init 进程探悉

前言 2

INIT配置文件分析 4

INIT的官方资料 9

INIT命令的手册 9

配置文件/ETC/INITTAB的手册 15

INIT详细分析 21

INIT PROCESS是怎么被启动的？ 21

INIT进程分析 26

init 1 的运行 27

主流程分析 27

辅助函数介绍 54

init 2 的运行 68

init 3 的运行 74

主流程分析 74

辅助函数 82

后记 98

联系 99

附录 100

环境 100

INITTAB中ACTION的注解 100

关机分析 101

关机流程介绍 101

Shutdown源码 106

前言

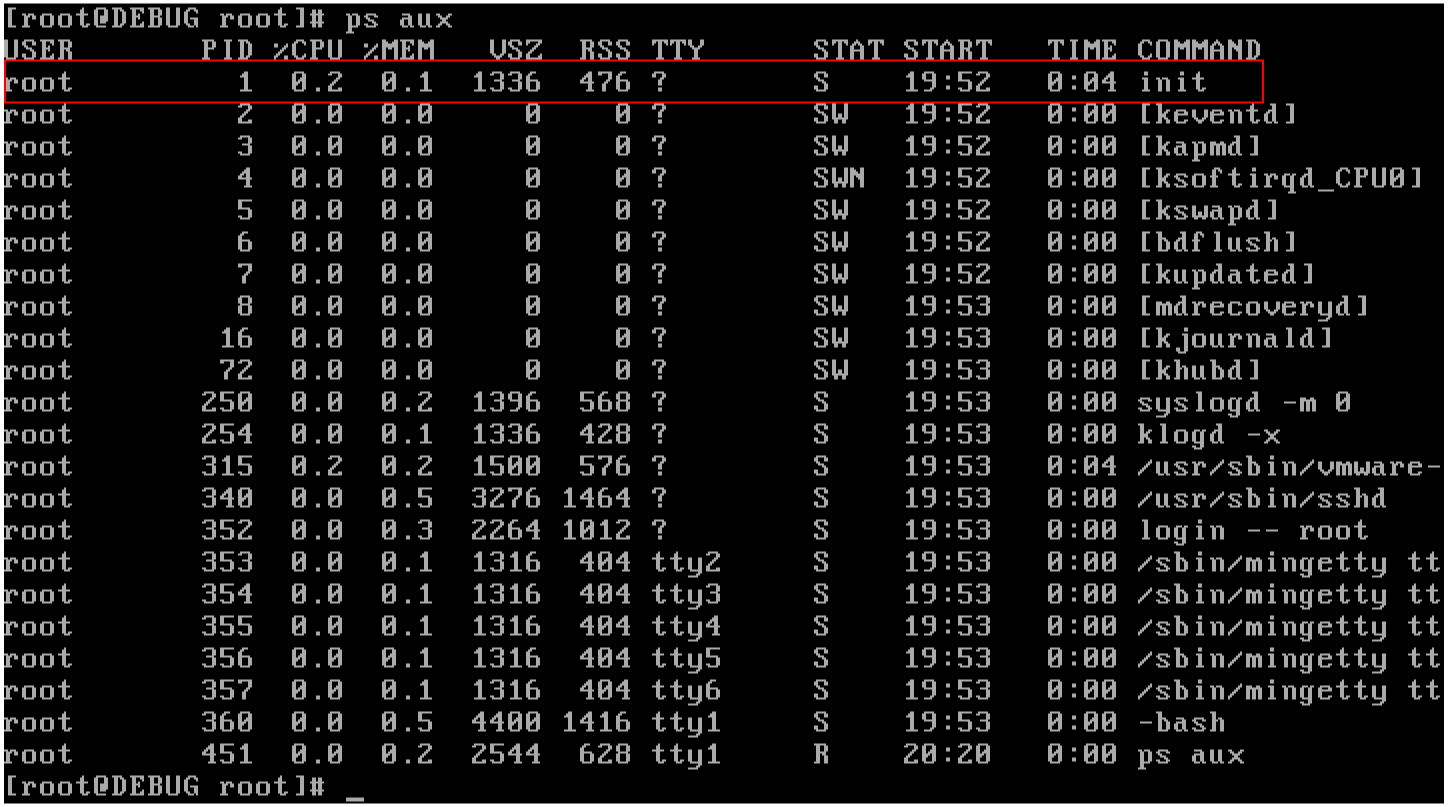
init是个普通的用户态进程，它是Unix系统内核初始化与用户态初始化的接合点，它是所有process的祖宗。在运行init以前是内核态初始化，该过程（内核初始化）的最后一个动作就是运行**/sbin/init**可执行文件。从init process运行开始进入Unix系统的用户态初始化。我对整个系统初始化的定义是从开机到屏幕上出现登录界面为止。这整个过程被init一分为二。当然init不单单启动了用户态的初始化，而且它在系统运行的整个期间都扮演着非常重要的角色。比如

- 在运行当中，具有 root 权限的用户可以通过再次运行 init 来切换到不同的运行级别（run level）
- init process 有认领系统中的所有孤儿进程的责任
- 当 root 权限用户想通过按 Ctrl-Alt-Del 三键来重启系统，也是由 init 最终来处理的
- 如果你想要一个 daemon 进程有这样的效果，它在整个系统运行期间一直要运行，即使它由于各种各样的原因（如在某种情况下它出错而退出了，或被某个用户 kill 掉了）停止运行了，也希望能马上被再次启动（当然不是依靠人力来手工启动），你可以在 init 运行的配置文件中加入类是与下面的一行：

  ```
  myrun::ondemand:/home/wzhou/mydaemon
  ```

  则/home/wzhou/mydaemon 这个脚本只要系统在运行，它必然也在运行。即使有人把它 kill 掉，等一会儿马上又会被 init process 启动
- 等等

而这一切都依赖于 init process。



```
[root@DEBUG root]# ps aux
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.1  1336  476 ?        S    19:52   0:04 init
root         2  0.0  0.0     0    0 ?        SW   19:52   0:00 [keventd]
root         3  0.0  0.0     0    0 ?        SW   19:52   0:00 [kapmd]
root         4  0.0  0.0     0    0 ?        SWN  19:52   0:00 [ksoftirqd_CPU0]
root         5  0.0  0.0     0    0 ?        SW   19:52   0:00 [kswapd]
root         6  0.0  0.0     0    0 ?        SW   19:52   0:00 [bdflush]
root         7  0.0  0.0     0    0 ?        SW   19:52   0:00 [kupdated]
root         8  0.0  0.0     0    0 ?        SW   19:53   0:00 [mdrecoveryd]
root        16  0.0  0.0     0    0 ?        SW   19:53   0:00 [kjournald]
root        72  0.0  0.0     0    0 ?        SW   19:53   0:00 [khubd]
root       250  0.0  0.2  1396  568 ?        S    19:53   0:00 syslogd -m 0
root       254  0.0  0.1  1336  428 ?        S    19:53   0:00 klogd -x
root       315  0.2  0.2  1500  576 ?        S    19:53   0:04 /usr/sbin/vmware-
root       340  0.0  0.5  3276 1464 ?        S    19:53   0:00 /usr/sbin/sshd
root       352  0.0  0.3  2264 1012 ?        S    19:53   0:00 login -- root
root       353  0.0  0.1  1316  404 tty2     S    19:53   0:00 /sbin/mingetty tt
root       354  0.0  0.1  1316  404 tty3     S    19:53   0:00 /sbin/mingetty tt
root       355  0.0  0.1  1316  404 tty4     S    19:53   0:00 /sbin/mingetty tt
root       356  0.0  0.1  1316  404 tty5     S    19:53   0:00 /sbin/mingetty tt
root       357  0.0  0.1  1316  404 tty6     S    19:53   0:00 /sbin/mingetty tt
root       360  0.0  0.5  4400 1416 tty1     S    19:53   0:00 -bash
root       451  0.0  0.2  2544  628 tty1     R    20:20   0:00 ps aux
[root@DEBUG root]# _
```

init 配置文件分析

init process的运行完全受其配置文件**/etc/inittab**的控制，这里分析一下该配置文件。

来个现实系统中的/etc/inittab 配置文件来解释一下。

```
[wzhou@dcmp10 ~]$ cat /etc/inittab
#
# inittab       This file describes how the INIT process should set up
#               the system in a certain run-level.
#
# Author:       Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#               Modified for RHS Linux by Marc Ewing and Donnie Barnes
#

# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
```

```
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"


# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

```
id:5:initdefault:
```

这一行表示系统启动后将运行在 run level 5，即 X Window 的 Full multiuser mode

```
si::sysinit:/etc/rc.d/rc.sysinit
```

sysinit 表示这是用户态系统启动，不管任何运行级别(run level)都要执行脚本/etc/rc.d/rc.sysinit
如果你要追踪操作系统内核态的初始化过程，则要从 init/main.c 中的 start_kernel()开始；而如果你想追踪操作系统用户态的启动过程，则可以从/etc/rc.d/rc.sysinit 脚本开始。

```
l0:0:wait:/etc/rc.d/rc 0      如果系统的 run level 是 0，则运行/etc/rc.d/rc 脚本，参数为 0
l1:1:wait:/etc/rc.d/rc 1      如果系统的 run level 是 1，则运行/etc/rc.d/rc 脚本，参数为 1
l2:2:wait:/etc/rc.d/rc 2      如果系统的 run level 是 2，则运行/etc/rc.d/rc 脚本，参数为 2
l3:3:wait:/etc/rc.d/rc 3      如果系统的 run level 是 3，则运行/etc/rc.d/rc 脚本，参数为 3
l4:4:wait:/etc/rc.d/rc 4      如果系统的 run level 是 4，则运行/etc/rc.d/rc 脚本，参数为 4
l5:5:wait:/etc/rc.d/rc 5      如果系统的 run level 是 5，则运行/etc/rc.d/rc 脚本，参数为 5
l6:6:wait:/etc/rc.d/rc 6      如果系统的 run level 是 6，则运行/etc/rc.d/rc 脚本，参数为 6
```

显然/etc/rc.d/rc 也是个系统初始化的很重要的脚本。上面的 wait action 表示 init process 在启动其他的动作以前，必须等待该行上的动作所代表的 process 的完成。

```
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
```

这一行表示无论在什么 run level，如果 root 用户按了 Ctrl+Alt+Del 三键则运行如下命令：
/sbin/shutdown -t3 -r now
即让 init 进程监视 Ctrl+Alt+Del 事件，一旦收到，它应当运行该命令。shutdown 命令会从现在(now)开始先向系统中的所有进程发 warning，然后等待 3 秒，再杀死进程，让系统重启。

```
# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
```

从上面的注释可以知道该行的动作。同样该行是不分 run level 的，只管是否发生“powerfail”的事件。

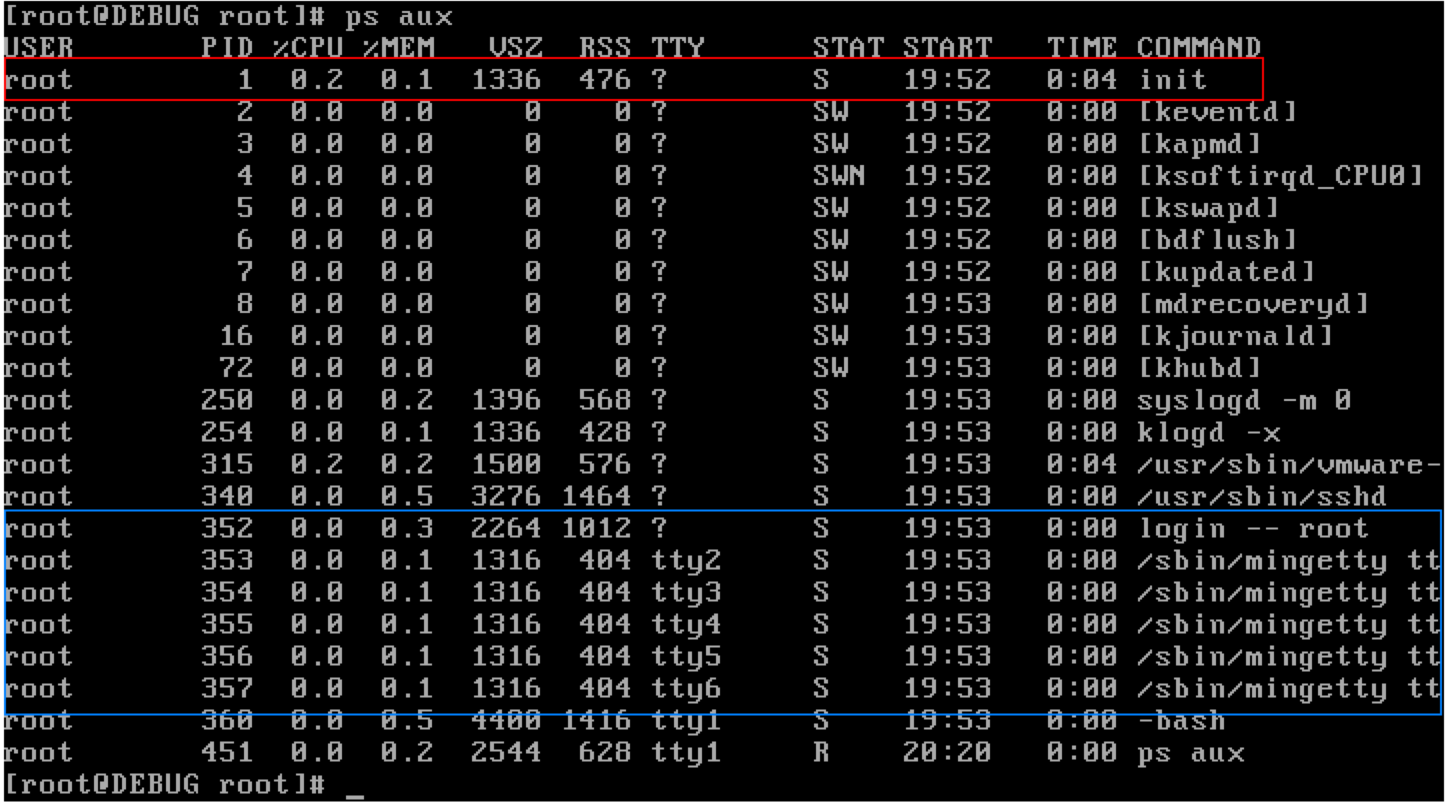
```
# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
```

在运行级别为 1,2,3,4,5 的情况下，如果发生“powerokwait” action，则运行命令/sbin/shutdown -c "Power Restored; Shutdown Cancelled"，即取消发出的关机指令。

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
```

```
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

上面的 6 行指示 init process 在 run level 是 2，3，4，5 的情况下，运行脚本/sbin/mingetty，并接受不同的参数。这里的功能是在从 tty1 到 tty6 的终端上启动字符登陆界面。



```
[root@DEBUG root]# ps aux
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.1  1336  476 ?        S    19:52   0:04 init
root         2  0.0  0.0     0    0 ?        SW   19:52   0:00 [keventd]
root         3  0.0  0.0     0    0 ?        SW   19:52   0:00 [kapmd]
root         4  0.0  0.0     0    0 ?        SWN  19:52   0:00 [ksoftirqd_CPU0]
root         5  0.0  0.0     0    0 ?        SW   19:52   0:00 [kswapd]
root         6  0.0  0.0     0    0 ?        SW   19:52   0:00 [bdflush]
root         7  0.0  0.0     0    0 ?        SW   19:52   0:00 [kupdated]
root         8  0.0  0.0     0    0 ?        SW   19:53   0:00 [mdrecoveryd]
root        16  0.0  0.0     0    0 ?        SW   19:53   0:00 [kjournald]
root        72  0.0  0.0     0    0 ?        SW   19:53   0:00 [khubd]
root       250  0.0  0.2  1396  568 ?        S    19:53   0:00 syslogd -m 0
root       254  0.0  0.1  1336  428 ?        S    19:53   0:00 klogd -x
root       315  0.2  0.2  1500  576 ?        S    19:53   0:04 /usr/sbin/vmware-
root       340  0.0  0.5  3276 1464 ?        S    19:53   0:00 /usr/sbin/sshd
root       352  0.0  0.3  2264 1012 ?        S    19:53   0:00 login -- root
root       353  0.0  0.1  1316  404 tty2     S    19:53   0:00 /sbin/mingetty tt
root       354  0.0  0.1  1316  404 tty3     S    19:53   0:00 /sbin/mingetty tt
root       355  0.0  0.1  1316  404 tty4     S    19:53   0:00 /sbin/mingetty tt
root       356  0.0  0.1  1316  404 tty5     S    19:53   0:00 /sbin/mingetty tt
root       357  0.0  0.1  1316  404 tty6     S    19:53   0:00 /sbin/mingetty tt
root       360  0.0  0.5  4400 1416 tty1     S    19:53   0:00 -bash
root       451  0.0  0.2  2544  628 tty1     R    20:20   0:00 ps aux
[root@DEBUG root]# _
```

上图中用蓝框围起来的就是启动的 6 个虚拟终端。我用 root 帐号登录在 tty1,所以该终端显示“ login -- root ”,而其他 5 个虚拟终端并没有用户登录，所以还是由 mingetty 在等待着。

```
# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

该行表示如果run level是5，则要运行脚本/etc/X11/prefdm –nodaemon，其实就是启动X Window,进入GUI界面。

上面是对inittab配置文件的静态的解释，下面解释init process依据该配置文件动态运行情况。

- init process由“initdefault”知道系统将在run level 5下运行
- init process首先运行“sysinit”标注的action,即运行/etc/rc.d/rc.sysinit脚本
- 运行identifier为“l5”的动作

  ```
  l5:5:wait:/etc/rc.d/rc 5
  ```

  由于该行告诉init process的反映是“wait”，即在init process继续执行inittab配置文件中其他action以前，必须等待“/etc/rc.d/rc 5”的结束
- 接下来执行下面的6个action

  ```
  1:2345:respawn:/sbin/mingetty tty1
  2:2345:respawn:/sbin/mingetty tty2
  3:2345:respawn:/sbin/mingetty tty3
  4:2345:respawn:/sbin/mingetty tty4
  5:2345:respawn:/sbin/mingetty tty5
  6:2345:respawn:/sbin/mingetty tty6
  ```

  由于上面6行的run level告诉init process，在2，3，4，5之下都要执行这里的命令“/sbin/mingetty tty5”。同时这里的“respawn”表示如果/sbin/mingetty所代表的process不运行了（无论哪种情况，是自己退出或出现问题而crash）,init process都有责任让他再次运行。当启动Linux后我们通过Alt-F1,…,Alt-F6可以切换到相应的终端，就是这几行运行的缘故。另外，当你登录到某个终端，比如tty1,然后在命令行上输入exit,在该终端上又会出现登录界面，这就是init process在响应“respawn”动作。当你输入exit时，/sbin/mingetty代表的process退出，被init process监控到，马上在该终端上又运行“/sbin/mingetty tty1”，从而在退出的tty1上再次出现登录界面。
- 最后运行的是/etc/X11/prefdm –nodaemon，即启动X Window登录。
- 在配置文件中的下面的配置行并不会执行，但会被init process纪录状态。只有当系统出现对应的情况时，才会运行。

  ```
  ca::ctrlaltdel:/sbin/shutdown -t3 -r now
  pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
  pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
  ```

  比如当root用户按了Ctrl-Alt-Del键以后，init process将执行如下命令行“/sbin/shutdown -t3 -r now”进行关机；而当UPS报告电源出现故障，马上要断电时，就执行“/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"”；当UPS报告从电源故障恢复以后，执行“/sbin/shutdown -c "Power Restored; Shutdown Cancelled"”。那么init process是怎么感知这些消息的呢？即该进程怎么知道root权限用户按下了Ctrl-Alt-Del键，UPS报告电源出现故障及电源恢复呢？都是通过Unix特有的signal机制。对init process来说，它只要正确处理对应的signal就好。

init 的官方资料

init 的作者亲手写了与 init 相关的手册，即 man init 与 man inittab。仔细看看，对理解 init process 有很大帮助。

init 命令的手册

```
NAME
       init, telinit - process control initialization

SYNOPSIS
       /sbin/init [ -a ] [ -s ] [ -b ] [ -z xxx ] [ 0123456Ss ]
       /sbin/telinit [ -t sec ] [ 0123456sSQqabcUu ]

DESCRIPTION
   Init
       Init  is the parent of all processes.  Its primary role is to create processes
       from a script stored in the file /etc/inittab  (see  inittab(5)).   This  file
       usually  has  entries which cause init to spawn gettys on each line that users
       can log in.  It also controls autonomous processes required by any  particular
       system.

RUNLEVELS
       A  runlevel  is  a  software  configuration  of the system which allows only a
       selected group of processes to exist.  The processes spawned by init for  each
       of  these  runlevels are defined in the /etc/inittab file.  Init can be in one
       of eight runlevels: 0â€“6 and S or s.  The runlevel is changed by having a priv-
       ileged  user  run telinit, which sends appropriate signals to init, telling it
       which runlevel to change to.
```

Runlevels 0, 1, and 6 are reserved. Runlevel 0 is used to halt the system, runlevel 6 is used to reboot the system, and runlevel 1 is used to get the system down into single user mode. Runlevel S is not really meant to be used directly, but more for the scripts that are executed when entering runlevel 1. For more information on this, see the manpages for shutdown(8) and inittab(5).

Runlevels 7-9 are also valid, though not really documented. This is because "traditional" Unix variants donâ€™t use them. In case youâ€™re curious, runlevels S and s are in fact the same. Internally they are aliases for the same runlevel.

BOOTING

After init is invoked as the last step of the kernel boot sequence, it looks for the file /etc/inittab to see if there is an entry of the type initdefault (see inittab(5)). The initdefault entry determines the initial runlevel of the system. If there is no such entry (or no /etc/inittab at all), a runlevel must be entered at the system console.

Runlevel S or s bring the system to single user mode and do not require an /etc/inittab file. In single user mode, a root shell is opened on /dev/console.

When entering single user mode, init initializes the consoles stty settings to sane values. Clocal mode is set. Hardware speed and handshaking are not changed.

When entering a multi-user mode for the first time, init performs the boot and bootwait entries to allow file systems to be mounted before users can log in. Then all entries matching the runlevel are processed.

When starting a new process, init first checks whether the file /etc/initscript exists. If it does, it uses this script to start the process.

Each time a child terminates, init records the fact and the reason it died in /var/run/utmp and /var/log/wtmp, provided that these files exist.

CHANGING RUNLEVELS

After it has spawned all of the processes specified, init waits for one of its descendant processes to die, a powerfail signal, or until it is signaled by telinit to change the systemâ€™s runlevel. When one of the above three conditions occurs, it re-examines the /etc/inittab file. New entries can be added to this file at any time. However, init still waits for one of the above three conditions to occur. To provide for an instantaneous response, the telinit Q or q command can wake up init to re-examine the /etc/inittab file.

If init is not in single user mode and receives a powerfail signal (SIGPWR), it reads the file /etc/powerstatus. It then starts a command based on the contents of this file:

F(AIL) Power is failing, UPS is providing the power. Execute the powerwait and powerfail entries.

O(K) The power has been restored, execute the powerokwait entries.

L(OW) The power is failing and the UPS has a low battery. Execute the powerfailnow entries.

