

Il processo di "boot" di Linux

0. il BIOS
 1. il boot loader
 2. setup(), setup_32(), setup_32()
 3. start_kernel()
 4. init
 5. /etc/rc
- 0,1,2: architecture dependent

1

La CPU e il BIOS

BIOS: un insieme di programmi cablati in ROM (i386-specific)

Linux usa il BIOS solo durante il boot

All'accensione la CPU esegue il codice all'indirizzo 0xfffff0 (mappato sul BIOS)

0. Power-On Self Test (POST)
 1. Inizializza il bus PCI
 2. Cerca un S.O. da caricare
 3. Copia il primo settore del device di boot in RAM, e ci salta dentro

2

Il boot loader

non è necessario: si può fare il boot da floppy senza

per fare il boot da hard disk occorre un boot loader

ce ne sono diversi:

- LILO (LIinux LOader)
- GRUB (GRand Unified Bootloader)
- Loadlin

3

Il boot da hard disk

Il primo settore è il Master Boot Record (MBR)

Contiene:

- la tabella delle partizioni
- codice

Il codice messo nel MBR da Windows cerca una partizione marcata "attiva" e carica il S.O. da quella partizione

⇒ in base a questo principio si può fare partire un solo S.O. :(

Usando LILO o GRUB si sostituisce il codice nel MBR con un boot loader sofisticato

4

Boot con LILO

Il boot loader di LILO è copiato dal MBR in RAM, ed eseguito

- carica il resto del boot loader
- presenta all'utente la scelta di quale S.O. caricare
- a seconda della scelta può:
 - caricare il boot sector di una partizione ed eseguirlo
 - caricare il kernel di Linux (usando il BIOS)
- se ha caricato Linux, esegue la funzione `setup()` del kernel

5

La funzione `setup()`

Il BIOS ha già inizializzato l'hardware, ma Linux non si fida...

... e reinizializza tutto a modo suo

`setup()` è realizzata in assembly

0. Chiede al BIOS quanta memoria è disponibile
1. (Re-)inizializza la tastiera, la scheda video, l'hard disk
2. Inizializza il mouse e l'Advanced Power Management (APM)
3. Riprogramma il Programmable Interrupt Controller (PIC)
4. Passa la CPU da "real mode" a "protected mode"
5. Esegue la funzione `startup_32()`

6

La funzione `setup_32()`

sta in `arch/i386/boot/compressed/head.S`

anch'essa scritta in assembly

0. inizializza uno stack temporaneo
 1. azzerare l'area di dati non inizializzati del kernel (proprio come se fosse un normale programma C)
 2. invoca `decompress_kernel()`
 - appare il messaggio "Uncompressing Linux..."
 - seguito da "OK, booting the kernel"
- L'immagine decompressa è copiata all'indirizzo `0x00100000` (2^{20})
3. salta all'indirizzo `0x00100000`

7

Un'altra `setup_32()`?!?

Il kernel decompresso inizia con un'altra `setup_32()`

Questa sta in `arch/i386/kernel/head.S`

0. Inizializza lo stack per il processo 0

fino ad ora stavamo eseguendo il processo hardware; ora stiamo per trasformarlo nel processo 0 se c'è un momento drammatico, è questo

1. Prepara i registri della CPU che gestiscono i segmenti
2. Prepara le page table del kernel
3. Fa partire la memoria virtuale paginata
4. esegue la funzione `start_kernel()`

8

La funzione start_kernel()

in init/main.c

Finalmente codice C e indipendente dall'architettura

Inizializza tutti i componenti del kernel (finora abbiamo inizializzato l'hardware)

Fa partire un kernel thread che esegue la funzione init()

Il processo 0 che ha eseguito fino a qui va a dormire

```
...
kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
unlock_kernel();
current->need_resched = 1;
cpu_idle();
}
```

9

Cos'è un kernel thread? (i)

In Unix tradizionale, alcuni compiti sono lasciati a processi che eseguono in background

Esempio: salvare i buffer sporchi su disco, reclamare pagine libere,...

