UNIVERSITÀ DEGLI STUDI DI ROMA

Tor Vergata

FACOLTÀ DI INGEGNERIA

Corso di Laurea in INGEGNERIA INFORMATICA

REALIZZAZIONE DI UN SISTEMA PER LA PRODUZIONE DI DISTRIBUZIONI LINUX

Relatore:
Daniel Pierre Bovet

Candidato:
Laurenziello Vincenzo

Anno Accademico 2006/2007

Indice

Ringraziamenti

Capitolo 1

Introduzione

Costruire un sistema operativo, da zero, ha da sempre fornito un alone di gloria al suo autore.

Con il movimento del software libero e del kernel Linux è stato possibile costruire dei veri sistemi operativi personalizzati. Questo, è presente dagli albori dell'informatica, e permette, *liberamente*, di condividere il codice tra persone, senza alcun legame contrattuale tra loro.

Il concetto di *software libero* può essere riassunto in quattro punti fondamentali:

- *libertà 0*: la libertà di eseguire il programma per qualunque scopo;
- *libertà 1*: la libertà di studiare come funziona il programma e di adattarlo alle proprie esigenze (in tal caso, deve essere disponibile il sorgente);
- *libertà 2*: la libertà di ridistribuire copie del programma;
- *libertà 3*: la libertà di migliorare il programma e di distribuire tali miglioramenti (anche per questo è necessario disporre dei sorgenti).

Con il passare degli anni, tramite la licenza `GPL`, introdotta da `GNU`, sono nati moltissimi progetti provenienti dalla comunità di Internet; ad esempio: Apache come web server, OpenOffice come suite per l'ufficio, Asterisk come server VoIP, Wikipedia come enciclopedia online, gcc come compilatore C, Compiz Fusion per l'interfaccia grafica 3D, CUPS per la gestione delle stampanti, eccetera.

Tra questi progetti, non si può non citare il primo e più importante, il kernel Linux. Questo, seppur inizialmente criticato dal professor Tanenbaum, ha ricevuto un forte supporto da parte degli sviluppatori di tutto il mondo. Ogni nuova versione del kernel Linux esce con cadenza bisettimanale ed

introduce sempre nuove funzionalità crescendo in maniera esponenziale. Il kernel controlla, gestisce e virtualizza l'hardware ed il software presente in un computer.

1.1 Distribuzioni

I programmi rilasciati sotto licenza GPL sono da sempre una valida alternativa alle più comuni applicazioni presenti nei sistemi UNIX™ closed source. Questi abbinati al kernel Linux permettono di avere un sistema operativo completo.

I sistemi operativi basati sul kernel Linux sono chiamati *distribuzioni Linux*.

Una tipica distribuzione Linux deve comprende almeno:

- un *filesystem*, che gestisce l'accesso in lettura e scrittura sulla memoria di massa;
- uno *spooler*, che gestisce i dati da stampare;
- un'*interfaccia utente*, che permette agli utenti di interagire con una macchina;
- un *kernel*.

Questa viene anche chiamata *distribuzione general purpose*.

La quasi totalità del software incluso in una distribuzione è FOSS[1], e viene distribuito dagli sviluppatori sia in forma precompilata che come codice sorgente. Alcune distribuzioni includono anche del software proprietario, che non è disponibile in formato sorgente.

Le prime distribuzioni Linux sono nate nel momento in cui alcune persone, esterne allo sviluppo del kernel, hanno iniziato ad usarlo. Infatti, gli sviluppatori del kernel sono più interessati a sviluppare il core del sistema operativo piuttosto che programmi applicativi, interfacce utente, e comode pacchettizzazioni.

Le prime distribuzioni furono:

- MCC Interim Linux, che fu reso disponibile per il download pubblico dal server ftp dell'Università di Manchester nel febbraio 1992;
- TAMU, creata da alcuni sviluppatori indipendenti della Texas A&M University più o meno nello stesso periodo, e

[1] Free and Open Source Software

- SLS (Softlanding Linux System).

Nessuna di queste distribuzioni era ben mantenuta, così Patrick Volkerding rilasciò una distribuzione bastata su SLS, che chiamò Slackware; questa è la distribuzione più longeva ancora attiva.

Le distribuzioni Linux sono state inizialmente costruite da gruppi di volontari, ma con il passare degli anni, anche differenti società — come Red Hat, Novell, Mandriva, Canonical, eccetera — sono entrate nel ruolo di distributori.

Questi gruppi di persone assemblano e testano le varie componenti software rilasciando distribuzioni personalizzate e variegate. Queste sono gratuitamente scaricabili da internet oppure tramite compenso è possibile avere un pacchetto comprensivo di manuali cartacei e assistenza tecnica.

1.2 Categorie di distribuzioni

Attualmente esistono più di 300 progetti di distribuzioni basate su Linux in continuo sviluppo.

Ci sono distribuzioni Linux che cercano di adattarsi al maggior numero di situazioni possibili, oppure quelle dedicate a scopi specifici.

Le distribuzioni vengono distinte in base a:

- la tipologia di utente a cui sono destinate: power user, normal user, newbie user.
- la tipologia di utilizzo: desktop, enterprise, live, embedded, real-time.

1.2.1 Tipologia di utenti

Power User

Un power user è un utente che utilizza funzionalità e programmi avanzati. Conoscono molto bene il sistema operativo che stanno utilizzando e preferiscono configurare i vari programmi installati manualmente ed in modalità testuale.

Esempi Slackware, Gentoo, Debian, Linux from Scratch.

Normal User

Questa categoria di utenti preferisce sfruttare tutte le funzionalità della distribuzione utilizzando tool di configurazione grafici ed user friendly — come

Adept, Synaptic, Anaconda, YaST, *Drake, tool del desktop manager KDE o Gnome. L'utilizzo di questi software permette di ottenere dei buoni risultati in breve tempo, ma sono limitati ad eseguire solo le più comuni tipologie di operazioni.

Esempi Ubuntu, Fedora, Suse, Mandriva.

Newbie User

Sono gli utenti che non hanno mai utilizzato un sistema operativo *nix. Tipicamente questi utenti hanno da sempre usato sistemi operativi Microsoft e per un qualsiasi motivo sono venuti a conoscenza di valide alternative, come le distribuzioni Linux.

Dato che i sistemi operativi *nix sono molto differenti da quelli di casa Microsoft, si consiglia di utilizzare distribuzioni live (vedi paragrafo 1.2.2) in modo da familiarizzare con un ambiente di lavoro preconfigurato e pieno di applicazioni grafiche.

Esempi Knoppix, Suse Live, Mandriva Live, Ubuntu.

1.2.2 Tipologia delle distribuzioni

Distribuzioni Desktop

Una distribuzione desktop deve gestire applicazioni di tipo multimediale ed essere di facile utilizzo. Questo tipo di distribuzioni vengono usate da un pubblico casalingo ed eterogeneo; sono le più personalizzate ed a volte per ottenere delle caratteristiche particolari si mette a rischio la stabilità della distribuzione.

Esempi Fedora, OpenSUSE, Mandriva, Ubuntu.

Distribuzioni Live

Una distribuzione live è un sistema operativo in grado di essere avviato ed eseguito senza richiedere l'installazione su hard disk. Questo tipo di distribuzione generalmente non altera lo stato della macchina e non utilizza il disco rigido, sebbene questa operazione sia possibile, se l'utente la richiede esplicitamente. Solitamente, vengono usati dei supporti di memorizzazione come i `CDROM`, `DVD`, memorie di massa `USB`.

Le distribuzioni live possono essere usate a scopo dimostrativo oppure per recuperare dati da sistemi operativi che non riescono più ad effettuare un avvio corretto.

Esempi Knoppix, System Rescue CD, Gobo Linux, BackTrack.

Distribuzioni Enterprise

Una distribuzione enterprise è specializzata per gestire le applicazioni critiche presenti nelle grandi aziende e istituzioni. Tipicamente queste distribuzioni lavoreranno in un `CED`[2] e devono garantire, a seconda delle politiche aziendali, alte prestazioni, alta affidabilità dei servizi oppure un bilanciamento del carico del sistema. Forniscono, inoltre, l'assistenza tecnica, 24 ore su 24 7 giorni su 7.

Esempi RedHat Enterprise Linux, Suse Linux Enterprise Server, CentOS, Ubuntu Server.

Distribuzioni Real-Time

Una distribuzione real-time viene utilizzata in ambito industriale e le applicazioni presenti devono garantire dei particolari tempi di risposta.

Per questo motivo, queste distribuzioni possono essere di tipo *hard* se richiedono una rigida precisione sui tempi di risposta oppure *soft* se la precisione sui tempi di risposta è tollerabile.

Esempi Jacklab Audio Distribution, MontaVista Linux, Zentropix RTLinux, REDSonic.

Distribuzioni Embedded

Una distribuzione embedded viene utilizzata per gestire tutte le funzionalità presenti su *piattaforme hardware esotiche*. Con il termine piattaforma hardware esotica intendo un sistema elettronico a microprocessore progettato appositamente per una determinata applicazione.

Contrariamente ai computer classici, un sistema embedded ha dei compiti conosciuti già durante lo sviluppo, pertanto si può ridurre l'hardware ai minimi termini in modo da ridurre lo spazio occupato, i consumi ed il costo di fabbricazione.

[2] Centro Elaborazione Dati

Esempi Linux Embedded Appliance Firewall, uCLinux, Qtopia, OpenWRT.

1.3 Famiglie di distribuzioni

Le distribuzioni desktop, precedentemente definite (vedi paragrafo 1.2.2), rappresentano le tipiche distribuzioni general-purpose.

Tra le tante distribuzioni attualmente esistenti, possiamo suddividerle in tre grandi famiglie:

1. Red Hat, come Fedora, Mandriva;
2. Slackware, come SUSE, Gentoo;
3. Debian, come Knoppix, Ubuntu.

Le due principali caratteristiche che differenziano queste famiglie di distribuzioni sono: gestione dei *runlevel* ed il sistema di pacchettizzazione (che vedremo nel paragrafo 2.5).

Con il termine runlevel si intende una configurazione software del sistema che permette solo ad un gruppo selezionato di processi di esistere. Il processo `init`[11] controlla autonomamente i processi richiesti da un particolare runlevel. Per convenzione esistono sette runlevel, numerati da zero a sei.

Quando un computer entra in runlevel zero, si spegne; quando entra nel runlevel sei, si riavvia. I restanti runlevel (1-5) cambiano da distribuzione a distribuzione. I runlevel con valore basso vengono, di solito, utilizzati per amministrazione o riparazione di emergenza del sistema operativo e non offrono alcun servizio di rete.

Il sistema di inizializzazione adottato da Debian e Red Hat è il classico *System V*, in cui per ogni runlevel esiste una directory — chiamata `rc`*`X`*`.d` dove *`X`* rappresenta il numero del runlevel. Per Debian queste directory si trovano all'interno della directory `/etc`, mentre Red Hat utilizza `/etc/rc.d`.

Tutti gli script che inizializzano i servizi del sistema si trovano all'interno della directory `/etc/init.d` e per ogni runlevel vengono creati dei link simbolici ai rispettivi script.

Slackware, d'altro canto, utilizza un metodo che la avvicina molto ai sistemi `*BSD`, dove per ogni runlevel ha uno ed un solo script di inizializzazione. Ciò permette una struttura organizzativa facile da gestire.

É possibile vedere l'albero genealogico delle distribuzioni su questo sito web:

http://futurist.se/gldt/

1.4 Lavoro di tesi

Le distribuzioni, come abbiamo visto precedentemente, possiedono delle caratteristiche che le rendono uniche.

La procedura di installazione sul disco rigido di un sistema operativo, tipicamente, essere riassunta nei seguenti punti:

- la preparazione dell'ambiente di lavoro,
- il partizionamento del disco rigido,
- la scelta e installazione degli strumenti di sistema,
- l'installazione del software,
- l'installazione del bootloader.

Queste operazioni rappresentano, quindi, la base comune in tutte le distribuzioni.

Il frutto del mio lavoro di tesi può essere espresso come:

> *"un costruttore di distribuzioni basate su Slackware"*.

In altre parole, lo script agevola la costruzione di un `CDROM`, che installa sul disco una distribuzione Linux, basata su Slackware. Durante il processo di costruzione, lo sviluppatore può personalizzare la distribuzione, creandone, di conseguenza, una nuova.

Tra le varie distribuzioni esistenti, è stata scelta Slackware, poiché, data la sua eleganza ed essenzialità, è facile da studiare e da modificare. Le altre distribuzioni, seppur orientate all'utente, adottano tecnologie più complesse e difficili da modificare.

Rendere semplice il processo di costruzione di una distribuzione permette allo sviluppatore di esprimere le proprie esigenze e condividerle in piena libertà con la comunità.

Supponiamo che uno sviluppatore deve gestire un cluster, potrebbe avere la necessità di ripetere all'incirca le stesse operazioni per ogni suo nodo. Tramite l'utilizzo di un *costruttore di distribuzioni*, lo sviluppatore può aggiungere regole adattabili durante la fase di installazione. Quindi, l'installazione dei nodi del cluster risulta veloce e accurata.

D'altro canto, l'utilizzatore di una distribuzione potrebbe avere esigenze simili, le quali, opportunamente integrate dallo sviluppatore, completano la distribuzione stessa.

L'obiettivo da raggiungere è ambizioso, poiché trasforma un utente che non ha approfondite conoscenze sull'argomento in un inventore di distribuzioni. Per realizzare quest'obiettivo si è utilizzato un linguaggio di programmazione molto astratto: lo *shell scripting*[8].

Lo shell scripting permette di usare facilmente differenti tecnologie presenti nelle applicazioni dell'ambiente di lavoro. Quindi, è particolarmente adatto per implementare un "costruttore di distribuzioni".

Nei capitoli successivi si tenterà di mettere il lettore nella condizione di scrivere un suo "costruttore di distribuzioni" ed in questo percorso saranno estratti alcuni esempi dal mio lavoro di tesi.

Il codice sorgente che ho implementato è liberamente scaricabile dal sito:

http://vinx.tuxfamily.org/my_distro

Capitolo 2

Di cosa abbiamo bisogno

In questo primo capitolo, gettiamo le fondamenta sugli strumenti necessari per poter affrontare la restante parte del testo.

E precisamente, qui vengono trattati i seguenti argomenti:

- il supporto multimediale usato e il suo filesystem (`CDROM` & `ISO9660`);
- il software che lancia il sistema operativo (bootloader);
- la tipologa dei kernel esistenti ed il kernel utilizzato (Linux);
- una memoria volatile costruita ad hoc e gli strumenti per la gestione del processo di installazione (`initrd` & shell & `udev`);
- la tipologia dei pacchetti software.

2.1 Supporto multimediale

Per poter installare una distribuzione sono state inventate molte tecniche efficaci che usano i dispositivi più disparati, come `CDROM`, `DVD`, periferiche di rete, dischi a stato solido, floppy, ecc...

Per rendere semplice il progetto, il dispositivo utilizzato è il `CDROM`.

Esso possiede una *struttura fisica*[15]. La più piccola entità presente viene chiamata *frame* ed è pari a 24 bytes. I dati all'interno di un `CDROM` vengono organizzati in frame e settori. Un settore del `CDROM` contiene 98 frame, ed è pari a 2351 bytes.

Un `CDROM` può avere due modalità di formati:

- La modalità 1 di un `CDROM`, viene generalmente utilizzato per i dati dei computer, dove suddivide i 2352 byte dell'area predisposta per i dati

in 12 byte per le informazioni di sincronizzazione, 2048 byte per i dati dell'utente e 288 byte per trovare e correggere gli errori.

- La modalità 2 può avere due formati:
  - il primo formato, partiziona ogni singolo settore con la stessa tecnica utilizzata per la modalità 1; ma il suo utilizzo non è raccomandato per ragioni di compatibilità.
  - il secondo formato, invece, viene utilizzato per dati di tipo audio e video, pertanto i 2352 byte vengono suddivisi in 12 byte per le informazioni di sincronizzazione, 4 byte per gli header dei dati e 2336 byte per i dati dell'utente. Dato che non sono presenti byte per effettuare il checksum avremo il 14% di spazio in più rispetto alla modalità 1.

Nel nostro caso usiamo il supporto con la modalità 1.