If /etc/powerstatus doesnâ€™t exist or contains anything else then the letters F, O or L, init will behave as if it has read the letter F.

Usage of SIGPWR and /etc/powerstatus is discouraged. Someone wanting to interact with init should use the /dev/initctl control channel - see the source code of the sysvinit package for more documentation about this.

When init is requested to change the runlevel, it sends the warning signal SIGTERM to all processes that are undefined in the new runlevel. It then waits 5 seconds before forcibly terminating these processes via the SIGKILL signal. Note that init assumes that all these processes (and their descendants) remain in the same process group which init originally created for them. If any process changes its process group affiliation it will not receive these signals. Such processes need to be terminated separately.

TELINIT

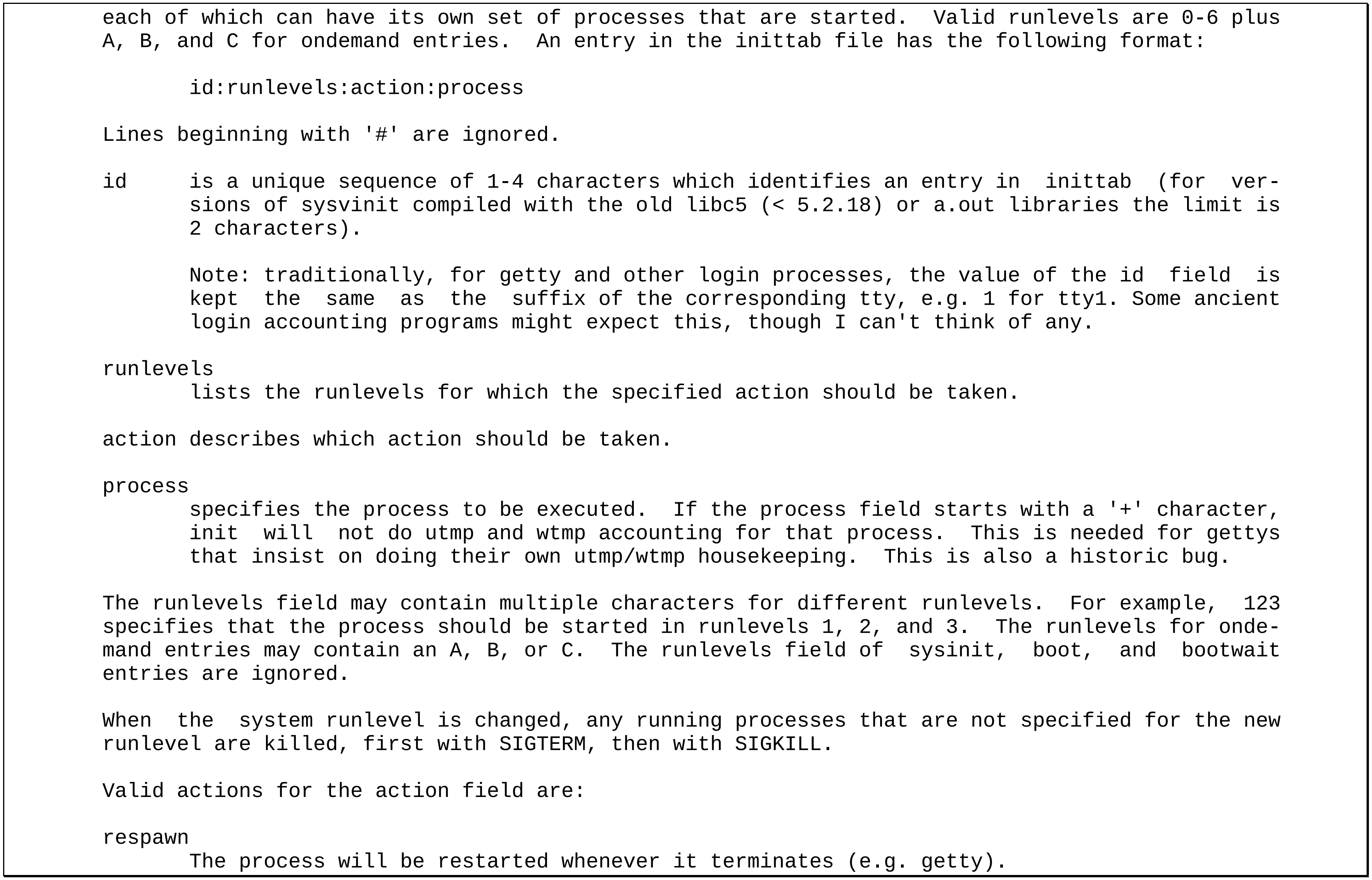
```
       /sbin/telinit is linked to /sbin/init.  It takes a one-character argument  and
       signals init to perform the appropriate action.  The following arguments serve
       as directives to telinit:

       0,1,2,3,4,5 or 6
              tell init to switch to the specified run level.

       a,b,c  tell init to process only those /etc/inittab file entries  having  run-
              level a,b or c.

       Q or q tell init to re-examine the /etc/inittab file.

       S or s tell init to switch to single user mode.

       U or u tell  init to re-execute itself (preserving the state). No re-examining
              of /etc/inittab file happens. Run level should be one of Ss12345,  oth-
              erwise request would be silently ignored.

       telinit  can  also tell init how long it should wait between sending processes
       the SIGTERM and SIGKILL signals.  The default is 5 seconds, but  this  can  be
       changed with the -t sec option.

       telinit can be invoked only by users with appropriate privileges.

       The  init binary checks if it is init or telinit by looking at its process id;
       the real initâ€™s process id is always 1.  From this it follows that instead  of
       calling telinit one can also just use init instead as a shortcut.

ENVIRONMENT
       Init sets the following environment variables for all its children:

       PATH   /usr/local/sbin:/sbin:/bin:/usr/sbin:/usr/bin

       INIT_VERSION
              As  the  name  says. Useful to determine if a script runs directly from
              init.
       RUNLEVEL
              The current system runlevel.

       PREVLEVEL
              The previous runlevel (useful after a runlevel switch).

       CONSOLE
              The system console. This is really inherited from the  kernel;  however
              if it is not set init will set it to /dev/console by default.

BOOTFLAGS
       It  is  possible  to pass a number of flags to init from the boot monitor (eg.
       LILO). Init accepts the following flags:

       -s, S, single
            Single user mode boot. In this mode  /etc/inittab  is  examined  and  the
            bootup  rc  scripts  are usually run before the single user mode shell is
            started.

       1-5  Runlevel to boot into.

       -b, emergency
            Boot directly into a single user shell without running any other  startup
            scripts.

       -a, auto
            The  LILO  boot  loader  adds  the  word "auto" to the command line if it
            booted the kernel with the default command line (without  user  interven-
            tion).  If this is found init sets the "AUTOBOOT" environment variable to
            "yes". Note that you cannot use this  for  any  security  measures  -  of
            course  the user could specify "auto" or -a on the command line manually.

       -z xxx
            The argument to -z is ignored. You can use this  to  expand  the  command
            line  a bit, so that it takes some more space on the stack. Init can then
            manipulate the command line so that ps(1) shows the current runlevel.
```

```
INTERFACE
       Init listens on a fifo in /dev, /dev/initctl, for messages.  Telinit uses this
       to  communicate  with  init. The interface is not very well documented or fin-
       ished. Those interested should study the initreq.h file in the src/  subdirec-
       tory of the init source code tar archive.

SIGNALS
       Init reacts to several signals:

       SIGHUP
            Has the same effect as telinit q.

       SIGUSR1
            On  receipt  of  this signals, init closes and re-opens its control fifo,
            /dev/initctl. Useful for bootscripts when /dev is remounted.

       SIGINT
            Normally the kernel sends  this  signal  to  init  when  CTRL-ALT-DEL  is
            pressed. It activates the ctrlaltdel action.

       SIGWINCH
            The  kernel  sends  this  signal  when the KeyboardSignal key is hit.  It
            activates the kbrequest action.

CONFORMING TO
       Init is compatible with the System V init. It works closely together with  the
       scripts  in the directories /etc/init.d and /etc/rc{runlevel}.d.  If your sys-
       tem uses this convention, there should be  a  README  file  in  the  directory
       /etc/init.d explaining how these scripts work.
FILES
       /etc/inittab
       /etc/initscript
       /dev/console
       /var/run/utmp
       /var/log/wtmp
       /dev/initctl
```

```
WARNINGS
       Init  assumes  that  processes and descendants of processes remain in the same
       process group which was originally created for them.  If the processes  change
       their  group, init canâ€™t kill them and you may end up with two processes read-
       ing from one terminal line.

DIAGNOSTICS
       If init finds that it is continuously respawning an entry more than  10  times
       in  2  minutes,  it  will assume that there is an error in the command string,
       generate an error message on the system console, and refuse  to  respawn  this
       entry  until  either 5 minutes has elapsed or it receives a signal.  This pre-
       vents it from eating up system resources when someone  makes  a  typographical
       error in the /etc/inittab file or the program for the entry is removed.

AUTHOR
       Miquel  van  Smoorenburg  (miquels@cistron.nl), initial manual page by Michael
       Haardt (u31b3hs@pool.informatik.rwth-aachen.de).

SEE ALSO
       getty(1), login(1),  sh(1),  runlevel(8),  shutdown(8),  kill(1),  inittab(5),
       initscript(5), utmp(5)

                                    18 April 2003                             INIT(8)
```

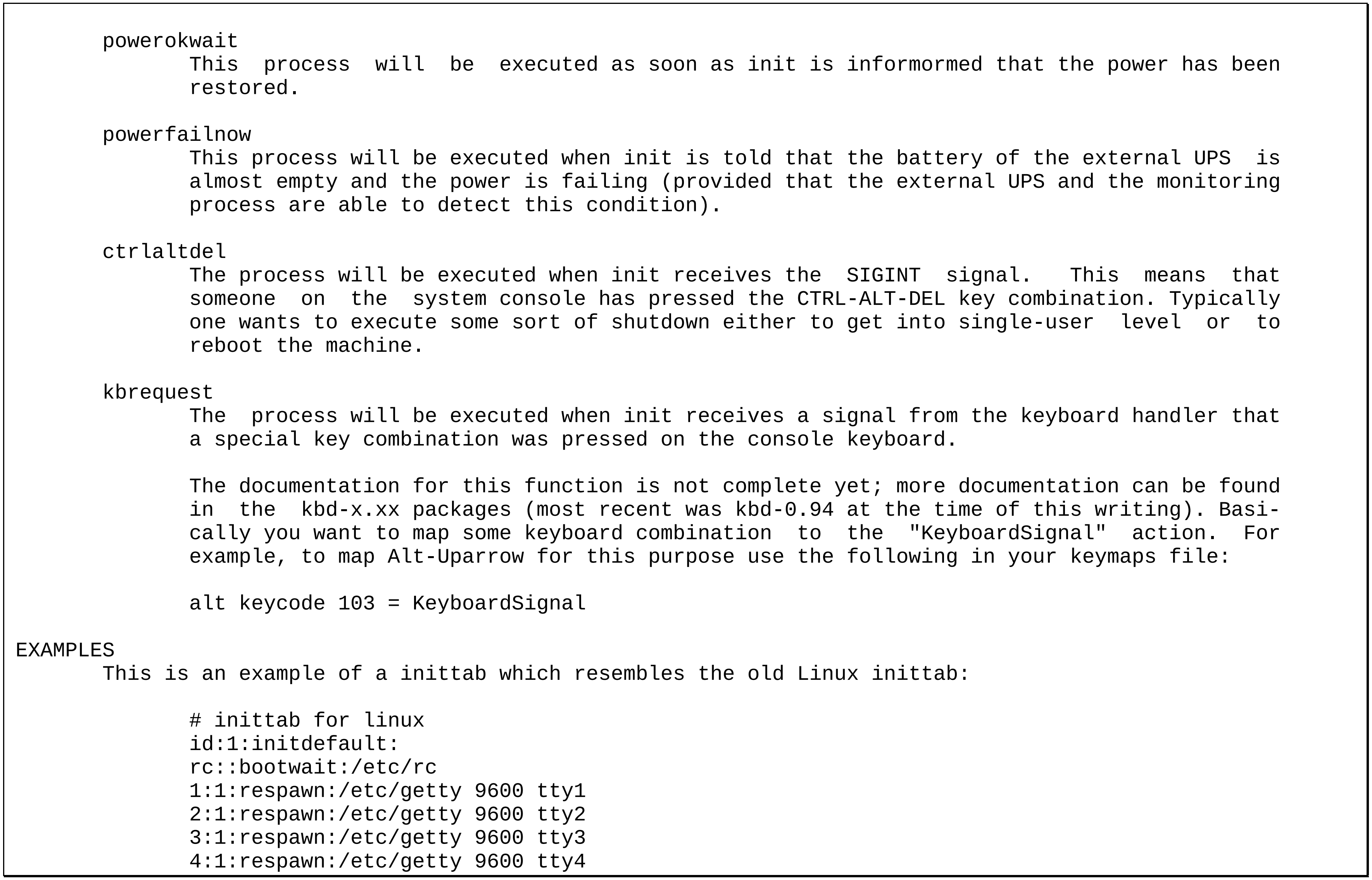
配置文件/etc/inittab 的手册

```
INITTAB(5)                        Linux System Administrator's Manual                       INITTAB(5)

NAME
       inittab - format of the inittab file used by the sysv-compatible init process

DESCRIPTION
       The  inittab  file  describes which processes are started at bootup and during normal operation
       (e.g. /etc/init.d/boot, /etc/init.d/rc, gettys...).  Init(8) distinguishes multiple  runlevels,
```



```
       each of which can have its own set of processes that are started.  Valid runlevels are 0-6 plus
       A, B, and C for ondemand entries.  An entry in the inittab file has the following format:

              id:runlevels:action:process

       Lines beginning with '#' are ignored.

       id     is a unique sequence of 1-4 characters which identifies an entry in  inittab  (for  ver-
              sions of sysvinit compiled with the old libc5 (< 5.2.18) or a.out libraries the limit is
              2 characters).

              Note: traditionally, for getty and other login processes, the value of the id  field  is
              kept  the  same  as  the  suffix of the corresponding tty, e.g. 1 for tty1. Some ancient
              login accounting programs might expect this, though I can't think of any.

       runlevels
              lists the runlevels for which the specified action should be taken.

       action describes which action should be taken.

       process
              specifies the process to be executed.  If the process field starts with a '+' character,
              init  will  not do utmp and wtmp accounting for that process.  This is needed for gettys
              that insist on doing their own utmp/wtmp housekeeping.  This is also a historic bug.

       The runlevels field may contain multiple characters for different runlevels.  For example,  123
       specifies that the process should be started in runlevels 1, 2, and 3.  The runlevels for onde-
       mand entries may contain an A, B, or C.  The runlevels field of  sysinit,  boot,  and  bootwait
       entries are ignored.

       When  the  system runlevel is changed, any running processes that are not specified for the new
       runlevel are killed, first with SIGTERM, then with SIGKILL.

       Valid actions for the action field are:

       respawn
              The process will be restarted whenever it terminates (e.g. getty).
       wait   The process will be started once when the specified runlevel is entered  and  init  will
              wait for its termination.

       once   The process will be executed once when the specified runlevel is entered.

       boot   The process will be executed during system boot.  The runlevels field is ignored.

       bootwait
              The  process  will  be executed during system boot, while init waits for its termination
              (e.g. /etc/rc).  The runlevels field is ignored.

       off    This does nothing.

       ondemand[1]
              A process marked with an ondemand runlevel will be executed whenever the specified onde-
              mand runlevel is called.  However, no runlevel change will occur (ondemand runlevels are
              'a', 'b', and 'c').

       initdefault
              An initdefault entry specifies the runlevel which should be entered after  system  boot.
              If  none  exists,  init  will  ask  for  a runlevel on the console. The process field is
              ignored.

       sysinit
              The process will be executed during system boot. It will be executed before any boot  or
              bootwait entries.  The runlevels field is ignored.

       powerwait
              The  process  will  be executed when the power goes down. Init is usually informed about
              this by a process talking to a UPS connected to the computer.  Init will  wait  for  the
              process to finish before continuing.

       powerfail
              As for powerwait, except that init does not wait for the process's completion.
```

[1] Ondemand与respawn的区别是其与运行级别（run level）无关



```
       powerokwait
              This  process  will  be  executed as soon as init is informormed that the power has been
              restored.

       powerfailnow
              This process will be executed when init is told that the battery of the external UPS  is
              almost empty and the power is failing (provided that the external UPS and the monitoring
              process are able to detect this condition).

       ctrlaltdel
              The process will be executed when init receives the  SIGINT  signal.   This  means  that
              someone  on  the  system console has pressed the CTRL-ALT-DEL key combination. Typically
              one wants to execute some sort of shutdown either to get into single-user  level  or  to
              reboot the machine.

       kbrequest
              The  process will be executed when init receives a signal from the keyboard handler that
              a special key combination was pressed on the console keyboard.

              The documentation for this function is not complete yet; more documentation can be found
              in  the  kbd-x.xx packages (most recent was kbd-0.94 at the time of this writing). Basi-
              cally you want to map some keyboard combination  to  the  "KeyboardSignal"  action.  For
              example, to map Alt-Uparrow for this purpose use the following in your keymaps file:

              alt keycode 103 = KeyboardSignal

EXAMPLES
       This is an example of a inittab which resembles the old Linux inittab:

              # inittab for linux
              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
```

This inittab file executes /etc/rc during boot and starts gettys on tty1-tty4.

A more elaborate inittab with different runlevels (see the comments inside):

```
# Level to run in
id:2:initdefault:

# Boot-time system configuration/initialization script.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~:S:wait:/sbin/sulogin

# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.

l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6

# What to do at the "3 finger salute".
ca::ctrlaltdel:/sbin/shutdown -t1 -h now

# Runlevel 2,3: getty on virtual consoles
# Runlevel   3: getty on terminal (ttyS0) and modem (ttyS1)
1:23:respawn:/sbin/getty tty1 VC linux
2:23:respawn:/sbin/getty tty2 VC linux
```

```
3:23:respawn:/sbin/getty tty3 VC linux
4:23:respawn:/sbin/getty tty4 VC linux
S0:3:respawn:/sbin/getty -L 9600 ttyS0 vt320
S1:3:respawn:/sbin/mgetty -x0 -D ttyS1
```

FILES

/etc/inittab

AUTHOR

Init was written by Miquel van Smoorenburg (miquels@cistron.nl). This manual page was written by Sebastian Lederer (lederer@francium.informatik.uni-bonn.de) and modified by Michael Haardt (u31b3hs@pool.informatik.rwth-aachen.de).

SEE ALSO

init(8), telinit(8)

init 详细分析

init process 是怎么被启动的？

init process 是 Linux 系统的第一个用户态进程，那自然没有父亲。它是由 Linux 内核直接启动的。

start_kernel()是内核的汇编与C语言的交接点，在该函数以前，内核的代码都是用汇编写的，完成一些最基本的初始化与环境设置工作，比如内核代码载入内存并解压缩（现在的内核一般都经过压缩），CPU 的最基本初始化，为 C 代码的运行设置环境（C 代码的运行是有一定环境要求的，比如 stack 的设置等）。这里一个不太确切的比喻是 start_kernel()就像是 C 代码中的 main()。我们知道对应用程序员而言，main()是他的入口，但实际上程序的入口是被包在了C库中，在链接阶段，linker 会把它链接入你的程序中。而它的任务中有一项就是为 main()准备运行环境。main()中的 argc，argv 等都不是平白无故来的，都是在调用 main()以前的代码做的准备。

在 start_kernel()中 Linux 将完成整个系统的内核初始化。内核初始化的最后一步就是启动 init 进程这个所有进程的祖先。

Linux-2.6.20/init/main.c

```
asmlinkage void __init start_kernel(void)       该函数是 Linux 内核的入口，其前面的代码都是用汇编编写
{
      char * command_line;
      extern struct kernel_param __start___param[], __stop___param[];

      smp_setup_processor_id();

      /*
       * Need to run as early as possible, to initialize the
       * lockdep hash:
       */
      unwind_init();
```

```
        lockdep_init();

        local_irq_disable();
        early_boot_irqs_off();
        early_init_irq_lock_class();

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
        lock_kernel();
        boot_cpu_init();
        page_address_init();
        printk(KERN_NOTICE);
        printk(linux_banner);

        。。。。。。

        cpuset_init();
        taskstats_init_early();
        delayacct_init();

        check_bugs();

        acpi_early_init(); /* before LAPIC and SMP init */

        /* Do the rest non-__init'ed, we're now alive */
        rest_init();                  这是 Linux 内核初始化的尾声
}
```

```
static void noinline rest_init(void)
        __releases(kernel_lock)
{
        kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);  创建一个内核线程，实际上就是内核进程，
        numa_default_policy();                                Linux 内核是不支持类似 Windows NT 一样的
        unlock_kernel();                                      线程概念的。Linux 本质上只支持进程。这里的
```

init 只是一个函数，不要与 init process 搞混淆了。该函数见下面。

```

        /*
         * The boot idle thread must execute schedule()
         * at least one to get things moving:
         */
        preempt_enable_no_resched();
        schedule();
        preempt_disable();

        /* Call into cpu_idle with preempt disabled */
        cpu_idle();
}
```

内核创建的内核线程运行本函数，在本函数的结尾就是启动 init process

```
static int init(void * unused)
{
        lock_kernel();
        /*
         * init can run on any cpu.
         */
        set_cpus_allowed(current, CPU_MASK_ALL);
        /*
         * Tell the world that we're going to be the grim
         * reaper of innocent orphaned children.
         *
         * We don't want people to have to make incorrect
         * assumptions about where in the task array this
         * can be found.
         */
        init_pid_ns.child_reaper = current;

        cad_pid = task_pid(current);

        smp_prepare_cpus(max_cpus);

        do_pre_smp_initcalls();

        smp_init();
        sched_init_smp();

        cpuset_init_smp();

        do_basic_setup();

        /*
         * check if there is an early userspace init.  If yes, let it do all
         * the work
         */

        if (!ramdisk_execute_command)
                ramdisk_execute_command = "/init";

        if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
                ramdisk_execute_command = NULL;
                prepare_namespace();
        }

        /*
         * Ok, we have completed the initial bootup, and
         * we're essentially up and running. Get rid of the
         * initmem segments and start the user-mode stuff..
         */
        free_initmem();
        unlock_kernel();
        mark_rodata_ro();
        system_state = SYSTEM_RUNNING;
        numa_default_policy();

        if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
                printk(KERN_WARNING "Warning: unable to open an initial console.\n");

        (void) sys_dup(0);
        (void) sys_dup(0);

```

```
	if (ramdisk_execute_command) {
		run_init_process(ramdisk_execute_command);
		printk(KERN_WARNING "Failed to execute %s\n",
				ramdisk_execute_command);
	}

	/*
	 * We try each of these until one succeeds.
	 *
	 * The Bourne shell can be used instead of init if we are
	 * trying to recover a really broken machine.
	 */
	if (execute_command) {
		run_init_process(execute_command);
		printk(KERN_WARNING "Failed to execute %s.  Attempting "
					"defaults...\n", execute_command);
	}
	run_init_process("/sbin/init");
	run_init_process("/etc/init");
	run_init_process("/bin/init");
	run_init_process("/bin/sh");

	panic("No init found.  Try passing init= option to kernel.");
}
```

run_init_process()实际上是通过嵌入汇编构建一个类似用户态代码一样的sys_execve()调用，其参数就是要执行的可执行文件名，也就是这里的init process在磁盘上的文件。

