\_main() 进入标准的 init 主函数中。

下一小节，我们将重点来介绍 init\_main 函数。

### 3.1.3 init\_main 函数流程分析

该函数主要完成的功能是：切换运行级别，检查出错情况，接受信号，启动相应服务例程。

1. 调用 `init_reboot` 宏定义（其实就是 `reboot` 函数）告诉内核，当 `ctrl + alt + del` 三个键被同时按下时，给当前进程发送 `SIGINT` 信号，以便 `init` 进程可以处理来自键盘的这一信号，进一步决定采取何种动作。

2. 接下来将会安装一些信号处理函数。如下：

`signal_handler()`，处理 `SIGALRM`, `SIGHUP`, `SIGINT`, `SIGPWR`, `SIGWINCH`, `SIGUSR1`  
`chld_handler()`，处理 `SIGCHLD`  
`stop_handler()`，处理 `SIGSTOP`, `SIGTSTP`  
`cont_handler()`，处理 `SIGCONT`  
`segv_handler()`，处理 `SIGSEGV`

3. 然后初始化终端，调用 `console_init` 函数。这个函数我们在下面也会再次详细分析。

4. 终端初始化完成后，接着对 `reload` 这个变量进行判别，是否属于是首次执行？

5. 如果是首次执行，则依次执行下列步骤：

5.1 关闭所有打开文件 0, 1, 2,

5.2 然后调用 `console_stty()` 函数对终端进行设置，主要是通过 `tcsetattr()` 函数来设置一些快捷键。

5.3 以覆盖 `overwrite` 方式设置 `PATH` 环境变量，通过 `PATH_DEFAULT` 宏定义，默认值是 `"/sbin:/usr/sbin:/bin:/usr/bin"`

5.4 初始化 `/var/run/utmp` 文件。通过日志输出 `booting` 信息

5.5 如果 `emerg_shell` 被设置（参数中有 `-b` 或者 `emergency`），表示需要启动 `emergency shell`，则通过调用 `spawn()` 初始化 `emergency shell` 子进程，并等待该子进程退出。

5.6 设置当前的 `runlevel = '#'`，表示这是正常的 `Kernel` 首次启动 `init` 的方式 `SYSINIT`。

5.7 当从 `emergency shell` 退出（或者不需要 `emergency shell` 的话），则调用 `read_inittab()` 来读入 `/etc/inittab` 文件。该函数主要将 `/etc/inittab` 文件解析的结果存入 `CHILD` 类型的链表 `family` 上，供之后的执行使用。

6. 如果不是首次执行，也就是 `reload` 为真，则只执行下列步骤：

6.1 通过日志输出 `reloading` 信息

6.2 以非覆盖 `non overwrite` 方式设置 `PATH` 环境变量，通过 `PATH_DEFAULT` 宏定义，默认值是 `"/sbin:/usr/sbin:/bin:/usr/bin"`

7. 5 或者 6 执行完之后，调用 `start_if_needed()` 函数，启动需要在相应运行级别中运行的程序和服务。而该函数主要又是通过调用 `startup()` 函数，继而调用 `spawn()` 来启动程序或者服务的运行的。

8. 在此之后，`init_main()` 就进入一个主循环中，主要完成切换运行级别，检查出错情况，接受信号，启动相应服务例程。



在这个主循环中，需要调用如下这些重要的函数：

```
boot_transitions() -> get_init_default() -> ask_runlevel()
check_init_fifo() -> console_init()
fail_check()
process_signals() -> console_stty()
start_if_needed() -> startup() -> spawn()
```

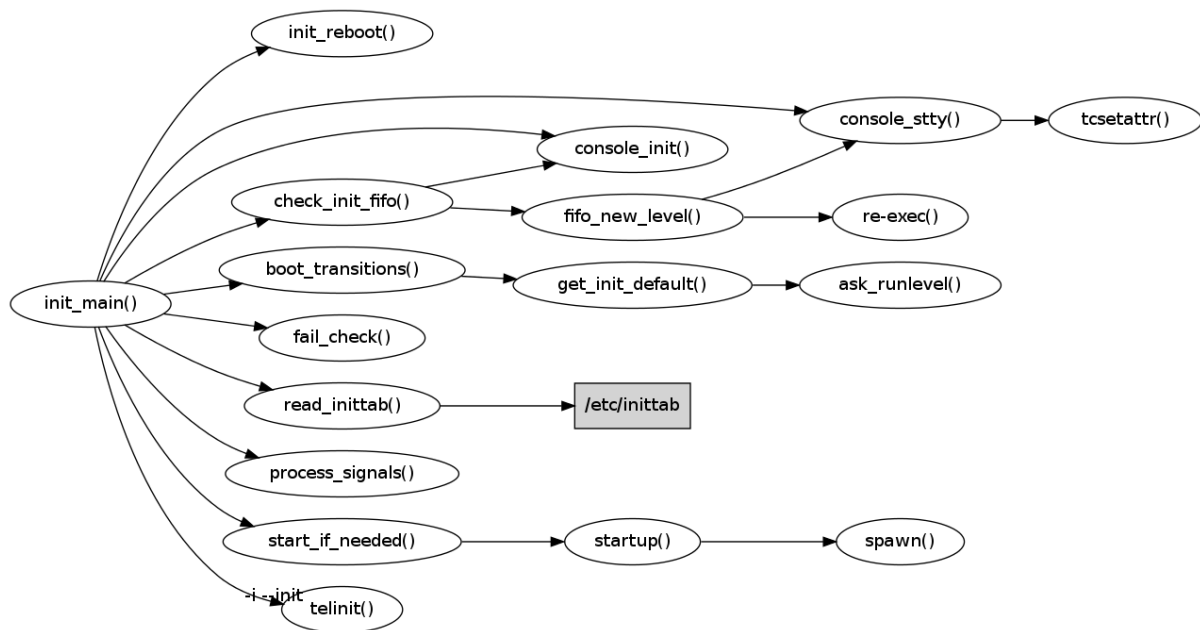


Figure 3.1: `init_main` 函数核心代码

### 3.1.4 `console_init` 函数流程分析

该函数主要完成的功能是：设置 `console_dev` 变量为一个可以工作的 `console` 函数执行流程分析：

1. 获取 `CONSOLE` 环境变量的值，赋值给 `console_dev` 全局变量（`char *` 类型）。
2. 以只读非阻塞方式打开 `console_dev` 所代表的设备文件。
3. 初始化成功，则关闭该设备文件；如果失败，则将 `console_dev` 置为 `/dev/null`。

### 3.1.5 read.inittab 函数流程分析

该函数主要完成的功能是：读取/etc/inittab 文件，解析其中的约定规则，形成一个 CHILD 链表数据结构中。

该函数中用到的重要数据结构有 CHILD (struct child) 和 actions 数组 (struct actions)

#### CHILD 机构体 (struct child)

这个链表数据结构在 init.h 头文件中，是实现根据 init 运行级别加载不同用户程序的最重要的数据结构。

```
00082 /* Information about a process in the in-core inittab */
00083 typedef struct _child_ {
00084     int flags;                                /* Status of this entry */
00085     int exstat;                               /* Exit status of process */
00086     int pid;                                 /* Pid of this process */
00087     time_t tm;                               /* When respawned last */
00088     int count;                               /* Times respawned in the last 2 minutes */
00089     char id[8];                               /* Inittab id (must be unique) */
00090     char rlevel[12];                          /* run levels */
00091     int action;                              /* what to do (see list below) */
00092     char process[128];                       /* The command line */
00093     struct _child_ *new;                     /* New entry (after inittab re-read) */
00094     struct _child_ *next;                    /* For the linked list */
00095 } CHILD;
00096
```

#### actions 数组 (struct actions)

这个数组保存的都是常量，包括常量字符串和宏定义，主要是一组对应关系，方便把/etc/inittab 文件中的字符串转换为整型数。

例如 respawn -> RESPAWN, sysinit -> SYSINIT, initdefault -> INITDEFAULT

```
00150
00151 /* ascii values for the `action' field. */
00152 struct actions {
00153     char *name;
```

```

00154  int act;
00155 } actions[] = {
00156  { "respawn",      RESPAWN      },
00157  { "wait",         WAIT         },
00158  { "once",         ONCE         },
00159  { "boot",         BOOT         },
00160  { "bootwait",     BOOTWAIT     },
00161  { "powerfail",    POWERFAIL    },
00162  { "powerfailnow", POWERFAILNOW },
00163  { "powerwait",    POWERWAIT    },
00164  { "powerokwait",  POWEROKWAIT  },
00165  { "ctrlaltdel",   CTRLALTDDEL  },
00166  { "off",          OFF          },
00167  { "ondemand",     ONDEMAND     },
00168  { "initdefault",  INITDEFAULT  },
00169  { "sysinit",      SYSINIT      },
00170  { "kbrequest",    KBREQUEST    },
00171  { NULL,          0             },
00172 };

```

## ACTIONS 宏定义

这一组宏定义的值，也是保存在 `init.h` 头文件中。

```

00065 /* Actions to be taken by init */
00066 #define RESPAWN      1
00067 #define WAIT         2
00068 #define ONCE         3
00069 #define BOOT         4
00070 #define BOOTWAIT     5
00071 #define POWERFAIL    6
00072 #define POWERWAIT    7
00073 #define POWEROKWAIT  8
00074 #define CTRLALTDDEL  9
00075 #define OFF          10
00076 #define ONDEMAND     11
00077 #define INITDEFAULT  12
00078 #define SYSINIT      13
00079 #define POWERFAILNOW 14
00080 #define KBREQUEST    15

```

## read\_inittab 函数执行流程分析

1. 读取 /etc/inittab 文件，按行读取，到 buf 数组中。
2. 遇到开头是空格或者 TAB 制表符的行，忽略直到第一个字母，如果发现是第一个字母是 # 开头的注释，或者 \n 开头的空行，都直接跳过。
3. 使用 strsep 函数，以 : 冒号作为间隔符号，依次找到 id, rlevel, action, process 这 4 个字段，分别代表的含义可参考下面的详细说明。同时将 action 字段中的字符串关键字转换为整型数 actionNo，方便后面的判别。
4. 检查当前的 id 字段，是否是唯一的，如果之前已经出现过，则忽略掉。
5. 通过 imalloc 函数，动态分配 CHILD 结构体节点 ch，结构体的定义见上面。然后将刚才分析的结果填入结构体中，并将这个节点，添加到链表 newFamily 中。其中包括 actionNo 填入 ch->action, id 填入 ch->id, process 填入 ch->process 等。
6. 关闭 /etc/inittab 文件。
7. 接下来，查看老的启动进程列表 family，看是否有进程需要被杀死的。这里有两轮检查，第一轮会给所有没有在新的运行级别中定义的进程发送一个警告信号 SIGTERM。如果在第一轮中有这样的进程，则会等待 5 秒，然后进入下一轮检查。在第 2 轮检查中，它会发送 SIGKILL 信号来强制中止所有子进程的运行。
8. 等所有子进程被杀死后，init 通过调用 write\_utmp\_wtmp() 来将终止信息和原因记录进这两个文件中。记录的信息包括子进程在 inittab 文件中的 id，子进程本身的 pid 等。
9. 这 2 个步骤 7, 8 完成之后，init 开始清除老的 family 链表上的所有节点，释放空间。
10. 最后 init 把刚才新建成的 newFamily 链表赋值给 -> family 链表，完成重建链表的操作即结束。