Per poter rendere usabile un `CDROM` dobbiamo utilizzare un *filesystem*, chiamato `ISO9660`.

Le specifiche `ISO9660` possiedono un'area riservata di 32768 byte all'inizio del disco, pari a 16 settori, chiamata *System Area*. Generalmente quest'area viene azzerata, rimanendo non utilizzata. Un disco `ISO9660` non è bootabile per default, vi sono però alcune estensioni standard che forniscono servizi di booting per questi dischi. Una delle estensioni di `ISO9660` è la "El Torito Specification"[16]. Tale estensione permette di sfruttare la System Area in modo da poter rendere il `CDROM` bootabile.

I due bootloader più conosciuti che seguono le specifiche "El Torito" sono: *isolinux*[1] e `GRUB`[2].

2.2 Bootloader

Il bootloader è un programma che permette di far partire un sistema operativo. È composto da un insieme di operazioni macchina che effettuano il cosiddetto bootstrapping[1]; è una tecnica che permette a un piccolo programma, il bootloader, di attivare un programma più grande e complesso, il sistema operativo, contenuto in un hard disk o in un altro sistema di memorizzazione permanente. La maggior parte dei bootloader offrono all'utente alcune opzioni sulla particolare modalità nel lanciare il sistema operativo o addirittura consentono di scegliere quale sistema operativo avviare (se sul computer ne sono installati diversi). Tali opzioni possono essere lanciate

[1] dal detto inglese "*to lift oneself by one's own bootstrap*"

in un'opportuna runtime shell oppure scritte nel file di configurazione del bootloader.

In questo contesto spiegheremo sia come utilizzare il bootloader `GRUB` che `isolinux`.

2.3 Kernel

Il kernel costituisce il *cuore* di un sistema operativo. Si tratta di un programma che, da un lato, comunica direttamente con l'hardware e, dall'altro, fornisce delle `API`[2] di più alto livello alla restante parte del sistema operativo (in modo da "nascondere" la complessità e semplificare il lavoro degli sviluppatori). I kernel si possono classificare in quattro grandi categorie:

- **Kernel monolitici**: implementano direttamente una completa astrazione dell'hardware sottostante.

  Forniscono un'insieme di `API` per implementare i servizi del sistema operativo come la gestione dei processi, multitasking e gestione della memoria, filesystem, stack di rete, ecc... Possono essere estesi tramite *moduli* che girano a livello kernel, in modo tale da adattarsi ai dispositivi presenti in un computer.

  Per modulo intendiamo una componente del kernel (ad esempio, per gestire i supporti rimovibili) che può essere inserita o rimossa dinamicamente con la parte statica del kernel.

  Esempi: i kernel `UNIX™`, il kernel Linux.

- **Microkernel**: implementano un insieme ristretto di operazioni rispetto ai kernel monolitici e utilizzano dei *device driver* per fornire maggiori funzionalità.

  Esempi: AIX, BeOS, Mach, Minix.

- **Kernel ibridi**: sono dei microkernel con delle funzionalità modificate per incrementare le prestazioni.

  Esempi: Microsoft Windows NT, XNU, DragonFly BSD, ReactOS.

- **Esokernel**: gestiscono unicamente l'accesso concorrente all'hardware, mentre le singole applicazioni useranno delle speciali librerie.

[2] `A`pplication `P`rogramming `I`nterface

Esempi: Nemesis.

La distribuzione sarà basata sulla versione 2.6.20 del kernel Linux[3].

2.3.1 Approccio statico e dinamico

Il `CDROM`, essendo un dipositivo che può essere condiviso tra più utenti, dovrà funzionare su differenti tipologie di hardware. Per poter gestire questa situazione, la lista dei dispositivi hardware da supportare viene scelta in base alle necessitá dello sviluppatore della distribuzione.

Tale lista viene definita nella configurazione del kernel. Il kernel linux puó essere configurato[12] usando due approcci differenti: *statico* e *dinamico*.

L'approccio statico crea un kernel monolitico e quindi, i vari dispositivi hardware vengono gestiti da driver collegati staticamente nel kernel. Questo implica che il kernel Linux è esente da moduli.

L'approccio dinamico crea un kernel modulare, ovvero i vari dispositivi hardware sono gestiti da driver collegati dinamicamente nel kernel.

Questi due approcci comprendono caratteristiche duali tra loro.

L'utilizzo di un kernel monolitico implica che:

- tutti i driver sono sempre caricati, anche quelli non utilizzati;
- se la lista dei dispositivi hardware è completa allora la directory `/dev` si popola correttamente;
- lo spazio occupato sul `CDROM` è contenuto;
- al momento dell'installazione di un nuovo kernel è difficile reperire le informazioni sui driver utilizzati.

Al contrario, con un kernel modulare abbiamo che:

- i moduli possono essere inseriti o rimossi dinamicamente dal kernel, con un conseguente risparmio di memoria volatile occupata;
- bisogna usare un metodo efficiente per inserire i moduli realmente utilizzati (vedi capitolo 5);
- richiede molto spazio sul `CDROM` poiché ogni modulo è definito da un file oggetto e possiede informazioni aggiuntive necessarie al suo inserimento/rimozione nel kernel;
- è semplice conoscere la lista dei moduli utilizzati ed adattarla ad un nuovo kernel;

	Statico	Dinamico
Vantaggi	spazio su disco gestione dei driver	memoria volatile occupata configurazione finale
Svantaggi	memoria volatile occupata configurazione finale	spazio su disco gestione dei driver

Tabella 2.1: Confronto tra l'approccio statico e quello dinamico

Per entrambi gli approcci è necessaria un'estensione del kernel: l'`initrd`. Questo viene caricato nella `RAM`[4] come `RAM disk`, subito dopo la fase di bootstrap del kernel.

In questo lavoro di tesi, è stato inizialmente seguito l'approccio statico per poter identificare la lista di driver necessari ad un generico computer. In seguito, tale lista è stata applicata all'approccio dinamico poiché quest'ultimo si adatta facilmente alle esigenze dello sviluppatore.

La tabella 2.1 riassume quanto precedentemente detto.

2.4 Initrd

`initrd`[10] significa *initial ramdisk* e contiene i programmi necessari per un corretto *start-up* del sistema operativo.

Nel nostro contesto abbiamo bisogno di uno "spazio virtuale" che non vada a sporcare il disco rigido e sia facile da rimuovere. Entrambe queste caratteristiche sono rappresentate implicitamente dalla `RAM`, dato che è in grado di contenere dei dati volatili.

Il processo di bootstrap[11] del kernel Linux consiste nei seguenti passi:

- chiama la funzione `setup()`, che:
  - re-inizializza i dispositivi hardware nel computer e imposta l'ambiente per l'esecuzione del kernel;
  - imposta l'`IDT` (`Interrupt Descriptor Table`) e il `GDT` (`Global Descriptor Table`);
  - passa in modalitá protetta;
- chiama la prima funzione `startup_32()`, che decomprime l'immagine del kernel;

[3]Linux è uno Unix-like, inventato da Linus Torvalds nel 1991 e rilasciato sotto licenza `GPL`, è reperibile sul sito *http://www.kernel.org*

[4]`R`andom `A`ccess `M`emory

- chiama la seconda funzione `startup_32()`, che imposta l'ambiente per l'esecuzione del primo processo di Linux, il *processo 0*;
- chiama la funzione `start_kernel()`, che completa l'inizializzazione del kernel e crea il processo `init`, il *processo 1*;
- chiama la funzione `init()`, che è l'ultima funzione chiamata durante la sua inizializzazione;
- infine, quest'ultima chiama la funzione `sys_execve()`, che esegue un file eseguibile, tipicamente `/sbin/init`.

Tramite l'`initrd` l'ultimo punto del processo di bootstrap esegue il file `/linuxrc`. Questo file viene utilizzato per inserire i moduli necessari al riconoscimento del disco rigido e del filesystem su di esso utilizzato.

É possibile omettere l'`initrd` solo nel caso in cui il kernel riesce automaticamente a riconoscere il disco e tipo di filesystem utilizzato; ma ciò non rientra nel nostro contesto.

L'`initrd` può essere popolato da una suite di programmi. Questi combinati con opportuni shell script costituiranno l'*installer* della distribuzione. Gli obiettivi dell'installer consistono:

- nel selezionare i moduli da inserire per il riconoscimento dei dispositivi hardware,
- nell'inizializzazione della directory `/dev` utilizzando il pacchetto `udev`[5] (vedi paragrafo 2.4.2),
- nel costruire e configurare la distribuzione sul disco.

L'`initrd` può essere costruito in due modi:

- come filesystem compresso. Per poter visualizzare i dati contenuti al suo interno, basta decomprimere il file utilizzando `gunzip` e successivamente montarlo utilizzando il comando:

  `mount -o loop` ***`file mntp`***

  dove ***`file`*** rappresenta l'`initrd`, mentre ***`mntp`*** rappresenta il punto di mount.

  L'opzione `loop` permette di montare un ***`file`*** che non è presente nella directory `/dev`. Per poter funzionare, il kernel deve avere abilitato il

[5] `udev` sostituisce definitivamente `devfs` a partire dal kernel 2.6.18

supporto per i *loop device*. Infatti, usando questa opzione viene messo in relazione un loop device (ad esempio, `/dev/loop0`) con il *`file`* che si ha intenzione di montare[6].

- come archivio cpio. Questo è il successore di `initrd`, chiamato *initramfs*. In questo caso per accedere ai dati si può utilizzare la seguente riga di comando:

  `gunzip -c <` *`file`* `| cpio -i -d` *`dest`*

  dove decomprime l'archivio contenuto nel *`file`* stampandolo, tramite l'opzione `-c`, sullo *standard output*. Tale archivio viene dato in pasto a `cpio` che estrae le informazioni nella directory *`dest`*.

In questo testo costruiremo l'`initrd` utilizzando il filesystem compresso.

Il suo funzionamento, per default, consiste nell'eseguire il programma `/linuxrc` contenuto al suo interno. Se tale programma non esiste oppure è un link simbolico alla *shell*, presente all'interno dell'`initrd`, allora esegue un programma definito nel file `/etc/inittab`.

Pertanto i programmi base per poter costruire un installer sono: la shell ed `udev`.

2.4.1 Shell

In un sistema operativo, la shell è il programma che permette agli utenti di avere un "ambiente di lavoro" in grado di comunicare con il sistema e di impartire dei comandi. Tali istruzioni testuali che vengono impartite attraverso la tastiera per eseguire programmi, visualizzare il filesystem, interagire in ogni modo con il computer, insomma un vero e proprio coltellino svizzero. Sono inoltre presenti potenti strumenti che permettono di unire tra loro più comandi in modo da effettuare operazioni molto complesse, come le pipe e la redirezione. La maggior parte delle shell possiedono un vero e proprio linguaggio di scripting, molto potente, con il quale si possono creare degli script per automatizzare l'amministrazione del sistema.

La shell che abbiamo scelto nel nostro caso è `BusyBox`[3]. Difatti, `BusyBox`, è più di una shell, include differenti comandi standard di `UNIX`™, come ad esempio `ls`, `cp`, `find`, `tar`, `sed`, ecc... Inoltre comprende una shell a scelta tra `ash`, `hush`, `lash` e `msh`. Può essere costruita in modo statico oppure linkata alle `glibc`[4] o alle `uClibc`[5]. Nel nostro caso, abbiamo optato nella costruzione di `BusyBox` in modalità statica, in modo da avere un ambiente di lavoro

[6] per ulteriori informazioni leggere `man mount`

minimale ed indipendente. Il paragrafo 4.1.1 spiega come inserire `BusyBox` nell'`initrd`.

2.4.2 udev

`udev` è un programma che gestisce la directory `/dev`. A differenza dei sistemi Linux tradizionali, dove le componenti venivano rappresentate nella directory `/dev` con un'insieme statico di file, `udev` fornisce dinamicamente solo i nodi dei dispositivi che sono attualmente presenti sul sistema.

Su una distribuzione Linux gira come un daemon ed ascolta gli eventi che il kernel manda tramite il filesystem `sysfs` (usando dei socket netlink) su di esso, in modo da creare o rimuovere un device file. È possibile generare un insieme di regole a seconda del tipo di dispositivo. Infatti, `udev` supporta il "persistent storage device naming"[17], che garantisce l'univocità del nome dei dispositivi presenti sul sistema secondo lo standard `LSB`[6]; e viene eseguito interamente in user space.

`udev` emette messaggi sul `D-BUS`, in modo tale che ogni altro programma in user space (come `HAL`[7]) può capire quali device file vengono creati o rimossi. Ad esempio se togliamo una penna `USB` udev reagisce nel seguente modo:

- rimuove i dipositivi associati alla penna `USB`:

```
/devices/pci0000:00/0000:00:03.3/usb1/1-2/1-2:1.0/ \
 usb_endpoint/usbdev1.5_ep81 (usb_endpoint)
/devices/pci0000:00/0000:00:03.3/usb1/1-2/1-2:1.0/ \
 usb_endpoint/usbdev1.5_ep02 (usb_endpoint)
```

- rimuove le risorse `SCSI` utilizzate:

```
/class/scsi_generic/sg1 (scsi_generic)
/class/scsi_device/2:0:0:0 (scsi_device)
/class/scsi_disk/2:0:0:0 (scsi_disk)
```

- rimuove, infine, il device file:

```
/block/sdb/sdb1 (block)
/block/sdb (block)
```

[7] Hardware Abstraction Layer

2.5 Codice Sorgente & Pacchetti

Un utente, a seconda delle sue necessitá, deve poter installare o rimuovere del software.

Tutto il software `FOSS` viene rilasciato sotto forma di codice sorgente. Il processo di compilazione può a volte essere molto macchinoso, sia da parte dello sviluppatore software che da parte dell'utente finale che deve generare il programma. Per agevolare questo compito, nella famiglia dei sistemi operativi `UNIX`™ ed unix-like, viene utilizzata la suite di programmi `autotool`[7]. Ad esempio, un qualsiasi utente per costruire un programma utilizzerà i seguenti comandi:

```
./configure
make
make install
```

dove

- `./configure` permette di adattare il codice sorgente alle volontà dell'utente finale.
- `make` legge il file `Makefile`, generato con il precedente comando, ed esegue tutte le operazioni necessarie per poter compilare il codice sorgente. Alla fine dell'esecuzione di questo comando vengono generati gli eseguibili e le librerie dell'applicazione.
- `make install` legge una particolare sezione del `Makefile` che ci permette di installare il programma. Quest'ultimo comando necessita dei privilegi dell'utente root poiché le directory di installazione, solitamente, sono amministrate proprio da quest'utente.

Tuttavia questo procedimento, anche se molto innovativo, non soddisfa completamente l'utente finale. Infatti, la compilazione fa perdere molto tempo, modificare il sorgente richiede da parte dell'utente una conoscenza approfondita del linguaggio di programmazione utilizzato, aggiornare il programma con una versione più recente potrebbe essere difficile da gestire, ecc... Per ovviare a questo tipo di inconvenienti la maggior parte delle distribuzioni utilizzano i *pacchetti software*.

Questo genere di pacchetti sono dipendenti dal tipo di distribuzione e non necessitano della presenza di un compilatore per poter essere installati. Le tipologie di pacchetto più famose nelle distribuzioni basate su kernel Linux sono: `.deb`, `.rpm` e `.tgz`[8].

[8] rispettivamente per distribuzioni basate su Debian, Red Hat e Slackware

Ogni pacchetto software viene mantenuto da una "persona esperta" che modifica il sorgente, lo compila e costruisce il pacchetto garantendo il suo funzionamento nella distribuzione. Tutti i pacchetti contengono una firma di tipo `PGP`[9] che ne garantisce l'univocità su tutta internet, dato che essi sono disponibili al pubblico nei cosiddetti *mirror*. Tale firma è sempre presente e viene gestita in modo diverso a seconda del tipo di pacchetto, ad esempio:

- i pacchetti `.rpm` contengono nel loro header una firma `PGP` che assicura la sua integrità ed originalità;
- la firma dei pacchetti `.deb` è contenuta in un file separato avente estensione `.dsc`;
- come i pacchetti `.deb`, i `.tgz` di slackware possiedono tale firma in un file con estensione `.asc`.