这里的run_init_process就是通过execve()来运行init程序。这里首先运行“/sbin/init”，如果失败再运行“/etc/init”,然后是“/bin/init”,然后是“/bin/sh”（也就是说，init可执行文件可以放在上面代码中寻找的4个目录中都可以）,如果都失败，则可以通过在系统启动时在添加的启动参数来指定init，比如init=/home/wzhou/init。这里是内核初始化结束并开始用户态初始化的阴阳界。

```
static void run_init_process(char *init_filename)
{
	argv_init[0] = init_filename;
	kernel_execve(init_filename, argv_init, envp_init);
}
```

Linux-2.6.20/arch/i386/kernel/sys_i386.c

```
/*
 * Do a system call from kernel instead of calling sys_execve so we
 * end up with proper pt_regs.
 */   构造 sys_execve 系统调用
int kernel_execve(const char *filename, char *const argv[], char *const envp[])
{
      long __res;
      asm volatile ("push %%ebx ; movl %2,%%ebx ; int $0x80 ; pop %%ebx"
      : "=a" (__res)
      : "0" (__NR_execve),"ri" (filename),"c" (argv), "d" (envp) : "memory");
      return __res;
}
```

init 进程分析

整个 init 的代码比较难读，倒不是真的 init process 要完成的工作有多么复杂，在我看来，这复杂大半的原因是设计者自找的。

init 的执行常规分成三种：

1. 在系统启动阶段，操作系统内核部分初始化阶段的结尾，将运行 init 这个第一个用户态的程序（它将作为所有用户态进程的共同祖先），它将依据/etc/inittab 配置文件来对系统进行用户态的初始化。
2. 在系统运行当中 root 用户可以运行 init 命令把系统切换到不同的运行级别（run level）。比如当前运行级别是 3（Console 界面的 Full multiuser mode），而 root 想维护系统，他可以运行如下命令：
   # init 1　　　　　　　切换到 Single user mode,即单用户模式，有点像 Windows 下的安全模式
   用户启动的 init 命令并不真正运行 run level 切换的工作，只是通过 pipe(管道)把命令打包成 request，然后传递给作为 daemon 进程运行的 init。
3. 在系统起来以后，init 作为一个 daemon 进程运行，一是监控/etc/inittab 配置文件中的相关命令的执行，二就是通过 pipe(管道)接受 2 中发来的切换 run level 的 request(请求)并处理之。

设计者把上面的功能合三为一，把整个逻辑完全搅和在一起，造成代码的难读。我不敢怀疑代码作者的水平，我只是想这可能是 Linux 下有些开发者的特点。就像 Linux 之父非常坚定地拒绝把 kernel 级的 debugger 支持引入其一手遮天的官方内核，虽然他有他的理由，但我想很多 Linux 下的内核黑客并不一定认同他的理由，只能无奈的接受他的这个“特点”。

我把 init process 的三种状态分别称为“init 1”，“init 2”，“init 3”，分别对应上面列举的三种状态。如果把这三种状态的 init 混在一块儿说的话，很容易搞成一团乱麻。我在下面把 init 按三种角色来说明，虽然实际上只有一个可执行文件，进程常规情况下是一个，当通过 init 切换 run level 时会是两个，绝不会是三个。

init 1 的运行

主流程分析

init 1 是由内核启动的，不带任何命令行参数，即直接执行/sbin/init。

```
/*
 * Main entry for init and telinit.
 */
int main(int argc, char **argv)            这时 argc 为 1，argv[0] = “/sbin/init”
{
        char                    *p;
        int                     f;
        int                     isinit;

        /* Get my own name */
        if ((p = strrchr(argv[0], '/')) != NULL)        argv[0] = /sbin/init
                p++;                                    则 p 指向 init
        else
                p = argv[0];
        umask(022);

        /* Quick check */
        if (geteuid() != 0) {          检查是否拥有 root 权限，运行 init 必须拥有该权限，内核当然拥有 root 权限
2612                fprintf(stderr, "%s: must be superuser.\n", p);
2613                exit(1);
2614        }
2615
2616        /*
2617         *     Is this telinit or init ?
2618         */
2619        isinit = (getpid() == 1);           内核启动的 init process 的 PID 为 1，满足，即 isinit = true

2620        for (f = 1; f < argc; f++) {        init 1 没有命令行参数，所以不尽如循环
2621                if (!strcmp(argv[f], "-i") || !strcmp(argv[f], "--init"))
2622                        isinit = 1;
2623                        break;
2624        }
2625        if (!isinit) exit(telinit(p, argc, argv));      init 1 不执行
2626
2627        /*
2628         *     Check for re-exec
2629         */
2630        if (check_pipe(STATE_PIPE)) {             检查 init 1 与 init 3 之间沟通的 pipe 是否建立，这时是内核启动
2631                                                  的 init，当然还白废待新，什么都没有呢。所以这里 check_pipe()
2632                receive_state(STATE_PIPE);        返回 0，init 0 不会进入该 if 分支，跳到 L2646 执行
2633
2634                myname = istrdup(argv[0]);
2635                argv0 = argv[0];
2636                maxproclen = 0;
2637                for (f = 0; f < argc; f++)
2638                        maxproclen += strlen(argv[f]) + 1;
2639                reload = 1;
2640                setproctitle("init [%c]",runlevel);
2641
2642                init_main();
2643        }
2644
2645        /* Check command line arguments */        由于启动 init 1 时没有带任何命令行参数，所以不会进入下面的循环
2646        maxproclen = strlen(argv[0]) + 1;         直接跳到 L2666 执行
```

```
      for(f = 1; f < argc; f++) {
            if (!strcmp(argv[f], "single") || !strcmp(argv[f], "-s"))
                  dfl_level = 'S';
            else if (!strcmp(argv[f], "-a") || !strcmp(argv[f], "auto"))
                  putenv("AUTOBOOT=YES");
            else if (!strcmp(argv[f], "-b") || !strcmp(argv[f],"emergency"))
                  emerg_shell = 1;
            else if (!strcmp(argv[f], "-z")) {
                  /* Ignore -z xxx */
                  if (argv[f + 1]) f++;
            } else if (strchr("0123456789sS", argv[f][0])
                  && strlen(argv[f]) == 1)
                  dfl_level = argv[f][0];
            /* "init u" in the very beginning makes no sense */
            if (dfl_level == 's') dfl_level = 'S';
            maxproclen += strlen(argv[f]) + 1;
      }

      /* Start booting. */
      argv0 = argv[0];                    到此，init 1 的 argv0 = /sbin/init
      argv[1] = NULL;
      setproctitle("init boot");
      init_main(dfl_level);               init 1 调用 init_main(0)，dfl_level 被静态的初始化为 0

      /*NOTREACHED*/
      return 0;
}
```

OK，init 1 进入主函数 init_main()。

```
/*
 *     The main loop
 */
int init_main()
{
  CHILD                 *ch;
  struct sigaction      sa;
  sigset_t        sgt;
  pid_t                 rc;
  int             f, st;

  if (!reload) {              init 1 不会修改 reload 值，所以还是 0，所以会进入这里的 if 分支

#if INITDEBUG                 debug init 用，忽略，debug init 1 可真有点技巧。因为/sbin/init 虽然是普通的用户
      /*                      程序，但别忘了它运行时的时机，debugger 根本还没机会介入呢
       * Fork so we can debug the init process.
       */
      if ((f = fork()) > 0) {
            static const char killmsg[] = "PRNT: init killed.\r\n";
            pid_t rc;

            while((rc = wait(&st)) != f)
                  if (rc < 0 && errno == ECHILD)
                        break;
            write(1, killmsg, sizeof(killmsg) - 1);
            while(1) pause();
      }
#endif

#ifdef __linux__              因为该 init 的代码,FreeBSD 也会用到，所以用该 Macro 来表示是为 Linux 编译的
      /*
       *    Tell the kernel to send us SIGINT when CTRL-ALT-DEL
       *    is pressed, and that we want to handle keyboard signals.
       */
      init_reboot(BMAGIC_SOFT);           调用 reboot(BMAGIC_SOFT)，使得当按下 CTRL-ALT-DEL 后，，将向 init
                                          process发SIGINT signal，见man 2 reboot

      if ((f = open(VT_MASTER, O_RDWR | O_NOCTTY)) >= 0) {
            (void) ioctl(f, KDSIGACCEPT, SIGWINCH);
            close(f);
      } else
            (void) ioctl(0, KDSIGACCEPT, SIGWINCH);
#endif

```

```
        /*
         *    Ignore all signals.
         */
        for(f = 1; f <= NSIG; f++)
              SETSIG(sa, f, SIG_IGN, SA_RESTART);
    }

```

把当前所有 signal handler 都设为 ignore，即不处理该 signals，因为下面要重设这些 signals 的 handler

背景介绍

对需要特殊处理的 signal 进行设置。

SIGALRM 为超时信号，即设置好时钟，当时钟到时后发该信号

SIGHUP 为连接断开信号，比如你通过 telnet 远程登陆到某台 Linux 机器上，启动 top 程序，然后从 telnet 退出，这是你刚才启动的 top 程序会收到该信号。

SIGINT 为中断操作信号，当用户按 Ctrl-C 时，前台进程组会收到该信号,系统将会把 CTRL-ALT-DEL 转换成该 signal

SIGCHLD 为当进程被终止或停止时会发该信号给其父进程

SIGPWR 为当电源失效，UPS 开始工作时，系统会发该信号给 init 进程

SIGWINCH（WINdow CHange）不太了了

SIGUSR1 为用户定义信号

SIGSTOP 停止信号

SIGTSTP 交互停止信号，当用户在终端上按 Ctrl-Z 后，当前进程会挂起

SIGCONT 为与 SIGSTOP 相对的 continue 信号

SIGSEGV 段违例，一般指访问了非法内存

当 init process 收到 SIGALRM
　　　　　　　SIGHUP
　　　　　　　SIGINT
　　　　　　　SIGPWR
　　　　　　　SIGWINCH
　　　　　　　SIGUSR1 时会执行 signal_handler（），signal_handler（）只是把收到的 signal 记录在全局变量 got_signals 中了事。

当 init process 收到 SIGCHLD，会执行 chld_handler（），即当 init process 的子进程死亡时，会执行 chld_handler（），取得该死亡儿子的退出码。

当 init process 收到 SIGSTOP，SIGTSTP，会执行 stop_handler （）
当 init process 收到 SIGCONT，会执行 cont_handler （）
当 init process 收到 SIGSEGV，会执行 segv_handler （），即 init process 访问非法内存后的处理，只是 init process 睡眠３０秒，然后接着干。一般 process 如果访问非法内存，都会死掉。

```
SETSIG(sa, SIGALRM,  signal_handler, 0);
SETSIG(sa, SIGHUP,   signal_handler, 0);
SETSIG(sa, SIGINT,   signal_handler, 0);
SETSIG(sa, SIGCHLD,  chld_handler, SA_RESTART);
SETSIG(sa, SIGPWR,   signal_handler, 0);
SETSIG(sa, SIGWINCH, signal_handler, 0);
SETSIG(sa, SIGUSR1,  signal_handler, 0);
SETSIG(sa, SIGSTOP,  stop_handler, SA_RESTART);
SETSIG(sa, SIGTSTP,  stop_handler, SA_RESTART);
SETSIG(sa, SIGCONT,  cont_handler, SA_RESTART);
SETSIG(sa, SIGSEGV,  (void (*)(int))segv_handler, SA_RESTART);
```

重设 init process 关心的 signal handler

让我们看看这里注册的几个 signal handler 到底干了点什么？
响应 SIGALRM，SIGHUP，SIGINT，SIGPWR，SIGWINCH，SIGUSR1 的是 signal_handler。

```
/*
 *    We got a signal (HUP PWR WINCH ALRM INT)
 */
void signal_handler(int sig)
{
      ADDSET(got_signals, sig);
}
```

对于 HUP PWR WINCH ALRM INT signal 只是记录一下了事，具体处理在 init_main()中的 process_signals()

got_signals 是一个全局变量。

```
sig_atomic_t got_signals;     /* Set if we received a signal. */
```

而 ADDSET （）只是用来把发生的 signal 纪录到该全局变量中。

```
#define ADDSET(set, val)   ((set) |=  (1 << (val)))
```

对这些纪录在 got_signals 变量中的 signal 的处理在函数 process_signals()中。

```
void process_signals()
{
```

处理 init process 受到的 SIGALRM，SIGHUP，SIGINT，SIGPWR，SIGWINCH，SIGUSR1 signal

```
CHILD           *ch;
int        pwrstat;
int        oldlevel;
int        fd;
char            c;

if (ISMEMBER(got_signals, SIGPWR)) {    收到过SIGPWR signal，即UPS报告电源fail
    INITDBG(L_VB, "got SIGPWR");
    /* See _what_ kind of SIGPWR this is. */
    pwrstat = 0;
    if ((fd = open(PWRSTAT, O_RDONLY)) >= 0) {      打开/etc/powerstatus文件，如果该文件存在，则可能是如
          c = 0;                                    下三个字符中的一个：“F”，“O”，“L”，应该是表示收到
          read(fd, &c, 1);                          SIGPWR  signal的原因吧。F表示fail，O表示OK，L表示low

          pwrstat = c;                        把电源fail的原因纪录在变量pwrstat中
          close(fd);
          unlink(PWRSTAT);                    删除/etc/powerstatus文件
    }
    do_power_fail(pwrstat);       根据powerfail的原因来对family链表中的action进行处理，而该连表中的
                                  action是完全按照配置文件inittab中的“powerfail”配置来实行的，见下面的

    pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
    这里init process就会执行/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

    DELSET(got_signals, SIGPWR);  从got_signals全局变量中删去SIGPWR标志
}

if (ISMEMBER(got_signals, SIGINT)) {    收到过SIGINT  signal
    INITDBG(L_VB, "got SIGINT");
    /* Tell ctrlaltdel entry to start up */
    for(ch = family; ch; ch = ch->next)
          if (ch->action == CTRLALTDEL)
                ch->flags &= ~XECUTED;                允许Ctrl-Alt-Del handler的运行
    DELSET(got_signals, SIGINT);
}
```

```

  if (ISMEMBER(got_signals, SIGWINCH)) {
      INITDBG(L_VB, "got SIGWINCH");
      /* Tell kbrequest entry to start up */
      for(ch = family; ch; ch = ch->next)
            if (ch->action == KBREQUEST)
                  ch->flags &= ~XECUTED;
      DELSET(got_signals, SIGWINCH);
  }

  if (ISMEMBER(got_signals, SIGALRM)) {         对收到的 SIGALRM signal，只是删除标志了事
      INITDBG(L_VB, "got SIGALRM");
      /* The timer went off: check it out */
      DELSET(got_signals, SIGALRM);
  }

  if (ISMEMBER(got_signals, SIGCHLD)) {         收到 init process 的子进程死亡而发来的消息
      INITDBG(L_VB, "got SIGCHLD");
      /* First set flag to 0 */
      DELSET(got_signals, SIGCHLD);             首先删除收到该 signal 的标志

      /* See which child this was */
      for(ch = family; ch; ch = ch->next)       对 family 链表进行枚举，如果发现僵尸，则清除三个标志
          if (ch->flags & ZOMBIE) {
            INITDBG(L_VB, "Child died, PID= %d", ch->pid);
            ch->flags &= ~(RUNNING|ZOMBIE|WAITING);
            if (ch->process[0] != '+')
                  write_utmp_wtmp("", ch->id, ch->pid, DEAD_PROCESS, NULL);
          }

  }

  if (ISMEMBER(got_signals, SIGHUP)) {    signal SIGHUP 一般用来通知进程重读其配置文件，这里就是通知 init
      INITDBG(L_VB, "got SIGHUP");        process 重新读取 inittab 文件中的配置
#if CHANGE_WAIT
      /* Are we waiting for a child? */
      for(ch = family; ch; ch = ch->next)
            if (ch->flags & WAITING) break;
      if (ch == NULL)
#endif
      {
            /* We need to go into a new runlevel */
            oldlevel = runlevel;
#ifdef INITLVL
            runlevel = read_level(0);
#endif
            if (runlevel == 'U') {
                  runlevel = oldlevel;
                  re_exec();
            } else {
                  if (oldlevel != 'S' && runlevel == 'S') console_stty();
                  if (runlevel == '6' || runlevel == '0' ||
                      runlevel == '1') console_stty();
                  read_inittab();
                  fail_cancel();
                  setproctitle("init [%c]", runlevel);
                  DELSET(got_signals, SIGHUP);
            }
      }
  }
  if (ISMEMBER(got_signals, SIGUSR1)) {         收到用户自定义 signal，这里用来重新打开 pipe 文件/dev/initctl
      /*
       *    SIGUSR1 means close and reopen /dev/initctl
       */
      INITDBG(L_VB, "got SIGUSR1");
      close(pipe_fd);
      pipe_fd = -1;
      DELSET(got_signals, SIGUSR1);
  }
}
```

继续刚才未完的 init_main()的执行。

```

console_init();       对系统console的初始化

if (!reload) {        前面已经分析过，在init 1时，该值为0，也即会进入下面的if分支

    /* Close whatever files are open, and reset the console. */
    close(0);         这里的一系列操作，关闭0，1，2，调用setsid()类似于Linux下Daemon编程的老套路。init进程也
    close(1);         是daemon process
    close(2);
    console_stty();
    setsid();

    /*
     *    Set default PATH variable.
     */
    putenv(PATH_DFL);       设置init 1执行时的环境变量中的PATH="PATH=/bin:/usr/bin:/sbin:/usr/sbin"

    /*
     *    Initialize /var/run/utmp (only works if /var is on
     *    root and mounted rw)
     */
    (void) close(open(UTMP_FILE, O_WRONLY|O_CREAT|O_TRUNC, 0644));    创建/var/run/utmp文件

    /*
     *    Say hello to the world
     */
    initlog(L_CO, bootmsg, "booting");

    /*
     *    See if we have to start an emergency shell.
     */
    if (emerg_shell) {    如果在命令行上指定启动紧急shell，但init 1的命令行是空的，所以这里的if分支不会执行到
          SETSIG(sa, SIGCHLD, SIG_DFL, SA_RESTART);
          if (spawn(&ch_emerg, &f) > 0) {
                while((rc = wait(&st)) != f)
                      if (rc < 0 && errno == ECHILD)
2436                                break;
2437              }
2438              SETSIG(sa, SIGCHLD,  chld_handler, SA_RESTART);
2439        }
2440
2441        /*
2442         *    Start normal boot procedure.
2443         */
2444        runlevel = '#';         表示现在还不知道 init 1 将要进入什么运行级别
2445        read_inittab();         读取/etc/inittab 中的设置，非常关键的函数，见对该函数的注解
2446
2447    } else {  re-exec 模式，即 init process 运行期间重读 inittab 配置文件的处理，init 1 不满足该条件，所以跳到 L2455
2448        /*
2449         *    Restart: unblock signals and let the show go on
2450         */
2451        initlog(L_CO, bootmsg, "reloading");
2452        sigfillset(&sgt);
2453        sigprocmask(SIG_UNBLOCK, &sgt, NULL);
2454    }
2455    start_if_needed();    枚举在 family 链表中的代表/etc/inittab 每一行的 action，并启动它们。该函数见辅助函数分析。
                          作为系统启动阶段，运行 inittab 文件中指定的命令行。
2456
2457    while(1) {      进入 init process 的主循环，init process 以后就在该无限循环中打转，永不出来。这实际上是 init 3，
2458                    也即作为 daemon process 运行的 init 的工作了。
2459       /* See if we need to make the boot transitions. */
2460       boot_transitions();
2461       INITDBG(L_VB, "init_main: waiting..");
2462
2463       /* Check if there are processes to be waited on. */
2464       for(ch = family; ch; ch = ch->next)
2465        if ((ch->flags & RUNNING) && ch->action != BOOT) break;


2466
2467  #if CHANGE_WAIT
```

```
2468       /* Wait until we get hit by some signal. */
2469       while (ch != NULL && got_signals == 0) {   daemon process 检查是否有关心的事件发生。
2470        if (ISMEMBER(got_signals, SIGHUP)) {      检查发生了 SIGHUP，即让 init process 重读 inittab 文件的事件吗
2471              /* See if there are processes to be waited on. */
2472              for(ch = family; ch; ch = ch->next)       标志为“wait”的配置行于“boot”配置行类似，不能与其他
2473                    if (ch->flags & WAITING) break;     配置行并发执行
2474        }
2475        if (ch != NULL) check_init_fifo();
2476       }
2477  #else /* CHANGE_WAIT */
2478       if (ch != NULL && got_signals == 0) check_init_fifo();
2479  #endif /* CHANGE_WAIT */
```

check_init_fifo()首先建立 init 2 与 init 3 之间沟通的渠道“/dev/initctl”pipe。

```
2480
2481       /* Check the 'failing' flags */
2482       fail_check();
2483
2484       /* Process any signals. */
2485       process_signals();       处理被记录下来的 signal，即当 init process 收到 signal，并纪录在 got_signals 变量
2486                                后，要在 init process 的主循环中才能执行，所以 init process 对相关 signal 的真正
                                处理并不是实时的，即受到 signal 就处理，而是要在 L2457 的循环中查询到后才能执行
2487       /* See what we need to start up (again) */
2488       start_if_needed();       有可能在 family 链表中的 node 状态已经改变，所以重新枚举整个链表，看是否有本来不能运行
2489    }                           的动作(action)可以执行了
2490    /*NOTREACHED*/
2491  }
```

下面是 init process 及其重要的读取配置文件 inittab 并生成自己管理的数据结构。Inittab 配置文件的格式已经在上面说明过。、每一个 init process 管理的进程都用如下结构来管理：