### 3.1.6 start\_if\_needed 函数流程分析

该函数主要完成的功能是：遍历 family 链表，调用 startup 启动链表上的子进程。

函数执行流程分析：

1. 从 family 链表的表头开始遍历该链表，根据每一个节点 ch 的 flags 标志来进行判别。
2. 如果当前节点 flags 表示 WAITING，则说明正在等待，之前的工作未完成，立即退出该函数。
3. 如果当前节点 flags 表示 RUNNING，则对这个正在运行的进程不做任何操作，继续下一个。
4. 如果当前节点的运行级别正好是当前 init 运行级别，则调用 startup 函数启动这个进程。
5. 如果当前节点不属于在当前运行级别中运行的程序，则将节点 flags 设置为 ~(RUNNING | WAITING) 表示不是运行中，也不是等待中。

### 3.1.7 startup 函数流程分析

该函数主要完成的功能是：执行 CHILD 节点所代表的配置行上的命令行，通常是个脚本程序。

函数执行流程分析：

1. 对于 CHILD \*ch 节点中的 action 字段来进行判别。如果是 SYSINIT, BOOTWAIT, WAIT, POWERWAIT, POWERFAILNOW, POWEROKWAIT, CTRLALTDDEL 这些情况，则设置标志为 WAITING，然后执行 spawn 函数。这个函数是完成启动子进程的真正的函数，spawn 名字的含义是产卵的意思，顾名思义就是产生后继的子进程。后面我们再对这个函数做详细分析。
2. 如果是 KBREQUEST, BOOT, POWERFAIL, ONCE 则直接退出，不进行后继的 spawn 函数调用。
3. 如果是 ONDEMAND, RESPAWN，则将 flags 设置为 RUNNING 后，立即执行 spawn 操作。

### 3.1.8 spawn 函数流程分析

该函数主要完成的功能是：调用 fork 和 execp 来启动子进程。这个函数非常长，但基本上是属于最底层的函数了。

函数执行流程分析：

1. spawn 整个程序比较长，从 927-1192 行约有 270 多行。整个代码逻辑以 fork 调用为分界线，可以分为 2 个部分。前面部分主要完成启动前的准备工作，后面通过 fork 和 execp 来实际创建出子进程执行 CHILD 节点上规定的程序。
2. 先分析第一部分。这部分代码主要处理三种情况，1 是 action 为“RESPAWN”与“ONDEMAND”类型的命令；2 是 /etc/initscript 初始化脚本为后继 execp 调用准备参数。
3. 第二部分进入到一个无限循环中，以便确保能够成功创建出子进程。在调用 fork 创建出 init 的子进程之后，init 的这个子进程将按照 daemon 进程的方式工作，包括需要关闭 0, 1, 2 打开文件。也就是说，真正用来创建用户子进程的，不是 pid = 1 的那个原始进程，而是原始进程的子进程再通过一个 fork 和 execp 才能够实现执行真正的用户程序。
4. 第 2 次执行 fork 之后，由子进程调用 execp 来完成加载用户程序，而父进程通过调用 waitpid 来等待子进程的结束。
5. 上述步骤完成之后，父进程又会创建出一个临时的子进程，来完成 setsid() 和 ioctl(f, TIOCSCTTY, 1) 这 2 个函数调用，来分配一个控制终端，创建一个新的会话，失去原有的控制终端的所有联系。

### 3.1.9 boot\_transitions 函数流程分析

该函数主要完成的功能是：实现一个启动过程中所需要的状态机，完成状态的迁移。

函数执行流程分析：

1. 以 runlevel 代表状态，如果当前 runlevel = '#' 状态开始，系统进入 SYSINIT -> BOOT 的转变。
2. 如果在 read\_inittab 时从文件中获得了 def\_level，则直接用这个变量的值，否则通过 get\_init\_default() 得到的是默认的运行级别并赋值给 newlevel
3. 如果 newlevel 是 'S'，则下一个状态为 'S'，否则下一个状态设为 '\*'
4. 如果当前 runlevel 是 '\*'，则系统从 BOOT -> NORMAL。

5. 如果当前 `runlevel` 是 'S', 则代表着 `SU` 模式已经结束, 重新调用 `get_init_default()` 得到新的运行级别 `newlevel`.
6. 将本次状态变迁的信息写入日志 `write_utmp_wtmp()`

### 3.1.10 `check_init_fifo()` 函数流程分析

该函数主要完成的功能是：主要用于 `init daemon` 程序中, 通过 `select` 函数监听来自于 `/dev/initctl` 管道的请求 `request`, 分析并执行该请求 `request`.

函数执行流程分析：

1. 如果 `/etc/initctl` 管道不存在, 则创建这个管道, 并设置权限 `0600`, 只允许 `root` 用户读写。
2. 如果管道已经打开, 则比较该管道是否是最初原始打开的管道。如果不是, 则关闭后, 重新打开。
3. 以读写 + 非阻塞方式打开管道, 并且使用 `dup2` 将采用 `PIPE_FD = 10` 来使用管道, 而不使用 `0, 1, 2`
4. 使用 `select` 调用在该管道上等待来自于 `init N` 的切换运行级别的请求 `request`
5. 一旦有来自这个管道的 `request`, 则检查这个 `request` 数据的合法性
6. 对于输入正确的 `request` 请求, 则分析是什么请求, 并判断要采取什么动作。
7. 请求包括进行
  - `INIT_CMD_RUNLVL` (`runlevel` 的切换) -> 调用 `fifo_new_level()`
  - `INIT_CMD_POWERFAIL`
  - `INIT_CMD_POWERFAILNOW`
  - `INIT_CMD_POWEROK` (以上三个请求都是和电源事件有关) -> 调用 `do_power_fail()`
  - `INIT_CMD_SETENV` (设置环境变量) -> 调用 `initcmd_setenv()`

### `struct init_request` 请求协议格式

通过 `/etc/initctl` 管道进行请求的数据, 需要遵循一定的格式, 也就是需要能够转换为如下的 `init_request` 结构体数据。

```

00073 struct init_request {
00074     int      magic;           /* Magic number           */
00075     int      cmd;             /* What kind of request   */
00076     int      runlevel;        /* Runlevel to change to  */
00077     int      sleeptime;       /* Time between TERM and KILL */
00078     union {
00079         struct init_request_bsd bsd;
00080         char                    data[368];
00081     } i;
00082 };
00083

```

### cmd 请求类别标识

所有正确的请求，都有一个唯一的标识，这些标识定义在 `initreq.h` 头文件中。

```

00035 #define INIT_CMD_START          0
00036 #define INIT_CMD_RUNLVL        1
00037 #define INIT_CMD_POWERFAIL     2
00038 #define INIT_CMD_POWERFAILNOW  3
00039 #define INIT_CMD_POWEROK       4
00040 #define INIT_CMD_BSD           5
00041 #define INIT_CMD_SETENV        6
00042 #define INIT_CMD_UNSETENV      7

```

### 3.1.11 fail\_check 函数流程分析

该函数主要完成的功能是：在每次信号处理完成之后，遍历 `family` 链表检查每个节点的状态

函数执行流程分析：

1. 首先调用 `time(&t)` 获得系统时间。
2. 从 `family` 链表头开始，遍历整个链表，直到结束。



3. 检查每一个节点 `ch` 的 `flags` 是否表示 `FAILING`
4. 如果是，并且这个进程已经睡眠 `sleep` 了至少 5 分钟，则会清除掉 `flags` 中的 `FAILING` 标识位。
5. 如果不是，则设置下一次 `alarm` 的时间为这个进程 `sleep` 的时间加上 5 分钟。

## **SLEEPTIME 数据**

睡眠时间超过 300 秒 = 5 分钟的进程，将会被清除标志位

```
#define SLEEPTIME 300
```

### **3.1.12 process\_signals 函数流程分析**

该函数主要完成的功能是：根据全局变量 `got_signals` 中哪些标志位被设置了，获得信号类型，进行相应的处理。

函数执行流程分析：

程序执行逻辑很简单，就是依次判别 `ISMEMBER(got_signals, SIGXXXX)` 对于以下信号进行相应处理。

1. `SIGPWR` 信号 -> `do_power_fail()`
2. `SIGINT` 信号 -> 通知 `ctrlaltdel` 入口启动
3. `SIGWINCH` 信号 -> 通知 `KBREQUEST` 入口启动
4. `SIGALRM` 信号 -> 定时器到时，忽略
5. `SIGCHLD` 信号 -> 查看是哪个子进程结束，调用 `write_utmp_wtmp()` 写入日志
6. `SIGHUP` 信号 -> 是否在等待子进程，进行 `runlevel` 切换
7. `SIGUSR1` 信号 -> 这个信号代表要求关闭然后重新打开 `/dev/initctl`

在 `set.h` 头文件中有关于这个宏定义的实现

```
#define ISMEMBER(set, val) ((set) & (1 << (val)))
#define DELSET(set, val) ((set) &= ~(1 << (val)))
#define ADDSET(set, val) ((set) |= (1 << (val)))
#define EMPTYSET(set) ((set) = 0)
```

通过 `kill -l` 可以得到这些 `SIGXXXX` 的具体赋值，如下：

```
$ kill -l
1) SIGHUP    2) SIGINT    3) SIGQUIT  4) SIGILL    5) SIGTRAP
6) SIGABRT   7) SIGBUS    8) SIGFPE   9) SIGKILL  10) SIGUSR1
11) SIGSEGV  12) SIGUSR2  13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN  22) SIGTTOU 23) SIGURG   24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO   30) SIGPWR
31) SIGSYS   34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
$
```

### 3.1.13 `console_stty` 函数流程分析

该函数主要完成的功能是：设置终端工作参数

函数执行流程分析：

1. 调用 `console_open` 打开 `console_dev` 设备，模式为读写 + 非阻塞方式。
2. 调用 `tcgetattr()` 函数获得当前终端属性 `tty` (`struct termios` 结构体)
3. 设置 `tty.c_cflag` 和 `tty.c_cc[]` 的参数配置。

4. 设置 `tty.c_iflag` 和 `tty.c_oflag` 以及 `tty.c_lflag` 参数配置。
5. 调用 `tcsetattr()` 和 `tcflush()` 完成设置终端属性的操作。
6. 调用 `close(fd)` 关闭终端设备文件。

### 3.1.14 `fifo_new_level` 函数流程分析

该函数主要完成的功能是：真正完成改变 `runlevel` 的 `request` 请求，目标为传入参数 `level`，通过重新读取 `inittab` 文件来启动与新 `runlevel` 匹配的命令脚本。

函数执行流程分析：

1. 如果传入参数 `level` 和当前的 `runlevel` 运行级别一致，则无需修改直接返回。
2. 如果新的 `runlevel = 'U'`，则通过调用 `re_exec()` 来执行改变 `runlevel` 的操作。
3. 如果新的 `runlevel != 'U'`，则通过调用 `read_inittab()` 来重新生成 `family` 链表。

### 3.1.15 `re_exec` 函数流程分析

该函数主要完成的功能是：强制 `init` 程序重新执行。

函数执行流程分析：

1. 该函数会创建 `STATE_PIPE`，并向 `STATE_PIPE` 写入 `Signature = "12567362"`
2. 接着 `fork()` 出一个子进程，通过子进程调用 `send_state()` 向 `STATE_PIPE` 写入父进程（当前 `init` 进程）的状态信息；
3. 然后父进程调用 `execle()` 重新执行 `init` 程序，并且传递参数 “`--init`”，也就是强制 `init` 重新执行。而这个重新执行的 `init` 进程，无需做初始化读取 `/etc/inittab` 就能调用 `init_main()`。