Anche i mirror della distribuzione, per essere identificati come mirror ufficiali, possiedono una firma `PGP`. Tramite l'utilizzo delle firme si evitano, quindi, eventuali attacchi alla sicurezza della distribuzione.

Ogni distribuzione possiede un tool per l'installazione/rimozione ed aggiornamento dei pacchetti: per Debian abbiamo `dpkg`, per Red Hat abbiamo `rpm` e per Slackware abbiamo i `pkgtools`. Questo genere di programmi possono risolvere le dipendenze tra i pacchetti in modo da garantire il corretto funzionamento dell'intera distribuzione. Infatti, per ogni nuovo pacchetto che viene installato, questi programmi salvano tutte le sue informazioni in un opportuno database.

Un singolo pacchetto può avere differenti versioni, pertanto i tool messi a disposizione da una distribuzione permettono di scegliere automaticamente la versione più adatta al sistema che abbiamo installato sul disco.

I pacchetti possiedono al loro interno:

- file eseguibili e librerie pre-compilate, che consistono nel programma vero e proprio,
- file di configurazione, che permettono la configurazione ed eventualmente l'aggiornamento del programma.

I pacchetti `.deb` di Debian sono in realtà dei file compressi con `ar`. Contengono tre archivi in formato `tar.gz`. Due di questi sono di particolare interesse - `data.tar.gz` e `control.tar.gz`. `data.tar.gz` contiene i file che verranno installati su disco. `control.tar.gz` contiene la descrizione del pacchetto e i vari script che verranno lanciati al momento dell'installazione.

[9] Pretty Good `Privacy`

I pacchetti `.rpm` di Red Hat sono degli archivi `cpio` modificati. Infatti per convertire un `.rpm` in un normale archivio `cpio` bisogna utilizzare un programma chiamato `rpm2cpio`. Queste modifiche servono ad effettuare il controllo dei dati del pacchetto (informazioni sulle dipendenze, descrizione del pacchetto - molto simile a `control.tar.gz` dei pacchetti Debian).

I pacchetti `.tgz` di Slackware sono degli archivi `tar` compressi tramite `gzip`. Ogni pacchetto è costituito da directory che contengono i dati ed una directory chiamata `install`, nella quale troviamo lo script di installazione ed il file che fornisce una breve descrizione del pacchetto.

Questi pacchetti possono essere convertiti, a discrezione dell'utente, da un formato all'altro, ad esempio:

- abbiamo dei programmi specifici come `rpm2targz` converte un pacchetto `.rpm` in un pacchetto `.tgz`;
- oppure tool più generici come `alien` che converte a seconda delle opzioni date un qualsiasi pacchetto in un pacchetto `.rpm`, `.tgz` o un `.deb`.

Capitolo 3

Costruzione di un CDROM bootabile

L'obiettivo di questo capitolo consiste nel costruire un `CDROM` bootabile supponendo di avere almeno un kernel e un `RAM` disk pronti. Questi ultimi due argomenti verranno trattati nel prossimo capitolo.

Le applicazioni utilizzate a questo scopo sono: un bootloader che gira sui supporti di tipo `CDROM` e `mkisofs`.

Il bootloader deve possedere l'estensione "El Torito" che permette la scrittura di dati all'interno della *System Area*, come specificato nello standard `ISO9660` (vedi paragrafo 2.1). I due bootloader più famosi sono `isolinux` e `GRUB`.

Il programma `mkisofs` consente di amalgamare il bootloader con i dati da inserire sul `CDROM`. Ad esempio, abbiamo bisogno di un kernel e un ramdisk per l'inizializzazione del sistema operativo su questo supporto. Il risultato finale consiste nell'ottenere un'immagine `.iso`, ovvero un filesystem di tipo `ISO9660`.

Tra le varie opzioni presenti in `mkisofs` utilizziamo l'estensione, chiamata *Joliet*, per abilitare il supporto ai file che possiedono nomi lunghi e caratteri non `ASCII`.

3.1 Isolinux

Il bootloader che supporta l'estensione "El Torito" secondo le specifiche `ISO9660` è `isolinux` è contenuto nel pacchetto `syslinux`[1].

Il pacchetto `syslinux` è basato su bootloader `LILO` e contiene:

- `pxelinux`, che ci permette di fare il boot utilizzando la propria `LAN`[1];
- `extlinux`, che viene usato per fare il boot da penne `USB`, `MMC` e supporti simili;
- `isolinux`, che ci servirà per fare il boot da `CDROM`.

Una volta ottenuti i sorgenti dal questo indirizzo:

http://www.kernel.org/pub/linux/utils/boot/syslinux/

per compilare, solo ed unicamente, `isolinux` è sufficiente scrivere il comando:

```
make isolinux.bin
```

Dato che è stato scritto in assembly con sintassi Intel deve essere presente il pacchetto `nasm` all'interno della distribuzione.

Per rendere bootabile il `CDROM` abbiamo bisogno di una gerarchia di directory, ossia il bootloader `isolinux` funziona solo se viene posizionato all'interno di una omonima directory nella root del `CDROM`.

3.1.1 Configurazione Isolinux

`isolinux` permette di avere un file di configurazione, chiamato `isolinux.cfg`. Questo file deve essere posizionato nella stessa directory in cui si trova il bootloader.

Il file di configurazione ci permette di specificare dove andare a prendere il kernel Linux e con quali opzioni lanciarlo, ad esempio una possibile configurazione può essere questa:

```
DEFAULT mydistro

PROMPT 1

TIMEOUT 100

LABEL mydistro

KERNEL /boot/vmlinuz

LABEL mydistro_rw

KERNEL /boot/vmlinuz

APPEND rw
```

[1]Local Area Network

In questo esempio, impostiamo il kernel di default utilizzando un'etichetta (in inglese *label*) che nel nostro caso si chiama `mydistro`.

L'etichetta ci fornirà le informazioni necessarie per trovare il kernel e le opzioni ad esso collegate, in questo caso il kernel si troverà nella directory `/boot` del `CDROM` e si chiamerà `vmlinuz`, ma non avrà nessuna opzione.

In alternativa, possiamo scegliere l'altra etichetta, `mydistro_rw`, in cui selezioniamo lo stesso kernel, ma viene passata l'opzione `rw` come parametro. In questo modo, si può leggere e scrivere il filesystem che verrà montato dopo la fase di boot del kernel stesso.

Per poter scegliere quest'opzione abbiamo a disposizione 100 secondi come indicato nella direttiva `TIMEOUT`.

3.2 Grub

Grub è un pacchetto molto noto in ambienti Linux, oltre a permettere il boot da un normale hard disk è possibile anche utilizzare una sua estensione per effettuare il boot da `CDROM`.

In questo caso però, al contrario di `isolinux` siamo costretti a compilare interamente il pacchetto utilizzando questa sequenza di comandi:

```
./configure --disable-ext2fs --disable-fat \
 --disable-ffs --disable-ufs2 \
 --disable-reiserfs --disable-minix \
 --disable-vstafs --disable-jfs \
 --disable-xfs
make
```

dove `./configure` sta si trova all'interno della directory dei sorgenti e serve a configurare l'intero pacchetto, pertanto si può usare per fornire delle direttive particolari che nel nostro caso consistono nel costruire il pacchetto abilitando soltanto il supporto per il filesystem `ISO9660`.

Anche in questo caso abbiamo bisogno di una particolare gerarchia di directory, infatti dobbiamo inserirlo nella directory chiamata `grub` che sarà costruita all'interno del `CDROM`.

3.2.1 Configurazione Grub

Anche il bootloader `GRUB` permette di avere un file di configurazione che per default prende il nome `menu.lst` e si trova nella stessa directory in cui si trova il bootloader. Vediamone un esempio:

```
default 0
timeout 100
title=mydistro
kernel (cd)/boot/vmlinuz
title=mydistro_rw
kernel (cd)/boot/vmlinuz rw
```

Come si vede la sintassi differisce da `isolinux` ma il suo comportamento rimane invariato.

3.3 Creazione della ISO

A questo punto, una volta scelto il bootloader da utilizzare e la gerarchia di directory, possiamo costruire l'immagine da masterizzare.

Il comando che useremo si chiama `mkisofs` ed avrà le seguenti opzioni:

```
mkisofs -R -b path_and_loader -c boot.catalog \
 -no-emul-boot -boot-load-size 4 \
 -boot-info-table -o isoname isodir
```

Analizziamo passo passo il significato di queste opzioni ed il valore delle variabili che ho introdotto. L'opzione:

- `-R` crea l'`ISO` secondo il protocollo Rock Ridge; senza questa estensione l'immagine si comporterà nello stesso modo in cui faceva il `DOS`: otto caratteri al massimo, seguiti da un punto e da un'estensione di un massimo di tre caratteri; senza contare la totale assenza degli attributi che possono avere i file. In questo modo si possono salvare tutte le informazioni tipiche dei sistemi Unix.

- `-b` *path_and_loader* indica il percorso e il nome del bootloader all'interno dell'immagine.

- `-c boot.catalog` indica il percorso e il nome del file che sfrutterà le specifiche "El Torito".

- `-no-emul-boot` specifica che l'immagine di boot usata per creare un `CDROM` bootabile non viene emulata, cioè quando il sistema caricherà ed eseguirà quest'immagine non verrà rappresentata come un'emulazione di un disco.

- `-boot-load-size 4` ci permette di specificare quanti settori virtuali del `CDROM` usare in modalità non emulata dato che su alcuni `BIOS` possono avere problemi se i settori non sono multipli di 4.
- `-boot-info-table`, imposta i 56 byte della tabella contenente le informazioni del `CDROM` nel bootloader in modo da rendere l'immagine non modificabile in seguito.
- `-o` *`isoname`* è l'immagine vera e propria che verrà generata e *`isoname`* rappresenta il suo nome.
- *`isodir`* rappresenta la directory in cui sono contenuti i dati (bootloader, kernel, varie ed eventuali) dell'immagine che vogliamo creare.

Per creare un'immgine `ISO` basata sul bootloader `GRUB` la directory *`isodir`* deve contenere il seguente albero di directory:

`isodir` → `boot` → `grub`

dove nella sottodirectory `boot` sarà presente il kernel Linux e l'`initrd`, mentre nella sottodirectory `grub` il bootloader. In questo caso, la variabile *`path_and_loader`* sarà pari a `boot/grub/stage2_eltorito`.

Mentre per creare un'immagine `ISO` basata sul bootloader `isolinux` la directory *`isodir`* contiene la sottodirectory `isolinux` nella quale sarà presente sia il kernel Linux che il bootloader. Quindi, la variabile *`path_and_loader`* sarà pari a `isolinux/isolinux.bin`.

Capitolo 4

Costruire l'initrd

Questo capitolo è sull'`initrd`.

Vedremo come viene costruito l'`initrd` e quali applicazioni ne faranno parte.

Verrà accennato come avviene la configurazione e la costruzione del kernel Linux in modo da abilitare l'utilizzo dell'`initrd`.

Infine, verranno forniti degli esempi su come configurare il bootloader per abilitare l'`initrd`.

4.1 La struttura dell'initrd

Come abbiamo precedentemente detto nel paragrafo 2.4, l'`initrd` serve a contenere tutti i programmi utili per poter installare la distribuzione sul disco e deve essere caricato in `RAM`. Nello specifico, è file compresso contenente un filesystem.

Per costruire questo file utilizziamo una comoda applicazione presente nel pacchetto `coreutils` di `GNU`, `dd`, acronimo di *disk dump*. Il comando che lanceremo é

```
dd if=/dev/zero of=name bs=brd count=size
```

dove

- `if=/dev/zero`, `/dev/zero` è file speciale che stampa il valore '0', pertanto lo usiamo per riempire il nostro file.
- `of=`*`name`*, *`name`* indica il nome che daremo all'`initrd` e verrà riempito da zeri.

- `bs=`*`brd`*, *`brd`* indica quant'é grande un singolo blocco del `RAM` disk, solitamente questo valore sarà uguale a quello impostato nella configurazione del kernel (vedi variabile *Z* del paragrafo 4.2.1).

- `count=`*`size`*, *`size`* indica quanti blocchi creare, questo valore varierà a seconda dello spazio totale occupato dai file che andranno a popolare l'`initrd`.

Una volta generato il file, viene formattato con un qualsiasi filesystem compilato in built-in nel kernel. Normalmente, l'`initrd` basato su un kernel Linux viene formattato in `ext2`, pertanto usiamo questo comando per crearlo:

```
mkfs -t ext2 name
```

Dopo la formattazione, il file viene montato in una directory di appoggio, rappresentata dalla variabile *`tdir`*, come si vede nel seguente comando:

```
mount -t ext2 name -o loop tdir
```

e creiamo l'albero delle directory necessario nella directory *`tdir`*, che è riassumibile tramite questo comando:

```
mkdir -p tdir/{{,s}bin,dev,mnt,sys,etc/{{init,conf}.d,i18n},\
    lib/modules,proc,tmp,usr/share/terminfo/l}
```

Dopo aver copiato tutti i file che faranno parte dell'`initrd` — come sarà spiegato nei sottoparagrafi successivi, smontiamo il filesystem ed eliminiamo la directory temporanea, con i seguenti comandi:

```
umount tdir
rm -rf tdir
```

Infine, il file creato viene compresso usando l'applicazione `gzip`, ossia:

```
gzip name
```

4.1.1 Costruire BusyBox

La suite di programmi, che rappresenta la struttura portante del processo di installazione della distribuzione, è contenuta all'interno del pacchetto `BusyBox`. Per costruire `BusyBox` si utilizza un procedimento molto simile a quello usato per costruire il kernel Linux (vedi paragrafo 4.2).

Prendiamo i sorgenti dal sito:

http://www.busybox.net

e li scompattiamo tramite questo comando:

```
tar xjvf busybox-versione.tar.bz2
```

A questo punto usiamo i seguenti comandi:

```
make menuconfig
```

```
make
```[1]

dove il primo comando indica la configurazione di `BusyBox`, mentre il secondo serve a compilarla.

Quello che non deve mancare nella configurazione é:

- impostare `BusyBox` in modalità statica, in questo modo evitiamo dipendenze con eventuali librerie,
- la generazione di link simbolici dei vari programmi contenuti all'interno di `BusyBox`, in modo tale da non occupare molto spazio,
- non utilizzare la directory `/usr`, per avere una piccola comodità sulla variabile *`$PATH`*. Infatti, se questa opzione viene abilitata la variabile viene dichiarata nel modo seguente:

  `PATH``=/bin:/sbin`

  Al contrario tale variabile equivale a:

  `PATH``=/bin:/sbin:/usr/bin:/usr/sbin`

- selezionare le applicazioni in grado di gestire i moduli del kernel (`insmod`, `modprobe`, `rmmod`),
- selezionare il programma `mdev`, in modo da avere un'implementazione minimale del pacchetto `udev`[2].

Una volta configurato e compilato il nostro "coltellino svizzero", lo installiamo con questo comando:

```
make PREFIX=tdir install
```

dove *`tdir`* indica una directory di appoggio.

[1] attenzione per compilare `BusyBox` abbiamo bisogno di `/bin/sh` e in alcune distribuzioni come `Ubuntu` deve essere sostituito con un link simbolico che punta a `/bin/bash`

[2] vedi il paragrafo 4.1.2 per utilizzare direttamente `udev`

Note su init & BusyBox

`init` è il processo 1. Viene instanziato appena dopo la fase di bootstrap del kernel Linux.

Per default, quando si usa l'`initrd` il processo `init` è rappresentato dal file eseguibile `/linuxrc`. Questo è definito passando la seguente opzione al kernel:

```
init=/linuxrc
```

Se utilizziamo `BusyBox` il file `/linuxrc` è in realtà un link simbolico al programma `/bin/busybox`. Questa applicazione è stata progettata in modo che a seconda del nome associato al link simbolico questa chiami l'applet[3] associata.

Come abbiamo precedentemente detto nel paragrafo 2.4, il processo `init` viene configurato utilizzando il file `/etc/inittab`. `BusyBox` implementa anche il caso in cui questo file non è presente, infatti, la procedura di `init` di `BusyBox` prova ad eseguire lo script shell `/etc/init.d/rcS`. Se anche quest'ultimo file non è presente sul filesystem allora, in automatico, avremo a disposizione un accesso in *single user mode* sulla shell.