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

该结构中的原有注释写得挺详细的，这里补充几点。

Inittab 中的配置行大约如下：

```
3:2345:respawn:/sbin/mingetty tty3
```

这里 init process 就会有一个 CHILD 与其对应，用上面的信息来填充该结构中的某些 field。

id[8]	rlevel[12]	action	process[128]
3	2345	Respawn	/sbin/mingetty tty3

而 flags 反映该 process 的状态

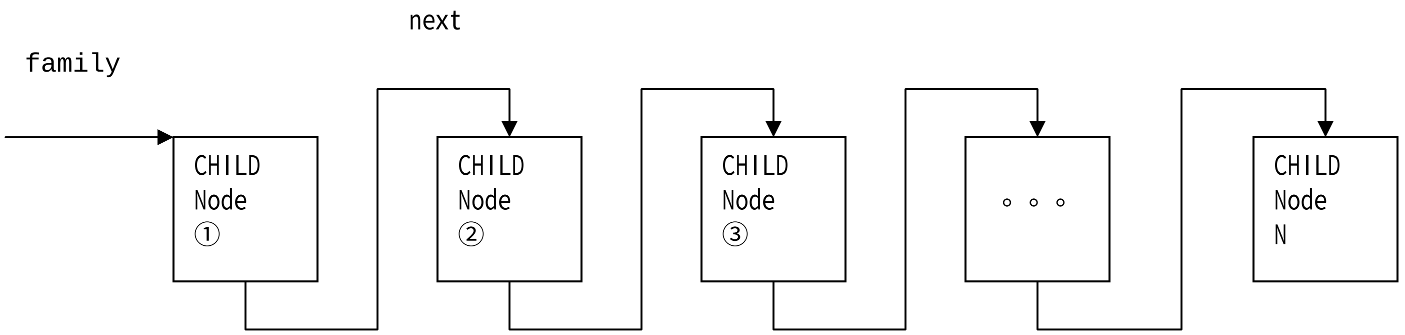
exstat 是该 process 在退出后的退出码，也就是提供给 exit()系统调用的参数

pid 是该 process 的 process identifier

tm 对 respawn 类型的 process 才有意义，即其被 init process respawn 时的时间戳

count 对 respawn 与 ondemanded 型的 process 有意义，即其在最近 2 分钟内被启动(spawn)的次数。

init process 把它管理的 process 通过 next 来链接在由全局变量 family 指向的链表中。



/etc/inittab
配置文件

```
id:5:initdefault:              CHILD Node ①

si::sysinit:/etc/rc.d/rc.sysinit
                               CHILD Node ②
l0:0:wait:/etc/rc.d/rc 0       CHILD Node ③
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6


ca::ctrlaltdel:/sbin/shutdown -t3 -r now

3:2345:respawn:/sbin/mingetty tty3  CHILD Node N
```

read_inittab（）就是把inittab文件中的每一行生成family链表中的一个节点（node），并用该行上的信息来填充该节点

```
void read_inittab(void)
{
  FILE            *fp;              /* The INITTAB file */
  CHILD           *ch, *old, *i;    /* Pointers to CHILD structure */
  CHILD           *head = NULL;     /* Head of linked list */
#ifdef INITLVL
  struct stat     st;               /* To stat INITLVL */
#endif
  sigset_t  nmask, omask;           /* For blocking SIGCHLD. */
  char            buf[256];         /* Line buffer */
  char            err[64];          /* Error message. */
  char            *id, *rlevel,
            *action, *process;      /* Fields of a line */
  char            *p;
  int       lineNo = 0;       /* Line number in INITTAB file */
  int       actionNo;         /* Decoded action field */
  int       f;                /* Counter */
  int       round;            /* round 0 for SIGTERM, 1 for SIGKILL */
  int       foundOne = 0;     /* No killing no sleep */
  int       talk;             /* Talk to the user */
  int       done = 0;         /* Ready yet? */

#if DEBUG
  if (newFamily != NULL) {
      INITDBG(L_VB, "PANIC newFamily != NULL");
      exit(1);
  }
  INITDBG(L_VB, "Reading inittab");
#endif

  /*
   *  Open INITTAB and real line by line.
   */
  if ((fp = fopen(INITTAB, "r")) == NULL)             打开/etc/inittab 文件
      initlog(L_VB, "No inittab file found");

```

```
    while(!done) {              每循环一次即处理 inittab 中一行，构造 newFamily 链表。注意是 newFamily 链表，不是
        /*                      family 链表
         *    Add single user shell entry at the end.
         */
        if (fp == NULL || fgets(buf, sizeof(buf), fp) == NULL) {
              done = 1;         已经处理完毕，退出循环
              /*
               *    See if we have a single user entry.
               */
              for(old = newFamily; old; old = old->next)            ？？？
                    if (strpbrk(old->rlevel, "S")) break;
              if (old == NULL)
                    snprintf(buf, sizeof(buf), "~~:S:wait:%s\n", SULOGIN);
              else
                    continue;
        }
        lineNo++;
        /*
         *    Skip comments and empty lines
         */
        for(p = buf; *p == ' ' || *p == '\t'; p++)      忽略前导空白字符
              ;
        if (*p == '#' || *p == '\n') continue;          以“#”开头的为注释，忽略

        /*
         *    Decode the fields
         */
  分解 id:runlevels:action:process 中的 4 部分
        id =      strsep(&p, ":");    由于文件中的配置行的各部分用“：”分割，所以这里通过 strsep 来分别提取各部分内
        rlevel =  strsep(&p, ":");    容
        action =  strsep(&p, ":");
        process = strsep(&p, "\n");

  从下面的代码可以看到在 init manual 中没有标明的限制，比如命令行的长度不能太长，超过 127 个字符等
        /*
         *    Check if syntax is OK. Be very verbose here, to
```

```
       *    avoid newbie postings on comp.os.linux.setup :)
       */
      err[0] = 0;
      if (!id || !*id) strcpy(err, "missing id field");
      if (!rlevel)     strcpy(err, "missing runlevel field");
      if (!process)    strcpy(err, "missing process field");
      if (!action || !*action)
                  strcpy(err, "missing action field");
      if (id && strlen(id) > sizeof(utproto.ut_id))
            sprintf(err, "id field too long (max %d characters)",
                  (int)sizeof(utproto.ut_id));
      if (rlevel && strlen(rlevel) > 11)
            strcpy(err, "rlevel field too long (max 11 characters)");
      if (process && strlen(process) > 127)
            strcpy(err, "process field too long");
      if (action && strlen(action) > 32)
            strcpy(err, "action field too long");
      if (err[0] != 0) {
            initlog(L_VB, "%s[%d]: %s", INITTAB, lineNo, err);
            INITDBG(L_VB, "%s:%s:%s:%s", id, rlevel, action, process);
            continue;
      }

      /*
       *    Decode the "action" field
       */
init 允许的 action 类型都记录在 actions[]数组中，这里通过比较字符串来把其转换成数字型 identifier
      actionNo = -1;
      for(f = 0; actions[f].name; f++)
            if (strcasecmp(action, actions[f].name) == 0) {
                  actionNo = actions[f].act;
                  break;
            }
      if (actionNo == -1) {         碰到非法的 action(不在 actions[]数组中的)则忽略
            initlog(L_VB, "%s[%d]: %s: unknown action field",
                  INITTAB, lineNo, action);
            continue;
```

```
        }

        /*
         *      See if the id field is unique
         */
```

配置行中的第一部分是所谓 identifier，必须唯一，但命名好像没有任何规定，可任意。已经处理过的配置行都被记录入 CHILD 的链表节点中，这里在处理当前行时检查一下已有节点中是否有与当前行的 id 相同的，如果有，则不是忽略该行，而是停止继续处理/etc/inittab 文件，可见 id 的唯一性是至关重要的

```
        for(old = newFamily; old; old = old->next) {
                if(strcmp(old->id, id) == 0 && strcmp(id, "~~")) {
                        initlog(L_VB, "%s[%d]: duplicate ID field \"%s\"",
                                INITTAB, lineNo, id);
                        break;
                }
        }
        if (old) continue;

        /*
         *      Allocate a CHILD structure
         */
        ch = imalloc(sizeof(CHILD));        为当前配置行分配一个 CHILD node

        /*
         *      And fill it in.         用当前配置行中的信息来填充 CHILD node
         */
        ch->action = actionNo;          action 类型
        strncpy(ch->id, id, sizeof(utproto.ut_id) + 1); /* Hack for different libs. */该行的唯一标示符
        strncpy(ch->process, process, sizeof(ch->process) - 1);     该行是要执行的命令行
        if (rlevel[0]) {                填上 run level
                for(f = 0; f < sizeof(rlevel) - 1 && rlevel[f]; f++) {
                        ch->rlevel[f] = rlevel[f];
                        if (ch->rlevel[f] == 's') ch->rlevel[f] = 'S';
                }
                strncpy(ch->rlevel, rlevel, sizeof(ch->rlevel) - 1);
        } else {     如果没有写 run level，则表示所有 run level 都要执行该行的 process 部分
                strcpy(ch->rlevel, "0123456789");
                if (ISPOWER(ch->action))
                        strcpy(ch->rlevel, "S0123456789");
        }
```

下面是对 action 的处理

```
        /*
         *     We have the fake runlevel '#' for SYSINIT  and
         *     '*' for BOOT and BOOTWAIT.
         */
```

从上面的注释看，SYSINIT action 用'#'表示，而 BOOT action 用'*'表示。而真正合法的 run level 是 0 到 9 加上'S'
“＃”与“＊”表示在任何 run level 都要执行，另外 SYSINIT 的优先级是最高的，所以它应该比 BOOT 中的 action 先执行

```
        if (ch->action == SYSINIT) strcpy(ch->rlevel, "#");
        if (ch->action == BOOT || ch->action == BOOTWAIT)
                strcpy(ch->rlevel, "*");

        /*
         *     Now add it to the linked list. Special for powerfail.
         */
```

在从/etc/inittab 中读取配置行并生成的链表的头为 newFamily。如果是系统的启动阶段，family 所表示的链表自然为空，如果只是通过运行 init 来切换 run level 等，则 family 记录的链表非空，也就是当前 init 通过上次读取/etc/inittab 后生成的链表。

```
        if (ISPOWER(ch->action)) {     如果 action 属于这几种（POWERWAIT，POWERFAIL，POWEROKWAIT，
                                       POWERFAILNOW，CTRLALTDEL），即与电源相关与用户按了 Ctrl+Alt+Del 键

                /*
                 *     Disable by default
                 */
                ch->flags |= XECUTED;  设置不执行标志。在 startup()种如果检测到该配置行所代表的 action 的 flag 设置
                                       了 XECUTED，则忽略对该行的处理。这可以理解，因为符合 ISPOWER()的 action 都不是
                /*                     在正常情况下需要运行的，只有对应的事件确实发生了，才需要执行。比如，如果用户从
                                       来没有按下过 Ctrl+Alt+Del 键，自然根本不需要执行/etc/inittab 中 CTRLALTDEL
```

Action 指定的 process。所以在默认情况下它是 disable 的(通过设置 XECUTED 标志)。当检测到按下 Ctrl+Alt+Del 键后，才需要 enable。

并且上述 action 被插在 family 链表的前面，这样如果它们需要执行的话，将先得到执行。可以理解，因为这几个动作都比较严重，所以优先权较高

```
                 *    Tricky: insert at the front of the list..
                 */
                old = NULL;
                for(i = newFamily; i; i = i->next) {
                        if (!ISPOWER(i->action)) break;
                        old = i;
                }
                /*
                 *    Now add after entry "old"
                 */
                if (old) {
                        ch->next = i;
                        old->next = ch;
                        if (i == NULL) head = ch;
                } else {
                        ch->next = newFamily;
                        newFamily = ch;
                        if (ch->next == NULL) head = ch;
                }
        } else {  其他的 action 都插到尾部，KBREQUEST 默认是不执行的。从 init manual 上看 SIGWINCH  signal 会触发该动作
                /*
                 *    Just add at end of the list
                 */
                if (ch->action == KBREQUEST) ch->flags |= XECUTED;
                ch->next = NULL;
                if (head)
                        head->next = ch;
                else
                        newFamily = ch;
                head = ch;
        }

```

```
1301      /*
1302       *    Walk through the old list comparing id fields
1303       */
1304      for(old = family; old; old = old->next)
1305         if (strcmp(old->id, ch->id) == 0) {
1306            old->new = ch;
1307            break;
1308         }
          到这里处理一行结束
1309    }
1310    /*
1311     *  We're done.
1312     */
1313    if (fp) fclose(fp);   关闭/etc/inittab 文件，对于 init 1，实际上基本上到此为止。
1314                          下面都是作为 daemon 进程的 init 即 init 3 的处理。我觉得整个代码应该整理得更清晰一点，由
                              kernel 启动 init(init 1)与作为 daemon 进程运行的 init （init 3） 的逻辑应该分开，而不要像现
                              在一样，绞和在一块，比较乱

1315    /*
1316     *  Loop through the list of children, and see if they need to
1317     *  be killed.
1318     */
1319
1320    INITDBG(L_VB, "Checking for children to kill");
1321    for(round = 0; round < 2; round++) {
1322      talk = 1;
1323      for(ch = family; ch; ch = ch->next) { 由于是系统启动阶段运行 init （init 1） ，则这时的 family 链表为空，应该
1324        ch->flags &= ~KILLME;               不进入循环，则这时 round = 0，talk = 1，foundOne = 0，代码应该跳转
1325                                            到 L1393 行执行。



1326      /*
1327       *    Is this line deleted?
1328       */
1329      if (ch->new == NULL) ch->flags |= KILLME;
```

```

        /*
         *    If the entry has changed, kill it anyway. Note that
         *    we do not check ch->process, only the "action" field.
         *    This way, you can turn an entry "off" immediately, but
         *    changes in the command line will only become effective
         *    after the running version has exited.
         */
        if (ch->new && ch->action != ch->new->action) ch->flags |= KILLME;

        /*
         *    Only BOOT processes may live in all levels
         */
        if (ch->action != BOOT &&
            strchr(ch->rlevel, runlevel) == NULL) {
              /*
               *    Ondemand procedures live always,
               *    except in single user
               */
              if (runlevel == 'S' || !(ch->flags & DEMAND))
                    ch->flags |= KILLME;
        }

        /*
         *    Now, if this process may live note so in the new list
         */
        if ((ch->flags & KILLME) == 0) {
              ch->new->flags  = ch->flags;
              ch->new->pid    = ch->pid;
              ch->new->exstat = ch->exstat;
              continue;
        }


        /*
         *    Is this process still around?
         */
```

```
1367        if ((ch->flags & RUNNING) == 0) {
1368              ch->flags &= ~KILLME;
1369              continue;
1370        }
1371        INITDBG(L_VB, "Killing \"%s\"", ch->process);
1372        switch(round) {
1373              case 0: /* Send TERM signal */
1374                    if (talk)
1375                          initlog(L_CO,
1376                                "Sending processes the TERM signal");
1377                    kill(-(ch->pid), SIGTERM);
1378                    foundOne = 1;
1379                    break;
1380              case 1: /* Send KILL signal and collect status */
1381                    if (talk)
1382                          initlog(L_CO,
1383                                "Sending processes the KILL signal");
1384                    kill(-(ch->pid), SIGKILL);
1385                    break;
1386        }
1387        talk = 0;
1388
1389      }
1390      /*
1391       *      See if we have to wait 5 seconds
1392       */
1393      if (foundOne && round == 0) {   对于系统启动阶段运行的 init（init 1），round = 0,但 foundOne = 0，所以
1394        /*                            将不会进入该 if 分支，跳转到 L1419 执行
1395         *    Yup, but check every second if we still have children.
1396         */
1397        for(f = 0; f < sltime; f++) {
1398              for(ch = family; ch; ch = ch->next) {
1399                    if (!(ch->flags & KILLME)) continue;
1400                    if ((ch->flags & RUNNING) && !(ch->flags & ZOMBIE))
1401                          break;
1402              }
1403              if (ch == NULL) {
```

```
1404            /*
1405             *    No running children, skip SIGKILL
1406             */
1407            round = 1;
1408            foundOne = 0; /* Skip the sleep below. */
1409            break;
1410         }
1411         do_sleep(1);
1412      }
1413     }
1414    }
1415
1416    /*
1417     *  Now give all processes the chance to die and collect exit statuses.
1418     */
1419    if (foundOne) do_sleep(1);              init 1 运行时，foundOne = 0，所以不会睡眠一秒
1420    for(ch = family; ch; ch = ch->next)     这时的 family 链表为空，不进入该循环，跳转到 L1437 执行
1421      if (ch->flags & KILLME) {
1422         if (!(ch->flags & ZOMBIE))
1423           initlog(L_CO, "Pid %d [id %s] seems to hang", ch->pid,
1424                   ch->id);
1425         else {
1426           INITDBG(L_VB, "Updating utmp for pid %d [id %s]",
1427                   ch->pid, ch->id);
1428           ch->flags &= ~RUNNING;
1429           if (ch->process[0] != '+')
1430             write_utmp_wtmp("", ch->id, ch->pid, DEAD_PROCESS, NULL);
1431         }
1432      }
1433
1434    /*
1435     *  Both rounds done; clean up the list.
1436     */
1437    sigemptyset(&nmask);
1438    sigaddset(&nmask, SIGCHLD);
1439    sigprocmask(SIG_BLOCK, &nmask, &omask);
1440    for(ch = family; ch; ch = old) {           init 1 运行时，family 链表还为空，所以不进入循环，跳转到 L1444
```

```
      old = ch->next;
      free(ch);
  }
  family = newFamily;                  newFamily 中就是在本函数头上读入的当前的/etc/inittab 配置行生成的链表，现在才
                                       把它赋给 family
  for(ch = family; ch; ch = ch->next) ch->new = NULL;
  newFamily = NULL;
  sigprocmask(SIG_SETMASK, &omask, NULL);

#ifdef INITLVL
  /*
   *  Dispose of INITLVL file.
   */ 删除/etc/initrunlvl，根据是文件还是 symbol link，删除方式是不一样的
  if (lstat(INITLVL, &st) >= 0 && S_ISLNK(st.st_mode)) {    检查/etc/initrunlvl 是文件还是 symbol link
      /*
       *    INITLVL is a symbolic link, so just truncate the file.
       */
      close(open(INITLVL, O_WRONLY|O_TRUNC));
  } else {
      /*
       *    Delete INITLVL file.
       */
      unlink(INITLVL);
  }
#endif
#ifdef INITLVL2
  /*
   *  Dispose of INITLVL2 file.
   */ 删除/var/log/initrunlvl，根据是文件还是 symbol link，删除方式是不一样的
  if (lstat(INITLVL2, &st) >= 0 && S_ISLNK(st.st_mode)) {
      /*
       *    INITLVL2 is a symbolic link, so just truncate the file.
       */
      close(open(INITLVL2, O_WRONLY|O_TRUNC));
  } else {
      /*
```

```
1476         *    Delete INITLVL2 file.
1477         */
1478        unlink(INITLVL2);
1479    }
1480  #endif
1481  }
```

从上面的代码分析可看出，对 init 1 而言，read_inittab()的处理到 L1313 行实际上就结束了，剩下的处理都是为作为 daemon 运行的 init 3 来服务的（将在分析 init 3 的运行时对其详细分析），它的功能就是读取 inittab 配置文件中的行，然后构建 CHILD node 构成的 family 链表，即把硬盘上的 init process 要实现的功能的配置转换成内存中的实现配置，以便 start_if_needed()根据该内存中的配置来执行。

```
1483  /*
1484   *    Walk through the family list and start up children.
1485   *    The entries that do not belong here at all are removed
1486   *    from the list.
1487   */
```

从上面的注释可看出本函数的功能就是枚举 family 链表，然后执行每个节点上指定的动作，即执行 CHILD 节点中 process[128]中指定的命令行

```
1488  void start_if_needed(void)
1489  {
1490        CHILD *ch;        /* Pointer to child */
1491        int delete;       /* Delete this entry from list? */
1492
1493        INITDBG(L_VB, "Checking for children to start");
```

通过 read_inittab()，所有/etc/inittab 配置文件中的合法的每一行都被用 family 链表中的一个 node 表示，现在枚举该链表

```
1494
1495        for(ch = family; ch; ch = ch->next) {
1496
1497  #if DEBUG
1498              if (ch->rlevel[0] == 'C') {
1499                    INITDBG(L_VB, "%s: flags %d", ch->process, ch->flags);
```

```
        }
#endif

        /* Are we waiting for this process? Then quit here. */
        if (ch->flags & WAITING) break;     WAITING 标志表示 init  process 必须等待该启动的子进程退出后才能
                                            继续下面的工作，即可能后面的工作依赖于当前的 process，所以必须
                                            等待其完成后才能开始新的工作

        /* Already running? OK, don't touch it */
        if (ch->flags & RUNNING) continue;  该 node 所代表的 process 处于 running 状态，当然不需要做什么

        /* See if we have to start it up */
        delete = 1;                         默认置删除标志
        if (strchr(ch->rlevel, runlevel) ||     runlevel 变量记录着当前的 run  level,对于在该 node 中包含
                                                了当前运行级别的 node 或者是 DEMAND(按需启动)或者
                                                “#*Ss”（代表 sysinit，boot 等）则启动该节点所代表的
                                                action。(“#*Ss”部分 run  level 都要执行)

            ((ch->flags & DEMAND) && !strchr("#*Ss", runlevel))) {
            startup(ch);        与执行该命令，具体间辅助函数介绍
            delete = 0;         启动了，当然不需要删除该 node，则 clear 该标志
        }

        if (delete) {      没有启动，表明该 node 不符合合法的启动 level
            /* FIXME: is this OK? */
            ch->flags &= ~(RUNNING|WAITING);
            if (!ISPOWER(ch->action) && ch->action != KBREQUEST)
                ch->flags &= ~XECUTED;
            ch->pid = 0;
        } else
            /* Do we have to wait for this process? */
            if (ch->flags & WAITING) break;
    }
    /* Done. */
}
```

辅助函数介绍

startup函数

```
1063  /*
1064   *     Start a child running!
1065   */
```

执行 CHILD 节点所代表的配置行上的命令行，比如 l2:2:wait:/etc/rc.d/rc 2 配置行，就是执行这里的“/etc/rc.d/rc 2”，“/etc/rc.d/rc”是执行的可执行脚本，而“2”是该脚本的参数

```
1066  void startup(CHILD *ch)
1067  {
1068        /*
1069         *     See if it's disabled
1070         */
1071        if (ch->flags & FAILING) return;
1072
1073        switch(ch->action) {
1074
1075              case SYSINIT:
1076              case BOOTWAIT:
1077              case WAIT:
1078              case POWERWAIT:
1079              case POWERFAILNOW:
1080              case POWEROKWAIT:
1081              case CTRLALTDEL:
1082                    if (!(ch->flags & XECUTED)) ch->flags |= WAITING;
```

对于上面的几种 action，如果已经允许执行（没有设 XECUTED 标志），则会设置 WAITING 标志，由于这里没有 break,所以将执行 L1091 行的 spawn()，即启动该命令。之所以这里要设 WAITING 标志，是因为上面的类型的动作都具有排它性。

```
1083              case KBREQUEST:
        case BOOT:
        case POWERFAIL:
        case ONCE:
            if (ch->flags & XECUTED) break;     XECUTED 是暂时禁止执行的标志
        case ONDEMAND:
        case RESPAWN:
            ch->flags |= RUNNING;
            if (spawn(ch, &(ch->pid)) < 0) break;     启动该脚本，ch->pid 为返回的 pid，spawn()见辅助
            /*                                        函数说明
             *    Do NOT log if process field starts with '+'
             *    FIXME: that's for compatibility with *very*
             *    old getties - probably it can be taken out.
             */
            if (ch->process[0] != '+')
                write_utmp_wtmp("", ch->id, ch->pid,
                    INIT_PROCESS, "");
            break;
    }
}
```

spawn函数

该函数是真正的去 launch 在 CHILD 中指定的命令行（process[128]）

```
/*
 *    Fork and execute.
 *
 *    This function is too long and indents too deep.
 *
 */   启动 ch 所对应的配置行上的命令，*res 为启动的这个新进程的 pid
```