Questi processi eseguono quasi totalmente in modo kernel

La parte user-mode di questi processi è inutile

In Linux questi processi sono diventati "kernel threads"

10

Cos'è un kernel thread? (ii)

È come un processo, ma:

- esegue solo in kernel mode
- non ha un file eseguibile
- esegue una specifica funzione C del kernel

11

Il kernel thread 0

Il kernel thread 0 è l'antenato di tutti i processi

Esegue la funzione `cpu_idle()` che non fa "niente"

- chiama ripetutamente la funzione `idle()`
- `idle()` è architecture dependent
- usualmente ferma il processore per risparmiare energia

Viene eseguito *solo* quando non c'è assolutamente nessun altro processo da eseguire

12

Il kernel thread 1 (init)

Esegue la funzione `init()` in `init/main.c`

- completa l'inizializzazione di vari moduli del kernel
- monta il filesystem di root
- esegue una `execve(2)` per caricare il comando `init(8)`

Dopo `execve(2)` il processo 1 non è più kernel thread ma un processo a tutti gli effetti

13

```
static int init(void * unused)
{
    lock_kernel();
    do_basic_setup();
    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();

    if (open("/dev/console", O_RDWR, 0) < 0)
        printk("Warning: unable to open an initial console.\n");

    (void) dup(0);
    (void) dup(0);

    if (execute_command)
        execve(execute_command, argv_init, envp_init);
    execve("/sbin/init", argv_init, envp_init);
    execve("/etc/init", argv_init, envp_init);
    execve("/bin/init", argv_init, envp_init);
    execve("/bin/sh", argv_init, envp_init);
    panic("No init found. Try passing init= option to kernel.");
}
```

14

Il processo 1: init

Init is the driving force that keeps our Linux box alive, and it is the one that can put it to death.

Quando il kernel ha finito di inizializzare l'hardware e ha montato il filesystem di root, invoca `init` e diventa un passivo esecutore di `syscall`

Non è uno specifico programma, ma una classe di programmi

Posso scegliere il programma `init` che preferisco

Posso sostituirlo con uno scritto da me

Posso sostituirlo con la shell (es. per recuperare un sistema danneggiato)

Posso sostituirlo con uno script di shell

Posso scegliere il programma `init` passando un'opzione al kernel

LILO: `Linux init=/bin/sh`

15

Il compito di init

È il primo e unico processo generato dal kernel

0. Deve generare tutti gli altri processi

- demoni
- sessione di login sulla console
- oppure, la sessione di login basata su X (`xdm`)

(tipicamente tutto ciò è fatto eseguendo uno script `/etc/rc`)

1. Legge lo stato di uscita di tutti i suoi figli con `waitpid(2)`

2. Per alcuni dei suoi figli, li fa ripartire quando terminano

- es. sessione di login sulla console

3. Gestisce lo shutdown

- riceve un segnale per `ctrl-alt-del`
- riceve un segnale dal gruppo di continuità quando sta per mancare la tensione

16

Nota sui processi zombie

Un processo è in stato zombie quando... ? (a questo punto dovrete saperlo!)

Il codice di `sys_exit()` fa in modo che tutti i processi "orfani" diventino figli di `init` (processo 1)

⇒ non ci sono mai orfani: ogni processo ha sempre un genitore

Tutti i programmi ben scritti dovrebbero leggere lo stato dei propri figli

`init` deve chiamare `waitpid(2)` su tutti i suoi figli, anche quelli "ereditati" da altri processi

17

Il processo di login

`init` fa partire un processo `getty(8)` per ogni "terminale" collegato

`getty(8)`

- apre un device (es. `/dev/tty1`)
- stampa il messaggio Login:
- legge la login dal terminale
- invoca il comando `/bin/login` con `exec(2)`