4.1.2 Costruire udev

L'applet `mdev` contenuta all'interno di `BusyBox` può essere limitata per le operazioni che abbiamo intenzione di effettuare.

A tal proposito è possibile usare `udev`, ad esempio, per effettuare il debug delle informazioni fornite dal kernel sul filesystem `sysfs`, come descritto nel paragrafo 5.4.1.

Per poter utilizzare `udev` scarichiamo, innanzitutto, i sorgenti da:

http://www.kernel.org/pub/linux/utils/kernel/hotplug/

e li scompattiamo utilizzando il comando:

```
tar xzvf udev-versione.tar.gz
```

dove *`versione`* è la versione di `udev` che intendiamo utilizzare.

Questo tipo di pacchetto non utilizza né gli `autotools` né la tecnica precedentemente descritta per `BusyBox`. Il nostro fine consiste nel riutilizzare alcune variabili presenti nel `Makefile` in modo da abilitare alcune funzionalità e generare gli eseguibili in staticamente. Per far questo utilizziamo il comando:

[3] rappresenta l'implementazione minimale di un comando supportato da `BusyBox`

```
make USE_STATIC=true \
 EXTRAS=‘‘extras/ata_id extras/dasd_id extras/edd_id \
 extras/floppy extras/path_id extras/scsi_id \
 extras/usb_id extras/volume_id
 extras/rule_generator’’
```

dove le variabili `USE_STATIC` e `EXTRAS` sono presenti nel `Makefile`. Usando questa sintassi andiamo a sovrascrivere i valori originali di tali variabili, in modo da abilitare la creazione dell'eseguibile linkato staticamente (`USE_STATIC`) e dei comodi file di configurazione per le varie periferiche (`EXTRAS`).

Dopo la fase di compilazione, installiamo gli eseguibili generati in questo modo:

```
make DESTDIR=tdir \
 EXTRAS=‘‘extras/ata_id extras/dasd_id extras/edd_id \
 extras/floppy extras/path_id extras/scsi_id \
 extras/usb_id extras/volume_id \
 extras/rule_generator’’ install-bin
```

dove *tdir* indica la directory dove vogliamo andare ad installare i binari. Il passo successivo all'installazione di `udev` consiste nel configurarlo. Per farlo tocca creare una directory:

```
mkdir -p tdir/etc/udev
```

copiarci i file di configurazione e le regole di default:

```
cp -r etc/udev/{udev.conf,rules.d} tdir/etc/udev
```

e a seconda delle necessità modificarle.

4.1.3 Costruire util-linux

Per poter partizionare l'hard disk abbiamo bisogno di utilizzare un'applicazione più user friendly rispetto a `fdisk` presente all'interno del pacchetto `BusyBox`. L'applicazione scelta é: `cfdisk`. Questa è la versione basata sulle librerie grafiche curses di `fdisk` contenuta nel pacchetto `util-linux`.

Per costruire l'applicazione, scarichiamo i sorgenti dal sito:

http://www.kernel.org/pub/linux/utils/util-linux/

e li estraiamo tramite il comando:

```
tar xzf util-linux-versione.tar.gz
```

`util-linux`, normalmente, utilizza le librerie `glibc`. Ma per linkare staticamente l'eseguibile dobbiamo usare il seguente comando prima di lanciare lo script `./configure`:

```
export LDFLAGS=''-static''
```

Successivamente dobbiamo utilizzare una patch per inibire l'utilizzo della libreria `glibc`. Infatti, nella versione utilizzata di `util-linux` (2.12r) viene utilizzata la macro `_syscall5` per la system call `lseek()`. Tale macro, nelle recenti versioni della libreria `glibc` (a partire dalla 2.4.0), non è più presente. Pertanto tale patch si occupa proprio nel ridefinire tale macro nei sorgenti di `util-linux`.

A questo punto generiamo l'eseguibile con i seguenti comandi:

```
cd fdisk
make cfdisk
```

Se vogliamo utilizzare anche il programma `fdisk` di questo pacchetto rispetto alla sua implementazione minimale fornita da `BusyBox` allora dobbiamo lanciare quest'altro comando:

```
make fdisk
```

Infine, copiamo l'eseguibile appena generato nella directory `sbin` presente all'interno dell'`initrd`:

```
cp cfdisk tdir/sbin/
```

e nel caso di `fdisk`:

```
cp fdisk tdir/sbin/
```

4.1.4 Costruire e2fsprogs

A partire dalla versione 1.4.0 di `BusyBox`, non sono più presenti le implementazioni minimali del pacchetto `e2fsprogs`, poiché considerate troppo grandi da mantenere. Questo pacchetto contiene programmi che servono a formattare le partizioni aventi il filesystem `ext2/3` presenti sul disco, pertanto la loro presenza è fondamentale in questo lavoro.

Scarichiamo i sorgenti da un qualsiasi mirror di sourceforge.net, ad esempio:

http://heanet.dl.sourceforge.net/sourceforge/e2fsprogs/

e li scompattiamo:

```
tar xzvf e2fsprogs-versione.tar.gz
```

Questo pacchetto è stato costruito utilizzando la suite `autotools`, pertanto il procedimento di configurazione consiste nell'eseguire il seguente comando:

```
./configure --with-cc='gcc -static'
```

in questo modo riusciremo a costruire l'eseguibile in modalità statica. Verrà compilato utilizzando il comando `make` ed infine lo installiamo nella directory di appoggio che verrà utilizzata dal nostro `initrd`:

```
cp e2fsck/e2fsck tdir/sbin
cp misc/mke2fs tdir/sbin
```

4.2 Costruire un kernel Linux

Il kernel, che verrà utilizzato nel `CDROM`, deve possedere un'opportuna configurazione per poter utilizzare l'`initrd`.

Vi sono vari modi per poter costruirsi un kernel su misura e molte note distribuzioni hanno creato delle applicazioni che automatizzano questo procedimento. La tecnica più comune è composta dai seguenti passi:

```
make menuconfig
make
```

dove il primo comando consiste nella configurazione del kernel, mentre il secondo rappresenta la sua compilazione.

Innanzitutto scarichiamo i sorgenti del kernel dal sito:

http://kernel.org

e li scompattiamo in una directory (solitamente si usa `/usr/src`).

Siccome dobbiamo garantire un alto riconoscimento delle periferiche dobbiamo includere il maggior numero di driver al suo interno.

La tecnica più semplice per costruire un kernel modulare è di costruire inizialmente un kernel monolitico avente solo le componenti "comuni", ad esempio la maggiorparte dei computer possiedono i controller `IDE` pertanto

sicuramente abbiamo bisogno di questa caratteristica per poter riconoscere le periferiche che sfruttano questa interfaccia.

Mentre i controller SCSI, molto utilizzati in ambito *enterprise*, sono difficilmente presenti su un personal computer. Dato che questi componenti costano molto, vengono raramente impiegati in una distribuzione di tipo general-purpose. Per questo motivo, è preferibile compilare i driver di tali periferiche come moduli invece di inserirli staticamente nel kernel.

Graficamente nella configurazione per abilitare un driver in built-in – un driver compilato staticamente all'interno del kernel – bisogna utilizzare l'asterisco, rimanendo nell'esempio precedente abbiamo:

```
<*> Include IDE/ATA-2 DISK support
```

mentre per ottenere un modulo bisogna impostare la lettera M, ad esempio:

```
<M> ATA device support
```

Una volta configurato (vedi il paragrafo 4.2.1) e compilato[12] il kernel dobbiamo installarlo, pertanto utilizzeremo i comandi:

```
cp arch/i386/boot/bzImage rdir/boot/vmlinuz
make INSTALL_MOD_PATH=tdir modules_install
```

dove *tdir* rappresenta la directory di appoggio per l'initrd, mentre la directory *rdir* è la direcory di appoggio per costruire l'immagine del CDROM.

4.2.1 Configurazione del kernel

Il primo passo consiste nel configurare il kernel in modo tale da poter abilitare il RAM disk e il supporto all'initrd stesso. Pertanto abilitiamo nella sua configurazione:

```
-> Device Drivers
    -> Block devices
        <*> RAM disk support
        (X) Default number of RAM disks
        (Y) Default RAM disk size (kbytes)
        (Z) Default RAM disk block size (bytes)
        [*] Initial RAM filesystem and RAM disk
         (initramfs/initrd) support
```

dove

- `X` rappresenta il numero di `RAM` disk ci necessitano,
- `Y` rappresenta quanto deve essere grande un singolo `RAM` disk,
- `Z` rappresenta quanto deve essere grande un singolo blocco del `RAM` disk,
- l'ultima opzione permette di abilitare l'`initrd`.

Un'altra caratteristica che bisogna includere è il supporto ai socket unix, dato che il file system `sysfs` è in grado di comunicare con `udev` (o `mdev`):

```
-> Networking

    <*> Networking support
    -> Networking options

        <*> Unix domain sockets
```

e il supporto alla tastiera, altrimenti l'utente non sarà in grado di inserire le proprie scelte durante l'installazione:

```
-> Device Drivers
 -> Input device support

    <*> Generic input layer
    <*> Support for memoryless force-feedback devices
    <*> Event interface

        [*] Keyboards
        -> Keyboars

            <*> AT keyboard
```

L'obiettivo, in una generica configurazione del kernel, consiste nel garantire, almeno, il supporto per i dispositivi di archiviazione dati, come dischi rigidi (`SATA`,`PATA`,`SCSI`), `CDROM`, penne `USB`, ecc... Senza di questi non è possibile installare alcuna distribuzione. Pertanto, *per tali periferice sono stati abilitati tutti i moduli ad essi associati.*

Anche i filesystem giocano un ruolo importante, sia quelli reali che quelli virtuali, è necessario che siano presenti almeno i seguenti filesystem: `ext2`, `ext3`, `proc` e `sysfs`.

4.3 Configurazione bootloader

Con l'aggiunta di `initrd` dobbiamo effettuare delle modifiche al file di configurazione del bootloader.

Per poter utilizzare il `RAM` disk rappresentato dall'`initrd` viene aggiunta l'opzione `root=/dev/ram0` nella configurazione del bootloader.

Inoltre, dobbiamo specificare qual'é il programma che rappresenti il processo `init`. Per default tale programma si chiama `linuxrc` e si trova nella root directory di `initrd`. `linuxrc` può essere un programma oppure uno shell script. L'opzione del kernel da aggiungere nel file di configurazione del bootloader è `init=/linuxrc`.

Nel caso di `Isolinux` avremo questa possibile configurazione:

```
KERNEL /boot/vmlinuz
APPEND ‘‘initrd=/boot/initrd root=/dev/ram0 load_ramdisk=1
    init=/linuxrc’’
```

Mentre se stiamo utilizzando `GRUB` allora la configurazione sarà:

```
kernel (cd)/boot/vmlinuz root=/dev/ram0 init=/linuxrc
initrd (cd)/boot/initrd
```

Tramite questi esempi stabiliamo come passare alcuni parametri direttamente al kernel e come specificare il file che rappresenta l'`initrd` appena costruito. Come al solito la sintassi tra i due bootloader cambia leggermente.

Capitolo 5

Gestione dei moduli

Il riconoscimento dell'hardware è una delle parti cruciali della fase di startup di una distribuzione. Tale procedimento, per distribuzioni giá installate sul disco, viene gestito in modo statico dato che si è giá a conoscenza delle periferiche presenti sul computer. Mentre, per sistemi che devono installare una distribuzione, questo procedimento deve essere dinamico.

In questo capitolo, pertanto, viene esposta una soluzione, basata sul caricamento/scaricamento dei moduli del kernel, a questo argomento, in modo da garantire un corretto riconoscimento dell'hardware di un qualsiasi computer.

5.1 Ambiente di lavoro

Prima di costruire le regole necessarie al corretto caricamento dei moduli nel kernel, dobbiamo costruirci un ambiente di lavoro adatto.

Ogni script shell inizia con l'identificatore dell'interprete che viene utilizzato. `BusyBox` crea un link simbolico per la shell di default identificato dal file `/bin/sh`. Quindi la prima riga dello script è costituita da:

```
#!/bin/sh
```

Successivamente si deve impostare la variabile globale *`$PATH`*:

```
PATH=/bin:/sbin
```

che rappresenta l'insieme delle directory in cui sono presenti gli eseguibili che sono necessari a costruire l'ambiente di lavoro. Nel nostro caso, avendo costruito `BusyBox` senza utilizzare la directory `/usr`, tutti gli eseguibili saranno presenti nelle directory `/bin` per le normali applicazioni e in `/sbin` per quelle che richiedono i privilegi dell'amministratore di sistema. Pertanto abbiamo

tutto l'occorrente per poter caricare, controllare ed eventualmente scaricare i moduli.

Per comunicare direttamente con il kernel si devono montare due pseudo-filesystem, `procfs` e `sysfs`:

```
mount -t proc none /proc
mount -t sysfs none /sys
```

Questi pseudo-filesystem vengono utilizzati per capire se il modulo appena caricato riesce a riconoscere nuovi dispositivi.

Inoltre, si devono creare dei dispositivi a carattere:

```
mknod /dev/console 5 1
mknod /dev/null c 1 3
```

tramite i quali possiamo visualizzare, tramite `/dev/console`, o inibire, tramite `/dev/null`, le informazioni.

Per tenere traccia, invece, dei vari punti di mount delle partizioni montate creiamo questo link simbolico:

```
ln -s /proc/mounts /etc/mtab
```

5.2 Introduzione a procfs

L'implementazione dello *pseudo-filesystem* `proc` in Linux è un clone di Plan 9[1]. Per pseudo-filesystem non si intende un filesystem reale, infatti non consuma spazio sul disco ma solo una limitata capacità di memoria.

Sotto Linux, `/proc` fornisce le informazioni di ogni processo che sta girando sulla macchina alla directory `/proc/`*`pid`*, ma in più include anche:

- un link simbolico al processo corrente in `/proc/self`;
- informazioni sull'hardware, il kernel e la configurazione dei moduli;
- accesso alle opzioni del kernel che possono essere modificate dinamicamente nella directory `/proc/sys`
- informazioni sul sistema, ad esempio `/proc/meminfo` che fornisce le statistiche sulla memoria.

[1]Sistema operativo distribuito, *http://plan9.bell-labs.com/plan9/*

Le applicazioni basilari che utilizzano `/proc` si trovano nel pacchetto `procps`, e richiedono che `/proc` sia già stato montato.

Il filesystem `proc` crea facilmente un ponte che mette in comunicazione il kernel space con l'user space. Ad esempio la versione GNU di `ps` opera interamente in user mode, usando questo pseudo filesystem per ottenere i dati.

5.3 Introduzione a sysfs

Durante il ciclo di sviluppo del kernel Linux 2.5, era stato introdotto un modello per i driver in modo da correggere alcuni problemi che erano apparsi nel kernel 2.4:

- nessun metodo unificato per rappresentare la relazione tra driver e dispositivi;
- nessun meccanismo di hotplug.

É un filesystem basato originariamente su `ramfs` e veniva chiamato `ddfs`[2]. `sysfs` viene utilizzato da alcune applicazioni per accedere alle informazioni riguardanti l'hardware e i suoi driver (i moduli del kernel) come `udev` o `HAL`.

`sysfs`[9] è un *filesystem virtuale* fornito dal kernel Linux 2.6. Esporta le informazioni riguardanti i dispositivi e i driver dal kernel space allo user space, viene anche utilizzato per configurarli. Oltre ad esportare questo tipo di informazioni, è stato progettato per ridurre l'utilizzo della memoria su grandi sistemi.

Per ogni oggetto aggiunto in `/dev` viene creata una directory in `sysfs`. Le relazioni si riflettono con la creazione di sottodirectory all'interno di `/sys/devices`. La sottodirectory `/sys/bus` viene popolata con link simbolici, in modo da identificare quale bus viene utilizzato dal dispositivo. `/sys/class/` fa vedere le componenti raggruppate in classi, come i componenti di rete, mentre `/sys/block` contiene i dispositivi a blocchi.

I moduli possono avere delle opzioni che ne modificano il comportamento. Queste opzioni possono essere, a discrezione dello sviluppatore, modificate dinamicamente utilizzando semplici file. La regola è che possono contenere solo un singolo valore. Oltre alle opzioni, lo sviluppatore può generare altri file che contengono informazioni sullo stato del dispositivo. Quindi, tutti i file legati ad un dispositivo saranno presenti all'interno della sottodirectory del modulo ad esso associato e vengono chiamati attributi.