下面以启动/etc/inittab 中的如下配置行为例：
si::sysinit:/etc/rc.d/rc.sysinit

则 CHILD *ch 的内容如下
ch->id = “si”

```
ch->rlevel = 0123456S
ch->action = SYSINIT
ch->process[128] = /etc/rc.d/rc.sysinit
```

```
int spawn(CHILD *ch, int *res)
{
  char *args[16];       /* Argv array */
  char buf[136];        /* Line buffer */
  int f, st, rc;        /* Scratch variables */
  char *ptr;            /* Ditto */
  time_t t;             /* System time */
  int oldAlarm;               /* Previous alarm value */
  char *proc = ch->process;   /* Command line */
  pid_t pid, pgrp;            /* child, console process group. */
  sigset_t nmask, omask;      /* For blocking SIGCHLD */
  struct sigaction sa;

  *res = -1;
  buf[sizeof(buf) - 1] = 0;

  /* Skip '+' if it's there */
  if (proc[0] == '+') proc++;

  ch->flags |= XECUTED;       当前正要启动的 ch，设置禁止启动标志，防止再进入

```

下面是根据不同的情况来拼装命令行字符串

init process 将监控 action 为“RESPAWN”与“ONDEMAND”类型的命令，如果其不运行了，则要启动它，使它在整个系统运行期间一直运行。下面的处理是为了防止由于该命令的不正常死亡，导致 init process 在短时间内太过频繁的启动。如果在 2 分钟内启动超过 10 次，则先要把该命令凉在一边 5 分钟，然后再启动。

```
  if (ch->action == RESPAWN || ch->action == ONDEMAND) {
      /* Is the date stamp from less than 2 minutes ago? */
      time(&t);   取得当前系统时间，ch->tm 为上次该 process 启动(spawn)时的时间
      if (ch->tm + TESTTIME > t) {  TESTTIME 为 2 分钟，即从上次启动到现在没到 2 分钟，则累加被启动的次数
```

```
854                 ch->count++;
855         } else {
856                 ch->count = 0;          超过 2 分钟则重新计数在 2 分钟内被启动次数
857                 ch->tm = t;             重设时间
858         }
859
860         /* Do we try to respawn too fast? */
861         if (ch->count >= MAXSPAWN) {  启动得太频繁了，2 分钟内被启动了 10 次（说明该程序在 2 分钟内死了超过 10 次）设置
862                                       FAILING 标志
863           initlog(L_VB,
864                 "Id \"%s\" respawning too fast: disabled for %d minutes",
865                 ch->id, SLEEPTIME / 60);
866           ch->flags &= ~RUNNING;
867           ch->flags |= FAILING;       临时设上 FAILING 标志
868
869           /* Remember the time we stopped */
870           ch->tm = t;
871
872           /* Try again in 5 minutes */
873           oldAlarm = alarm(0);        取消 alarm，返回剩余时间
874           if (oldAlarm > SLEEPTIME || oldAlarm <= 0) oldAlarm = SLEEPTIME;      最多暂时 disable 5 分钟
875           alarm(oldAlarm);            启动 alarm，最多等待 5 分钟
876           return(-1);           没有启动就返回了（-1）
877         }
878     }
879
880     /* See if there is an "initscript" (except in single user mode). */
881     if (access(INITSCRIPT, R_OK) == 0 && runlevel != 'S') {   文件/etc/initscript 可读吗，实际上就是是否存在
882         /* Build command line using "initscript" */           如果/etc/initscript 存在，并且当前 run level
883         args[1] = SHELL;                                      不为 Single Mode(单用户模式)
884         args[2] = INITSCRIPT;
885         args[3] = ch->id;                   为 L1045 行的 execvp()调用准备参数
886         args[4] = ch->rlevel;               这里执行的命令大致是这样的
887         args[5] = "unknown";                execvp(“/bin/sh”,args + 1),而 args[1]=“/bin/sh”
888         for(f = 0; actions[f].name; f++) {                         args[2] = “/etc/initscript”
```

```
          if (ch->action == actions[f].act) {
                args[5] = actions[f].name;
                break;
          }
    }
    args[6] = proc;
    args[7] = NULL;
} else if (strpbrk(proc, "~`!$^&*()=|\\{}[];\"'<>?")) {
/* See if we need to fire off a shell for this command */
    /* Give command line to shell */
    args[1] = SHELL;
    args[2] = "-c";
    strcpy(buf, "exec ");
    strncat(buf, proc, sizeof(buf) - strlen(buf) - 1);
    args[3] = buf;
    args[4] = NULL;
} else {
    /* Split up command line arguments */
    buf[0] = 0;
    strncat(buf, proc, sizeof(buf) - 1);
    ptr = buf;
    for(f = 1; f < 15; f++) {
          /* Skip white space */
          while(*ptr == ' ' || *ptr == '\t') ptr++;
          args[f] = ptr;

          /* May be trailing space.. */
          if (*ptr == 0) break;

          /* Skip this `word' */
          while(*ptr && *ptr != ' ' && *ptr != '\t' && *ptr != '#')
                ptr++;

          /* If end-of-line, break */
          if (*ptr == '#' || *ptr == 0) {
                f++;
```

args[3] = “si”
args[4] = “0123456S”
args[5]=“ sysinit”
args[6]=“/etc/rc.d/rc.sysinit”
args[7]= NULL

在命令行上查找以“~`!$^&*()=|\\{}[];\"'<>?”中任何一个字符匹配的第一个字符的位置

运行/bin/sh –c exec /etc/rc.d/rc.sysinit

```
                *ptr = 0;
                break;
            }
            /* End word with \0 and continue */
            *ptr++ = 0;
        }
        args[f] = NULL;
    }
    args[0] = args[1];
    while(1) {
        /*
         *    Block sigchild while forking.
         */
        sigemptyset(&nmask);
        sigaddset(&nmask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &nmask, &omask);

        按 daemon 进程方式来启动进程


        if ((pid = fork()) == 0) {    init process 用它的子进程来运行启动的命令

            close(0);         关闭 file handle 0,1,2, 即 STDIN, STDOUT, STDERR
            close(1);
            close(2);
            if (pipe_fd >= 0) close(pipe_fd);

            sigprocmask(SIG_SETMASK, &omask, NULL);

            /*
             *    In sysinit, boot, bootwait or single user mode:
             *    for any wait-type subprocess we _force_ the console
             *    to be its controlling tty.
             */
            if (strchr("*#sS", runlevel) && ch->flags & WAITING) {
                /*
                 *    We fork once extra. This is so that we can
```

```
959                      *    wait and change the process group and session
960                      *    of the console after exit of the leader.
961                      */
962                     setsid();
963                     if ((f = console_open(O_RDWR|O_NOCTTY)) >= 0) {
964                           /* Take over controlling tty by force */
965                           (void)ioctl(f, TIOCSCTTY, 1);
966                           dup(f);
967                           dup(f);
968                     }
969                     if ((pid = fork()) < 0) {     fork 失败，则退出，原始 init process 当然不会退出，这里退出的
970                           initlog(L_VB, "cannot fork");       是其子进程，该进程由 L942 的 fork()产生
971                           exit(1);
972                     }
973                     if (pid > 0) {    父进程等待子进程（通过 waitpid 系统调用），也就是执行命令行的子进程返回
974                           /*
975                            *    Ignore keyboard signals etc.
976                            *    Then wait for child to exit.
977                            */
978                           SETSIG(sa, SIGINT, SIG_IGN, SA_RESTART);
979                           SETSIG(sa, SIGTSTP, SIG_IGN, SA_RESTART);
980                           SETSIG(sa, SIGQUIT, SIG_IGN, SA_RESTART);
981                           SETSIG(sa, SIGCHLD, SIG_DFL, SA_RESTART);
982
983                           while ((rc = waitpid(pid, &st, 0)) != pid)
984                                 if (rc < 0 && errno == ECHILD)
985                                       break;
986
987                           /*
988                            *    Small optimization. See if stealing
989                            *    controlling tty back is needed.
990                            */
991                           pgrp = tcgetpgrp(f);
992                           if (pgrp != getpid())
993                                 exit(0);
994
995                           /*
996                            *    Steal controlling tty away. We do
997                            *    this with a temporary process.
998                            */
999                           if ((pid = fork()) < 0) {
1000                                initlog(L_VB, "cannot fork");
1001                                exit(1);
1002                          }
1003                          if (pid == 0) {
1004                                setsid();
1005                                (void)ioctl(f, TIOCSCTTY, 1);
1006                                exit(0);
1007                          }
1008                          while((rc = waitpid(pid, &st, 0)) != pid)
1009                                if (rc < 0 && errno == ECHILD)
1010                                      break;
1011                          exit(0);
1012                    }
1013
1014                    /* Set ioctl settings to default ones */
1015                    console_stty();
1016
1017              } else {    子进程，命令行将在该 process 的 context 内执行
1018                    setsid();
1019                    if ((f = console_open(O_RDWR|O_NOCTTY)) < 0) {
1020                          initlog(L_VB, "open(%s): %s", console_dev,
1021                                strerror(errno));
1022                          f = open("/dev/null", O_RDWR);
1023                    }
1024                    dup(f);     由于在 L944，L945，L946 上关闭了 0,1,2 的 handle，所以自然子进程也继承了父进程的
1025                    dup(f);     handle，即这时候没有 0，1，2 handle。这里打开 system console 作为新的 1,2,3
                                    Handle。这样当调用 console_open()打开/dev/console 时返回的 f 是 0，而 L1024 则置
                                    当前 process 的 1 handle 为/dev/console，L1025 则置当前 process 的 2 handle 为
                                    /dev/console。这样 STDIN(标准输入)，STDOUT（标准输出），STDERR（标准错误输出）都
                                    被正确设置。

                                    注意点：
```

在 Unix 环境下，启动任何程序后，其默认的前 3 个 handle(0,1,2)都已经设定，它的根源就在这里。

```
                }

                /* Reset all the signals, set up environment */
                for(f = 1; f < NSIG; f++) SETSIG(sa, f, SIG_DFL, SA_RESTART);
                                                    把该 process 的 signal 都设为忽略

                environ = init_buildenv(1);          建立命令行执行的环境

                /*
                 *    Execute prog. In case of ENOEXEC try again
                 *    as a shell script.
                 */
                execvp(args[1], args + 1);           执行命令的调用
                if (errno == ENOEXEC) {
                        args[1] = SHELL;
                        args[2] = "-c";
                        strcpy(buf, "exec ");
                        strncat(buf, proc, sizeof(buf) - strlen(buf) - 1);
                        args[3] = buf;
                        args[4] = NULL;
                        execvp(args[1], args + 1);
                }
                initlog(L_VB, "cannot execute \"%s\"", args[1]);
                exit(1);
        }
        *res = pid;       把运行的命令的进程的 pid 返回给 init process
        sigprocmask(SIG_SETMASK, &omask, NULL);

        INITDBG(L_VB, "Started id %s (pid %d)", ch->id, pid);

        if (pid == -1) {
                initlog(L_VB, "cannot fork, retry..");
                do_sleep(5);
```

```
            continue;
      }
      return(pid);
  }
}
```

check_init_fifo函数

```
/*
 *    Read from the init FIFO. Processes like telnetd and rlogind can
 *    ask us to create login processes on their behalf.
 *
 *    FIXME:      this needs to be finished. NOT that it is buggy, but we need
 *          to add the telnetd/rlogind stuff so people can start using it.
 *          Maybe move to using an AF_UNIX socket so we can use
 *          the 2.2 kernel credential stuff to see who we're talking to.
 *
 */
void check_init_fifo(void)
{
  struct init_request   request;
  struct timeval  tv;
  struct stat           st, st2;
  fd_set          fds;
  int             n;
  int             quit = 0;

  /*
   *  First, try to create /dev/initctl if not present.
   */
  if (stat(INIT_FIFO, &st2) < 0 && errno == ENOENT)         如果没有建立命名管道(name pipe)/dev/initctl,
      (void)mkfifo(INIT_FIFO, 0600);                        则建立之，只允许 root 用户读写

  /*
   *  If /dev/initctl is open, stat the file to see if it
   *  is still the _same_ inode.
```

```
2019     */
2020    if (pipe_fd >= 0) {         pipe_fd 是管道/dev/initctl 的 file handle，如果 >= 0 表示已被 open
2021        fstat(pipe_fd, &st);
2022        if (stat(INIT_FIFO, &st2) < 0 ||
2023            st.st_dev != st2.st_dev ||      比较此管道是否是原始打开时的管道（通过 ino 来比较）
2024            st.st_ino != st2.st_ino) {
2025              close(pipe_fd);               如果不一致，则关闭之，这样下面的代码将再次 open 该 pipe
2026              pipe_fd = -1;                 此赋值，会造成满足 L2033 的 if 分支
2027        }
2028    }
2029
2030    /*
2031     *   Now finally try to open /dev/initctl
2032     */
2033    if (pipe_fd < 0) {                                                      pipe_fd 为/dev/initctl 的 file handle，-1
2034        if ((pipe_fd = open(INIT_FIFO, O_RDWR|O_NONBLOCK)) >= 0) {          表示还没有打开，这里非阻塞打开
2035              fstat(pipe_fd, &st);
2036              if (!S_ISFIFO(st.st_mode)) {
2037                    initlog(L_VB, "%s is not a fifo", INIT_FIFO);
2038                    close(pipe_fd);
2039                    pipe_fd = -1;
2040              }
2041        }
2042        if (pipe_fd >= 0) {
2043              /*
2044               *    Don't use fd's 0, 1 or 2.
2045               */
2046              (void) dup2(pipe_fd, PIPE_FD);      把/dev/initctl 的 file handle 设为 PIPE_FD（10）
2047              close(pipe_fd);
2048              pipe_fd = PIPE_FD;                  使得/dev/initctl 管道的 file handle 为 10(PIPE_FD)
2049
2050              /*
2051               *    Return to caller - we'll be back later.
2052               */
2053        }
2054    }
```

到了这里，/dev/initctl 管道应该被正确打开

```
/* Wait for data to appear, _if_ the pipe was opened. */
if (pipe_fd >= 0) while(!quit) {

    /* Do select, return on EINTR. */
    FD_ZERO(&fds);
    FD_SET(pipe_fd, &fds);
    tv.tv_sec = 5;                设 select 调用在该管道上最多阻塞 5 秒(timeout 为 5)
    tv.tv_usec = 0;
    n = select(pipe_fd + 1, &fds, NULL, NULL, &tv);       通过 select 调用来等待/dev/initctl 的输入，如有
    if (n <= 0) {                                          输入则马上返回，否则就阻塞次调用
          if (n == 0 || errno == EINTR) return;            即 init  3 等待 init  2 发来的 request
          continue;
    }

```

从 select 调用返回，表示有进程往/dev/initctl 管道写东西，实际上就是用户通过运行 init X 把要切换到 X run level 的 request 写入该管道，造成 select 调用返回

```
    /* Read the data, return on EINTR. */
    n = read(pipe_fd, &request, sizeof(request));
```

读取写入/dev/initctl 中的 request，该 request 的格式如下：

```
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;
};

        if (n == 0) {
              /*
               *    End of file. This can't happen under Linux (because
               *    the pipe is opened O_RDWR - see select() in the
               *    kernel) but you never know...
               */
              close(pipe_fd);
              pipe_fd = -1;
              return;
        }
        if (n <= 0) {
              if (errno == EINTR) return;         被 signal 打断了 select（）系统调用的等待，所以继续
              initlog(L_VB, "error reading initrequest");
              continue;
        }

```

正常情况下，运行到这儿，表示确有 request 被写入/dev/initctl

```
        /*
         *    This is a convenient point to also try to
         *    find the console device or check if it changed.
         */
        console_init();

        /*
         *    Process request.
         */
        if (request.magic != INIT_MAGIC || n != sizeof(request)) {  检查被写入的 request 的格式的合法性
              initlog(L_VB, "got bogus initrequest");
              continue;
        }
        switch(request.cmd) {
```

输入的 request 的格式合法，则判断要采取什么动作。即 init 2 可以向 daemon 进程的 init 3 发送的命令列表。具体 init 2 向 Init 2 发 request 的代码在 telinit()中，代码分析见对 init 2 的代码分析。

下面的全局变量 sltime 用来设定 SIGTERM 与 SIGKILL 间的间隔。当 init process 要杀死某个其管理的 process 时，先发送 SIGTERM，然后等待 sltime 秒（默认为 5 秒），再发送 SIGKILL。

```
        case INIT_CMD_RUNLVL:                 切换到新的 run level
              sltime = request.sleeptime;           取得等待时间
              fifo_new_level(request.runlevel);     设置新的 run level，通过重新读取 inittab 文件来
                                                    启动与新 run level 匹配的命令脚本
              quit = 1;
              break;
        case INIT_CMD_POWERFAIL:
              sltime = request.sleeptime;
              do_power_fail('F');
              quit = 1;
              break;
        case INIT_CMD_POWERFAILNOW:
              sltime = request.sleeptime;
              do_power_fail('L');
              quit = 1;
              break;
        case INIT_CMD_POWEROK:
              sltime = request.sleeptime;
              do_power_fail('O');
              quit = 1;
              break;
```

上面 3 个 request 都与 UPS 发来的与电源 event 有关。其处理代码都是 do_power_fail()，只不过通过不同的参数来区分状态

```
        case INIT_CMD_SETENV:
              initcmd_setenv(request.i.data, sizeof(request.i.data));
              break;
        case INIT_CMD_CHANGECONS:
              if (user_console) {
                    free(user_console);
```

```
                        user_console = NULL;
                  }
                  if (!request.i.bsd.reserved[0])
                        user_console = NULL;
                  else
                        user_console = strdup(request.i.bsd.reserved);
                  console_init();
                  quit = 1;
                  break;
            default:
                  initlog(L_VB, "got unimplemented initrequest.");
                  break;
      }
  }

  /*
   *  We come here if the pipe couldn't be opened.
   */
  if (pipe_fd < 0) pause();

}
```

init 2 的运行

init 2 是指具有 root 权限的用户通过运行 init 来切换运行级别，或设置一些 init 3(daemon 进程)在运行当中的参数，比如指定 init 在向 process 发 SIGTERM 与 SIGKILL 之间的间隔秒数(sltime)及 init 启动程序时传递的环境。

还是从 main()开始分析。

```
int main(int argc, char **argv)
{
      char              *p;
      int               f;
      int               isinit;

      /* Get my own name */
```

```
        if ((p = strrchr(argv[0], '/')) != NULL)
                p++;
        else
                p = argv[0];
        umask(022);                    argv[0] = /sbin/init，则p指向init

        /* Quick check */
        if (geteuid() != 0) {          检查是否拥有root权限，运行init必须拥有该权限
                fprintf(stderr, "%s: must be superuser.\n", p);
                exit(1);
        }

        /*
         *     Is this telinit or init ?
         */
        isinit = (getpid() == 1);      这时启动的init process(init 2)的pid当然不可能是1（1正被init 2给占着
                                       呢），所以isinit = 0
        for (f = 1; f < argc; f++) {   init 2可以模拟init 1的执行，就是通过传递命令行参数-i或–init，在手册上
                                       没有提到这一点
                if (!strcmp(argv[f], "-i") || !strcmp(argv[f], "--init"))
                        isinit = 1;
                        break;
        }
        if (!isinit) exit(telinit(p, argc, argv));      这时isinit = 0，所以init 2的执行进入telinit()的执
                                                        行
        /*
         *     Check for re-exec
         */

。。。。。。
```

telinit()也是init 2运行的主要函数。这里的参数programe为init process的可执行文件名“init”。

```
int telinit(char *progname, int argc, char **argv)
```

```
{
#ifdef TELINIT_USES_INITLVL
        FILE                *fp;
#endif
        struct init_request      request;
```

init 2 于 init 3 的沟通通过“/dev/initctl”这个有名 pipe（其由 init 3 创建），在该 pipe 中传递的即是 init 2 对 init 3 提出的动作要求。即 init 2 本身不干任何事情，只是把用户的请求发送给 init 3 来处理。结构 init_request 就是“动作请求”的格式

```
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;
};
```

```
        struct sigaction    sa;
        int                 f, fd, l;
        char                *env = NULL;

        memset(&request, 0, sizeof(request));
        request.magic       = INIT_MAGIC;               请求的签名