### 3.1.16 get\_init\_default 函数流程分析

该函数主要完成的功能是：查找/etc/inittab 文件中的 initdefault 默认运行级别，如果有则返回，如果没有则请用户输入。

函数执行流程分析：

1. 实际上这个函数是从 family 链表中遍历，取出每一个节点 ch
2. 如果 ch->action == INITDEFAULT ，则将当前 ch 的运行级别赋值给 lvl
3. 判断如果 lvl 是小写，则转换为大写。并且对 lvl 进行判别，看它是否属于 “0123456789S “ 的其中之一。
4. 如果从文件中得到的 lvl 正确，则返回 lvl；
5. 如果从文件中无法得到正确的 lvl，则调用 ask\_runlevel() 函数返回。这个函数中会通过终端来询问用户，并要求用户输入一个默认运行级别。

### 3.1.17 telinit 函数流程分析

在执行 telinit 函数时，实际上是通过向 INIT\_FIFO ( /dev/initctl ) 写入命令的方式，通知 init 执行相应的操作。Telinit() 根据不同请求，构造如下结构体类型的变量并向 INIT\_FIFO ( /dev/initctl ) 写入该请求来完成其使命：

```
struct init_request { int magic; /* Magic number / int cmd; / What kind of
request / int runlevel; / Runlevel to change to / int sleeptime; / Time
between TERM and KILL */ union { struct init_request_bsd bsd; char data[368]; } i; };
```

## 3.2 init 进程执行流程分析汇总

通过以上这些子函数的分析，我们可以总结一下关于 init 进程的运行状态和相应的执行流程。

### 3.2.1 init 程序的 3 种启动执行方式

#### 方式 1- Kernel 启动 init

在内核启动代码中，start\_kernel 函数初始化代码的结束，会通过 command\_line 来找出 init=execute\_command 字符串中的程序来执行，或者按照默认的 4 个 init 程序的顺序依次来调用 execve() 执行 init 进程。这种方

式启动的 `init` 进程，会完成读取 `/etc/inittab` 文件，建立 family 链表，依次执行各个子进程，并等待子进程的结束。当 `init` 进程运行到最后会进入一个无限循环中，变成一个 daemon `init` 进程。

这种启动方式，也是在整个操作系统启动过程中，`init` 程序的初次执行。这种启动是在内核空间启动 `init`。

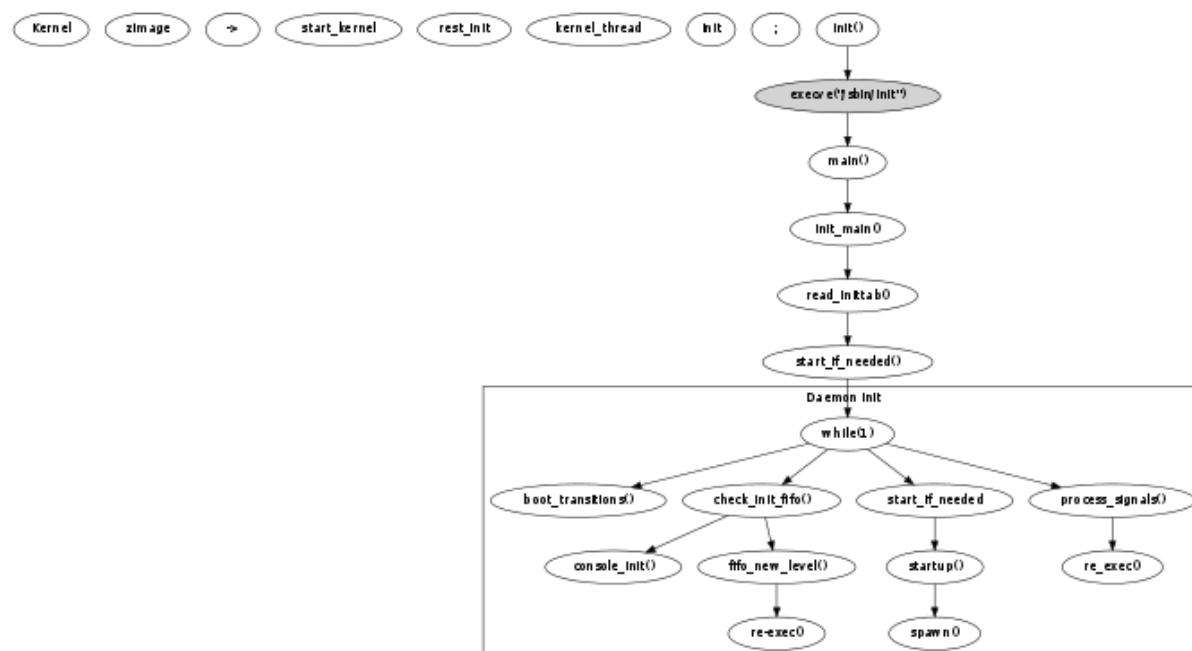


Figure 3.2: 方式 1- Kernel 启动 `init`

## 方式 2- 用户命令 `telinit` 启动 `init`

在 `init` 进程启动之后，用户通过终端可以完成登录进入 `Bash` 中，执行 `telinit` 命令的时候，因为 `telinit` 命令本身就是一个指向 `init` 程序的软链接，所以会导致 `init` 程序再次被执行。通过这种方式运行起来的 `init` 进程，因为 `pid != 1` 因此可以判断不是 Kernel 创建的 `init` 进程，此时会转为调用 `telinit()` 函数来执行。

这种情况下，`telinit()` 函数只负责打开 `INIT_FIFO (/dev/initctl)` 并按照传入参数，组织为一个 `struct request` 结构体，写入 `FIFO` 中，通知方式 1 中的 `init` 进程，就完成任务了。

这种启动方式通常会涉及到 `runlevel` 的切换，例如执行 `telinit 1` 或者 `init 1` 就会引起系统切换到单用户模式下。这是 `init` 程序的第 2 种常用的启动方式。我们可以看成是在用户空间启动 `init`。

## 方式 3- 在程序中通过 `re_exec()` 函数启动 `init`

这种方式发生在通过方式 1 启动了 `init` 之后，在 `init` 执行的最后，进入了一个无限循环等待中。此时，用户如果在终端下执行 `telinit U` 命令，则代表着用

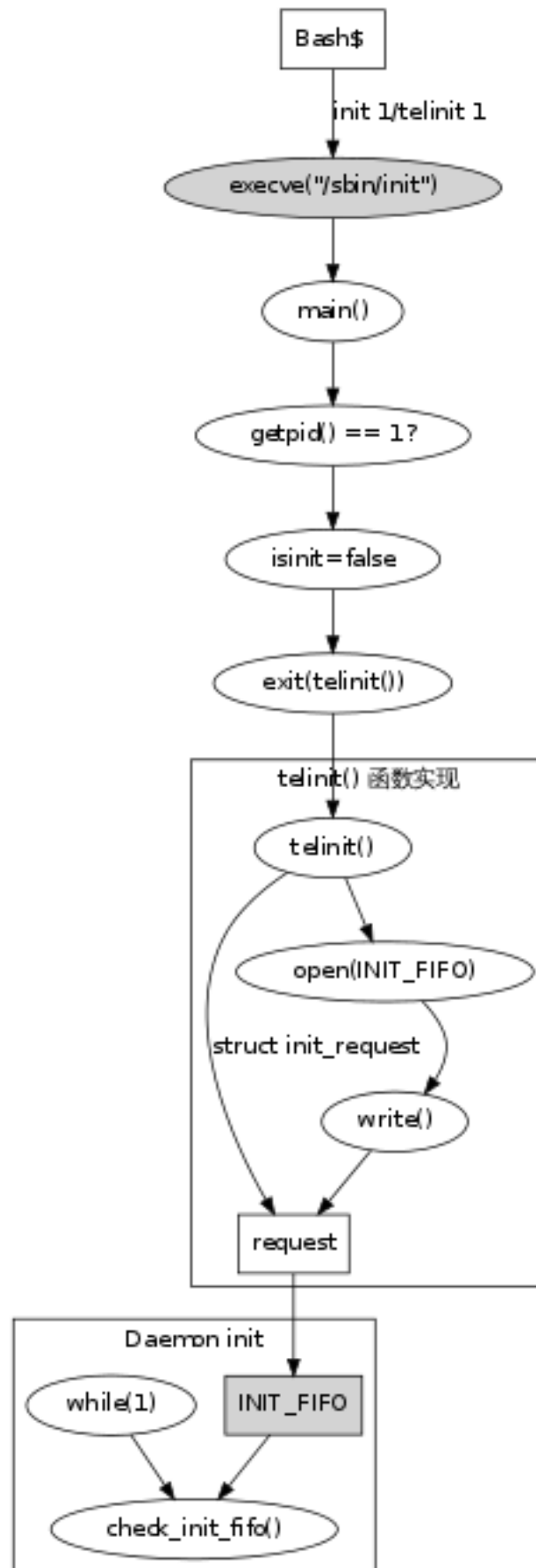


Figure 3.3: 方式 2- 用命令 telinit 启动 init

户希望 re-execute itself, 那么在方式 2 启动 init 之后, 新的 init 进程会发送 U 命令给方式 1 启动的 init 进程, 这个最原始的进程在循环中会调用 process\_signals() 来处理 U 命令, 处理方法是调用 re\_exec() 函数。在这个函数中, 会 fork 出一个子进程, 子进程通过管道向父进程发送消息, 由父进程通过 execl() 重新执行 init 程序, 并传递 --init 参数, 强制 init 重新执行。

和方式 1 的执行所不同的是, 方式 1 在执行的后期, 会读取/etc/inittab 文件, 建立 family 链表; 而方式 3 因为是通过 telinit U 的方式告诉方式 1 启动的那个 daemon init 进程, 调用 re\_exec() 函数, 因此最原始的那个 init 进程, 不会进行之前的 readinittab() 初始化操作, 而是直接进入无限循环, 又一次进入 daemon 的等待/处理循环中。

### 方式 1, 2, 3 的比较区别

- 通过方式 1 启动的 init 进程 pid=1。
- 通过方式 3, 又让 pid=1 的进程调用了 re\_exec() -> fork() -> execl() 来 (让父进程) 重新加载了一次的 init 进程, 本质上其实都是 1 号进程。
- 通过方式 2 启动的 init 进程, pid 一定不是 1, 所以这个进程和前面的这个 init 进程完全不同。它们是分别属于内核空间和用户空间的 2 个不同的进程 (前者进程 1 其实应该称为内核线程, 因为它通过 kernel\_thread() 创建出来的, 而后者是通过 shell 在用户空间 fork 出来的, 是真正的用户。