[2] Device Driver Filesystem

5.4 Le regole

Per costruire delle regole valide, dobbiamo trovare un comportamento univoco per ogni dispositivo trovato, in modo da riconoscere solo i moduli necessari al funzionamento del computer. Iniziamo a costruire le regole basandoci sui messaggi che ci fornisce `sysfs`.

5.4.1 Il comportamento dinamico di sysfs

Un modo molto semplice per vedere cosa accade sul filesystem `/sys` consiste nell'utilizzare il programma `udevmonitor` contenuto nel pacchetto `udev`. Infatti, appena dopo la fase di bootstrap del kernel, nessun modulo è stato ancora caricato e all'interno della directory `/sys` ci saranno solo i file e le directory inerenti ai driver caricati staticamente all'interno del kernel.

Lanciando il comando:

```
udevmonitor &
```

faremo lavorare questo programma in background e ci stamperà a video tutti gli eventi che riceve `udev`.

Successivamente carichiamo un qualsiasi modulo del kernel presente sul `CDROM`; ad esempio il modulo che permette il riconoscimento di un lettore `CDROM`, è chiamato `ide-cd`.

Quindi lanciamo il comando:

```
modprobe mod
```

dove *mod* è il modulo del kernel che abbiamo intenzione di caricare.

Ad esempio, se abbiamo caricato il modulo `ide-cd`, `udevmonitor` fornirà le seguenti informazioni:

```
UEVENT[1173868127.192947] add@/module/ide_cd/drivers
UEVENT[1173868127.203628] add@/module/ide_cd
UEVENT[1173868127.213055] add@/bus/ide/drivers/ide_cdrom
UEVENT[1173868127.192947] add@/block/hdc
```

dove, subito dopo il caricamento del modulo, notiamo che sarà presente una directory chiamata `ide-cdrom` all'interno di `/sys/bus/ide/driver`. Entrando in questa directory, se è stato riconosciuto un dispositivo troveremo un link simbolico, rappresentato da una sequenza di numeri e punti (ad esempio `1.0`), che punta alla directory identificativa del dispositivo a blocchi. All'interno di questa directory troveremo un insieme di file, che rappresentano gli attributi

del dispositivo. Tra i tanti file presenti, quello che rappresenta univocamente il dispositivo è `media`, infatti rappresenta il tipo di dispositivo multimediale, che nel nostro caso è *cdrom*.

5.5 Scenario

Una volta capito come vengono generati i file e le directory all'interno del pseudo-filesystem `sysfs`, possiamo scrivere uno script shell che automatizzi il riconoscimento dell'hardware presente sulla macchina.

Tornando all'esempio precedente, una possibile funzione che controlla l'esistenza di un `CDROM` con bus `IDE`:

```
test_ide_cdrom () {

    temp=$(cat \
     /sys/bus/ide/drivers/ide-cdrom/[0-9]*/media \
     2>/dev/null)
    for a in temp
    do
        [ ``a'' = ``cdrom'' ] && return 1
    done
    return 0

}
```

A questo punto analizziamo la funzione scritta. In uno script shell, una funzione possiede una propria signature ed il suo corpo viene racchiuso da parentesi graffe, ad esempio:

```
signature() {

    #corpo della funzione

}
```

Se alla funzione dobbiamo passarci dei parametri, essi non verranno inclusi all'interno delle parentesi tonde che si trovano subito dopo la signature, ma saranno rappresentati da particolari variabili: *$1*,*$2*,... dove indicano rispettivamente il primo argomento, il secondo, eccetera. Infine, gli unici valori di ritorno ammessi da una qualsiasi funzione in uno script shell sono solo di tipo intero.

Il comando:

```
temp=$(cat /sys/bus/ide/drivers/ide-cdrom/[0-9]*/media \
 2>/dev/null)
```

fa più operazioni contemporaneamente.

Innanzitutto la stringa `[0-9]*` rappresenta una semplice espressione regolare. In questo caso significa che prende tutte le stringe che iniziano con un numero e continuano con un qualsiasi altro carattere. Così facendo selezioneremo solo i link simbolici presenti all'interno della directory:

```
/sys/bus/ide/drivers/ide-cdrom
```

che, come abbiamo precedentemente detto, rappresentano i device che sono stati riconosciuti dal kernel. Usiamo inoltre questa espressione regolare perché è supportata all'interno di `ash` — la shell che utilizzo all'interno di `BusyBox`.

L'operazione successiva è composta dal comando `cat` che stampa sullo standard output il contenuto del file `media`, mentre lo standard error viene rimandato sul device `/dev/null`[3] tramite:

```
2>/dev/null
```

dove `2` rappresenta lo standard error mentre `>` rappresenta l'operatore di redirezione. Utilizziamo la redirezione dello standard error dato che se il kernel non riesce a riconoscere nessun dispositivo con il modulo che è stato appena caricato allora il percorso scritto precedentemente non esiste ed il comando `cat` ritorna l'errore.

Il contenuto dello standard output, invece, sarà contenuto dalla variabile *temp*.

In uno shell script il costrutto iterativo `for` è costituito dalla seguente sintassi:

```
for var in lista
do
    #corpo del ciclo for
done
```

dove *lista* rappresenta l'insieme di valori che si devono iterare, nel nostro esempio i valori sono contenuti all'interno della variabile *temp*; la variabile *var* assume un singolo valore contenuto nella variabile *lista* per ogni iterazione; infine, il corpo del ciclo `for` viene aperto dalla parola chiave `do` e viene chiuso da `done`.

Nell'esempio, il corpo del ciclo `for` è questo:

[3] il buco nero dei sistemi UNIX™ e unix-like

```
[ ‘‘a’’ = ‘‘cdrom’’ ] && return 1
```

dove viene confrontato ogni singolo valore con la stringa `cdrom` e se ne esiste uno esce dalla funzione ritornando il valore `1`, altrimenti controlla tutti i valori e ritorna il valore `0`.

Un possibile utilizzo delle funzioni shell precedentemente descritte potrebbe essere:

```
modpbobe ide-cd
test_ide_cdrom
[ $?  = 0 ] && rmmod ide-cd
```

dove `$?` è una variabile speciale che contiene il valore di ritorno dell'ultimo comando lanciato. Infatti, se la funzione ritorna `0` rimuoviamo il modulo, dato che non è stato riconosciuto nessun dispositivo.

Tale ragionamento viene esteso anche ad altri differenti tipi di dispositivi, come ad esempio periferiche `USB`, `SCSI`, `SATA`, eccetera.

La tabella 5.1 riassume tali relazioni.

Dispositivo	Percorso
USB	/sys/bus/usb/drivers/*nmodulo*/ \ *symlink*/host[0-9]/scsi_host:host[0-9]/ \ proc_name
IDE	/sys/bus/ide/drivers/*nmodulo*/ \ *symlink*/media
SCSI	
SATA	/sys/class/scsi_host/host[0-9]/proc_name

Tabella 5.1: Rappresentazione dei dispositivi nel filesystem sysfs

Capitolo 6

Interfaccia grafica: dialog

Il processo di installazione deve essere facilmente utilizzabile da parte dell'utente finale. Per creare tale interazione si è scelto di utilizzare l'interfaccia grafica chiamata `dialog`.

Questo capitolo mostra come utilizzare tale interfaccia grafica tramite alcuni esempi[13] e fornisce dei suggerimenti riguardanti la creazione del programma.

6.1 Textual User Interface

L'installer è un'applicazione che permette di configurare la distribuzione sul disco a seconda delle esigenze che ha l'utente.

La comunicazione tra l'utente e l'installer avviene tramite un'interfaccia. È stata scelta una *Textual User Interface*, cioè stampa le informazioni occupando l'intero schermo ed utilizza i simboli tipici disponibili sui terminali[1].

Nel mondo `UNIX™`, le interfacce testuali solitamente sono state costruite utilizzando la libreria `curses`, o `ncurses`[2].

L'applicazione basata su `ncurses` che utilizzeremo per creare l'interfaccia utente è: `dialog`.

Quest'applicazione è stata sviluppata inizialmente per essere utilizzata all'interno di script shell, ma è possibile utilizzare le sue funzionalità anche in un programma scritto in `C` caricando la libreria `libdialog` in fase di compilazione. È composto da otto tipi di finestre: **calendar**, **checklist**, **form**, **fselect**, **gauge**, **infobox**, **inputbox**, **inputmenu**, **menu**, **msgbox**, **pas-**

[1] solitamente sono device a caratteri

[2] che sta per New Curses

swordbox, **passwordform**, **pause**, **radiolist**, **tailbox**, **tailboxbg**, **textbox**, **timebox**, **progressbox**, **yesno**.

`dialog` è molto facile da usare. Se ad esempio scriviamo:

```
dialog --title 'Finestra' --msgbox 'Hello, world!'  5 20
```

`dialog` crea un messaggio con il titolo "Finestra" e con il testo "Hello, world!". Questa finestra è fatta da 5 linee in altezza e 20 caratteri in lunghezza. Un bottone "OK" appare al di sotto; premendo *<ENTER>* la finestra viene chiusa.

6.2 Alcuni esempi sui tipi di finestre

Molti tipi di finestre possiedono un formato simile.

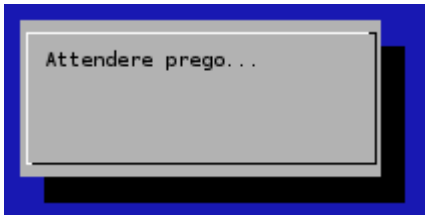
Il menu **yesno** è molto simile al primo esempio:

```
dialog --title 'Finestra' --yesno 'Sei contento?'  6 25
```

Se si prova questo esempio, si vedrà che ci sono due bottoni nella parte bassa della finestra, etichettati con "Yes" e "No". I bottoni possono essere selezionati usando i tasti di direzione o il tasto *<TAB>* e premere successivamente il tasto *<ENTER>* per confermare la scelta. Lo stato di uscita che viene ritornato sulla shell sarà 0 se "Yes" viene scelto e 1 se scegliamo "No".

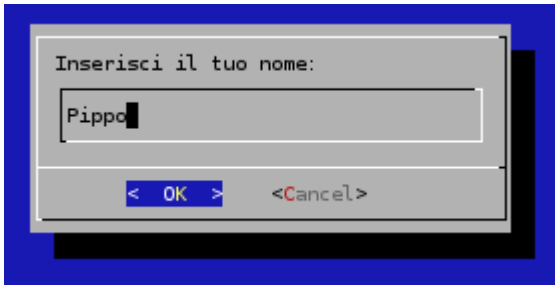
La finestra **infobox** è simile a **msgbox** ma non aspetta che l'utente seleziona il bottone "OK". Questo è molto utile per stampare un messaggio quando si sta effettuando un'operazione che richiede tempo, ad esempio

```
dialog --infobox 'Attendere prego...'  10 30 ; sleep 4
```



Per inserire una stringa si utilizza **inputbox**. Dopo che i dati dell'utente sono stati inseriti, essi verranno stampati sullo standard error. Ad esempio:

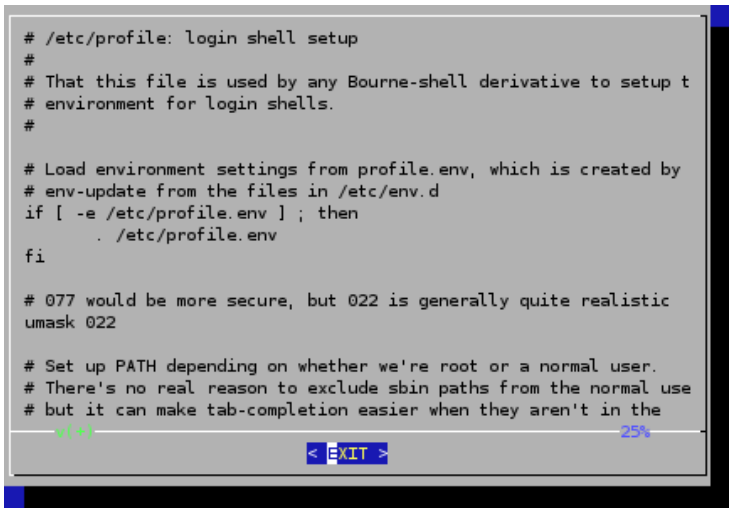
```
dialog --inputbox Inserisci il tuo nome:  8 40 2>risposta
```



dove la risposta viene rigirata sul file chiamato "risposta".

Utilizzando **textbox** possiamo vedere il contenuto di un file; esso prende il nome del file come parametro:

```
dialog --textbox /etc/profile 22 70
```



Per spostarsi all'interno del file si usano i tasti di direzione, *<Page Up>*, *<Page Down>*, *<Home>*, eccetera. Si può uscire premendo *<ESC>* o *<ENTER>*.

Il tipo di finestra **menu** permette di creare un menu a scelta sul quale un utente può scegliere. Il formato è:

```
dialog --menu <text> <height> <width> <menu-height> [tag item]
```

Ogni entry del menu è composta da due stringhe: una *tag* e da un *item*. L'utente può prendere una scelta utilizzando i tasti di direzione e successivamente premendo *<ENTER>*. La *tag* selezionata verrà scritta sullo standard error. Questo è un semplice esempio:

```
dialog --menu 'Scegli:'  10 30 3 \
    '1' 'rosso' '2' 'verde' '3' 'blu'
```

Il prossimo tipo è **checklist**. All'utente viene presentata una lista di scelte che può attivare o disattivare individualmente usando la barra spaziatrice:

```
dialog --checklist 'Scegli:'  10 40 3 \
    '1' 'Ferrari' 'on' \
    '2' 'Aston Martin' 'on' \
    '3' 'Porsche' 'off'
```

Il terzo campo in ogni scelta è lo stato iniziale che può essere "on" o "off".

Utilizzando **radiolist**, è simile alla finestra di tipo **checklist** tranne per il fatto che un utente può selezionare solo un entry presente nella lista di opzioni. Ad esempio:

```
dialog --backtitle 'Selezione della CPU' \

    --radiolist 'Seleziona il tipo di CPU:' 10 40 4 \

        '1' 'Pentium' 'off' \
        '2' 'Athlon' 'on' \
        '3' 'Core Duo' 'off' \
        '4' 'Athlon64' 'off'
```

Questi esempi non comprendono tutti i tipi di finestre generabili tramite l'applicazione `dialog`, per approfondire è consigliabile leggere l'apposito `man`.

6.3 Costruire dialog

Per poter usufruire di questa comoda applicazione bisogna installarla all'interno dell'`initrd` presente nella `ISO` che andremo a generare; pertanto dobbiamo compilarla in modalità statica.

In questo caso, il procedimento per costruire il programma staticamente è leggermente più complesso poiché `dialog` dipende dalle librerie `ncurses`. Questa libreria solitamente viene compilata da parte degli sviluppatori della distribuzione abilitando il supporto a `GPM`[3], tale feature però non permette di costruire l'applicazione `dialog` staticamente.

Per costruire le librerie `ncurses` senza il supporto per `GPM`, scarichiamo i sorgenti da:

[3] General Purpose Mouse

`ftp://invisible-island.net/ncurses/`*`nver`*`.tar.gz`

e lo esplodiamo usando il comando:

`tar xzf `*`nver`*`.tar.gz`

dove *`nver`* è la versione di `ncurses` che abbiamo intenzione di utilizzare.

A questo punto configuriamo il pacchetto:

```
./configure --without-gpm --without-debug --without-dlsym
```

dove disabilitiamo:

- il supporto per `GPM`, che rappresenta il supporto per l'utilizzo del mouse sui bottoni che `dialog` crea sulle sue finestre.
- le informazioni di debug, in modo da rendere la libreria più leggera,
- l'utilizzo della funzione `dlsym`, in modo da non utilizzare questa `API` per il linking dinamico.

Una volta configurato il pacchetto utilizziamo il comando `make` per compilarlo.

Il passo successivo consiste nel costruire l'applicazione `dialog` basandola sulla libreria `ncurses` appena costruita. I sorgenti possono essere presi da un qualsiasi mirror Debian, ad esempio:

http://ftp.debian.org/debian/pool/main/d/dialog/ `dver.orig.tar.gz`

ed estraiamo il pacchetto con il comando:

`tar xzf `*`dver`*`.orig.tar.gz`

dove *`dver`* è la versione di `dialog` che abbiamo intenzione di installare.