```

下面就是根据命令行上的参数来构建 request

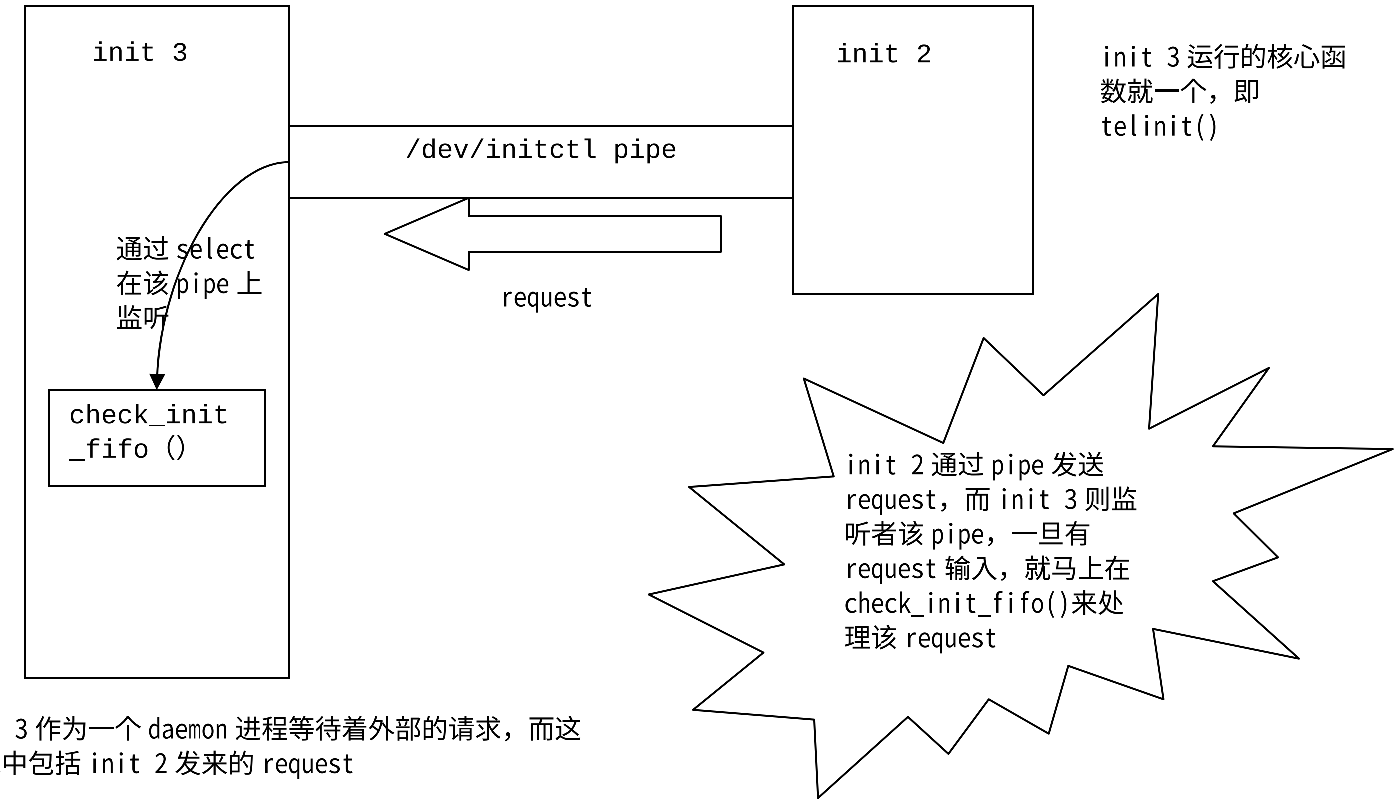
```
        while ((f = getopt(argc, argv, "t:e:")) != EOF) switch(f) {
                case 't':         t 参数指定 init 在向 process 发 SIGTERM 与 SIGKILL 之间的间隔秒数(sltime)
                        sltime = atoi(optarg);
                        break;
                case 'e':         手册中没有描述该参数，但看代码是用户可以通过该参数设置环境变量
```

```
2520                            if (env == NULL)
2521                                    env = request.i.data;
2522                            l = strlen(optarg);
2523                            if (env + l + 2 > request.i.data + sizeof(request.i.data)) {
2524                                    fprintf(stderr, "%s: -e option data "
2525                                            "too large\n", progname);
2526                                    exit(1);
2527                            }
2528                            memcpy(env, optarg, l);
2529                            env += l;
2530                            *env++ = 0;
2531                            break;
2532                    default:
2533                            usage(progname);
2534                            break;
2535            }
2536
2537            if (env) *env++ = 0;
2538
2539            if (env) {
2540                    if (argc != optind)
2541                            usage(progname);
2542                    request.cmd = INIT_CMD_SETENV;
2543            } else {
2544                    if (argc - optind != 1 || strlen(argv[optind]) != 1)
2545                            usage(progname);
2546                    if (!strchr("0123456789SsQqAaBbCcUu", argv[optind][0]))
2547                            usage(progname);
2548                    request.cmd = INIT_CMD_RUNLVL;            处理该 request 的代码在 check_init_fifo () 中(L2102)
2549                    request.runlevel  = env ? 0 : argv[optind][0];  这里的 request.runlevel 即要切换过去的 run level
2550                    request.sleeptime = sltime;
2551            }
2552
2553            /* Open the fifo and write a command. */
2554            /* Make sure we don't hang on opening /dev/initctl */
2555            SETSIG(sa, SIGALRM, signal_handler, 0);
2556            alarm(3);         设置 alarm，即 3 秒后本进程收到 SIGALRM signal
```

```
2557       if ((fd = open(INIT_FIFO, O_WRONLY)) >= 0 &&    打开有名管道“/dev/initctl”来写
2558           write(fd, &request, sizeof(request)) == sizeof(request)) {    写入该 request
2559             close(fd);
2560             alarm(0);          取消原先设置的 alarm 即上面 line 2556 行的 alarm(3)
                                  Line 2556 与 Line 2560 是为了保证对有名管道/dev/initctl 的写如果正常的话，肯定应该
2561             return 0;          小于 3 秒。如果超过 3 秒，则本进程会收到 SIGALRM signal，则在下面 Line 2585 中的
2562       }                        ISMEMBER（）将返回 true,则报错
2563
2564 #ifdef TELINIT_USES_INITLVL
2565       if (request.cmd == INIT_CMD_RUNLVL) {
2566             /* Fallthrough to the old method. */
2567
2568             /* Now write the new runlevel. */
2569             if ((fp = fopen(INITLVL, "w")) == NULL) {       打开/etc/initrunlvl 文件供写
2570                   fprintf(stderr, "%s: cannot create %s\n",
2571                         progname, INITLVL);
2572                   exit(1);
2573             }
2574             fprintf(fp, "%s %d", argv[optind], sltime);
2575             fclose(fp);
2576
2577             /* And tell init about the pending runlevel change. */
2578             if (kill(INITPID, SIGHUP) < 0) perror(progname);      向 1 号进程发 SIGHUP signal，也就是让
2579                                                                  init 2 重新读取/etc/inittab

2580             return 0;
2581       }
2582 #endif
2583
2584       fprintf(stderr, "%s: ", progname);
2585       if (ISMEMBER(got_signals, SIGALRM)) {
2586             fprintf(stderr, "timeout opening/writing control channel %s\n",
2587                   INIT_FIFO);
2588       } else {
2589             perror(INIT_FIFO);
2590       }
```

```
        return 1;
}
```

下图可以表示 init 2 与 init 3 之间的关系。



init 3 运行的核心函数就一个，即 telinit()

init 2 通过 pipe 发送 request，而 init 3 则监听者该 pipe，一旦有 request 输入，就马上在 check_init_fifo()来处理该 request

init 3 作为一个 daemon 进程等待着外部的请求，而这请求中包括 init 2 发来的 request

init 3 的运行

init 3 作为一个 daemon process 运行，它其实与 init 1 是完全一体的，即 init 1 完成系统用户态的启动，然后 init process 就进入类似 Client-Server 架构中的 server process 的运行。它本身不会主动的发起新的动作，而是当有 init process 关注的事件时，它才会被动的响应这些动作。我这里把原本连为一体的 init process 分成 init 1 与 init 3，纯粹是为了说明清楚的缘故。

主流程分析

当 init 1 结束后，即进入下面的一个无限循环。

```
/*
 *    The main loop
 */
int init_main()
{

。。。。。。    （这里省略的我把它归类为 init 1）


  while(1) {      进入无限循环，init process 以后就在该无限循环中打转，永不出来，除非关机

     /* See if we need to make the boot transitions. */
     boot_transitions();
     INITDBG(L_VB, "init_main: waiting..");

     /* Check if there are processes to be waited on. */
     for(ch = family; ch; ch = ch->next)              枚举当前 init process 管理的进程
      if ((ch->flags & RUNNING) && ch->action != BOOT) break;     如果发觉有进程处于运行状态则马上退出枚举
```

```

#if CHANGE_WAIT
     /* Wait until we get hit by some signal. */
     while (ch != NULL && got_signals == 0) {
      if (ISMEMBER(got_signals, SIGHUP)) {
            /* See if there are processes to be waited on. */
            for(ch = family; ch; ch = ch->next)
                  if (ch->flags & WAITING) break;
      }
      if (ch != NULL) check_init_fifo();
     }
#else /* CHANGE_WAIT */
     if (ch != NULL && got_signals == 0) check_init_fifo();
#endif /* CHANGE_WAIT */
```

check_init_fifo（）的注解见“init 1”时的解释。这里是 init 3 检查是否有来自 init 2 的 request。

```

     /* Check the 'failing' flags */
     fail_check();

     /* Process any signals. */
     process_signals();
```

在 init 运行期间，其可能会收到关心的 signal。在 signal handler 中只是被纪录下来，而真正的处理则在该函数中。该函数注解见“init 1”的说明

```
     /* See what we need to start up (again) */
     start_if_needed();
```

该函数注解见“init 1”的说明

```
  }
  /*NOTREACHED*/
}
```

init process 等候着其关心的事件（通过 signal 的方式来通知），主要有如下几个事件

- 其通过 select 系统调用等候在管道**/dev/initctl** 的一端，一旦有进程往另一端写入希望 init process 执行的 request，init 3 就会分析该 request，然后执行要求的动作（见 check_init_fifo 函数注释）。
- 当有 signal 发送给 init process 时，init 在 signal handler 中并不马上处理该 signal，而仅仅是在全局变量 got_signals 中置一下该 signal 发生过的标志，真正的处理是在 init 3 的循环中的 process_signals()中。

设置 init process 的信号(signal)处理器。

```
  SETSIG(sa, SIGALRM,  signal_handler, 0);
  SETSIG(sa, SIGHUP,   signal_handler, 0);
  SETSIG(sa, SIGINT,   signal_handler, 0);
  SETSIG(sa, SIGPWR,   signal_handler, 0);
  SETSIG(sa, SIGWINCH, signal_handler, 0);
  SETSIG(sa, SIGUSR1,  signal_handler, 0);
```

在 init 1 中把 HUP PWR WINCH ALRM INT signal 的处理器设置好。当 signal 发生时，只是简单的置一下位，真正的对 signal 的处理是在函数 process_signals()中。

```
/*
 *    We got a signal (HUP PWR WINCH ALRM INT)
 */
void signal_handler(int sig)
{
      ADDSET(got_signals, sig);
}
```

而 ADDSET 是个 macro。简单设个标志位了事。

```
#define ADDSET(set, val)   ((set) |=  (1 << (val)))
```

在 process_signals()中检查哪些标志位置位，然后依次处理。

```
  if (ISMEMBER(got_signals, SIGPWR)) {          ISMEMBER 是个 macro，定义如下
      INITDBG(L_VB, "got SIGPWR");                    #define ISMEMBER(set, val) ((set) & (1 << (val)))
      /* See _what_ kind of SIGPWR this is. */  用于检查某个 signal 是否被触发
      pwrstat = 0;
      if ((fd = open(PWRSTAT, O_RDONLY)) >= 0) {
            c = 0;
            read(fd, &c, 1);                    电源 fail 的处理
            pwrstat = c;
            close(fd);
            unlink(PWRSTAT);
      }
      do_power_fail(pwrstat);
      DELSET(got_signals, SIGPWR);              处理完后把标志清调
  }

  if (ISMEMBER(got_signals, SIGINT)) {          当用户按了 Ctrl+Alt+Del 后，内核将发送 SIGINT signal 给 init
      INITDBG(L_VB, "got SIGINT");              process
```

```
        /* Tell ctrlaltdel entry to start up */
        for(ch = family; ch; ch = ch->next)
              if (ch->action == CTRLALTDEL)
                    ch->flags &= ~XECUTED;        去掉 disable 执行的标志
        DELSET(got_signals, SIGINT);
    }

    if (ISMEMBER(got_signals, SIGWINCH)) {
        INITDBG(L_VB, "got SIGWINCH");
        /* Tell kbrequest entry to start up */
        for(ch = family; ch; ch = ch->next)
              if (ch->action == KBREQUEST)
                    ch->flags &= ~XECUTED;
        DELSET(got_signals, SIGWINCH);
    }

    if (ISMEMBER(got_signals, SIGALRM)) {
        INITDBG(L_VB, "got SIGALRM");
        /* The timer went off: check it out */
        DELSET(got_signals, SIGALRM);
    }

    if (ISMEMBER(got_signals, SIGCHLD)) {
        INITDBG(L_VB, "got SIGCHLD");
        /* First set flag to 0 */
        DELSET(got_signals, SIGCHLD);

        /* See which child this was */
        for(ch = family; ch; ch = ch->next)
            if (ch->flags & ZOMBIE) {
              INITDBG(L_VB, "Child died, PID= %d", ch->pid);
              ch->flags &= ~(RUNNING|ZOMBIE|WAITING);
              if (ch->process[0] != '+')
                    write_utmp_wtmp("", ch->id, ch->pid, DEAD_PROCESS, NULL);
            }

    }
```

```
  if (ISMEMBER(got_signals, SIGHUP)) {
      INITDBG(L_VB, "got SIGHUP");
#if CHANGE_WAIT
      /* Are we waiting for a child? */
      for(ch = family; ch; ch = ch->next)
            if (ch->flags & WAITING) break;
      if (ch == NULL)
#endif
      {
            /* We need to go into a new runlevel */
            oldlevel = runlevel;
#ifdef INITLVL
            runlevel = read_level(0);
#endif
            if (runlevel == 'U') {
                  runlevel = oldlevel;
                  re_exec();
            } else {
                  if (oldlevel != 'S' && runlevel == 'S') console_stty();
                  if (runlevel == '6' || runlevel == '0' ||
                      runlevel == '1') console_stty();
                  read_inittab();
                  fail_cancel();
                  setproctitle("init [%c]", runlevel);
                  DELSET(got_signals, SIGHUP);
            }
      }
  }
  if (ISMEMBER(got_signals, SIGUSR1)) {
      /*
       *    SIGUSR1 means close and reopen /dev/initctl
       */
      INITDBG(L_VB, "got SIGUSR1");
      close(pipe_fd);
      pipe_fd = -1;
      DELSET(got_signals, SIGUSR1);
  }
```

■ 当受到 UPS 发来的电源 fail signal 时

```
if ((fd = open(PWRSTAT, O_RDONLY)) >= 0) {
      c = 0;
      read(fd, &c, 1);
      pwrstat = c;
      close(fd);
      unlink(PWRSTAT);
}
do_power_fail(pwrstat);
```

首先从/etc/powerstatus 获得原因，“F”，“L”，“O”，分别表示电源 Fail，电源 Low，电源 Ok(恢复)。

参数 pwrstat 为 powerfail 原因，根据原因来启动/etc/inittab 中要求当对因事件发生时要执行的动作
参数 pwrstat 反映了电源（power）的状态
“O”表示电源 OK
“L”表示电源 Low
“F” 表示电源 Fail（故障）

```
/*
 *    Start up powerfail entries.
 */
void do_power_fail(int pwrstat)
{
      CHILD *ch;

      /*
       *    Tell powerwait & powerfail entries to start up
       */
      for (ch = family; ch; ch = ch->next) {    枚举 family 链表
            if (pwrstat == 'O') {               在收到电源恢复（Ok）的 signal 后
                  /*
                   *    The power is OK again.
                   */
```

```
                        if (ch->action == POWEROKWAIT)        XECUTED 是 disable 该 process 运行
                                ch->flags &= ~XECUTED;        清除 disable 标志，即属性为 POWEROKWAIT 的 process 可以
                                                              运行了
                } else if (pwrstat == 'L') {                  在收到电源 Low 的 signal 后
                        /*
                         *      Low battery, shut down now.
                         */
                        if (ch->action == POWERFAILNOW)
                                ch->flags &= ~XECUTED;        允许属性为 POWERFAILNOW 的 process 的运行
                } else {
                        /*
                         *      Power is failing, shutdown imminent        在收到电源 Fail 的 signal 后
                         */
                        if (ch->action == POWERFAIL || ch->action == POWERWAIT)
                                ch->flags &= ~XECUTED;        允许属性为 POWERFAIL 与 POWERWAIT 的 process 的运行
                }
        }
}
```

对应到例子文件中

```
# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
```

也就是在默认情况下/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"（属性为 POWERFAIL）与 /sbin/shutdown -c "Power Restored; Shutdown Cancelled"(属性为 POWEROKWAIT)都在 init process 管理的进程链表中是被 disable 的，即置了 XECUTED 标志的，当收到对应的消息后，会 enable 它。当然这里只是打开了允许他们运行，并没有真正的运行。真正的运行在下面的 start_if_needed()中。

```
        /* Process any signals. */
        process_signals();          处理被记录下来的 signal
```

```

       /* See what we need to start up (again) */
       start_if_needed();       有可能在 family 链表中的 node 状态已经改变，所以重新枚举整个链表，看是否有本来不能运行
    }                           的动作(action)可以执行了
```

- 当用户按了中断键（Delete 键或 Ctrl-C），则会向前台 process group 发 SIGINT signal。

```
        /* Tell ctrlaltdel entry to start up */
        for(ch = family; ch; ch = ch->next)
              if (ch->action == CTRLALTDEL)
                    ch->flags &= ~XECUTED;  允许 Ctrl-Alt-Del handler 的运行
```

- 按《UNIX 环境高级编程》上的说法如下：

当有进程通过 ioctl 接口来改变终端窗口大小时会发送该消息。

```
        /* Tell kbrequest entry to start up */
        for(ch = family; ch; ch = ch->next)
              if (ch->action == KBREQUEST)
                    ch->flags &= ~XECUTED;  允许属性为 KBREQUEST 的 process 运行
```

- 对 timeout 的 alarm，几乎什么都不干

```
        INITDBG(L_VB, "got SIGALRM");
        /* The timer went off: check it out */
        DELSET(got_signals, SIGALRM);
```

- 当收到有子进程死亡时，要在 utmp 和 wtmp 文件中记上一笔。

```
        /* See which child this was */
        for(ch = family; ch; ch = ch->next)       对 family 链表进行枚举，如果发现僵尸，则清除三个标志
            if (ch->flags & ZOMBIE) {
              INITDBG(L_VB, "Child died, PID= %d", ch->pid);
              ch->flags &= ~(RUNNING|ZOMBIE|WAITING);
              if (ch->process[0] != '+')
                    write_utmp_wtmp("", ch->id, ch->pid, DEAD_PROCESS, NULL);
            }
```

- 当用户如果修改了/etc/inittab，希望不用重新启动就生效（在正常情况下，init process 只在启统启动阶段才会读取该配置文件），则可以发 SIGHUP signal，让 init process 重读 inittab，并根据新的配置文件来运行。

```
#if CHANGE_WAIT
        /* Are we waiting for a child? */
        for(ch = family; ch; ch = ch->next)          如果这时有 process 处于必须等待其结束的状态，则此时不能
                if (ch->flags & WAITING) break;      执行重读 inittab 文件的操作
        if (ch == NULL)        如果有 process 处于必须等待其结束的状态，则 ch 就不会为 NULL，即如果为 NULL，表示此
#endif                         时重新读取 inittab 文件是安全的
        {
                /* We need to go into a new runlevel */
                oldlevel = runlevel;        把当前的 run level 存入 oldlevel 中
#ifdef INITLVL
                runlevel = read_level(0);   获得当前 inittab 文件中的 run level
#endif
                if (runlevel == 'U') {      按照 init 手册的说法如下“U or u tell  init to re-execute itself
                        runlevel = oldlevel;  (preserving the state). No re-examining of /etc/inittab
                        re_exec();            file happens”，其中 re_exec()即时重新启动一遍 init 管理的链表中的
                                              Process
                } else {
                        if (oldlevel != 'S' && runlevel == 'S') console_stty();
                        if (runlevel == '6' || runlevel == '0' ||
                          runlevel == '1') console_stty();
                        read_inittab();       我们在分析 init 1 时说过，read_inittab()中有部分分支是不在系统时执行
                        fail_cancel();        的，是在 init 3 时执行的。
                        setproctitle("init [%c]", runlevel);
```

- 当 init process 收到 SIGUSR1 signal(用户自定义信号)后，会关闭/dev/initctl pipe。

```
close(pipe_fd);         init 在接受到该 signal 后，init 关闭和重新打开/dev/initctl
pipe_fd = -1;
```

辅助函数

fail_check函数

```
/*
 *    This procedure is called after every signal (SIGHUP, SIGALRM..)
 *
 *    Only clear the 'failing' flag if the process is sleeping
 *    longer than 5 minutes, or inittab was read again due
 *    to user interaction.
 */
void fail_check(void)
{
      CHILD *ch;                    /* Pointer to child structure */
      time_t      t;                /* System time */
      time_t      next_alarm = 0;   /* When to set next alarm */

      time(&t);         取得当前系统时间

      for(ch = family; ch; ch = ch->next) {     枚举 init process 管理的进程

            if (ch->flags & FAILING) {          枚举整个 family 链表检查状态为 fail 的 node
                  /* Can we free this sucker? */
                  if (ch->tm + SLEEPTIME < t) {
                        ch->flags &= ~FAILING;
                        ch->count = 0;
                        ch->tm = 0;
                  } else {
                        /* No, we'll look again later */
                        if (next_alarm == 0 ||
                            ch->tm + SLEEPTIME > next_alarm)
                              next_alarm = ch->tm + SLEEPTIME;
                  }
            }
      }
      if (next_alarm) {
            next_alarm -= t;
            if (next_alarm < 1) next_alarm = 1;
            alarm(next_alarm);
      }
}
```

re_exec函数

```
/*
 *     Attempt to re-exec.
 */
void re_exec(void)
{
      CHILD       *ch;
      sigset_t    mask, oldset;
      pid_t       pid;
      char        **env;
      int         fd;

      if (strchr("S12345",runlevel) == NULL)    这两行对应 telinit 手册中的说明“Run level should be one of
            return;                             Ss12345,otherwise request would be silently ignored“

      /*
       *     Reset the alarm, and block all signals.
       */
      alarm(0);                     取消所有 alarm signal
      sigfillset(&mask);
      sigprocmask(SIG_BLOCK, &mask, &oldset);

      /*
       *     construct a pipe fd --> STATE_PIPE and write a signature
       */
      fd = make_pipe(STATE_PIPE);

```

```
/*
 * It's a backup day today, so I'm pissed off.  Being a BOFH, however,
 * does have it's advantages...
 */
fail_cancel();
close(pipe_fd);
pipe_fd = -1;
DELSET(got_signals, SIGCHLD);
DELSET(got_signals, SIGHUP);
DELSET(got_signals, SIGUSR1);

/*
 *    That should be cleaned.
 */
for(ch = family; ch; ch = ch->next)       对 init process 管理的进程链表进行枚举
    if (ch->flags & ZOMBIE) {
      INITDBG(L_VB, "Child died, PID= %d", ch->pid);
      ch->flags &= ~(RUNNING|ZOMBIE|WAITING);
      if (ch->process[0] != '+')
            write_utmp_wtmp("", ch->id, ch->pid, DEAD_PROCESS, NULL);
    }

if ((pid = fork()) == 0) {
      /*
       *    Child sends state information to the parent.
       */
      send_state(fd);         由 init 的子进程往 state pipe 中写入当前 init process 管理的 process 的状态
      exit(0);
}

/*
 *    The existing init process execs a new init binary.
 */
env = init_buildenv(0);
execl(myname, myname, "--init", NULL, env);     原有的 init process 执行/sbin/init，环境用
                                                init_buildenv()构造的，而非父进程
      /*
       *    We shouldn't be here, something failed.
       *    Bitch, close the state pipe, unblock signals and return.
       */
      close(fd);
      close(STATE_PIPE);
      sigprocmask(SIG_SETMASK, &oldset, NULL);
      init_freeenv(env);
      initlog(L_CO, "Attempt to re-exec failed");
}
```

read_inittab函数在init 3 时的分析

```
void read_inittab(void)
{
  FILE            *fp;              /* The INITTAB file */
  CHILD           *ch, *old, *i;    /* Pointers to CHILD structure */
  CHILD           *head = NULL;     /* Head of linked list */
#ifdef INITLVL
  struct stat     st;               /* To stat INITLVL */
#endif
  sigset_t  nmask, omask;           /* For blocking SIGCHLD. */
  char            buf[256];         /* Line buffer */
  char            err[64];          /* Error message. */
  char            *id, *rlevel,
            *action, *process;      /* Fields of a line */
  char            *p;
  int       lineNo = 0;       /* Line number in INITTAB file */
  int       actionNo;         /* Decoded action field */
  int       f;                /* Counter */
  int       round;            /* round 0 for SIGTERM, 1 for SIGKILL */
  int       foundOne = 0;     /* No killing no sleep */
  int       talk;             /* Talk to the user */
  int       done = 0;         /* Ready yet? */

#if DEBUG
  if (newFamily != NULL) {
      INITDBG(L_VB, "PANIC newFamily != NULL");
      exit(1);
  }
  INITDBG(L_VB, "Reading inittab");
#endif

  /*
   *  Open INITTAB and real line by line.
   */
  if ((fp = fopen(INITTAB, "r")) == NULL)              打开/etc/inittab 文件
      initlog(L_VB, "No inittab file found");

  while(!done) {                  每循环一次即处理 inittab 中一行，构造 newFamily 链表。注意是 newFamily 链表，不是
      /*                          family 链表
       *    Add single user shell entry at the end.
       */
      if (fp == NULL || fgets(buf, sizeof(buf), fp) == NULL) {
            done = 1;             已经处理完毕，退出循环
            /*
             *    See if we have a single user entry.
             */
            for(old = newFamily; old; old = old->next)           ？？？
                  if (strpbrk(old->rlevel, "S")) break;
            if (old == NULL)
                  snprintf(buf, sizeof(buf), "~~:S:wait:%s\n", SULOGIN);
            else
                  continue;
      }
      lineNo++;
      /*
       *    Skip comments and empty lines
       */
      for(p = buf; *p == ' ' || *p == '\t'; p++)       忽略前导空白字符
            ;
      if (*p == '#' || *p == '\n') continue;           以“＃”开头的为注释，忽略