### 3.2.2 halt 命令分析

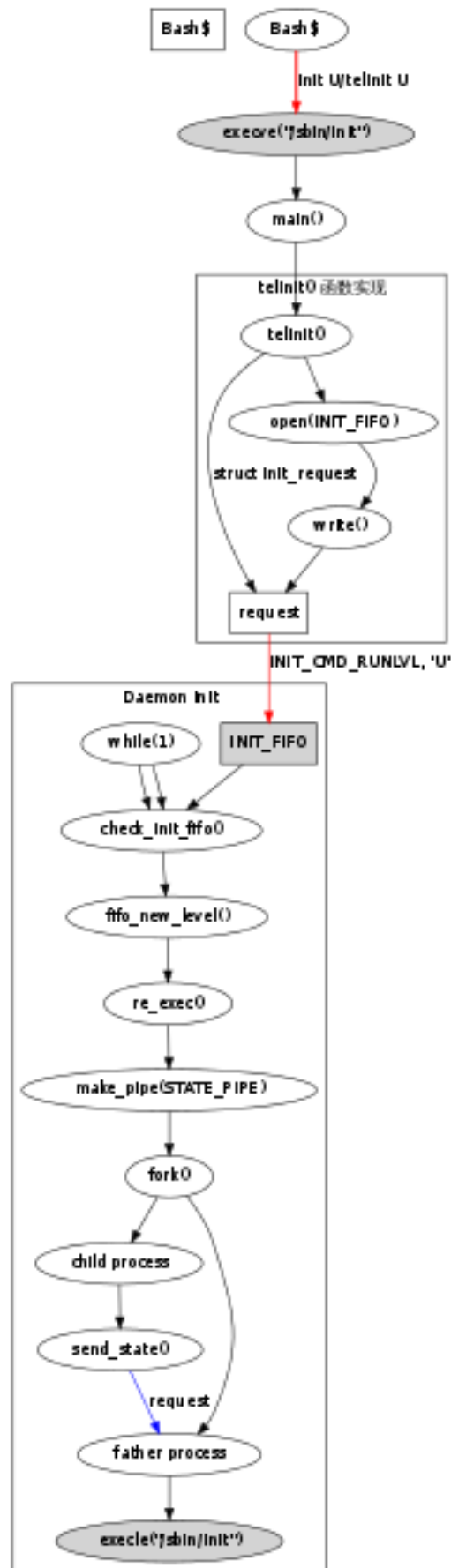


Figure 3.4: 方式 3- 在程序中通过 `re_exec()` 函数启动 `init`



## Chapter 4

### Sysvinit 项目安全漏洞

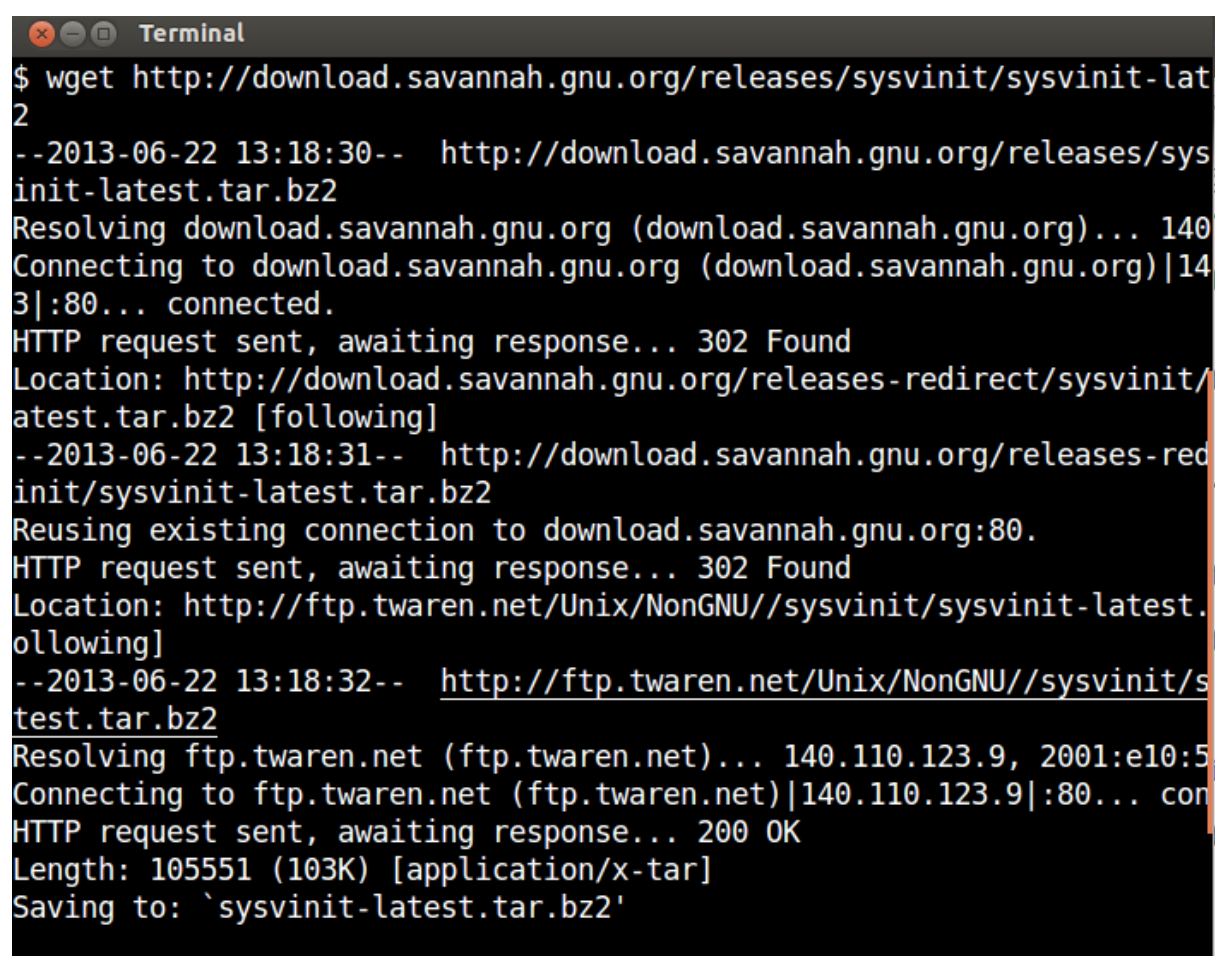


## Chapter 5

# Sysvinit 项目运行时调试图

### 5.1 编译安装运行调试图

#### 5.1.1 wget 下载源码包



```
Terminal
$ wget http://download.savannah.gnu.org/releases/sysvinit/sysvinit-latest.tar.bz2
--2013-06-22 13:18:30-- http://download.savannah.gnu.org/releases/sysvinit-latest.tar.bz2
Resolving download.savannah.gnu.org (download.savannah.gnu.org)... 140.110.123.9
Connecting to download.savannah.gnu.org (download.savannah.gnu.org)|140.110.123.9|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://download.savannah.gnu.org/releases-redirect/sysvinit-latest.tar.bz2 [following]
--2013-06-22 13:18:31-- http://download.savannah.gnu.org/releases-redirect/sysvinit-latest.tar.bz2
Reusing existing connection to download.savannah.gnu.org:80.
HTTP request sent, awaiting response... 302 Found
Location: http://ftp.twaren.net/Unix/NonGNU//sysvinit/sysvinit-latest.tar.bz2 [following]
--2013-06-22 13:18:32-- http://ftp.twaren.net/Unix/NonGNU//sysvinit/sysvinit-latest.tar.bz2
Resolving ftp.twaren.net (ftp.twaren.net)... 140.110.123.9, 2001:e10:5...
Connecting to ftp.twaren.net (ftp.twaren.net)|140.110.123.9|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 105551 (103K) [application/x-tar]
Saving to: `sysvinit-latest.tar.bz2'
```

Figure 5.1: wget 下载源码包

```

$ wget http://download.savannah.gnu.org/releases/sysvinit/sysvinit-latest.tar.bz2
--2013-06-22 13:18:30-- http://download.savannah.gnu.org/releases/sysvinit/sysvinit-l
Resolving download.savannah.gnu.org (download.savannah.gnu.org)... 140.186.70.73
Connecting to download.savannah.gnu.org (download.savannah.gnu.org)|140.186.70.73|:80.
HTTP request sent, awaiting response... 302 Found
Location: http://download.savannah.gnu.org/releases-redirect/sysvinit/sysvinit-latest
--2013-06-22 13:18:31-- http://download.savannah.gnu.org/releases-redirect/sysvinit/s
Reusing existing connection to download.savannah.gnu.org:80.
HTTP request sent, awaiting response... 302 Found
Location: http://ftp.twaren.net/Unix/NonGNU//sysvinit/sysvinit-latest.tar.bz2 [followi
--2013-06-22 13:18:32-- http://ftp.twaren.net/Unix/NonGNU//sysvinit/sysvinit-latest.t
Resolving ftp.twaren.net (ftp.twaren.net)... 140.110.123.9, 2001:e10:5c00:5::9
Connecting to ftp.twaren.net (ftp.twaren.net)|140.110.123.9|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 105551 (103K) [application/x-tar]
Saving to: `sysvinit-latest.tar.bz2'

100%[=====>] 105,551      45.1K/s   in 2.3s

2013-06-22 13:18:35 (45.1 KB/s) - `sysvinit-latest.tar.bz2' saved [105551/105551]

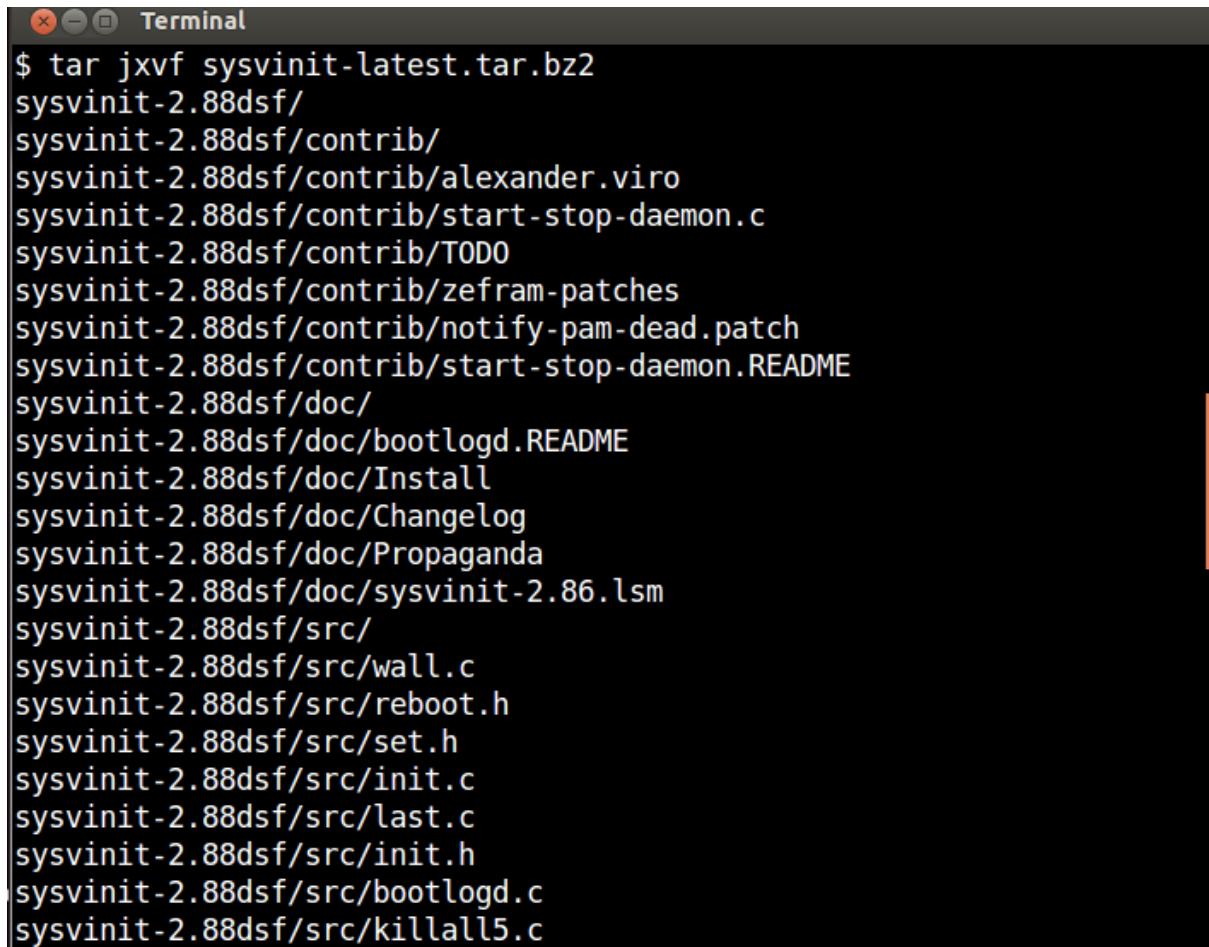
```