Per quanto riguarda la configurazione del pacchetto utilizziamo il comando:

```
./configure CC='gcc --static'
```

in modo tale da generare un eseguibile non dipendente da librerie esterne.

Dato che il comando `./configure` genera il file `makefile` dobbiamo modificare la variabile *`LIBS`*, come abbiamo precedentemente detto, per poter utilizzare le librerie `ncurses` costruite precedentemente, con questa riga:

`LIBS = -L`*`ndir`*`/lib/ -lncurses -lm -L/lib`

dove *ndir* è la directory dove abbiamo costruito le nuove librerie `ncurses`.

L'opzione `-l` utilizzata nella variabile *LIBS* del `makefile` serve a linkare dinamicamente una determinata libreria all'eseguibile che stiamo generando; questa opzione cerca le librerie specificate all'interno di una lista di directory definita solitamente in `/etc/ld.so.conf`. Tramite questa modifica aggiungiamo la directory che contiene la libreria `ncurses` appena costruita a questa lista di directory. Inoltre, essendo l'ultima directory inserita sarà la prima ad essere sottoposta alla scansione; in questo modo non utilizzeremo la libreria `ncurses` installata per default dalla distribuzione, bensì quella che abbiamo personalizzato.

Dopo aver modificato correttamente il `makefile`, lanciamo il comando `make` per poter compilare l'applicazione.

Infine, installiamo l'eseguibile generato con questo comando:

```
cp dialog_dir/dialog adir/bin
```

dove *dialog_dir* è la directory dove abbiamo compilato l'applicazione, mentre *adir* è la directory di appoggio dedicata all'`initrd`.

Nota Questo tipo di pacchetto soffre di un problema riguardante la compilazione per sistemi a 64 bit. Se utilizziamo la tecnica precedentemente descritta – generare un eseguibile statico – la parte grafica non viene disegnata. Per ovviare questo problema, dobbiamo impostare un'ulteriore flag al compilatore. Pertanto, esportiamo la variabile di ambiente `CC` in questo modo:

```
export CC=‘‘gcc -m32 -static’’
```

Tramite l'opzione `-m32` in realtà effettuiamo una cross-compilazione, cioè generiamo eseguibili o librerie per un'architettura diversa.

Dato che stiamo generando un eseguibile statico non avremo problemi durante l'esecuzione a patto di compilare anche le librerie `ncurses` con questa tecnica.

Capitolo 7

Installer

Dopo aver costruito l'ambiente di lavoro, in questo capitolo vedremo come costruire il programma che installerà la distribuzione sul disco: l'*installer*.

L'installer che costruiremo sarà interamente scritto in script shell e farà parte dell'`initrd`. La shell utilizzata è un'implementazione minimale di `ash` presente nel pacchetto `BusyBox`.

7.1 Organizzazione dello script

La filosofia `UNIX`™ può essere riassunta da una frase:

> ***Keep it simple, stupid!***

cioè, ogni programma deve:

- essere molto semplice da capire a livello di sorgenti,
- risolvere un unico compito,
- cooperare con altri comandi.

Uno shell script applica alla lettera questa filosofia perché

- possiede una sintassi di alto livello, pertanto è semplice da capire;
- al suo interno si possono chiamare tutti programmi presenti nella variabile d'ambiente *`$PATH`* necessari per risolvere un determinato obiettivo;
- tramite gli operatori di redirezione è possibile far cooperare due o più comandi tra loro;

tali caratteristiche lo rendono un'ambiente di sviluppo potente e allo stesso tempo semplice.

L'installer dovrà svolgere compiti differenti durante la sua esecuzione:

- identificare la mappatura della tastiera,
- selezionare i dischi,
- gestire le partizioni,
- selezionare il software,
- installare i pacchetti,
- configurare il bootloader.

Ogni singola parte, precedentemente elencata, sarà associata ad un unico script.

Mentre, uno script, più generale, gestisce il flusso di esecuzione dell'installer, specializzandosi tramite il comando `source` per chiamare i vari script esterni.

7.2 Mappatura della tastiera

Come qualsiasi distribuzione che si rispetti abbiamo bisogno di una mappatura della tastiera adatta a quella che stiamo utilizzando. Il comando utilizzato per caricare la tastiera è `loadkmap` incluso all'interno di `BusyBox`.

Il programma `loadkmap` legge un file binario dallo standard input. Questo file rappresenta la mappatura della tastiera e può essere creato utilizzando il comando `dumpkmap`[1].

Per poter abilitare la corretta mappatura della tastiera è sufficiente scrivere:

```
loadkmap < kmapfile
```

dove *kmapfile* è il file binario contenente la mappatura della tastiera.

Una possibile implementazione per poter impostare la tastiera è la seguente:

```
set_keymap() {
```

[1] contenuto anch'esso all'interno di `BusyBox`

```
    KEYMAP=us
    if [ -e temp ]
    then
        KEYMAP=`cat temp`
        rm temp
    fi
    cd /tmp/
    tar xzf /etc/i18n/kmaps.tgz
    loadkmap < /tmp/$KEYMAP
    rm /tmp/*

}
```

dove

- *KEYMAP* è la variabile che contiene il nome del file binario, per default imposta la mappatura della tastiera americana;
- *temp* è un file che contiene il nome del file binario, se questo file esiste sarà il futuro valore della variabile *KEYMAP*
- *kmaps.tgz* è un archivio che contiene le mappature della tastiera più utilizzate.

Possiamo integrare questa funzione utilizzando `dialog` per l'interfaccia testuale, ad esempio:

```
show_keymap() {

    dialog --title ‘‘KEYBOARD MAP SELECTION’’ \
        --menu ‘‘You may select one of the \
         following keyboard maps.  If you do \
         not select a keyboard map, the US \
         keyboard map is the default.’’  \
        22 55 11 \
        ‘‘it’’ ‘‘Italian Keyboard Map’’ \
        ‘‘us’’ ‘‘American Keyboard Map’’ 2>temp
    set_keymap

}
```

dove il contenuto di *temp* assumerà o il valore `it` oppure `us`.

I valori `it` e `us` rappresenteranno i nomi dei file contenuti all'interno dell'archivio *kmaps.tgz*

7.3 Selezionare i dischi

Il riconoscimento dei dischi è l'argomento cardine della distribuzione poiché senza di esso non possiamo effettuare nessuna installazione. Esso viene inizializzato tramite il caricamento di opportuni moduli del kernel e l'utilizzo di `mdev`.

L'installer, in questo caso, deve capire quali siano i device files presenti nella directory `/dev` che rappresentano gli hard disk e fornire all'utente un'interfaccia per selezionarne almeno uno di essi.

La maggiorparte dei nomi associati ai dispositivi che rappresentano un disco possono essere identificati da questa regular expression:

```
[hs]d[a-z]
```

Questa regular expression significa che possiamo avere dei dispositivi chiamati `hda`, `hdb`, ..., `hdz` oppure `sda`, `sdb`, ..., `sdz`.

7.3.1 Dischi E/IDE

Per i dischi di tipo `E/IDE` utilizziamo il comando `hdparm` incluso in `BusyBox`. Infatti tramite l'opzione `-i` si ottengono le informazioni sul disco, tra le quali: marca, modello e numero di serie del disco.

A livello di scripting bash una possibile funzione che ci permette di identificare un disco `E/IDE` può essere:

```
find_disk_ide() {

    for i in /dev/hd[a-z]
    do
        ret=‘‘`hdparm -i i 2>&1 | \
            grep Model | \
            grep -v -i -E ’(cd|dvd)(.|)(rom|)’`’’
```

```
            if [ !  ‘‘ret’’ = ‘‘’’ ]
            then
                echo i e’ un disco IDE
            fi
        done
    }
```

In questa funzione bash ritroviamo il seguente comando:

```
hdparm i 2>&1 | \
    grep Model | \
    grep -v -i -E ’(cd|dvd)(.|)(rom|)’
```

Questo significa che le informazioni presenti nel device *i* vengono filtrate prendendo la sola riga riguardante il modello. Da qui filtreremo ulteriormente la frase scartando tutte le frasi che contengono le parole `CDROM`, `CD-ROM`, `DVD`, `DVDROM` scritte sia con lettere maiuscole che con lettere minuscole.

Se sono presenti dischi `E/IDE` che evitano questi filtri allora sicuramente non saranno dei lettori `CD/DVD-ROM`.

7.3.2 Dischi SATA/SCSI

Questa tipologia di dischi ha delle caratteristiche differenti rispetto ai normali dischi `E/IDE`. Ad esempio, i dischi `SATA` comunicano con un bus a 16 bit mentre i dischi `E/IDE` utilizzano 32 bit, pertanto le `ioctl` non sono state implementate[2] per questo tipo di supporti, rendendo inutilizzabile il programma `hdparm`.

Per poter risolvere il problema si è implementato un piccolo programma scritto in linguaggio `C`.

Faremo uso dei seguenti header files:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Creiamo delle funzioni per rendere la scrittura del programma leggermente più comoda.

La funzione `my_alloc` verrà utilizzata per l’allocazione di memoria:

[2]anche se esistono delle patch fino al kernel 2.6.17

```
void *my_alloc(ssize_t byte)
{
    void *ptr = NULL;
    ptr=malloc(byte*sizeof(char));
    if(ptr == NULL)
        exit(EXIT_FAILURE);
    memset(ptr, 0, byte*sizeof(char));
    return ptr;
}
```

Al contrario, la funzione `my_free` gestirà la deallocazione della memoria:

```
void my_free(void *ptr)
{
    if(ptr != NULL)
        free(ptr);
}
```

Le seguenti funzioni serviranno ad effettuare il parsing delle stringhe.

La funzione `delete_chars` servirà ad eliminare tutti i caratteri che possono fornire un output con caratteri particolari, quali ad esempio tabulazioni, carriage return, eccetera:

```
char *delete_chars(char *str)
{
    int i;
    for(i=0; str[i] != 0; i++)
    {
        if(str[i] < 32)
        {
            str[i] = 0;
            break;
        }
    }
    return str;
```

```
}
```

La funzione `parse_device` servirà a tagliare una stringa ritornando solo l'ultima parte della stessa, precisamente partirà dall'ultimo /(slash) e terminerà a fine stringa:

```
char *parse_device(char *dev)
{
    while(dev++)
        if(*dev == '/')
            return dev++;
    return dev;
}
```

Infine, `read_file` gestisce la lettura dei file:

```
char *read_file(char *file)
{
    int f, length, result;
    char *string, buf;
    length = result = f = -1;
    f=open(file, O_RDONLY);
    if(f == -1)
        exit(EXIT_FAILURE);
    while( read(f, &buf, 1) > 0 )
        length++;
    lseek(f, 0, SEEK_SET);
    string=my_alloc(length);
    result = read(f,string,length);
    close(f);
    if (result != length)
        exit (EXIT_FAILURE);
    return delete_chars(string);
}
```

Infine la funzione `main`:

```
int main(int argc, char *argv[])
{
    ssize_t length;
    char *device, *vendor, *model, *rev, *file;
    if(argc != 2)
        return EXIT_FAILURE;
    device=parse_device(argv[1]);
    if(device == NULL)
        return EXIT_FAILURE;
    length = strlen(device);
    file=my_alloc(25+length);
    sprintf(file, ``%s%s%s'', ``/sys/block'',
        device, ``/device/vendor'');
    vendor=read_file(file);
    file=realloc(file,24+length);
    sprintf(file, ``%s%s%s'', ``/sys/block'',
        device, ``/device/model'');
    model=read_file(file);
    file=realloc(file,22+length);
    sprintf(file, ``%s%s%s'', ``/sys/block'',
        device, ``/device/rev'');
    rev=read_file(file);
    my_free(file);
    printf(``Model=%s %s FwRev=%s\n'',
        vendor, model, rev);
    return EXIT_SUCCESS;
}
```

Il funzionamento del programma consiste nello stampare a video il contenuto di più file che contengono le informazioni riguardanti i dischi di tipo `SATA/SCSI`. Tali file sono contenuti all'interno dello pseudo filesystem

sysfs. Il programma prende un solo argomento, che serve per poter navigare all'interno del suddetto filesystem.

Per compilare questo codice sorgente C è sufficiente lanciare questo comando:

```
gcc file.c -o eseguibile -Wall -pedantic -static
```

dove *file.c* è il nome del file che contiene il sorgente C appena visto, mentre *eseguibile* è il nome da dare al programma.

Utilizzando questo programma è possibile effettuare le stesse operazioni viste per i dischi E/IDE. Le uniche differenze riguardano il nome dei device e il nome del programma da utilizzare.

7.3.3 dialog: selezionare uno o più dischi

Per integrare questa ricerca all'interno di un'interfaccia dialog utilizzeremo una tecnica particolare.

Siccome non sappiamo quanti e quali dischi ha un comune pc popoleremo la lista di dialog utilizzando un file di appoggio.

La funzione inizialmente scriverà la parte 'statica' di dialog:

```
    cat > /tmp/sldisk << OOK
dialog --separate-output --title ``SELECT DISK'' \\
    --checklist ``Select one or more disk which \\
    you would build this distribution 12 70 4 \\
OOK
```

Successivamente chiamerà un'altra funzione che cerca sia i dischi E/IDE che quelli di tipo SATA/SCSI e ne scrive il risultato in un file, che nel nostro caso si chiama /tmp/sldisk:

```
find_disks
```

Questa funzione oltre a costituire la parte 'dinamica' del comando che stiamo costruendo imposterà un ulteriore file dove salverà il risultato della selezione chiamato:

```
/tmp/sldiskr
```

Se il file non contiene nessun disco, allora avvertirà l'utente prima di effettuare il reboot della macchina:

```
if [ ‘‘`grep off /tmp/sldisk`’’ = ‘‘’’ ]
then
    dialog --title ‘‘NO DISK FOUND’’ \
        --ok-label ‘‘Reboot’’ \
        --msgbox ‘‘Have you a disk on your \
         machine?!?’’  7 30
    ret=$?
    reboot
fi
```

In alternativa se esiste almeno un disco eseguiremo il file appena generato utilizzando il comando:

```
source /tmp/sldisk
```

Anche in questo caso possiamo avere dei problemi, l'utente non ha selezionato nessun disco presente nella lista. Pertanto il comando precedente verrà posizionato in un ciclo `while` in modo da gestire questa situazione:

```
while [ 1 ]
do
    source /tmp/sldisk
    res=$?
    if [ res -ne 0 -o ‘‘`cat /tmp/sldiskr`’’ = ‘‘’’ ]
    then
        dialog --title ‘‘WARNING’’ \
            --msgbox ‘‘You must select at \
             least one disk!!’’  10 30
    else
        break
    fi
done
```

Infine una volta usciti dal loop infinito possiamo rimuovere il file dato che non verrà ulteriormente utilizzato:

```
rm /tmp/sldisk
```

7.4 Gestione delle partizioni

Dopo aver riconosciuto i dischi ed averli selezionati viene chiesto all'utente se modificare la tabella delle partizioni tramite l'interfaccia `dialog`:

```
dialog --title ‘‘EDIT PARTITON TABLE’’ \
    --yesno ‘‘Do you create or edit partitions?’’
     10 30
```

Se l'utente decide di modificare la tabella delle partizioni allora ogni disco viene gestito dal programma `cfdisk`, in questo modo:

```
for i in ̀cat /tmp/sldiskr`
do
    dialog --sleep 2
        --infobox ‘‘Selecting /dev/i...’’  3 30
    /sbin/cfdisk /dev/i
done
```

7.4.1 Selezione delle partizioni

Per ogni disco selezionato dall'utente verrà effettuata una ricerca di tutte le partizioni presenti di tipo Linux.

A questo proposito si utilizza il comando `fdisk` e l'opzione `-l`. Questa opzione permette di listare le partizioni presenti su un determinato device.

Solitamente Linux dedica dei nomi particolari a questo tipo di periferiche[3], ad esempio:

- `hd`*`Xy`* per i dischi `E/IDE` oppure
- `sd`*`Xy`* per i dischi `SATA/SCSI`

dove

- *`X`* è una lettera e rappresenta il dispositivo, ad esempio la lettera `a` solitamente viene associata al disco master del primo canale;
- *`y`* è un numero e rappresenta il numero della partizione.