```

```
        /*
         *    Decode the fields
         */
```

分解 id:runlevels:action:process 中的 4 部分

```
        id =      strsep(&p, ":");
        rlevel =  strsep(&p, ":");
        action =  strsep(&p, ":");
        process = strsep(&p, "\n");

```

由于文件中的配置行的各部分用“：”分割，所以这里通过 strsep 来分别提取各部分内容

从下面的代码可以看到在 init manual 中没有标明的限制，比如命令行的长度不能太长，超过 127 个字符等

```
        /*
         *    Check if syntax is OK. Be very verbose here, to
         *    avoid newbie postings on comp.os.linux.setup :)
         */
        err[0] = 0;
        if (!id || !*id) strcpy(err, "missing id field");
        if (!rlevel)     strcpy(err, "missing runlevel field");
        if (!process)    strcpy(err, "missing process field");
        if (!action || !*action)
                    strcpy(err, "missing action field");
        if (id && strlen(id) > sizeof(utproto.ut_id))
              sprintf(err, "id field too long (max %d characters)",
                    (int)sizeof(utproto.ut_id));
        if (rlevel && strlen(rlevel) > 11)
              strcpy(err, "rlevel field too long (max 11 characters)");
        if (process && strlen(process) > 127)
              strcpy(err, "process field too long");
        if (action && strlen(action) > 32)
              strcpy(err, "action field too long");
        if (err[0] != 0) {
              initlog(L_VB, "%s[%d]: %s", INITTAB, lineNo, err);
              INITDBG(L_VB, "%s:%s:%s:%s", id, rlevel, action, process);
              continue;
        }

        /*
```

```
 *    Decode the "action" field
 */
```

init 允许的 action 类型都记录在 actions[]数组中，这里通过比较字符串来把其转换成数字型 identifier

```
actionNo = -1;
for(f = 0; actions[f].name; f++)
      if (strcasecmp(action, actions[f].name) == 0) {
            actionNo = actions[f].act;
            break;
      }
if (actionNo == -1) {
```

碰到非法的 action(不在 actions[]数组中的)则忽略

```
      initlog(L_VB, "%s[%d]: %s: unknown action field",
            INITTAB, lineNo, action);
      continue;
}

/*
 *    See if the id field is unique
 */
```

配置行中的第一部分是所谓 identifier，必须唯一，但命名好像没有任何规定，可任意。已经处理过的配置行都被记录入 CHILD 的链表节点中，这里在处理当前行时检查一下已有节点中是否有与当前行的 id 相同的，如果有，则不是忽略该行，而是停止继续处理/etc/inittab 文件，可见 id 的唯一性是至关重要的

```
for(old = newFamily; old; old = old->next) {
      if(strcmp(old->id, id) == 0 && strcmp(id, "~~")) {
            initlog(L_VB, "%s[%d]: duplicate ID field \"%s\"",
                  INITTAB, lineNo, id);
            break;
      }
}
if (old) continue;

/*
 *    Allocate a CHILD structure
 */
ch = imalloc(sizeof(CHILD));
```

为当前配置行分配一个 CHILD node

```

	/*
	 *	And fill it in.         用当前配置行中的信息来填充 CHILD node
	 */
	ch->action = actionNo;          action 类型
	strncpy(ch->id, id, sizeof(utproto.ut_id) + 1); /* Hack for different libs. */该行的唯一标示符
	strncpy(ch->process, process, sizeof(ch->process) - 1);      该行是要执行的命令行
	if (rlevel[0]) {                填上 run level
		for(f = 0; f < sizeof(rlevel) - 1 && rlevel[f]; f++) {
			ch->rlevel[f] = rlevel[f];
			if (ch->rlevel[f] == 's') ch->rlevel[f] = 'S';
		}
		strncpy(ch->rlevel, rlevel, sizeof(ch->rlevel) - 1);
	} else {    如果没有写 run level，则表示所有 run level 都要执行该行的 process 部分
		strcpy(ch->rlevel, "0123456789");
		if (ISPOWER(ch->action))
			strcpy(ch->rlevel, "S0123456789");
	}
```

下面是对 action 的处理

```
	/*
	 *	We have the fake runlevel '#' for SYSINIT  and
	 *	'*' for BOOT and BOOTWAIT.
	 */
```

从上面的注释看，SYSINIT action 用'#'表示，而 BOOT action 用'*'表示。而真正合法的 run level 是 0 到 9 加上'S'
“＃”与“＊”表示在任何 run level 都要执行，另外 SYSINIT 的优先级是最高的，所以它应该比 BOOT 中的 action 先执行

```
	if (ch->action == SYSINIT) strcpy(ch->rlevel, "#");
	if (ch->action == BOOT || ch->action == BOOTWAIT)
		strcpy(ch->rlevel, "*");

	/*
	 *	Now add it to the linked list. Special for powerfail.
	 */
```

在从/etc/inittab 中读取配置行并生成的链表的头为 newFamily。如果是系统的启动阶段，family 所表示的链表自然为空，如果只是通过运行 init 来切换 run level 等，则 family 记录的链表非空，也就是当前 init 通过上次读取/etc/inittab 后生成的链表。

```
if (ISPOWER(ch->action)) {
```

如果 action 属于这几种（POWERWAIT，POWERFAIL，POWEROKWAIT，POWERFAILNOW，CTRLALTDEL），即与电源相关与用户按了 Ctrl+Alt+Del 键

```
      /*
       *    Disable by default
       */
      ch->flags |= XECUTED;
```

设置不执行标志。在 startup()种如果检测到该配置行所代表的 action 的 flag 设置了 XECUTED，则忽略对该行的处理。这可以理解，因为符合 ISPOWER()的 action 都不是在正常情况下需要运行的，只有对应的事件确实发生了，才需要执行。比如，如果用户从来没有按下过 Ctrl+Alt+Del 键，自然根本不需要执行/etc/inittab 中 CTRLALTDEL Action 指定的 process。所以在默认情况下它是 disable 的(通过设置 XECUTED 标志)。当检测到按下 Ctrl+Alt+Del 键后，才需要 enable。

并且上述 action 被插在 family 链表的前面，这样如果它们需要执行的话，将先得到执行。可以理解，因为这几个动作都比较严重，所以优先权较高

```
      /*
       *    Tricky: insert at the front of the list..
       */
      old = NULL;
      for(i = newFamily; i; i = i->next) {
            if (!ISPOWER(i->action)) break;
            old = i;
      }
      /*
       *    Now add after entry "old"
       */
      if (old) {
            ch->next = i;
            old->next = ch;
            if (i == NULL) head = ch;
      } else {
            ch->next = newFamily;
            newFamily = ch;
```

```
                if (ch->next == NULL) head = ch;
          }
    } else {  其他的 action 都插到尾部，KBREQUEST 默认是不执行的。从 init manual 上看 SIGWINCH  signal 会触发该动作
          /*
           *    Just add at end of the list
           */
          if (ch->action == KBREQUEST) ch->flags |= XECUTED;
          ch->next = NULL;
          if (head)
                head->next = ch;
          else
                newFamily = ch;
          head = ch;
    }

    /*
     *    Walk through the old list comparing id fields
     */
    for(old = family; old; old = old->next)
          if (strcmp(old->id, ch->id) == 0) {
                old->new = ch;
                break;
          }
    到这里处理一行结束
}
/*
 *  We're done.
 */
if (fp) fclose(fp);   关闭/etc/inittab 文件，对于 init 1，实际上基本上到此为止。
                      下面都是作为 daemon 进程的 init 即 init 3 的处理。我觉得整个代码应该整理得更清晰一点，由
                      kernel 启动 init(init 1)与作为 daemon 进程运行的 init (init 3) 的逻辑应该分开，而不要像现
                      在一样，绞和在一块，比较乱

/*
 *  Loop through the list of children, and see if they need to
 *  be killed.
```

```
1318   */
1319
1320  INITDBG(L_VB, "Checking for children to kill");
1321  for(round = 0; round < 2; round++) {
1322    talk = 1;
1323    for(ch = family; ch; ch = ch->next) {
1324      ch->flags &= ~KILLME;
1325
```

循环两遍，round 0 for SIGTERM, 1 for SIGKILL，即第一遍给要杀死的 process 发 SIGTERM signal，第二遍发 SIGKILL signal

由于是系统启动阶段运行 init（init 1），则这时的 family 链表为空，应该不进入循环，则这时 round = 0，talk = 1，foundOne = 0，代码应该跳转到 L1393 行执行。

在 init 3 中，family 的链表当然不为空，所以自然会进入循环
从 L1321 到 L1414 的代码是 init 3 执行的代码，init 1 不会执行。

这部分代码就是首先把当前 init 管理的 process(在 family 链表中)需要杀死的就 kill 掉，为重新 inittab 中要启动的 process 清理场地。

1. 对如下属性的 process 是不能杀死的，即忽略这些进程

   BOOT
   S --- Single User Mode
   DEMAND

2. 杀死 process 的方式是这样的，先发送 SIGTERM signal，然后等待 sltime 秒（sltime 默认为 5 秒，但用户可以设定），最后再发 SIGKILL signal。

```
1326      /*
1327       *    Is this line deleted?
1328       */
1329      if (ch->new == NULL) ch->flags |= KILLME;
1330
1331      /*
1332       *    If the entry has changed, kill it anyway. Note that
1333       *    we do not check ch->process, only the "action" field.
1334       *    This way, you can turn an entry "off" immediately, but
1335       *    changes in the command line will only become effective
1336       *    after the running version has exited.
1337       */
1338      if (ch->new && ch->action != ch->new->action) ch->flags |= KILLME;
1339
1340      /*
```

```
         *    Only BOOT processes may live in all levels
         */
        if (ch->action != BOOT &&
            strchr(ch->rlevel, runlevel) == NULL) {
              /*
               *    Ondemand procedures live always,
               *    except in single user
               */
              if (runlevel == 'S' || !(ch->flags & DEMAND))
                    ch->flags |= KILLME;
        }

        /*
         *    Now, if this process may live note so in the new list
         */
        if ((ch->flags & KILLME) == 0) {
              ch->new->flags  = ch->flags;
              ch->new->pid    = ch->pid;
              ch->new->exstat = ch->exstat;
              continue;
        }


        /*
         *    Is this process still around?
         */
        if ((ch->flags & RUNNING) == 0) {
              ch->flags &= ~KILLME;
              continue;
        }
        INITDBG(L_VB, "Killing \"%s\"", ch->process);
        switch(round) {                                 先发 SIGTERM signal
              case 0: /* Send TERM signal */
                    if (talk)
                          initlog(L_CO,
                                "Sending processes the TERM signal");
                    kill(-(ch->pid), SIGTERM);          将 SIGTERM signal 发给其进程组 id 为 ch->pid 的
```

```
1378                    foundOne = 1;                      process
1379                    break;
1380              case 1: /* Send KILL signal and collect status */
1381                    if (talk)                          发送 SIGKILL signal
1382                          initlog(L_CO,
1383                                "Sending processes the KILL signal");
1384                    kill(-(ch->pid), SIGKILL);
1385                    break;
1386        }
1387        talk = 0;
1388
1389      }
1390      /*
1391       *      See if we have to wait 5 seconds
1392       */
1393      if (foundOne && round == 0) {   对于系统启动阶段运行的 init (init 1) , round = 0,但 foundOne = 0, 所以
1394        /*                            将不会进入该 if 分支，跳转到 L1419 执行
1395         *    Yup, but check every second if we still have children.
1396         */                            同样，init 3 将执行执行这里 L1397 到 L1411 的代码
1397        for(f = 0; f < sltime; f++) {
1398              for(ch = family; ch; ch = ch->next) {
1399                    if (!(ch->flags & KILLME)) continue;
1400                    if ((ch->flags & RUNNING) && !(ch->flags & ZOMBIE))
1401                          break;
1402              }
1403              if (ch == NULL) {
1404                    /*
1405                     *    No running children, skip SIGKILL
1406                     */
1407                    round = 1;
1408                    foundOne = 0; /* Skip the sleep below. */
1409                    break;
1410              }
1411              do_sleep(1);           整个循环将等待 1 * sltime 秒
1412        }
1413      }
```

```
1414    }
1415
1416    /*
1417     *  Now give all processes the chance to die and collect exit statuses.
1418     */
1419    if (foundOne) do_sleep(1);             init 1 运行时，foundOne = 0，所以不会睡眠一秒
1420    for(ch = family; ch; ch = ch->next)    这时的 family 链表为空，不进入该循环，跳转到 L1437 执行
1421        if (ch->flags & KILLME) {          init 3 将执行这里 L1420 到 L1430 的代码
1422              if (!(ch->flags & ZOMBIE))
1423                  initlog(L_CO, "Pid %d [id %s] seems to hang", ch->pid,
1424                          ch->id);
1425              else {
1426                  INITDBG(L_VB, "Updating utmp for pid %d [id %s]",
1427                          ch->pid, ch->id);
1428                  ch->flags &= ~RUNNING;
1429                  if (ch->process[0] != '+')
1430                    write_utmp_wtmp("", ch->id, ch->pid, DEAD_PROCESS, NULL);
1431              }
1432        }
1433
1434    /*
1435     *  Both rounds done; clean up the list.
1436     */
1437    sigemptyset(&nmask);
1438    sigaddset(&nmask, SIGCHLD);
1439    sigprocmask(SIG_BLOCK, &nmask, &omask);
1440    for(ch = family; ch; ch = old) {             init 1 运行时，family 链表还为空，所以不进入循环，跳转到 L1444
1441        old = ch->next;         init 3 运行时，需要先释放原来 family 链表中的节点，重读 inittab 文件后生成的链表由
                                newFamily 指向
1442        free(ch);
1443    }
1444    family = newFamily;                   newFamily 中就是在本函数头上读入的当前的/etc/inittab 配置行生成的链表，现在才
                                          把它赋给 family

1445    for(ch = family; ch; ch = ch->next) ch->new = NULL;
1446    newFamily = NULL;
```

```
  sigprocmask(SIG_SETMASK, &omask, NULL);

#ifdef INITLVL
  /*
   *  Dispose of INITLVL file.
   */ 删除/etc/initrunlvl，根据是文件还是 symbol link，删除方式是不一样的
  if (lstat(INITLVL, &st) >= 0 && S_ISLNK(st.st_mode)) {    检查/etc/initrunlvl 是文件还是 symbol link
      /*
       *    INITLVL is a symbolic link, so just truncate the file.
       */
      close(open(INITLVL, O_WRONLY|O_TRUNC));
  } else {
      /*
       *    Delete INITLVL file.
       */
      unlink(INITLVL);
  }
#endif
#ifdef INITLVL2
  /*
   *  Dispose of INITLVL2 file.
   */ 删除/var/log/initrunlvl，根据是文件还是 symbol link，删除方式是不一样的
  if (lstat(INITLVL2, &st) >= 0 && S_ISLNK(st.st_mode)) {
      /*
       *    INITLVL2 is a symbolic link, so just truncate the file.
       */
      close(open(INITLVL2, O_WRONLY|O_TRUNC));
  } else {
      /*
       *    Delete INITLVL2 file.
       */
      unlink(INITLVL2);
  }
#endif
}
```

后记

本笔记是我想写的Linux系统初始化系列的一部分。（主要分三部分，一是init进程分析，也就是本文；二是多用户模式启动分析，即进入console登录界面的启动；三是GUI模式启动，即X Window登录界面的启动[1]）Linux内核初始化后的动作是启动init process，而以后的用户态的初始化全部由其启动，所以它是研究系统用户态启动的源头，非学习不可。

[1] 这里不涉及Linux内核本身的初始化，也就是上面介绍的start_kernel函数及其调用的相关函数。因为这个话题实在太大，几乎涉及内核所有子系统。

联系

Walter Zhou

mailto:z-l-dragon@hotmail.com

附录

环境

1. sysvinit-2.86 包
2. VMware + Redhat 8.0

inittab 中 action 的注解

- **respawn**：如果process字段指定的进程不存在，则启动该进程，init不等待处理结束，而是继续扫描inittab文件中的后续进程，当这样的进程终止时，init会重新启动它，如果这样的进程已存在，则什么也不做。
- **wait**：启动process字段指定的进程，并等到处理结束才去处理inittab中的下一记录项。
- **once**：启动process字段指定的进程，不等待处理结束就去处理下一记录项。当这样的进程终止时，也不再重新启动它，在进入新的运行级别时，如果这样的进程仍在运行，init也不重新启动它。
- **boot**：只有在系统启动时，init才处理这样的记录项，启动相应进程，并不等待处理结束就去处理下一个记录项。当这样的进程终止时，系统也不重启它。
- **bootwait**：系统启动后，当第一次从单用户模式进入多用户模式时处理这样的记录项，init启动这样的进程，并且等待它的处理结束，然后再进行下一个记录项的处理，当这样的进程终止时，系统也不重启它。
- **powerfail**：当init接到断电的信号（SIGPWR）时，处理指定的进程。
- **powerwait**：当init接到断电的信号（SIGPWR）时，处理指定的进程，并且等到处理结束才去检查其他的记录项。
- **off**：如果指定的进程正在运行，init就给它发SIGTERM警告信号，在向它发出信号SIGKILL强制其结束之前等待 5 秒，如果这样的进程不存在，则忽略这一项。
- **ondemand**：功能通respawn，不同的是，与具体的运行级别无关，只用于rstate字段是a、b、c的那些记录项。
- **sysinit**：指定的进程在访问控制台之前执行，这样的记录项仅用于对某些设备的初始化，目的是为了使init在这样的设备上向用户提问有关运行级别的问题，init需要等待进程运行结束后才继续。

- **initdefault**：指定一个默认的运行级别，只有当init一开始被调用时才扫描这一项，如果rstate字段指定了多个运行级别，其中最大的数字是默认的运行级别，如果rstate字段是空的，init认为字段是0123456，于是进入级别6，这样便陷入了一个循环，如果inittab文件中没有包含initdefault的记录项，则在系统启动时请求用户为它指定一个初始运行级别

关机分析

在init process的配置文件inittab中有多个动作涉及到shutdown(关机命令)。

```
# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now                    当按下CTRL-ALT-DELETE的组合键

# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.                      当UPS报告电源fail时
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

# If power was restored before the shutdown kicked in, cancel it.       当UPS报告电源恢复
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
```

在上面三种情况下，init process都会调用shutdown命令，只不过参数不同而已。比如当按下CTRL-ALT-DELETE的组合键后，“-t3 -r now”表示从现在(now)开始等待3秒(t3)，然后重启系统(r)。具体参数请看该命令的手册。

为什么在介绍系统启动的init process时要介绍关机命令呢？很简单，shutdown命令与init process息息相关。shutdown真正的关机或重启实际上是通过init process来实现的。

关机流程介绍

Shutdown命令有如下的参数选项

```
/*
 *     Show usage message.
 */
void usage(void)
```

```
{
      fprintf(stderr,
      "Usage:\t  shutdown [-akrhHPfnc] [-t secs] time [warning message]\n"
      "\t\t  -a:      use /etc/shutdown.allow\n"
      "\t\t  -k:      don't really shutdown, only warn.\n"
      "\t\t  -r:      reboot after shutdown.\n"
      "\t\t  -h:      halt after shutdown.\n"
      "\t\t  -P:      halt action is to turn off power.\n"
      "\t\t  -H:      halt action is to just halt.\n"
      "\t\t  -f:      do a 'fast' reboot (skip fsck).\n"
      "\t\t  -F:      Force fsck on reboot.\n"
      "\t\t  -n:      do not go through \"init\" but go down real fast.\n"
      "\t\t  -c:      cancel a running shutdown.\n"
      "\t\t  -t secs: delay between warning and kill signal.\n"
      "\t\t  ** the \"time\" argument is mandatory! (try \"now\") **\n");
      exit(1);
}
```

-a 表示只有记录在/etc/shutdown.allow文件中的用户才允许运行shutdown命令。
-k 表示并不是真正要关机或重启，仅仅发给登录该系统的用户警告。
-r 表示系统重新启动。
-h 表示系统halt。
-P 表示系统关机。
-f 表示重新启动，同时重启时不运行fsck(检查文件系统)
-F 表示重新启动，同时重启时强制运行fsck(检查文件系统)
-c 表示取消已进入shutdown状态的系统。
-t secs: 指定在发警告与杀死系统中运行的进程之间的间隔秒数

与shutdown相关的一些配置文件。

/etc/nologin	如果存在该文件，则不允许登陆该系统。如果系统要禁止用户登录，则只要建立该文件即可。该文件只要存在及可，有无内容无所谓。

```
/*
 *    Create the /etc/nologin file.
 */
void donologin(int min)
```

```
{
      FILE *fp;
      time_t t;

      time(&t);
      t += 60 * min;

      if ((fp = fopen(NOLOGIN, "w")) != NULL) {        create /etc/nologin 文件
            fprintf(fp, "\rThe system is going down on %s\r\n", ctime(&t));
            if (message[0]) fputs(message, fp);
            fclose(fp);
      }
}
```

/fastboot	如果希望在重启时不要检查文件系统（fsck），就建立该文件，有无内容无所谓。

```
      while((c = getopt(argc, argv, "HPacqkrhnfFyt:g:i:")) != EOF) {    分析 shutdown 命令行参数
            switch(c) {
                  case 'H':
                        halttype = "HALT";
                        break;
                  case 'P':
                        halttype = "POWERDOWN";
                        break;
                  case 'a': /* Access control. */
                        useacl = 1;
                        break;
                  case 'c': /* Cancel an already running shutdown. */
                        cancel = 1;
                        break;
                  case 'k': /* Don't really shutdown, only warn.*/
                        dontshut = 1;
                        break;
                  case 'r': /* Automatic reboot */
                        down_level[0] = '6';
                        break;
                        case 'h': /* Halt after shutdown */
                                down_level[0] = '0';
                                break;
                        case 'f': /* Don't perform fsck after next boot */
                                fastboot = 1;     表示希望重启时不要检查文件系统，置标志
                                break;
```

```
        chdir("/");
        if (fastboot)  close(open(FASTBOOT,  O_CREAT | O_RDWR, 0644));    如果 fastboot 标志置位建立/fastboot 文件
```

/forcefsck	如果希望在重启时强制检查文件系统（fsck），就建立该文件，有无内容无所谓。

```
        if (forcefsck) close(open(FORCEFSCK, O_CREAT | O_RDWR, 0644));    如果 fastfsck 标志置位建立/forcefsck 文
                                                                          件
```