### 5.1.2 tar 解压源码包

```

$ tar jxvf sysvinit-latest.tar.bz2
sysvinit-2.88dsf/
sysvinit-2.88dsf/contrib/
sysvinit-2.88dsf/contrib/alexander.viro
sysvinit-2.88dsf/contrib/start-stop-daemon.c
sysvinit-2.88dsf/contrib/TODO
sysvinit-2.88dsf/contrib/zefram-patches
sysvinit-2.88dsf/contrib/notify-pam-dead.patch
sysvinit-2.88dsf/contrib/start-stop-daemon.README
sysvinit-2.88dsf/doc/
sysvinit-2.88dsf/doc/bootlogd.README
sysvinit-2.88dsf/doc/Install
sysvinit-2.88dsf/doc/Changelog
sysvinit-2.88dsf/doc/Propaganda
sysvinit-2.88dsf/doc/sysvinit-2.86.lsm
sysvinit-2.88dsf/src/

```

A terminal window titled "Terminal" with a dark background and light-colored text. The window shows the output of a tar command. The first line is the command prompt followed by the command. The subsequent lines are the names of the files and directories extracted from the archive, listed one per line.

```
$ tar jxvf sysvinit-latest.tar.bz2
sysvinit-2.88dsf/
sysvinit-2.88dsf/contrib/
sysvinit-2.88dsf/contrib/alexander.viro
sysvinit-2.88dsf/contrib/start-stop-daemon.c
sysvinit-2.88dsf/contrib/TODO
sysvinit-2.88dsf/contrib/zefram-patches
sysvinit-2.88dsf/contrib/notify-pam-dead.patch
sysvinit-2.88dsf/contrib/start-stop-daemon.README
sysvinit-2.88dsf/doc/
sysvinit-2.88dsf/doc/bootlogd.README
sysvinit-2.88dsf/doc/Install
sysvinit-2.88dsf/doc/Changelog
sysvinit-2.88dsf/doc/Propaganda
sysvinit-2.88dsf/doc/sysvinit-2.86.lsm
sysvinit-2.88dsf/src/
sysvinit-2.88dsf/src/wall.c
sysvinit-2.88dsf/src/reboot.h
sysvinit-2.88dsf/src/set.h
sysvinit-2.88dsf/src/init.c
sysvinit-2.88dsf/src/last.c
sysvinit-2.88dsf/src/init.h
sysvinit-2.88dsf/src/bootlogd.c
sysvinit-2.88dsf/src/killall5.c
```

Figure 5.2: tar 解压源码包

sysvinit-2.88dsf/src/wall.c  
sysvinit-2.88dsf/src/reboot.h  
sysvinit-2.88dsf/src/set.h  
sysvinit-2.88dsf/src/init.c  
sysvinit-2.88dsf/src/last.c  
sysvinit-2.88dsf/src/init.h  
sysvinit-2.88dsf/src/bootlogd.c  
sysvinit-2.88dsf/src/killall5.c  
sysvinit-2.88dsf/src/utmpdump.c  
sysvinit-2.88dsf/src/shutdown.c  
sysvinit-2.88dsf/src/mountpoint.c  
sysvinit-2.88dsf/src/sulogin.c  
sysvinit-2.88dsf/src/fstab-decode.c  
sysvinit-2.88dsf/src/initreq.h  
sysvinit-2.88dsf/src/dowall.c  
sysvinit-2.88dsf/src/hddown.c  
sysvinit-2.88dsf/src/paths.h  
sysvinit-2.88dsf/src/utmp.c  
sysvinit-2.88dsf/src/ifdown.c  
sysvinit-2.88dsf/src/initscript.sample  
sysvinit-2.88dsf/src/halt.c  
sysvinit-2.88dsf/src/oldutmp.h  
sysvinit-2.88dsf/src/mesg.c  
sysvinit-2.88dsf/src/Makefile  
sysvinit-2.88dsf/src/runlevel.c  
sysvinit-2.88dsf/COPYING  
sysvinit-2.88dsf/COPYRIGHT  
sysvinit-2.88dsf/man/  
sysvinit-2.88dsf/man/bootlogd.8  
sysvinit-2.88dsf/man/killall5.8  
sysvinit-2.88dsf/man/shutdown.8  
sysvinit-2.88dsf/man/bootlogd.8.todo  
sysvinit-2.88dsf/man/sulogin.8  
sysvinit-2.88dsf/man/fstab-decode.8  
sysvinit-2.88dsf/man/mesg.1  
sysvinit-2.88dsf/man/initscript.5  
sysvinit-2.88dsf/man/inittab.5  
sysvinit-2.88dsf/man/poweroff.8  
sysvinit-2.88dsf/man/wall.1  
sysvinit-2.88dsf/man/halt.8  
sysvinit-2.88dsf/man/reboot.8  
sysvinit-2.88dsf/man/last.1  
sysvinit-2.88dsf/man/runlevel.8  
sysvinit-2.88dsf/man/lastb.1  
sysvinit-2.88dsf/man/pidof.8

```

sysvinit-2.88dsf/man/init.8
sysvinit-2.88dsf/man/utmpdump.1
sysvinit-2.88dsf/man/mountpoint.1
sysvinit-2.88dsf/man/telinit.8
sysvinit-2.88dsf/obsolete/
sysvinit-2.88dsf/obsolete/powerd.c
sysvinit-2.88dsf/obsolete/powerd.8
sysvinit-2.88dsf/obsolete/utmpdump.c.OLD
sysvinit-2.88dsf/obsolete/README.RIGHT.NOW
sysvinit-2.88dsf/obsolete/bootlogd.init
sysvinit-2.88dsf/obsolete/powerd.README
sysvinit-2.88dsf/obsolete/powerd.cfg
sysvinit-2.88dsf/Makefile
sysvinit-2.88dsf/README
$

$ ls
Makefile  pdf  sysvinit-2.88dsf  sysvinit-latest.tar.bz2

$ ls sysvinit-2.88dsf/
contrib  COPYRIGHT  Makefile  obsolete  src
COPYING  doc        man       README
$

```

### 5.1.3 编译项目源码

```

$ cd sysvinit-2.88dsf/
$ make
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o mountpoint.o mountpoint.
cc  mountpoint.o -o mountpoint
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o init.o init.c
init.c: In function 'telinit' :
init.c:2737:7:  warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'get_record' :
init.c:377:11:  warning: ignoring return value of 'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:380:11:  warning: ignoring return value of 'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:383:11:  warning: ignoring return value of 'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]

```

```

init.c:386:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:389:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:392:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:395:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:398:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:401:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:404:11: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:423:10: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:426:10: warning: ignoring return value of      'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'spawn' :
init.c:1064:10: warning: ignoring return value of      'dup'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:1065:10: warning: ignoring return value of      'dup'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:1133:7: warning: ignoring return value of      'dup'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:1134:7: warning: ignoring return value of      'dup'
, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'ask_runlevel' :
init.c:1673:10: warning: ignoring return value of      'write'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:1675:9: warning: ignoring return value of      'read'
, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'make_pipe' :
init.c:1960:6: warning: ignoring return value of      'pipe'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:1965:7: warning: ignoring return value of      'write'
, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'process_signals' :
init.c:2411:7: warning: ignoring return value of      'read'
, declared with attribute warn_unused_result [-Wunused-result]
init.c:2420:7: warning: ignoring return value of      'read'
, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'coredump' :
init.c:666:7: warning: ignoring return value of      'chdir'

```



```

, declared with attribute warn_unused_result [-Wunused-result]
init.c: In function 'print' :
init.c:821:8: warning: ignoring return value of 'write'
, declared with attribute warn_unused_result [-Wunused-result]
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -DINIT_MAIN -c -o init_utmp.o ut
cc init.o init_utmp.o -o init
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o halt.o halt.c
halt.c: In function 'main' :
halt.c:242:2: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o ifdown.o ifdown.c
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o hddown.o hddown.c
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o utmp.o utmp.c
cc halt.o ifdown.o hddown.o utmp.o reboot.h -o halt
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o shutdown.o shutdown.c
shutdown.c: In function 'main' :
shutdown.c:485:10: warning: variable 'realuid'
set but not used [-Wunused-but-set-variable]
shutdown.c:630:9: warning: ignoring return value of 'fscanf'
, declared with attribute warn_unused_result [-Wunused-result]
shutdown.c:719:7: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
shutdown.c: In function 'spawn' :
shutdown.c:289:7: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o dowall.o dowall.c
cc shutdown.o dowall.o utmp.o reboot.h -o shutdown
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o runlevel.o runlevel.c
cc runlevel.o -o runlevel
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o sulogin.o sulogin.c
sulogin.c: In function 'sushell' :
sulogin.c:407:2: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
sulogin.c:427:8: warning: ignoring return value of 'getcwd'
, declared with attribute warn_unused_result [-Wunused-result]
cc sulogin.o -o sulogin
sulogin.o: In function `main':
sulogin.c:(.text.startup+0x1e2): undefined reference to `crypt'
collect2: ld returned 1 exit status
make: *** [sulogin] Error 1
$

```

```
Terminal
69
70 ifeq ($(WITH_SELINUX),yes)
71     SELINUX_DEF = -DWITH_SELINUX
72     INITLIBS    += -lsepol -lselinux
73     SLOGINLIBS  = -lselinux
74 else
75     SELINUX_DEF =
76     INITLIBS    =
77     SLOGINLIBS  =
78 endif
79
80 SLOGINLIBS    += -lcrypt
81 # Additional libs for GNU libc.
82 ifneq ($(wildcard /usr/lib*/libcrypt.a),)
83     SLOGINLIBS += -lcrypt
84 endif
85
86 all:          $(BIN) $(SBIN) $(USBIN)
87
88 #%: %.o
89 #      $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LDLIBS)
90 #%.o: %.c
91 #      $(CC) $(CFLAGS) $(CPPFLAGS) -c $^ -o $@
"Makefile" 184L, 4343C written      80,1      42%
```