[3]come tutti gli altri UNIX™

7.4.2 Partizioni Linux

Per identificare l'esistenza di partizioni di tipo Linux, andremo a leggere la tabella delle partizioni di ogni singolo disco selezionato da parte dell'utente:

```
disks=```cat /tmp/sldiskr```
for i in disks
do
    part=```fdisk -l /dev/i 2>/dev/null | \
        grep '^/dev/*' | grep -v swap | \
        grep Linux | awk 'print $1'` part''
done
```

Analizzando questo pezzo di script viene popolata la variabile *part* che conterrà tutte le partizioni Linux non-swap.

Se non è stata trovata nessuna partizione allora lo script avvertirà l'utente con il seguente messaggio e successivamente la macchina verrà riavviata:

```
dialog --title ''NO LINUX PARTITION FOUND'' \
    --ok-label ''Reboot'' \
    --msgbox ''Have you create a partition table \
     with linux partitions?!?''  6 30
res=$?
reboot
```

Controllo del disco

Per ogni partizione trovata verrà chiesto se si vuole controllare il filesystem se esso è presente:

```
dialog --title ''CHECK PARTITION'' --defaultno \
    --yesno ''Would you check partition \
    filesystem?''  6 30
```

dove la variabile *partition* indica il nome della partizione.

Per default la scelta consiste nel saltare questo passo, dato che a volte è stata solo modificata la tabella delle partizioni.

Se invece l'utente vuole controllare il filesystem verrà utilizzato il comando:

```
e2fsck -y partition
```

Per essere sicuri dell'assenza di errori questo comando si troverà in ciclo `while` infinito fin quando non verranno corretti tutti gli errori, pertanto questa operazione richiederà del tempo per essere eseguita.

Formattazione del disco

Il passo successivo consiste nel chiedere all'utente se formattare o meno la partizione selezionata:

```
dialog --backtitle ``Do you want to format \
     Linux partition partition?''  \
    --title ``FORMAT PARTITION partition'' \
    --yesno ``This partition has nbyte bytes.  \
        If this partition has not been \
        formatted, you should format it.  \
        NOTE: This will erase all data on it.  \
         Would you like to format this \
         partition?''  12 70
```

dove *partition* indica il nome della partizione, mentre *nbytes* ne indica la sua dimensione nell'ordine dei bytes.

A questo punto se l'utente decide di formattare la partizione, essa verrà formattata in formato `ext3`.

A volte ci sono dei problemi con il filesystem `ext3` nel generare il journalizing del filesystem, pertanto se la formattazione non va a buon fine, allora la partizione verrà formattata utilizzando il filesystem `ext2`, che è paragonabile ad `ext3` senza journalizing. Ossia:

```
mke2fs -j partition
[ !  $?  = 0 ] && mke2fs partition
```

dove *partition* indica la partizione da formattare.

Punti di mount

L'utente deve impostare i punti di mount per ogni partizione. Per farlo utilizziamo la seguente funzione bash:

```
read_mountpoint() {
```

```
    while [ 1 ]
    do
        dialog --title ‘‘MOUNT POINT’’ \
            --separate-output \
            --inputbox ‘‘Please, insert a \
             mount point for partition’’ \
            8 50 2>mpoint
        [ $?  = 0 ] && break
    done

}
```

Questa funzione prenderà una stringa scritta da parte dell'utente e verrà salvata nel file *mpoint*, tale stringa rappresenterà il punto di montaggio per la partizione *partition*.

Il contenuto del file *mpoint* viene controllato, se è stata passata una stringa vuota allora verrà impostato il valore di default pari a `/`.

Un ulteriore controllo consiste nel verificare se la partizione è già stata montata sullo stesso punto di mount:

```
pmounted=‘‘‘cat /proc/mounts | \
    awk ’print $2’ | \
    grep /mnt/my_distro‘’’
for i in pmounted
do
    if [ ‘‘i’’ = ‘‘/mnt/my_distro’’ -a \
     ‘‘mpoint’’ = ‘‘/’’ ]
    then
        already_mounted
        amounted=1
        break
    elif [ ‘‘i’’ = \
     ‘‘/mnt/my_distro/mpoint’’ ]
    then
        already_mounted
        amounted=1
        break
```

```
        fi

    done
```

dove:

- la variabile *pmounted* è la lista delle partizioni che sono già state montate e *i* rappresenta ogni singola entry della lista;
- *mpoint* rappresenta il mount point inserito dall'utente;
- se è già stato impostato un punto di mount abbiamo:
  - la funzione `already_mounted` invita l'utente a scegliere un punto di mount alternativo tramite un messaggio;
  - *amounted* è un flag che per default vale `0` mentre se ci ritroviamo nel caso in cui la partizione è già stata montata con uno specifico mount point viene impostata ad `1`.

Se questi controlli vanno a buon fine, la partizione verrà montata e verrà aggiunta una riga nel file `/etc/fstab`[4] con la seguente sintassi:

```
pdev mpoint fstype options
```

dove

- *pdev* indica il nome del dispositivo che rappresenta la partizione;
- *mpoint* indica il punto di mount;
- *fstype* indica il filesystem presente sulla partizione;
- *options* indica le opzioni di montaggio della partizione, che per default sono:

```
default 0 0
```

[4]Questo file è caratteristico di molti UNIX™. Contiene la tabella dei dischi e delle partizioni utilizzate, indicando come devono essere usate dal sistema operativo.

7.4.3 Partizioni di swap

Solitamente quando viene installata una distribuzione sul disco bisogna avere anche una partizione di `swap`.

Questa è molto utile nel caso in cui la macchina ha poca memoria volatile libera. In questa situazione, i dati presenti in `RAM` da molto tempo senza essere utilizzati vengono spostati in questa speciale partizione del disco in modo da liberare lo spazio necessario sulla `RAM` per un normale utilizzo del computer.

Se ad esempio non abbiamo una partizione di `swap` appena un singolo processo occupa tutta la memoria volatile disponibile viene automaticamente *killato* dal kernel generando eventuali disservizi e instabilità dell'intero sistema operativo.

L'installer in questo caso cercherà tutte le partizioni di tipo `swap`. Se non sono presenti verrà stampato un messaggio all'utente:

```
dialog --title ‘‘NO SWAP PARTITION FOUND’’ \
    --yes-label ‘‘Reboot’’ \
    --no-label ‘‘Continue’’ \
    --yesno ‘‘Warning, you have not \
     created a swap partition with \
     Linux fdisk.  Do you want to continue \
     installing without one?’’  6 60
```

In questo modo l'utente deve scegliere se continuare con l'installazione oppure riavviare il computer.

Formattazione

Ogni partizione di tipo `swap` viene sempre formattata, pertanto questo passaggio non richiede alcuna interazione da parte dell'utente.

Per formattare questo tipo di partizioni utilizzeremo il seguente comando:

```
mkswap partition
```

dove *partition* indica il nome del dispositivo da formattare.

Questo tipo di partizione viene utilizzata sia da parte della distribuzione che verrà installata sul disco che dall'installer che gira su `RAM` disk. Pertanto per attivare una partizione di swap utilizzeremo il comando:

```
swapon partition
```

dove *partition* indica la partizione di `swap`.

Aggiornamento /etc/fstab

Il file `/etc/fstab` identifica anche questo tipo di partizioni secondo la sintassi:

```
pdev none swap defaults 0 0
```

dove *pdev* rappresenta il nome della partizione di `swap`.

7.5 Selezione del CDROM

Una volta identificate e montate le partizioni dobbiamo cercare il `CDROM` che contiene i pacchetti di installazione.

Secondo lo standard `ISO9660`, un `CDROM` è composto da settori di grandezza pari a `2048`. Pertanto utilizzeremo il comando `fdisk` per trovare il giusto dispositivo presente in `/dev`:

```
for i in `find /dev | \
    grep -E '/dev/(hd[a-z]|s(r[0-9]|d[a-z]))$'`
do
    if [ ``2048'' = ``fdisk -l i 2>&1 | \
        awk '/^Units/{print $7 }'`'' ]
    then
        ln -sf i /dev/cdrom
        mount -t iso9660 -o ro i \
            /mnt/my_distro/cdrom
        if [ !  -d /mnt/my_distro/cdrom/packs ]
        then
            umount i
        else
            break
        fi
    fi
done
```

In questo pezzo di codice evidenziamo:

- l'espressione regolare estesa `(hd[a-z]|s(r[0-9]|d[a-z]))$` accetta solo i nomi dei dispositivi che iniziano con `hd` o `sd` e terminano con una lettera, oppure iniziano con `sr` e terminano con un numero.
- la variabile *i* rappresenta ogni singolo dispositivo trovato.
- dato che il programma `fdisk` stampa le informazioni riguardanti la grandezza dei settori di un disco sullo *standard error* utilizziamo questa sequenza di caratteri `2>&1` per redirigere lo standard error (identificato dal numero 2) sullo standard output (identificato dal numero 1).
- il comando

  ```
  awk '/^Units/{print $7 }'
  ```

  seleziona la riga che inizia con la parola `Units` e ne stampa la settima parola, cioè la dimensione del singolo settore.

Se la dimensione del settore equivale al numero `2048` allora:

- viene creato un link simbolico chiamato `/dev/cdrom` che punta al dispositivo identificato dalla variabile *i*;
- il dispositivo viene montato sul filesystem nella directory

  ```
  /mnt/my_distro/cdrom
  ```

- viene controllato se nel dispositivo appena montato esiste una directory chiamata `packs`, così garantiamo l'univocità del `CDROM` che serve per installare la distribuzione.

7.6 Selezionare il software

Questo argomento verrà suddiviso in due punti di vista, quello dello sviluppatore e quello dell'utente.

Lo sviluppatore, è colui che utilizza lo script shell per costruire una distribuzione personalizzata, quindi, deve poter scegliere le applicazioni che serviranno per la distribuzione.

L'utente, d'altro canto, è colui che utilizza il `CDROM` fatto dallo sviluppatore, e scegliere quali applicazioni utilizzare tra quelle selezionate dallo sviluppatore.

7.6.1 Lato sviluppatore

Per agevolare l'inserimento dei vari pacchetti software da parte dello sviluppatore è stato creato un file chiamato `slack_packages`. Con questo file lo sviluppatore decide di quali pacchetti software sarà composta la propria distribuzione.

Il file prevede l'utilizzo di un'apposita sintassi dove:

- si possono aggiungere dei commenti in questo file aggiungendo il simbolo `#` all'inizio del commento.

  Ad esempio:

  ```
  # questo è un commento
  scrivo qualcosa # questo è un altro commento
  ```

- si possono aggiungere i pacchetti software secondo tre modalità differenti:

  - se si vuole prendere un pacchetto da un mirror ufficiale slackware avremo la seguente sintassi:

    directory`/`*npackage*`:`*flag*

    dove

    * *directory* indica il path relativo presente sul mirror
    * *npackage* indica il nome del pacchetto software, specificando eventualmente la versione da utilizzare
    * *flag* può assumere i valori `on` oppure `off`, tramite i quali lo sviluppatore abilita o disabilita il pacchetto software nel menu dell'utente.

    Ad esempio:

    ```
    a/bash:on
    a/glibc-solibs-2.3.6-i486-6.tgz:off
    ```

  - se il pacchetto software è presente su un sito web differente dai mirror ufficiali slackware utilizziamo quest'altra sintassi:

    `url:`*paddress*`:`*flag*

    dove *paddress* è il percorso completo per poter scaricare il pacchetto software. Attualmente sono supportati i protocolli http ed ftp, se nel percorso si omette il protocollo allora quello di default è http.

    Ad esempio:

`url:http://`*somesite*`/yourpackage.tgz:on`

– infine, se il pacchetto software è già presente sul disco dello sviluppatore si utilizza la seguente sintassi:

`local:`*pabsolutepath*`:`*flag*

dove *pabsolutepath* indica il percorso assoluto del pacchetto software presente sul disco.

Ad esempio

`local:/home/vinx/Desktop/rox-1.2.2-i686-1.tgz:on`

- ogni doppio cancelletto (`##`) indica l'inizio o la fine di un menu.

  Per default, esistono quattro menu, rispettivamente `BASE`, `NETWORK`, `DEVEL`, `GUI`.

  Mentre la direttiva `END`

Internals su slack_packages

Per poter effettuare il parsing dei pacchetti faremo largo uso del programma `sed`.

Innanzitutto verranno prese solo le righe riguardanti il singolo menu, ad esempio il comando:

`sed -n -e '/##GUI/,/##END/p'` *psp*

stampa solo le righe che identificano i pacchetti del menu `GUI`. Ossia sono quelle righe comprese tra la direttiva `##GUI` e `##END`. La variabile *psp* indica il percorso del file `slack_packages`.

Queste righe vengono parsate in modo da eliminare i commenti e le righe vuote:

- per eliminare i commenti che prendono una riga intera useremo il comando:

  `sed -e '/^#/d'`

  dato che i commenti, nella sintassi di uno script shell, iniziano con il carattere `#`.

- per eliminare le righe vuote useremo il comando:

  `sed -e '/./!d'`

- infine per eliminare i commenti misti a righe di codice bash useremo il comando:

```
sed 's/#.*//'
```

dove eliminiamo solo la parte finale della riga a partire dal carattere `#`.

Funzione: fetch_packs Dopo aver filtrato ogni singola riga possiamo analizzarne la sintassi in modo da poter prendere ogni singolo pacchetto. La funzione che fa questo si chiama `fetch_packs` che prende tre argomenti:

```
fetch_packs row dest smirror
```

dove

- la variabile *row* indica il contenuto di ogni singola riga;
- la variabile *dest* indica la directory di destinazione, nella quale viene copiato il pacchetto;
- la variabile *smirror* indica il mirror slackware da utilizzare per scaricare il pacchetto.

Questa funzione inizialmente controlla se la riga *row* inizia con le parole chiave `url` o `local`:

```
url=`echo $1 |grep '^url'`
plocal=`echo $1 |grep '^local'`
```

Se la variabile *url* non è vuota allora il pacchetto viene scaricato seguendo l'indirizzo specificato utilizzando il comando `wget`. Infine, il pacchetto viene copiato nella directory di destinazione *dest*.

Se la variabile *plocal* non è vuota allora il pacchetto viene semplicemente copiato nella directory di destinazione *dest*.

Infine, se entrambe le variabili sono nulle allora utilizzeremo il mirror *smirror*. Pertanto si verifica la presenza del pacchetto sul mirror utilizzando il file chiamato `FILE_LIST`[5]. Successivamente si utilizza il comando `wget` per scaricare il pacchetto che viene copiato nella directory *dest*.

[5]Tale file è presente sul mirror

Funzione: parse_packs_default Ogni singola riga del file `slack_packages` contiene un flag, che può assumere o il valore `on` oppure `off`. Questa flag viene usata per definire le preferenze dello sviluppatore riguardanti la selezione dei pacchetti.

La funzione `parse_packs_default` utilizza la seguente sintassi:

```
parse_packs_default row dfile sfile
```

dove

- *row* indica la riga contenente le informazioni sul pacchetto;
- *dfile* specifica il nome del file che contiene le impostazioni di default sui pacchetti, e quindi, rappresenta la preferenza dello sviluppatore sul pacchetto;
- *sfile* è il file che contiene l'elenco dei pacchetti software con la flag impostata ad `on`.

Pertanto questa funzione si occuperà di popolare i file *dfile* e *sfile*:

- estrapola il nome del pacchetto dal primo argomento *row*:

```
pack_name=`echo row | awk -F\:  '{
    if ($0 ~ /^url/ || $0 ~ /^local/) {
        print $2
    }else {
        print $1
    }
}'`
name=`basename pack_name .tgz`
```

- estrae il flag impostato per il pacchetto e se non esiste nessun valore allora viene impostato ad `off`:

```
flag=`echo row |awk -F\:  'print $NF'`
[ ''flag'' = ''' ] && flag=''off''
```

- infine vengono salvati i dati appena trovati nei file *dfile* e *sfile*:

```
echo ''\''name\'' \''\'' \''flag\'' \\'' >> dfile
[ ''flag'' = ''on'' ] && echo ''name'' >> sfile
```

dove la sintassi del file *dfile* rispecchia in parte a quella che si usa con il programma `dialog` (vedi 7.6.2).