/etc/shutdown.allow	如果该文件存在，shutdown 命令会查看该文件，只有列在该文件中的用户才可以运行 shutdown 命令；如果没有该文件，则只有 root 用户才有权运行 shutdown 命令。

```
        /* Process the options. */
        while((c = getopt(argc, argv, "HPacqkrhnfFyt:g:i:")) != EOF) {
                switch(c) {
                        case 'H':
                                halttype = "HALT";
                                break;
                        case 'P':
                                halttype = "POWERDOWN";
                                break;
                        case 'a': /* Access control. */     -a option，表示查看/etc/shutdown.allow 文件
                                useacl = 1;                 置标志
                                break;
        。。。。。。
```

```
      /* Do we need to use the shutdown.allow file ? */
      if (useacl && (fp = fopen(SDALLOW, "r")) != NULL) {   如果标志置位，则打开/etc/shutdown.allow 文件，里面每一
                                                             行是一个用户登录名
            /* Read /etc/shutdown.allow. */
            i = 0;
            while(fgets(buf, 128, fp)) {       循环读取一行
                  if (buf[0] == '#' || buf[0] == '\n') continue;  以#开头的为注释，忽略该行
                  if (i > 31) continue;        该文件最多可以有 32 个用户登录名，我不知道为什么要有这个限制
                  for(sp = buf; *sp; sp++) if (*sp == '\n') *sp = 0;    提取用户名
                  downusers[i++] = strdup(buf);      把提取出的用户名放入 downusers 数组
            }
            if (i < 32) downusers[i] = 0;
            fclose(fp);

            /* Now walk through /var/run/utmp to find logged in users. */
            while(!user_ok && (ut = getutent()) != NULL) {

                  /* See if this is a user process on a VC. */
                  if (ut->ut_type != USER_PROCESS) continue;
                  sprintf(term, "/dev/%.*s", UT_LINESIZE, ut->ut_line);
                  if (stat(term, &st) < 0) continue;
#ifdef major /* glibc */
                  if (major(st.st_rdev) != 4 ||
                      minor(st.st_rdev) > 63) continue;
#else
                  if ((st.st_rdev & 0xFFC0) != 0x0400) continue;
#endif
                  /* Root is always OK. */
                  if (strcmp(ut->ut_user, "root") == 0) {   root 用户总是可以关机的
                        user_ok++;
                        break;
                  }

                  /* See if this is an allowed user. */
                  for(i = 0; i < 32 && downusers[i]; i++)
                        if (!strncmp(downusers[i], ut->ut_user,   检查运行 shutdown 命令的用户名是否在 downusers
```

数组中
置位，可以运行 shutdown 命令标志

```
                      UT_NAMESIZE)) {
                        user_ok++;
                        break;
                  }
            }
            endutent();

            /* See if user was allowed. */
            if (!user_ok) {
                  if ((fp = fopen(CONSOLE, "w")) != NULL) {
                        fprintf(fp, "\rshutdown: no authorized users "
                                    "logged in.\r\n");
                        fclose(fp);
                  }
                  exit(1);
            }
      }
```

Shutdown 源码

```
/*
 * shutdown.c     Shut the system down.
 *
 * Usage:   shutdown [-krhfnc] time [warning message]
 *            -k: don't really shutdown, only warn.
 *            -r: reboot after shutdown.
 *            -h: halt after shutdown.
 *            -f: do a 'fast' reboot (skip fsck).
 *            -F: Force fsck on reboot.
 *            -n: do not go through init but do it ourselves.
 *            -c: cancel an already running shutdown.
 *            -t secs: delay between SIGTERM and SIGKILL for init.
 *
 * Author:  Miquel van Smoorenburg, miquels@cistron.nl
 *
 * Version: @(#)shutdown  2.86-1  31-Jul-2004  miquels@cistron.nl
 *
 *          This file is part of the sysvinit suite,
 *          Copyright 1991-2004 Miquel van Smoorenburg.
 *
 *          This program is free software; you can redistribute it and/or
 *          modify it under the terms of the GNU General Public License
 *          as published by the Free Software Foundation; either version
 *          2 of the License, or (at your option) any later version.
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <stdarg.h>
#include <utmp.h>
#include <syslog.h>
#include "paths.h"
#include "reboot.h"
#include "initreq.h"

char *Version = "@(#) shutdown 2.86-1 31-Jul-2004 miquels@cistron.nl";

#define MESSAGELEN      256

int dontshut = 0; /* Don't shutdown, only warn  */
char down_level[2];     /* What runlevel to go to.    */
int dosync = 1;         /* Sync before reboot or halt */
int fastboot = 0; /* Do a 'fast' reboot         */
int forcefsck = 0;      /* Force fsck on reboot       */
char message[MESSAGELEN];     /* Warning message      */
char *sltime = 0; /* Sleep time                 */
char newstate[64];      /* What are we gonna do       */
int doself = 0;         /* Don't use init       */
int got_alrm = 0;

char *clean_env[] = {
      "HOME=/",
      "PATH=/bin:/usr/bin:/sbin:/usr/sbin",
      "TERM=dumb",
      NULL,
};

/* From "wall.c" */
extern void wall(char *, int, int);

/* From "utmp.c" */
extern void write_wtmp(char *user, char *id, int pid, int type, char *line);

/*
 *    Sleep without being interrupted.
 */
void hardsleep(int secs)
{
      struct timespec ts, rem;

      ts.tv_sec = secs;
      ts.tv_nsec = 0;

      while(nanosleep(&ts, &rem) < 0 && errno == EINTR)
            ts = rem;
}

/*
 *    Break off an already running shutdown.
 */
```

```
void stopit(int sig)
{
      unlink(NOLOGIN);
      unlink(FASTBOOT);
      unlink(FORCEFSCK);
      unlink(SDPID);
      printf("\r\nShutdown cancelled.\r\n");
      exit(0);
}

/*
 *    Show usage message.
 */
void usage(void)
{
      fprintf(stderr,
      "Usage:\t  shutdown [-akrhHPfnc] [-t secs] time [warning message]\n"
      "\t\t  -a:      use /etc/shutdown.allow\n"
      "\t\t  -k:      don't really shutdown, only warn.\n"
      "\t\t  -r:      reboot after shutdown.\n"
      "\t\t  -h:      halt after shutdown.\n"
      "\t\t  -P:      halt action is to turn off power.\n"
      "\t\t  -H:      halt action is to just halt.\n"
      "\t\t  -f:      do a 'fast' reboot (skip fsck).\n"
      "\t\t  -F:      Force fsck on reboot.\n"
      "\t\t  -n:      do not go through \"init\" but go down real fast.\n"
      "\t\t  -c:      cancel a running shutdown.\n"
      "\t\t  -t secs: delay between warning and kill signal.\n"
      "\t\t  ** the \"time\" argument is mandatory! (try \"now\") **\n");
      exit(1);
}


void alrm_handler(int sig)
{
      got_alrm = sig;
}
```

```
/*
 *    Set environment variables in the init process.
 */
int init_setenv(char *name, char *value)
{
      struct init_request     request;
      struct sigaction  sa;
      int               fd;
      int               nl, vl;

      memset(&request, 0, sizeof(request));
      request.magic = INIT_MAGIC;
      request.cmd = INIT_CMD_SETENV;
      nl = strlen(name);
      vl = value ? strlen(value) : 0;

      if (nl + vl + 3 >= sizeof(request.i.data))
            return -1;

      memcpy(request.i.data, name, nl);
      if (value) {
            request.i.data[nl] = '=';
            memcpy(request.i.data + nl + 1, value, vl);
      }

        /*
       *    Open the fifo and write the command.
         *  Make sure we don't hang on opening /dev/initctl
       */
      memset(&sa, 0, sizeof(sa));
      sa.sa_handler = alrm_handler;
      sigaction(SIGALRM, &sa, NULL);
      got_alrm = 0;
        alarm(3);
        if ((fd = open(INIT_FIFO, O_WRONLY)) >= 0 &&
            write(fd, &request, sizeof(request)) == sizeof(request)) {
                close(fd);
                alarm(0);
                return 0;
        }

        fprintf(stderr, "shutdown: ");
        if (got_alrm) {
                fprintf(stderr, "timeout opening/writing control channel %s\n",
                        INIT_FIFO);
        } else {
                perror(INIT_FIFO);
        }
        return -1;
}


/*
 *    Tell everyone the system is going down in 'mins' minutes.
 */
void warn(int mins)
{
      char buf[MESSAGELEN + sizeof(newstate)];
      int len;

      buf[0] = 0;
      strncat(buf, message, sizeof(buf) - 1);
      len = strlen(buf);

      if (mins == 0)
            snprintf(buf + len, sizeof(buf) - len,
                  "\rThe system is going down %s NOW!\r\n",
                  newstate);
      else
            snprintf(buf + len, sizeof(buf) - len,
                  "\rThe system is going DOWN %s in %d minute%s!\r\n",
                        newstate, mins, mins == 1 ? "" : "s");
      wall(buf, 1, 0);
}

/*
 *    Create the /etc/nologin file.
 */
void donologin(int min)
{
      FILE *fp;
      time_t t;

      time(&t);
      t += 60 * min;

      if ((fp = fopen(NOLOGIN, "w")) != NULL) {
            fprintf(fp, "\rThe system is going down on %s\r\n", ctime(&t));
            if (message[0]) fputs(message, fp);
            fclose(fp);
      }
}

/*
 *    Spawn an external program.
 */
int spawn(int noerr, char *prog, ...)
{
      va_list     ap;
      pid_t pid, rc;
      int   i;
      char  *argv[8];

      i = 0;
      while ((pid = fork()) < 0 && i < 10) {
            perror("fork");
            sleep(5);
            i++;
      }
      if (pid < 0) return -1;

      if (pid > 0) {
            while((rc = wait(&i)) != pid)
                  if (rc < 0 && errno == ECHILD)
                        break;
            return (rc == pid) ? WEXITSTATUS(i) : -1;
      }

      if (noerr) fclose(stderr);

      argv[0] = prog;
      va_start(ap, prog);
      for (i = 1; i < 7 && (argv[i] = va_arg(ap, char *)) != NULL; i++)
            ;
      argv[i] = NULL;
      va_end(ap);

      chdir("/");
      environ = clean_env;

      execvp(argv[0], argv);
      perror(argv[0]);
      exit(1);

      /*NOTREACHED*/
      return 0;
}

/*
 *    Kill all processes, call /etc/init.d/halt (if present)
 */
void fastdown()
{
      int do_halt = (down_level[0] == '0');
      int i;
```

```
#if 0
      char cmd[128];
      char *script;

      /*
       *    Currently, the halt script is either init.d/halt OR rc.d/rc.0,
       *    likewise for the reboot script. Test for the presence
       *    of either.
       */
      if (do_halt) {
            if (access(HALTSCRIPT1, X_OK) == 0)
                  script = HALTSCRIPT1;
            else
                  script = HALTSCRIPT2;
      } else {
            if (access(REBOOTSCRIPT1, X_OK) == 0)
                  script = REBOOTSCRIPT1;
            else
                  script = REBOOTSCRIPT2;
      }
#endif

      /* First close all files. */
      for(i = 0; i < 3; i++)
            if (!isatty(i)) {
                  close(i);
                  open("/dev/null", O_RDWR);
            }
      for(i = 3; i < 20; i++) close(i);
      close(255);

      /* First idle init. */
      if (kill(1, SIGTSTP) < 0) {
            fprintf(stderr, "shutdown: can't idle init.\r\n");
            exit(1);
      }
      /* Kill all processes. */
      fprintf(stderr, "shutdown: sending all processes the TERM signal...\r\n");
      kill(-1, SIGTERM);
      sleep(sltime ? atoi(sltime) : 3);
      fprintf(stderr, "shutdown: sending all processes the KILL signal.\r\n");
      (void) kill(-1, SIGKILL);

#if 0
      /* See if we can run /etc/init.d/halt */
      if (access(script, X_OK) == 0) {
            spawn(1, cmd, "fast", NULL);
            fprintf(stderr, "shutdown: %s returned - falling back "
                        "on default routines\r\n", script);
      }
#endif

      /* script failed or not present: do it ourself. */
      sleep(1); /* Give init the chance to collect zombies. */

      /* Record the fact that we're going down */
      write_wtmp("shutdown", "~~", 0, RUN_LVL, "~~");

      /* This is for those who have quota installed. */
      spawn(1, "accton", NULL);
      spawn(1, "quotaoff", "-a", NULL);

      sync();
      fprintf(stderr, "shutdown: turning off swap\r\n");
      spawn(0, "swapoff", "-a", NULL);
      fprintf(stderr, "shutdown: unmounting all file systems\r\n");
      spawn(0, "umount", "-a", NULL);

      /* We're done, halt or reboot now. */
      if (do_halt) {
            fprintf(stderr, "The system is halted. Press CTRL-ALT-DEL "
                        "or turn off power\r\n");
            init_reboot(BMAGIC_HALT);
            exit(0);
      }

      fprintf(stderr, "Please stand by while rebooting the system.\r\n");
      init_reboot(BMAGIC_REBOOT);
      exit(0);
}

/*
 *    Go to runlevel 0, 1 or 6.
 */
void shutdown(char *halttype)
{
      char  *args[8];
      int   argp = 0;
      int   do_halt = (down_level[0] == '0');

      /* Warn for the last time */
      warn(0);
      if (dontshut) {
            hardsleep(1);
            stopit(0);
      }
      openlog("shutdown", LOG_PID, LOG_USER);
      if (do_halt)
            syslog(LOG_NOTICE, "shutting down for system halt");
      else
            syslog(LOG_NOTICE, "shutting down for system reboot");
      closelog();

      /* See if we have to do it ourself. */
      if (doself) fastdown();

      /* Create the arguments for init. */
      args[argp++] = INIT;
      if (sltime) {
            args[argp++] = "-t";
            args[argp++] = sltime;
      }
      args[argp++] = down_level;
      args[argp]   = (char *)NULL;

      unlink(SDPID);
      unlink(NOLOGIN);

      /* Now execute init to change runlevel. */
      sync();
      init_setenv("INIT_HALT", halttype);
      execv(INIT, args);

      /* Oops - failed. */
      fprintf(stderr, "\rshutdown: cannot execute %s\r\n", INIT);
      unlink(FASTBOOT);
      unlink(FORCEFSCK);
      init_setenv("INIT_HALT", NULL);
      openlog("shutdown", LOG_PID, LOG_USER);
      syslog(LOG_NOTICE, "shutdown failed");
      closelog();
      exit(1);
}

/*
 *    returns if a warning is to be sent for wt
 */
static int needwarning(int wt)
{
      int ret;

      if (wt < 10)
            ret = 1;
      else if (wt < 60)
            ret = (wt % 15 == 0);
      else if (wt < 180)
            ret = (wt % 30 == 0);
      else
            ret = (wt % 60 == 0);

      return ret;
}

/*
 *    Main program.
 *    Process the options and do the final countdown.
 */
int main(int argc, char **argv)
{
      FILE              *fp;
      extern int        getopt();
      extern int        optind;
      struct sigaction  sa;
      struct tm         *lt;
      struct stat       st;
      struct utmp       *ut;
      time_t                  t;
      uid_t             realuid;
      char              *halttype;
      char              *downusers[32];
      char              buf[128];
      char              term[UT_LINESIZE + 6];
      char              *sp;
      char              *when = NULL;
      int               c, i, wt;
      int               hours, mins;
      int               didnolog = 0;
      int               cancel = 0;
      int               useacl = 0;
      int               pid = 0;
      int               user_ok = 0;

      /* We can be installed setuid root (executable for a special group) */
      realuid = getuid();
      setuid(geteuid());

      if (getuid() != 0) {
            fprintf(stderr, "shutdown: you must be root to do that!\n");
            exit(1);
      }
      strcpy(down_level, "1");
      halttype = NULL;

      /* Process the options. */
      while((c = getopt(argc, argv, "HPacqkrhnfFyt:g:i:")) != EOF) {
            switch(c) {
                  case 'H':
                        halttype = "HALT";
                        break;
                  case 'P':
                        halttype = "POWERDOWN";
                        break;
                  case 'a': /* Access control. */
                        useacl = 1;
                        break;
                  case 'c': /* Cancel an already running shutdown. */
                        cancel = 1;
                        break;
                  case 'k': /* Don't really shutdown, only warn.*/
                        dontshut = 1;
                        break;
                  case 'r': /* Automatic reboot */
                        down_level[0] = '6';
                        break;
                  case 'h': /* Halt after shutdown */
                        down_level[0] = '0';
                        break;
                  case 'f': /* Don't perform fsck after next boot */
                        fastboot = 1;
                        break;
                  case 'F': /* Force fsck after next boot */
```

```
                        forcefsck = 1;
                        break;
                  case 'n': /* Don't switch runlevels. */
                        doself = 1;
                        break;
                  case 't': /* Delay between TERM and KILL */
                        sltime = optarg;
                        break;
                  case 'y': /* Ignored for sysV compatibility */
                        break;
                  case 'g': /* sysv style to specify time. */
                        when = optarg;
                        break;
                  case 'i': /* Level to go to. */
                        if (!strchr("0156aAbBcCsS", optarg[0])) {
                              fprintf(stderr,
                              "shutdown: `%s': bad runlevel\n",
                              optarg);
                              exit(1);
                        }
                        down_level[0] = optarg[0];
                        break;
                  default:
                        usage();
                        break;
            }
      }

      /* Do we need to use the shutdown.allow file ? */
      if (useacl && (fp = fopen(SDALLOW, "r")) != NULL) {

            /* Read /etc/shutdown.allow. */
            i = 0;
            while(fgets(buf, 128, fp)) {
                  if (buf[0] == '#' || buf[0] == '\n') continue;
                  if (i > 31) continue;
                  for(sp = buf; *sp; sp++) if (*sp == '\n') *sp = 0;
                  downusers[i++] = strdup(buf);
            }
            if (i < 32) downusers[i] = 0;
            fclose(fp);

            /* Now walk through /var/run/utmp to find logged in users. */
            while(!user_ok && (ut = getutent()) != NULL) {

                  /* See if this is a user process on a VC. */
                  if (ut->ut_type != USER_PROCESS) continue;
                  sprintf(term, "/dev/%.*s", UT_LINESIZE, ut->ut_line);
                  if (stat(term, &st) < 0) continue;
#ifdef major /* glibc */
                  if (major(st.st_rdev) != 4 ||
                      minor(st.st_rdev) > 63) continue;
#else
                  if ((st.st_rdev & 0xFFC0) != 0x0400) continue;
#endif
                  /* Root is always OK. */
                  if (strcmp(ut->ut_user, "root") == 0) {
                        user_ok++;
                        break;
                  }

                  /* See if this is an allowed user. */
                  for(i = 0; i < 32 && downusers[i]; i++)
                        if (!strncmp(downusers[i], ut->ut_user,
                            UT_NAMESIZE)) {
                              user_ok++;
                              break;
                        }
            }
            endutent();

            /* See if user was allowed. */
            if (!user_ok) {
                  if ((fp = fopen(CONSOLE, "w")) != NULL) {
                        fprintf(fp, "\rshutdown: no authorized users "
                                    "logged in.\r\n");
                        fclose(fp);
                  }
                  exit(1);
            }
      }

      /* Read pid of running shutdown from a file */
      if ((fp = fopen(SDPID, "r")) != NULL) {
            fscanf(fp, "%d", &pid);
            fclose(fp);
      }

      /* Read remaining words, skip time if needed. */
      message[0] = 0;
      for(c = optind + (!cancel && !when); c < argc; c++) {
            if (strlen(message) + strlen(argv[c]) + 4 > MESSAGELEN)
                  break;
            strcat(message, argv[c]);
            strcat(message, " ");
      }
      if (message[0]) strcat(message, "\r\n");

      /* See if we want to run or cancel. */
      if (cancel) {
            if (pid <= 0) {
                  fprintf(stderr, "shutdown: cannot find pid "
                              "of running shutdown.\n");
                  exit(1);
            }
            init_setenv("INIT_HALT", NULL);
            if (kill(pid, SIGINT) < 0) {
                  fprintf(stderr, "shutdown: not running.\n");
                  exit(1);
            }
            if (message[0]) wall(message, 1, 0);
            exit(0);
      }

      /* Check syntax. */
      if (when == NULL) {
            if (optind == argc) usage();
            when = argv[optind++];
      }

      /* See if we are already running. */
      if (pid > 0 && kill(pid, 0) == 0) {
            fprintf(stderr, "\rshutdown: already running.\r\n");
            exit(1);
      }

      /* Extra check. */
      if (doself && down_level[0] != '0' && down_level[0] != '6') {
            fprintf(stderr,
            "shutdown: can use \"-n\" for halt or reboot only.\r\n");
            exit(1);
      }

      /* Tell users what we're gonna do. */
      switch(down_level[0]) {
            case '0':
                  strcpy(newstate, "for system halt");
                  break;
            case '6':
                  strcpy(newstate, "for reboot");
                  break;
            case '1':
                  strcpy(newstate, "to maintenance mode");
                  break;
            default:
                  sprintf(newstate, "to runlevel %s", down_level);
                  break;
      }
```

```
        /* Create a new PID file. */
        unlink(SDPID);
        umask(022);
        if ((fp = fopen(SDPID, "w")) != NULL) {
              fprintf(fp, "%d\n", getpid());
              fclose(fp);
        } else if (errno != EROFS)
              fprintf(stderr, "shutdown: warning: cannot open %s\n", SDPID);

        /*
         *    Catch some common signals.
         */
        signal(SIGQUIT, SIG_IGN);
        signal(SIGCHLD, SIG_IGN);
        signal(SIGHUP,  SIG_IGN);
        signal(SIGTSTP, SIG_IGN);
        signal(SIGTTIN, SIG_IGN);
        signal(SIGTTOU, SIG_IGN);

        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = stopit;
        sigaction(SIGINT, &sa, NULL);

        /* Go to the root directory */
        chdir("/");
        if (fastboot)  close(open(FASTBOOT,  O_CREAT | O_RDWR, 0644));
        if (forcefsck) close(open(FORCEFSCK, O_CREAT | O_RDWR, 0644));

        /* Alias now and take care of old '+mins' notation. */
        if (!strcmp(when, "now")) strcpy(when, "0");
        if (when[0] == '+') when++;

        /* Decode shutdown time. */
        for (sp = when; *sp; sp++) {
              if (*sp != ':' && (*sp < '0' || *sp > '9'))
                    usage();
```

```
      }
      if (strchr(when, ':') == NULL) {
            /* Time in minutes. */
            wt = atoi(when);
            if (wt == 0 && when[0] != '0') usage();
      } else {
            /* Time in hh:mm format. */
            if (sscanf(when, "%d:%2d", &hours, &mins) != 2) usage();
            if (hours > 23 || mins > 59) usage();
            time(&t);
            lt = localtime(&t);
            wt = (60*hours + mins) - (60*lt->tm_hour + lt->tm_min);
            if (wt < 0) wt += 1440;
      }
      /* Shutdown NOW if time == 0 */
      if (wt == 0) shutdown(halttype);

      /* Give warnings on regular intervals and finally shutdown. */
      if (wt < 15 && !needwarning(wt)) warn(wt);
      while(wt) {
            if (wt <= 5 && !didnolog) {
                  donologin(wt);
                  didnolog++;
            }
            if (needwarning(wt)) warn(wt);
            hardsleep(60);
            wt--;
      }
      shutdown(halttype);

      return 0; /* Never happens */
}
```