Figure 5.3: 修改 Makefile

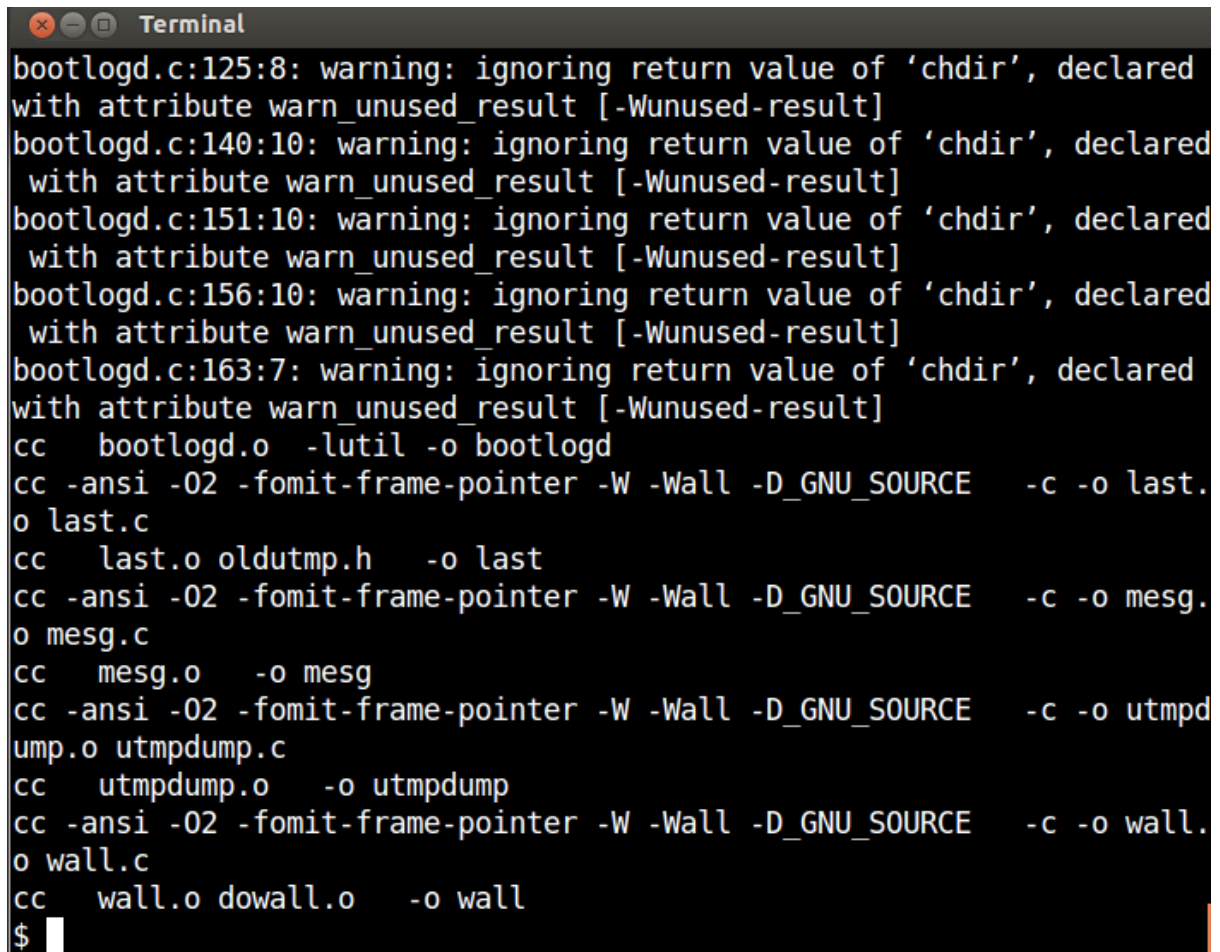
### 5.1.4 修改 Makefile 使之能够编译通过

```
$ vi Makefile
69
70 ifeq ($(WITH_SELINUX),yes)
71     SELINUX_DEF    = -DWITH_SELINUX
72     INITLIBS        += -lsepol -lselinux
73     SULOGINLIBS     = -lselinux
74 else
75     SELINUX_DEF    =
76     INITLIBS        =
77     SULOGINLIBS     =
78 endif
79
80 SULOGINLIBS        += -lcrypt
81 # Additional libs for GNU libc.
82 ifneq ($(wildcard /usr/lib*/libcrypt.a),)
83     SULOGINLIBS     += -lcrypt
84 endif
85
86 all:                $(BIN) $(SBIN) $(USRBIN)
87
88 #%: %.o
89 #      $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LDLIBS)
90 #%.o: %.c
91 #      $(CC) $(CFLAGS) $(CPPFLAGS) -c $^ -o $@
```

在 80 行处添加 83 行处的赋值，增加链接时 `-lcrypt` 选项

### 5.1.5 继续编译项目源码，成功

```
$ make
cc  sulogin.o  -lcrypt  -o sulogin
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o bootlogd.o bootlogd.c
bootlogd.c: In function 'findtty':
bootlogd.c:125:8:  warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
bootlogd.c:140:10: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
```

A terminal window titled "Terminal" with standard window controls (close, minimize, maximize). The terminal displays the output of a compilation process. It shows several warning messages from bootlogd.c regarding the ignoring of return values of 'chdir'. Following these warnings, a series of compilation commands are executed using 'cc' (the C compiler). These commands compile various object files (bootlogd.o, last.o, mesg.o, utmpdump.o, wall.o) and source files (oldutmp.h, utmpdump.c, dowall.o) into a final executable named 'wall'. The terminal ends with a prompt character '\$' and a cursor.

```
bootlogd.c:125:8: warning: ignoring return value of 'chdir', declared
with attribute warn_unused_result [-Wunused-result]
bootlogd.c:140:10: warning: ignoring return value of 'chdir', declared
  with attribute warn_unused_result [-Wunused-result]
bootlogd.c:151:10: warning: ignoring return value of 'chdir', declared
  with attribute warn_unused_result [-Wunused-result]
bootlogd.c:156:10: warning: ignoring return value of 'chdir', declared
  with attribute warn_unused_result [-Wunused-result]
bootlogd.c:163:7: warning: ignoring return value of 'chdir', declared
with attribute warn_unused_result [-Wunused-result]
cc  bootlogd.o -lutil -o bootlogd
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o last.
o last.c
cc  last.o oldutmp.h -o last
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o mesg.
o mesg.c
cc  mesg.o -o mesg
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o utmpd
ump.o utmpdump.c
cc  utmpdump.o -o utmpdump
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o wall.
o wall.c
cc  wall.o dowall.o -o wall
$
```

Figure 5.4: make 编译源码包

```

bootlogd.c:151:10: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
bootlogd.c:156:10: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
bootlogd.c:163:7: warning: ignoring return value of 'chdir'
, declared with attribute warn_unused_result [-Wunused-result]
cc bootlogd.o -lutil -o bootlogd
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o last.o last.c
cc last.o oldutmp.h -o last
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o mesg.o mesg.c
cc mesg.o -o mesg
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o utmpdump.o utmpdump.c
cc utmpdump.o -o utmpdump
cc -ansi -O2 -fomit-frame-pointer -W -Wall -D_GNU_SOURCE -c -o wall.o wall.c
cc wall.o dowall.o -o wall
$

```

### 5.1.6 查看生成的可执行文件

```

$ ls -l | grep "x "
-rwxrwxr-x 1 akaedu akaedu 17677 Jun 22 13:28 a.out
-rwxrwxr-x 1 akaedu akaedu 18162 Jun 22 13:36 bootlogd
-rwxrwxr-x 1 akaedu akaedu 7402 Jun 22 13:27 fstab-decode
-rwxrwxr-x 1 akaedu akaedu 17625 Jun 22 13:30 halt
-rwxrwxr-x 1 akaedu akaedu 42121 Jun 22 13:30 init
-rwxr-xr-x 1 akaedu akaedu 706 Sep 10 2009 initscript.sample
-rwxrwxr-x 1 akaedu akaedu 22259 Jun 22 13:27 killall5
-rwxrwxr-x 1 akaedu akaedu 22117 Jun 22 13:36 last
-rwxrwxr-x 1 akaedu akaedu 7730 Jun 22 13:36 mesg
-rwxrwxr-x 1 akaedu akaedu 7708 Jun 22 13:30 mountpoint
-rwxrwxr-x 1 akaedu akaedu 7368 Jun 22 13:30 runlevel
-rwxrwxr-x 1 akaedu akaedu 27547 Jun 22 13:30 shutdown
-rwxrwxr-x 1 akaedu akaedu 17677 Jun 22 13:36 sulogin
-rwxrwxr-x 1 akaedu akaedu 12638 Jun 22 13:36 utmpdump
-rwxrwxr-x 1 akaedu akaedu 13243 Jun 22 13:36 wall
$

```

```
Terminal
$ ls -l | grep "x "
-rwxrwxr-x 1 akaedu akaedu 17677 Jun 22 13:28 a.out
-rwxrwxr-x 1 akaedu akaedu 18162 Jun 22 13:36 bootlogd
-rwxrwxr-x 1 akaedu akaedu 7402 Jun 22 13:27 fstab-decode
-rwxrwxr-x 1 akaedu akaedu 17625 Jun 22 13:30 halt
-rwxrwxr-x 1 akaedu akaedu 42121 Jun 22 13:30 init
-rwxr-xr-x 1 akaedu akaedu 706 Sep 10 2009 initscript.sample
-rwxrwxr-x 1 akaedu akaedu 22259 Jun 22 13:27 killall5
-rwxrwxr-x 1 akaedu akaedu 22117 Jun 22 13:36 last
-rwxrwxr-x 1 akaedu akaedu 7730 Jun 22 13:36 mesg
-rwxrwxr-x 1 akaedu akaedu 7708 Jun 22 13:30 mountpoint
-rwxrwxr-x 1 akaedu akaedu 7368 Jun 22 13:30 runlevel
-rwxrwxr-x 1 akaedu akaedu 27547 Jun 22 13:30 shutdown
-rwxrwxr-x 1 akaedu akaedu 17677 Jun 22 13:36 sulogin
-rwxrwxr-x 1 akaedu akaedu 12638 Jun 22 13:36 utmpdump
-rwxrwxr-x 1 akaedu akaedu 13243 Jun 22 13:36 wall
$
```

Figure 5.5: 查看可执行文件

## 5.2 Linux 内核启动 init 进程

### 5.2.1 start\_kernel

```
545 asmlinkage void __init start_kernel(void)
546 {
547     char * command_line;
548     unsigned long mempages;
549     extern char saved_command_line[];
550 /*
551  * Interrupts are still disabled. Do necessary setups, then
552  * enable them
553  */
554     lock_kernel();
555     printk(linux_banner);
556     setup_arch(&command_line);
557     printk("Kernel command line: %s\n", saved_command_line);
558     parse_options(command_line);
559     trap_init();
560     init_IRQ();
561     sched_init();
562     softirq_init();
563     time_init();
564
565     .....
622     /*
623      *      We count on the initial thread going ok
624      *      Like idlers init is an unlocked kernel thread, which will
625      *      make syscalls (and thus be locked).
626      */
627     smp_init();
628     rest_init();
629 }
630
```

### 5.2.2 parse\_options

```
426 static void __init parse_options(char *line)
427 {
428     char *next,*quote;
```

```
Terminal
545 asmlinkage void __init start_kernel(void)
546 {
547     char * command_line;
548     unsigned long mempages;
549     extern char saved_command_line[];
550 /*
551  * Interrupts are still disabled. Do necessary setups, then
552  * enable them
553  */
554     lock_kernel();
555     printk(linux_banner);
556     setup_arch(&command_line);
557     printk("Kernel command line: %s\n", saved_command_line);
558     parse_options(command_line);
559     trap_init();
560     init_IRQ();
561     sched_init();
562     softirq_init();
563     time_init();
564
565     /*
566      * HACK ALERT! This is early. We're enabling the console before
567      * we've done PCI setups etc, and console_init() must be aware of
567,17-24 66%
```

Figure 5.6: 内核 start\_kernel 函数



```

429         int args, envs;
430
431         if (!*line)
432             return;
433         args = 0;
434         envs = 1;          /* TERM is set to 'linux' by default */
435         next = line;
436         while ((line = next) != NULL) {
437             quote = strchr(line, '"');
438             next = strchr(line, ' ');
439             while (next != NULL && quote != NULL && quote < next) {
440                 /* we found a left quote before the next blank
441                  * now we have to find the matching right quote
442                  */
443                 next = strchr(quote+1, '"');
444                 if (next != NULL) {
445                     quote = strchr(next+1, '"');
446                     next = strchr(next+1, ' ');
447                 }
448             }
449             if (next != NULL)
450                 *next++ = 0;
451             if (!strncmp(line, "init=", 5)) {
452                 line += 5;
453                 execute_command = line;
454             } /* In case LIL0 is going to boot us with default com

```

### 5.2.3 rest\_init

```

532
533 static void rest_init(void)
534 {
535     kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
536     unlock_kernel();
537     current->need_resched = 1;
538     cpu_idle();
539 }
540