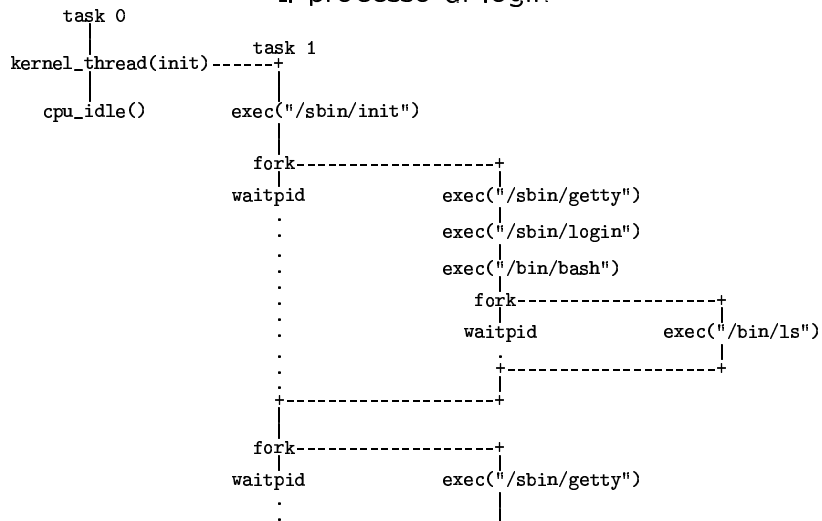
`login(8)`

- chiede la password
- verifica la password (con `/etc/shadow`)
- invoca la shell dell'utente (trovata in `/etc/passwd`) con `exec(2)`

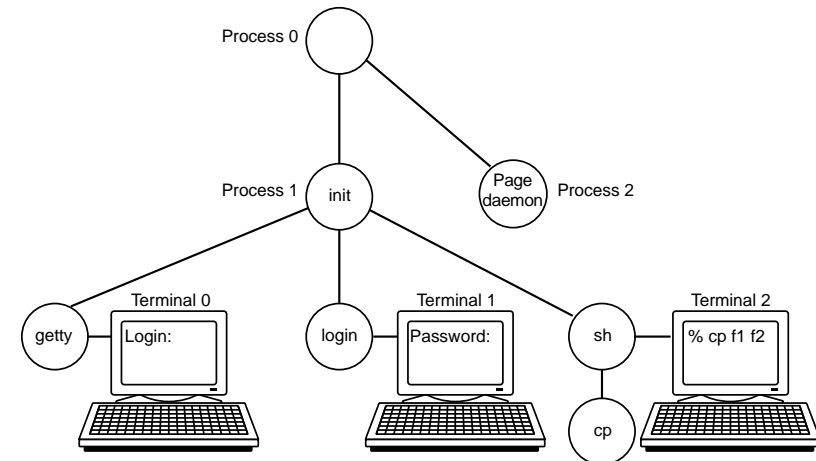
Quando l'utente chiude la shell, `init` si accorge che il suo processo figlio è terminato e fa ripartire `getty`

18

Il processo di login



19



20

Un init minimale realizzato in shell

```
#!/bin/sh
# avoid typing full pathnames
export PATH=/usr/bin:/bin:/sbin:/usr/sbin
# remount root read-write, and mount all
mount -n -o remount,rw /
mount -a
swapon -a
# system log
syslogd
klogd
# start your lan
modprobe eth0 2> /dev/null
ifconfig eth0 192.168.0.1
route add 192.168.0.0 eth0
route add default gw 192.168.0.254
# start lan services
inetd
sendmail -bd -q30m
# Anything else: crond, named, ...
# And run one getty with a sane path
export PATH=/usr/bin:/bin
/sbin/mingetty tty1
```

21

Analisi dell'init minimale

OK: la shell /bin/sh automaticamente chiama waitpid(2) su tutti i suoi figli

KO: il getty non viene fatto ripartire quando termina

KO: non gestisce lo shutdown

22

Risolviamo il problema dello shutdown

Aggiungiamo questo codice dopo la chiamata a getty

```
# kill anything you started
killall inetd
killall sendmail
killall klogd
killall syslogd
# kill anything else
kill -TERM -1
sleep 2
kill -KILL -1
# release the disks
swapoff -a
umount -a
mount -n -o remount,ro /
echo "The system is halted"
exit
```

23

Risolviamo il problema di getty

```
# And run one getty with a sane path
export PATH=/usr/bin:/bin
while true
do
  /sbin/mingetty tty1
done
```

Problema: ora non c'è più verso di uscire dal loop. Come segnaliamo a init che è ora di smetterla?

24

Una possibile soluzione

Inventiamo la convenzione che per invocare lo shutdown creiamo un file /etc/ORA_DI_SMETTERLA e terminiamo la sessione

```
# touch /etc/ORA_DI_SMETTERLA
# exit
The system is halted
```

Modifichiamo lo script

```
# Remove old file
rm /etc/ORA_DI_SMETTERLA

# And run one getty with a sane path
export PATH=/usr/bin:/bin
while [ ! -f /etc/ORA_DI_SMETTERLA ]
do
  /sbin/mingetty tty1
done
```

25

Lo shutdown

Questo ci insegna che il programma shutdown(8) non fa altro che comunicare a init(8) che è ora di scendere

26

Ricapitolando

```
#!/bin/sh

# avoid typing full pathnames
export PATH=/usr/bin:/bin:/sbin:/usr/sbin

# remount root read-write, and mount all
mount -n -o remount,rw /
mount -a
swapon -a

# system log
syslogd
klogd

# start your lan
modprobe eth0 2> /dev/null
ifconfig eth0 192.168.0.1
route add 192.168.0.0 eth0
route add default gw 192.168.0.254

# start lan services
inetd
sendmail -bd -q30m

# Anything else: crond, named, ...

# Remove old file
rm /etc/ORA_DI_SMETTERLA

# And run one getty with a sane path
export PATH=/usr/bin:/bin
while [ ! -f /etc/ORA_DI_SMETTERLA ]
do
  /sbin/mingetty tty1
done

# kill anything you started
killall inetd
killall sendmail
killall klogd
killall syslogd

# kill anything else
kill -TERM -1
sleep 2
kill -KILL -1

# release the disks
swapoff -a
umount -a
mount -n -o remount,ro /
echo "The system is halted"
exit
```

27

Il "vero" init

il *runlevel* (0-6 e "S") è lo "stato" in cui si trova init

i runlevel sono configurabili dall'amministratore in /etc/inittab

per ogni runlevel si possono specificare:

- processi da eseguire
- processi che devono essere restartati sempre (respawn)

tipica configurazione dei runlevel:

- runlevel 0: shutdown
- runlevel 1: single user
- runlevel 2: non usato
- runlevel 3: multiuser con console in modo testo
- runlevel 4: multiuser con console in modo X
- runlevel 5: non usato
- runlevel 6: reboot
- runlevel S: esegue una serie di script che portano al runlevel 1

per cambiare runlevel: si invoca /sbin/telinit

28

Un esempio di /etc/inittab semplificato

```
# il runlevel di default è 3
id:3:initdefault:

# esegui /etc/rc al boot, aspetta che termini
rc::bootwait:/etc/rc

# What to do at the "Three Finger Salute".
ca::ctrlaltdel:/sbin/shutdown -t5 -r now

# apri quattro getty e falli ripartire sempre
1:3:respawn:/etc/getty 9600 tty1
2:3:respawn:/etc/getty 9600 tty2
3:3:respawn:/etc/getty 9600 tty3
4:3:respawn:/etc/getty 9600 tty4
```

29

Le implementazioni di init(8) più comuni

- `init(8)` di Miquel van Smoorenburg
(usato quasi universalmente)
- `simpleinit`
più semplice, adatto a scopi didattici

30

La discussione di `init` è basata sull'articolo di Alessandro Rubini

“Take command: `init`”

<http://www.linux.it/kerneldocs/init/init.html>

31