```

```
Terminal
425 */
426 static void __init parse_options(char *line)
427 {
428     char *next,*quote;
429     int args, envs;
430
431     if (!*line)
432         return;
433     args = 0;
434     envs = 1; /* TERM is set to 'linux' by default */
435     next = line;
436     while ((line = next) != NULL) {
437         quote = strchr(line, '"');
438         next = strchr(line, ' ');
439         while (next != NULL && quote != NULL && quote < next) {
440             /* we found a left quote before the next blank
441              * now we have to find the matching right quote
442              */
443             next = strchr(quote+1, '"');
444             if (next != NULL) {
445                 quote = strchr(next+1, '"');
446                 next = strchr(next+1, ' ');
447             }
448         }
449         if (next != NULL)
450             *next++ = 0;
451         if (!strncmp(line,"init=",5)) {
452             line += 5;
453             execute_command = line;
454             /* In case LILO is going to boot us with default com
mand line,
@
425,1 52%
```

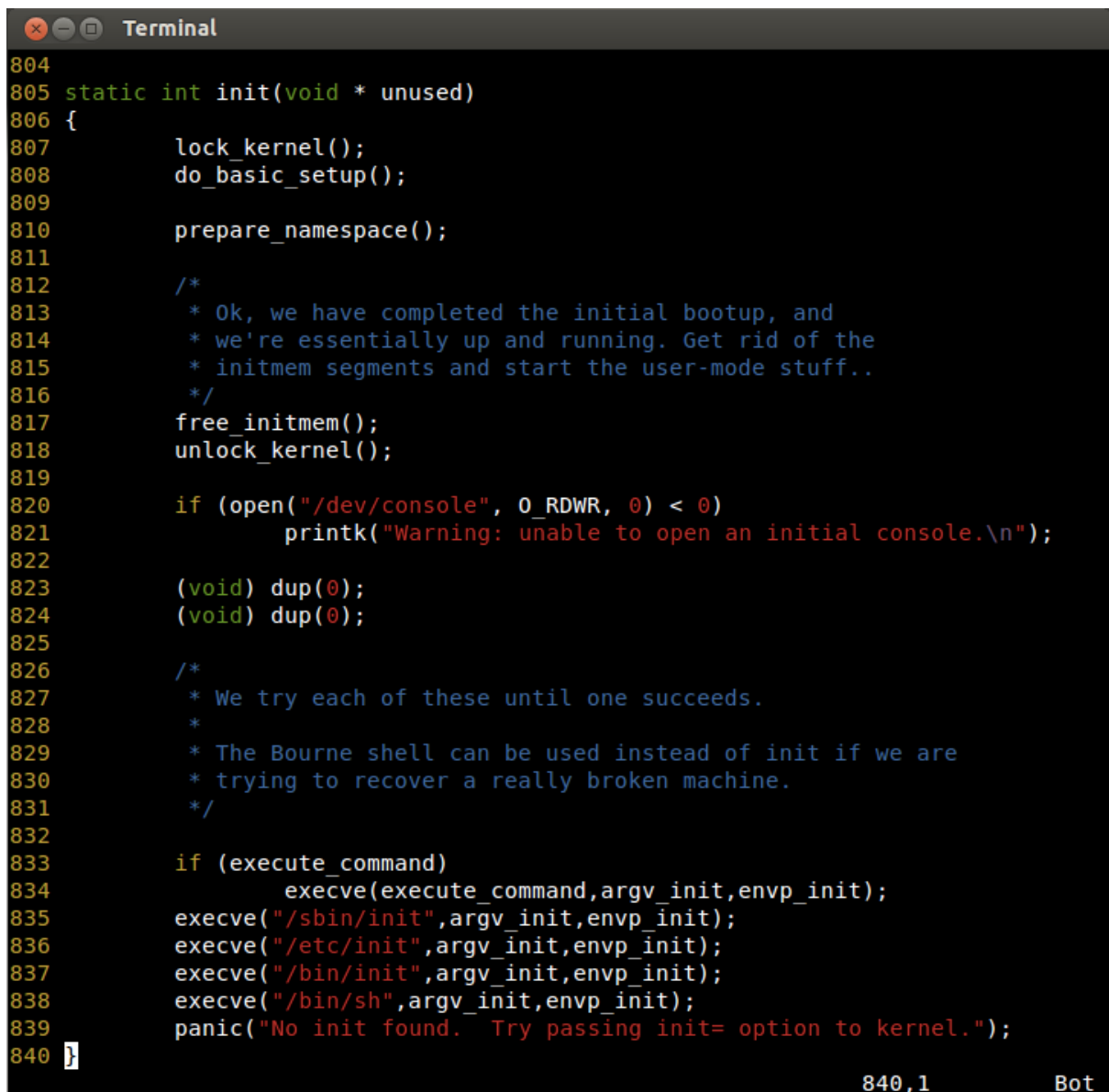
Figure 5.7: 内核 parse\_options 函数

## 5.2.4 init 函数

```
805 static int init(void * unused)
806 {
807     lock_kernel();
808     do_basic_setup();
809
810     prepare_namespace();
811
812     /*
813      * Ok, we have completed the initial bootup, and
814      * we're essentially up and running. Get rid of the
815      * initmem segments and start the user-mode stuff..
816      */
817     free_initmem();
818     unlock_kernel();
819
820     if (open("/dev/console", O_RDWR, 0) < 0)
821         printk("Warning: unable to open an initial console.\n");
822
823     (void) dup(0);
824     (void) dup(0);
825
826     /*
827      * We try each of these until one succeeds.
828      *
829      * The Bourne shell can be used instead of init if we are
830      * trying to recover a really broken machine.
831      */
832
833     if (execute_command)
834         execve(execute_command, argv_init, envp_init);
835     execve("/sbin/init", argv_init, envp_init);
836     execve("/etc/init", argv_init, envp_init);
837     execve("/bin/init", argv_init, envp_init);
838     execve("/bin/sh", argv_init, envp_init);
839     panic("No init found. Try passing init= option to kernel.");
840 }
```

至此我们找到了一条路径，使得内核从 `start_kernel` 的主函数，进入到 `init` 进程。这里涉及到了 4 个重要的函数和 1 个重要的变量，这些都是和 `init` 进程如何启动直接相关的，对于我们了解在 `init` 进程启动之前的逻辑流程有重要作用。

- `start_kernel()`

A terminal window titled "Terminal" with a dark background and light-colored text. It displays the source code for the kernel's `init` function, with line numbers 804 through 840 on the left margin. The code includes comments in blue and function calls in green. At the bottom right of the terminal, the text "840,1" and "Bot" are visible.

```
804
805 static int init(void * unused)
806 {
807     lock_kernel();
808     do_basic_setup();
809
810     prepare_namespace();
811
812     /*
813     * Ok, we have completed the initial bootup, and
814     * we're essentially up and running. Get rid of the
815     * initmem segments and start the user-mode stuff..
816     */
817     free_initmem();
818     unlock_kernel();
819
820     if (open("/dev/console", O_RDWR, 0) < 0)
821         printk("Warning: unable to open an initial console.\n");
822
823     (void) dup(0);
824     (void) dup(0);
825
826     /*
827     * We try each of these until one succeeds.
828     *
829     * The Bourne shell can be used instead of init if we are
830     * trying to recover a really broken machine.
831     */
832
833     if (execute_command)
834         execve(execute_command, argv_init, envp_init);
835     execve("/sbin/init", argv_init, envp_init);
836     execve("/etc/init", argv_init, envp_init);
837     execve("/bin/init", argv_init, envp_init);
838     execve("/bin/sh", argv_init, envp_init);
839     panic("No init found. Try passing init= option to kernel.");
840 }
```

Figure 5.8: 内核 `init` 函数

- `parse_options()`
- `rest_init()`
- `init()`
- `execute_command`

我们用下面这张图来表示这些函数和变量之间的关系，可以更直观的看到内核启动 `init` 流程。

## `/sbin/init`

`init` 命令的大致工作流程如下：

首先，由于 `init` 本身具有两面性（既是 `init`，又是 `telinit`），因此 `init` 通过检查自己的进程号来判断自己是 `init` 还是 `telinit`；真正的 `init` 的进程号（`pid`）永远都是 1。此外，用户还可通过参数 `-i`，或者 `---init` 明确指定强制执行 `init`（源码中有相关处理，但是 `man page` 没有给出说明）。

如果 `init` 发现要执行的是 `telinit`，它会调用 `telinit()` 函数：`if (!isinit) exit(telinit(p, argc, argv));` `telinit()` 函数的原型如下：

```
int telinit(char *progrname, int argc, char **argv);
```

实际调用 `telinit()` 函数时，是将用户的输入参数列表完全传递给 `telinit()` 函数的。在执行 `telinit` 时，实际上是通过向 `INIT_FIFO (/dev/initctl)` 写入命令的方式，通知 `init` 执行相应的操作。`Telinit()` 根据不同请求，构造如下结构体类型的变量并向 `INIT_FIFO (/dev/initctl)` 写入该请求来完成其使命：

```
struct init_request { int magic; /* Magic number / int cmd; / What kind of
request / int runlevel; / Runlevel to change to / int sleeptime; / Time
between TERM and KILL */ union { struct init_request_bsd bsd; char data[368]; } i; };
```

如果执行的是真正的 `init`，则又分为两种情形：

对 `init` 的重新执行（`re-exec`）标准 `init` 的执行（首次执行）

在从判断是否 `telinit()` 之后的第一步就是检查是否是对 `init` 的重新执行（`re-exec`）（通过读取 `STATE_PIPE`，看是否收到一个 `Signature = ``12567362``` 的字符串来确定）。如果是 `re-exec`，则继续从 `STATE_PIPE` 读取完整的 `state` 信息（这些信息被保存在 `CHILD` 类型的链表 `family` 上），然后调用 `init_main()` 来重新执行 `init`（注意，这里没有对 `/etc/inittab` 进行解析，这也就是 `re-exec` 的特点）。下面在对标准 `init` 的执行过程的描述中会谈到如何发起对 `init` 的重新执行。

如果不是对 `init` 的重新执行（`re-exec`），则是标准 `init` 的执行（首次执行）。首先，会通过检查命令参数，设置 `dfll_level`，`emerg_shell` 变量，如果参数有 `-a, auto` 的话，还会设置环境变量 `AUTOBOOT=YES`。

如果 `sysvinit` 编译时使能了 `SELINUX`，即定义了 `WITH_SELINUX`，则首先检查 `SELINUX_INIT` 是否被设置。如果 `SELINUX_INIT` 未被设置，则装

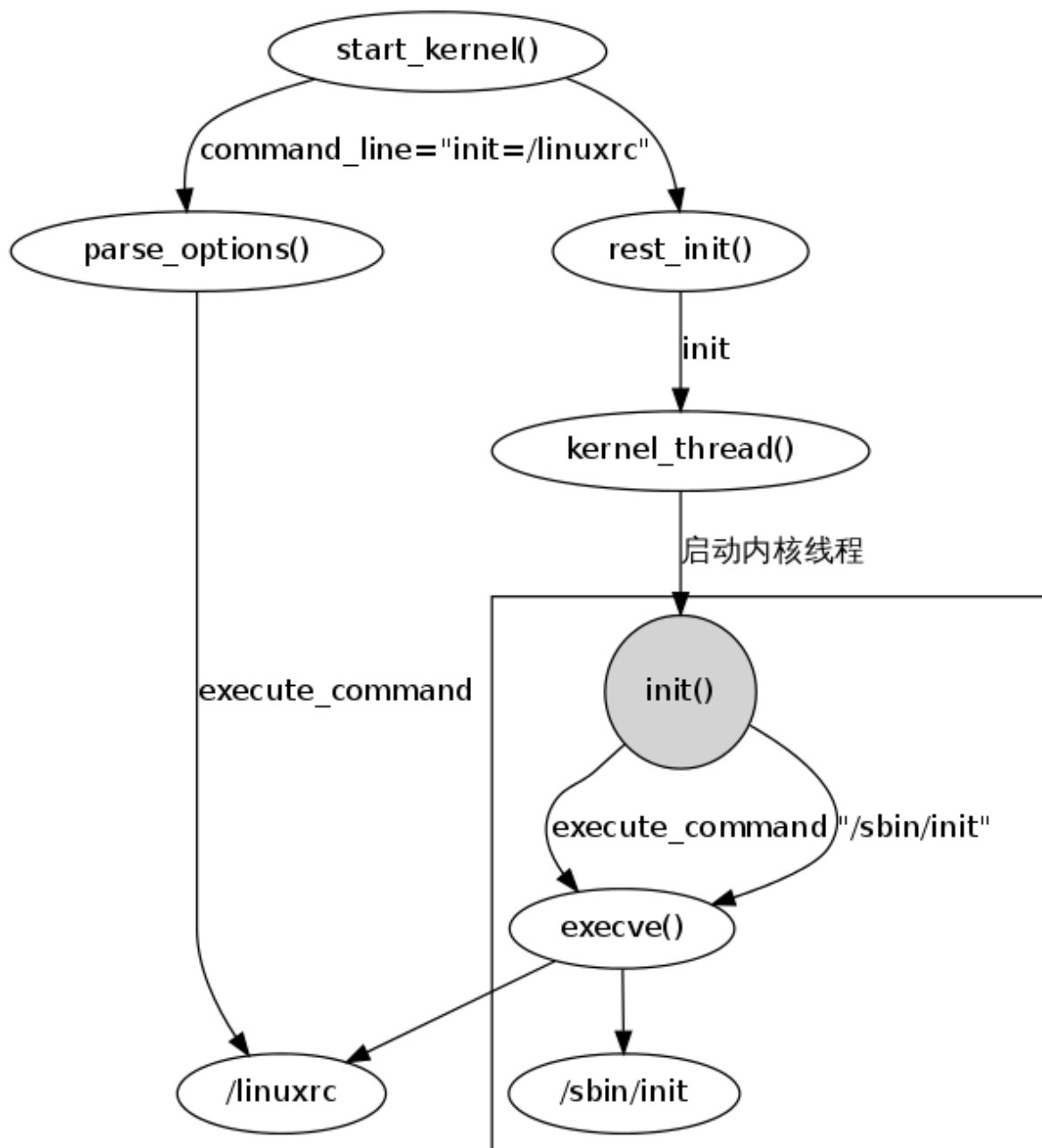


Figure 5.9: Linux 内核启动 init 进程

载/proc 文件系统（实际上这是为了确保/proc 文件系统已经被装载上）。之后用 `is_selinux_enabled()` 判断是否系统真的使能了 SELINUX。如果是的话，则卸载掉/proc 文件系统，然后再调用 `selinux_init_load_policy()` 加载策略，并在成功时调用 `execv()` 再执行 `init`；否则若 SELINUX 处于强制模式，则输出警告消息 ``Unable to load SELinux Policy...`` 并退出。此处似乎有一个问题，参加下面的链接：

<http://us.generation-nt.com/answer/bug-580272-sysvinit-2-88-selinux-policy-help-198006521.html>

在进行前面的一系列检测之后，最终开始调用 `init_main()` 进入标准的 `init` 主函数。下面对该函数做初步分析。

首先，会通过调用 `reboot(RB_DISABLE_CAD)` 禁止标准的 CTRL-ALT-DEL 组合键的响应，从而当按下这个组合键时，会发送 SIGINT 给 `init` 进程，让 `init` 来进一步决定采取何种动作（负责该组合键会导致系统直接重启）。

接着，安装一些默认的信号处理函数，包括：

`signal_handler()`，处理 SIGALRM, SIGHUP, SIGINT, SIGPWR, SIGWINCH, SIGUSR1 `chld_handler()`，处理 SIGCHLD `stop_handler()`，处理 SIGSTOP, SIGTSTP `cont_handler()`，处理 SIGCONT `segv_handler()`，处理 SIGSEGV

再之后，考虑首次运行 `init` 的情形 (`reload=0`)，`init_main()` 会初始化终端，并对终端进行一些默认的设置（在 `console_stty()` 函数中通过 `tcsetattr()` 实现），设置有一些快捷键，例如：`ctrl+d` 退出登陆，等效于 `logout` 命令 `ctrl+c` 杀死应用程序 `ctrl+s` 暂停应用程序运行，可用 `ctrl+q` 恢复运行 `ctrl+z` 挂起应用程序，此时 `ps` 显示进程状态变为 T

紧接着，`init_main()` 设置 PATH 环境变量，并初始化 `/var/run/utmp`。如果 `emerg_shell` 被设置（参数中有 `-b` 或者 `emergency`），表示需要启动 Emergency shell，则通过调用 `spawn()` 初始化 Emergency shell 子进程，并等待该子进程退出。

当从 Emergency shell 退出（或者不需要 Emergency shell 的话），`init_main()` 会调用 `read_inittab()` 来读入 `/etc/inittab` 文件。该函数主要将 `/etc/inittab` 文件解析的结果存入 CHILD 类型的链表 `family` 上，供之后的执行使用。

紧接着，调用 `start_if_needed()`，启动需要在相应运行级别中运行的程序和服务。而该函数主要又是通过调用 `startup()` 函数，继而调用 `spawn()` 来启动程序或者服务的运行的。

在此之后，`init_main()` 进入其主循环，该循环大致如下：

```
while(1) { /* See
if we need to make the boot transitions. / boot_transitions();
/ Check if there are processes to be waited on. / for(ch =
family; ch; ch = ch->next) if ((ch->flags & RUNNING) && ch-
>action != BOOT) break; if (ch != NULL && got_signals == 0)
check_init_fifo(); / Check the 'failing' flags / fail_check(); /
Process any signals. / process_signals(); / See what we need to
start up (again) */ start_if_needed(); }
```

该主循环的大致功能是，先判断是否有需要切换运行级别，然后等待需要被等待退出的进程退出；并检测是否有任何失败情形并发出警告；之后处理接收到的信号（检查 `got_signals`）；然后再看有没有需要被启动的程序或者服务。

下面是对上述循环中的一些需要注意的特殊点的描述。

1. 对于首次运行，上述代码中会调用 `get_init_default()`，解析 `/etc/inittab` 文件查找是否有 `initdefault` 记录。 `initdefault` 记录决定系统初始运行级别。如果没有这条记录，就调用 `ask_runlevel()`，让用户在系统控制台输入想要进入的运行级别。此后， `init` 会解析 `/etc/inittab` 文件中的各个条目并执行相应操作。
2. 在正常运行期间，也会对 `/etc/inittab` 文件重新扫描，当发现 `runlevel` 为 ‘U’ 时，便会调用 `re_exec()`；而该函数实际上会创建 `STATE_PIPE`，并向 `STATE_PIPE` 写入 `Signature = "12567362"`，接着 `fork()` 出一个子进程，通过子进程向 `STATE_PIPE` 写入父进程（当前 `init` 进程）的状态信息；接着，父进程调用 `execle()` 重新执行 `init` 程序，并且传递参数 “--init”，也就是强制 `init` 重新执行。而这个重新执行的 `init` 进程，就会进入前面的 `re-exec` 一段代码（见前面的分析），从而无需做初始化就能调用 `init_main()`。
3. 运行级别 `S` 或 `s` 把系统带入单用户模式，此模式不需要 `/etc/inittab` 文件。单用户模式中， `/sbin/sulogin` 会在 `/dev/console` 这个设备上打开。
4. 当第一次进入多用户模式时， `init` 会执行 `boot` 和 `bootwait` 记录以便在用户可以登录之前挂载文件系统。然后再执行相应指定的各进程。
5. 当调用 `spawn()` 启动新进程时， `init` 会检查是否存在 `/etc/initscript` 文件。如果存在该文件，则使用该脚本来启动该进程。
6. 如果系统中存在文件 `/var/run/utmp` 和 `/var/log/wtmp`，那么当每个子进程终止时， `init` 会将终止信息和原因记录进这两个文件中。
7. 当 `init` 启动了所有指定的子进程后，它会不断地监测系统进程情况，例如：某个子进程被终止、电源失效、或由 `telinit` 发出的改变运行级别的信号。当它接收到以上的这些信号时，会自动重新扫描 `/etc/inittab` 文件，并执行相应操作。因此，新的记录可以随时加入到 `/etc/inittab` 文件中。在更新了各种系统文件后，如果希望及时更新，就可以使用 `telinit Q` 或 `q` 命令来唤醒 `init` 让它即刻重新检测 `/etc/inittab` 文件。
8. 当 `init` 得到更新运行级别的请求， `init` 会向所有没有在新运行级别中定义的进程发送一个警告信号 `SIGTERM`。在等待 5 秒钟之后，它会发出的信号 `SIGKILL`（强制中断所有进程的运行）。 `init` 假设所有的这些进程（包括它们的后代）都仍然在 `init` 最初创建它们的同一进程组里。如果有进程改变了自己的进程组，那么它就收不到这些信号。这样的进程，就需要分别进行手工终止。



```

<mydebug> begin to call init_main
<mydebug> init_main()
<mydebug> console init ok
<mydebug> reload = 0
<mydebug> 0
<mydebug> 1
<mydebug> 2
<mydebug> reload = 0
<mydebug> reload = 0
<mydebug> log buf = version 2.88 booting
<mydebug> begin to read_inittab()
<mydebug> buf = id:1:initdefault:

<mydebug> id = id
<mydebug> rlevel = 1
<mydebug> action = initdefault
<mydebug> process =
<mydebug> ch->id = id
<mydebug> ch->process =
<mydebug> buf =

<mydebug> buf = rc::bootwait:/bin/date

<mydebug> id = rc
<mydebug> rlevel =
<mydebug> action = bootwait
<mydebug> process = /bin/date
<mydebug> ch->id = rc
<mydebug> ch->process = /bin/date
<mydebug> buf =

<mydebug> buf = 1:1:respawn:/etc/getty 9600 tty1

<mydebug> id = 1
<mydebug> rlevel = 1
<mydebug> action = respawn
<mydebug> process = /etc/getty 9600 tty1
<mydebug> ch->id = 1
<mydebug> ch->process = /etc/getty 9600 tty1
<mydebug> buf =

<mydebug> buf = 2:1:respawn:/etc/getty 9600 tty2

<mydebug> id = 2

```

```
<mydebug> rlevel = 1
<mydebug> action = respawn
<mydebug> process = /etc/getty 9600 tty2
<mydebug> ch->id = 2
<mydebug> ch->process = /etc/getty 9600 tty2
<mydebug> buf =

<mydebug> buf = 3:1:respawn:/etc/getty 9600 tty3

<mydebug> id = 3
<mydebug> rlevel = 1
<mydebug> action = respawn
<mydebug> process = /etc/getty 9600 tty3
<mydebug> ch->id = 3
<mydebug> ch->process = /etc/getty 9600 tty3
<mydebug> buf =

<mydebug> buf = 4:1:respawn:/etc/getty 9600 tty4

<mydebug> id = 4
<mydebug> rlevel = 1
<mydebug> action = respawn
<mydebug> process = /etc/getty 9600 tty4
<mydebug> ch->id = 4
<mydebug> ch->process = /etc/getty 9600 tty4
<mydebug> buf = ~:S:wait:/sbin/sulogin
```