7.6.2 Lato utente

L'utente deve poter utilizzare un'interfaccia grafica per selezionare i pacchetti software.

Per default può scegliere tra quattro menu:

- `BASE`, contiene tutti i pacchetti fondamentali;
- `NETWORK`, dove ci sono i pacchetti utilizzati per collegarsi ad Internet;
- `DEVEL`, comprende i tool per lo sviluppo del software;
- `GUI`, è il menu che contiene i programmi e librerie per avere un'interfaccia grafica.

Ogni menu è composto dall'elenco di pacchetti software scelti dallo sviluppatore.

Pertanto avremo la *testata* di `dialog` molto simile per tutti i menu:

```
dialog --title ‘‘title’’ \
    --separate-output \
    --ok-label ‘‘Continue’’ \
    --cancel-label ‘‘Reset’’ \
    --checklist ‘‘’’ 22 50 16 \
```

dove possiamo scegliere se reimpostare i valori di default utilizzando il bottone `Reset` oppure dopo aver scelto l'elenco dei pacchetti da installare usare il bottone `Continue`. Questa testata viene poi salvata all'interno di un file nella directory `/tmp` che chiameremo *tfile* in questo modo:

```
cat > tfile << EOF
    Qui inseriamo la testata sopra descritta con gli opportuni escape
EOF
```

Dato che non sappiamo di quali pacchetti sarà composta la distribuzione, per costruire un'interfaccia grafica con `dialog` utilizziamo il file *dfile*, costruito in precedenza:

```
cat dfile >> tfile
echo ‘‘2>packs_selected’’ >> tfile
```

dove il primo comando serve a concatenare il file che contiene l'elenco dei pacchetti all'interno del file temporaneo *tfile*; mentre il secondo comando salva i programmi selezionati dall'utente all'interno del file *packs_selected*.

A questo punto abbiamo un file che contiene un comando `dialog`, per poterlo lanciare usiamo la seguente riga:

```
source tfile
```

7.7 Installare i pacchetti

Prima di partire direttamente con l'installazione dei singoli pacchetti si deve controllare se è stato creato il file *packs_selected* per ogni menu.

Se non è stato creato allora viene usato l'elenco dei pacchetti di default generato dalla funzione `parse_packs_default`, ovvero *sfile*:

```
if [ !  -e packs_selected ]
then
     cp sfile packs_selected
fi
```

Il passo successivo consiste nell'installazione di ogni singolo pacchetto:

```
for i in `cat packs_selected`
do
     package=`ls /mnt/my_distro/cdrom/packs/menu/i*.tgz`
     package=`echo package |cut -d -f1`
     install_package ‘‘package’’
done
```

dove, il primo comando all'interno del ciclo `for` serve a cercare all'interno del `CDROM` il pacchetto rappresentato dalla variabile `i`. Siccome esistono pacchetti con nomi molto simili allora viene preso solo il primo della lista dei pacchetti che possiedono quel nome. Infine viene chiamata la funzione `install_package`.

Funzione: install_package Questa funzione è basata sullo script contenuto nella suite di programmi Slackware: `installpkg`.

L'installazione di un pacchetto è composta da due argomenti:

- il controllo sull'itegrità del pacchetto;
- estrarre i file contenuti nel pacchetto e configurarlo.

Per garantire che il pacchetto sia integro si effettuano due test:

1. Si utilizza il comando `gzip` per garantire la consistenza del pacchetto:

   `gzip -l` *`package`*

   se il suo *exit status* è diverso da zero allora il pacchetto è corrotto.

2. Si utilizza il comando `tar` per controllare se i singoli file sono integri:

   `tar tzf` *`package`*

   anche in questo caso se l'exit status del programma è diverso da zero l'installazione del pacchetto viene bloccata.

Il passo successivo consiste nell'estrarre i file contenuti nel pacchetto. I pacchetti di Slackware sono degli archivi `tar` compressi con `gzip`, solitamente vengono estratti nella root directory che nel nostro caso è `/mnt/my_distro`. Il comando utilizzato è:

`tar xzf` *`package`*

Tramite questo comando involontariamente installiamo i singoli file poiché scriviamo nella root directory. Nella maggiorparte dei pacchetti esistono le librerie che servono al corretto funzionamento del programma appena installato, per poter aggiornare l'intero sistema con le nuove librerie appena installate usiamo il comando:

`sbin/ldconfig -r .`

in modo tale che il linker `ld` saprà dove andare a prendere la libreria di cui ha bisogno un programma.

Per quanto riguarda l'installazione ogni pacchetto Slackware contiene una directory chiamata `install`. Al suo interno esistono due file:

- uno chiamato `slack-desc` che contiene la descrizione del pacchetto;
- l'altro è lo script di configurazione chiamato: `doinst.sh`.

Pertanto per configurare il pacchetto usiamo il comando:

```
sh install/doinst.sh -install
```

Infine usiamo il comando:

```
rm -rf install
```

per pulire il filesystem dai files che non verranno più usati; nel nostro caso, la directory `install`.

7.8 Configurazione del bootloader e chroot

Il bootloader di default in una distribuzione Slackware è `LILO`.

Questo bootloader possiede un file di configurazione che nella maggiorparte dei pacchetti è rappresentato dal file `/etc/lilo.conf`. Nel file di configurazione, tra le tante keyword presenti, quella più importante è *boot*. Essa rappresenta il dispositivo presente in `/dev` dove il bootloader stesso viene installato. Per conoscere questo dispositivo, e quindi impostare correttamente la variabile *boot*, troviamo inizialmente la partizione root che viene sempre montata sulla directory `/mnt/my_distro` con il seguente comando:

```
root=`awk '{if($2 == ``/mnt/my_distro'') print $1 }' \
    /proc/mounts`
```

Le partizioni, come è stato precedentemente detto, sono rappresentate da un file il cui nome termina con uno o più numeri, mentre se questi numeri vengono eliminati allora il file rappresenta l'intero disco. Dato che un bootloader viene solitamente installato nel `MBR`[6], ossia i primi 512 bytes del disco allora la variabile *boot* è pari proprio al dispositivo che rappresenta il disco per intero. Quindi con il seguente comando eliminiamo i numeri dalla variabile *root* precedentemente computata:

```
boot=`echo root | sed 's/[0-9]//g'`
```

Usando queste due variabili è possibile costruire il file di configurazione necessario ad effettuare il boot del nuovo sistema operativo, ossia:

```
default = linux
prompt
```

[6]Master Boot Record

```
lba32
boot = boot
root = root
read-only
image = /boot/vmlinuz
label = linux
```

Dal file si nota che l'opzione scelta per default è `linux`. Questa carica l'immagine del kernel `/boot/vmlinuz` che rappresenta l'ultimo kernel installato dalla distribuzione — se ne è presente più di uno. Imposta la partizione di root alla variabile *root* precedentemente calcolata e quando il sistema verrà avviato il filesystem viene montato in sola lettura.

A questo punto siamo pronti per poter installare il bootloader. Per farlo lanciamo il comando:

```
lilo -v
```

Nella versione corrente dello script shell non sono supportate le configurazioni dinamiche per effettuare il boot da sistemi operativi differenti rispetto alla distribuzione appena installata. Sarà compito dell'utente finale riconfigurare il file di configurazione di `LILO` per poter usare gli altri sistemi operativi presenti sulla stessa macchina.

7.8.1 Cambiare la directory di root

Il comando per installare `LILO` nel `MBR` deve essere lanciato nell'ambiente di lavoro insito della distribuzione e non in quello del `RAM` disk.

Quindi questo comando viene posizionato in uno script che si troverà all'interno della partizione di root.

Inoltre, lo sviluppatore può scrivere ulteriori comandi in questo script per poter personalizzare la distribuzione. Ad esempio aggiungendo il comando:

```
passwd root
```

per impostare la password dell'utente root prima che la distribuzione effettui il suo primo avvio.

Pertanto questo script viene lanciato con il comando:

```
chroot /mnt/my_distro script
```

dove la variabile *script* identifica il percorso completo dello script posizionato all'interno della partizione di root.

Ad esempio, se il file reale si chiama `initialize` ed ha questo percorso assoluto:

rscript=/mnt/my_distro/scripts/initialize

nel comando `chroot` la variabile *script* assumerà questo valore:

/scripts/initialize

Lo script deve avere i privilegi di esecuzione, pertanto prima di lanciare il comando `chroot`, i permessi vengono cambiati con il seguente comando:

chmod 755 *rscript*

dove la variabile `rscript` rappresenta il percorso assoluto reale dello script.

Dopo aver eseguito quest'ultimo script allora è possibile rimuoverlo con il comando:

rm *rscript*

7.9 Riavvio del pc

Dopo aver installato la distribuzione sul disco ed aver effettuato gli ultimi ritocchi eseguendo lo script in `chroot`, l'installer inizia la procedura di riavvio della macchina.

La prima operazione consiste nel smontare tutti i filesystem attualmente montati con il comando:

```
umount -a > /dev/null
```

dove l'opzione `-a` và a leggere il file `/etc/mtab`.

Ma come abbiamo visto nel paragrafo 5.1, questo file è un link simbolico al file `/proc/mounts`, e quindi smonterà tutti i filesystem attualmente montati.

Successivamente, facciamo uscire il cassetto del `CDROM` utilizzando il comando:

```
eject
```

Per default `eject` effettua una chiamata di sistema `ioctl()` sul dispositivo `/dev/cdrom`. Durante la fase di riconoscimento del `CDROM`, come è stato precedentemente descritto nel paragrafo 7.5, si è costruito un link simbolico

del dispositivo che rappresenta il `CDROM` proprio su `/dev/cdrom`, pertanto non c'è bisogno di passare alcun argomento su questa applicazione.

A questo punto, viene chiesto all'utente di togliere il `CDROM` e di premere un tasto qualsiasi della tastiera per effettuare il reboot della macchina. Cioè viene laciata la seguente sequenza di comandi:

```
echo ‘‘Please, remove the cdrom
 and press a key to reboot!!’’
read a
reboot
```

Capitolo 8

Messa a punto e Testing

8.1 Setup dello sviluppatore

Lo script da me creato è liberamente scaricabile su questo indirizzo:

http://vinx.tuxfamily.org/my_distro

Prima di lanciarlo lo sviluppatore deve impostare alcune variabili globali presenti nelle prime righe dello script bash:

```
version=‘‘ver’’
home_dir=‘‘dpath’’
home_files=‘‘fpath’’
oredir=‘‘redirection’’
makeopt=‘‘options’’
optflags=‘‘flags’’
arch=‘‘type’’
```

dove vengono impostate:

- la versione della distribuzione;
- l'ambiente di lavoro è rappresentato dal percorso *dpath*;
- i pacchetti da scaricare verranno salvati nella directory *fpath*;
- per abilitare o disabilitare l'output di ogni singolo comando;
- le opzioni del comando `make`;

- le opzioni da passare al compilatore `gcc`;
- il tipo di piattaforma hardware, al momento sono supportate: `i386` e `x86_64`.

Successivamente verranno impostati i permessi di esecuzione sullo script:

`chmod 755` *`script`*

e verrà eseguito dall'utente `root`.

8.2 Macchine virtuali e reali

8.2.1 Macchine virtuali

Lo script è stato inizialmente testato su piattaforme virtuali sia open-source quali `Qemu` e `VirtualBox` che closed-source come varie versioni `VMware` liberamente scaricabili da Internet.

Utilizzando `Qemu`[1] si ottiene un comportamento instabile del `CDROM` di installazione appena creato dato che non esiste alcuna emulazione `SCSI` e le periferiche `E/IDE` a volte non leggono correttamente i pacchetti presenti sul `CDROM` e il kernel ritorna questo tipo di errore:

```
hdc:  command error:  status=0x41 {DriveReady Error }
hdc:  command error:  error=0x50 {LastFailedSense=0x05 }
ide:  failed opcode was:  unknown
ATAPI device hdc:
    Error:  Illegal request -- (Sense key=0x05)
    Invalid command operation code -- (asc=0x20, ascq=0x00)
    The failed ‘‘Read 10’’ packet command was:
    28 00 00 00 1e 9e 00 00 01 00 00 00 00 00 00 00
end_request:  I/O error, dev hdc, sector 31356
Buffer I/O error on device hdc, logical block 7838
```

Se si riesegue a mano l'operazione che ha generato questo fallimento invece viene eseguita senza nessun problema. L'instabilità viene fuori anche utilizzando la stessa `ISO` per due installazioni identiche eseguite da `Qemu`, in questo caso i pacchetti che vengono considerati *corrotti* per colpa dell'errore

[1] evoluzione di `Bochs`

precedente in maniera randomica. Ad esempio, nella prima installazione i pacchetti che falliscono sono `bash` e `glibc`; mentre nella seconda installazione fallisce solo `readline`. Utilizzando la stessa `ISO` dovremmo ottenere lo stesso elenco di pacchetti corrotti.

A tal proposito è stato implementato un sistema di *checksumming* per i pacchetti. Praticamente viene confrontata la stringa `MD5` generata per il pacchetto scaricato all'interno della directory ***fpath*** con lo stesso identico pacchetto presente sulla `ISO` generata dallo script. Ma anche con questo controllo i test sulla piattaforma di virtualizzazione `Qemu` falliscono per i motivi precedentemente specificati.

Utilizzando `VirtualBox`[2] abbiamo un comportamento stabile della distribuzione. Questo virtualizzatore supporta solo la virtualizzazione di dischi `E/IDE`.

Utilizzando varie versioni di `VMware` si è potuto testare anche hardware virtuale di tipo `SCSI`. VMware utilizza i moduli `Fusion MPI` per collegarsi tramite il metodo di trasporto `iScsi` ai dischi `SCSI` presenti sul delle SAN[3], in alternativa è possibile utilizzare anche i moduli `BusLogic` sempre per virtualizzare dei dischi `SCSI` presenti sulla macchina. Inoltre supporta la virtualizzazione dei dischi `E/IDE`. I vari test effettuati sono stati positivi per tutti i tipi di disco che possono essere creati con questa piattaforma.

Pertanto è stata rinnovata la configurazione del kernel presente sul `CDROM` abilitando questo tipo di periferiche.

8.2.2 Macchine reali

Dopo aver testato l'hardware "virtuale", si sono utilizzati pc "reali" con hardware differente.

Le macchine virtuali associano dei nomi "virtuali" ai dischi con dei nomi standardizzati, mentre le macchine reali forniscono dei nomi nettamente differenti ed univoci. Data la varia natura dei nomi di un disco reale, si è deciso di estendere l'espressione regolare in questo modo.

Le macchine reali utilizzate per i test sono state effettuate su:

- un vecchio portatile: Processore Celeron 450MHz, 64MByte di RAM, 4GByte di hard disk
- un pc fisso con: Processore AMD Athlon 1800+, 256MByte di RAM, 80GByte di hard disk

[2] basato sul codice di `Qemu`

[3] Storage Area Network

- un mac book con: Processore Intel Core duo 2GHz, 1GByte di RAM, 80GByte di hard disk

Su hardware differente non ha dato alcun tipo di problema. Scegliendo l'installazione minimale, la distribuzione è operativa mediamente in 5 minuti; mentre installando tutti i pacchetti presenti sul `CDROM` abbiamo bisogno di circa 15 minuti.

8.3 Setup dell'utente

Qui di seguito vedremo gli screenshot durante il processo di installazione.

Innanzitutto, l'utente dovrà impostare il `CDROM` come primo disco di boot nel `BIOS`.

Dopo aver inserito il `CDROM`, partirà, ad esempio, `isolinux`. Premendo il tasto `INVIO` oppure scrivendo la parola `mydistro` si farà partire la procedura di installazione normale. Mentre se si preme il tasto `F2`, il bootloader farà vedere le opzioni che si possono aggiungere alla riga di comando. Attualmente l'unica altra opzione è `mydistro single_user` che genera un sistema minimale di lavoro sul ramdisk.

Nella figura 8.1 vediamo il momento in cui il bootloader analizza la richiesta dell'utente e carica il kernel e l'`initrd` precedentemente costruito.

Figura 8.1: Startup del bootloader

Dopo che il kernel ha finito la sua fase di inizializzazione, la prima interazione che l'utente ha con l'installazione consiste nel selezionare la corretta mappatura della tastiera.

Nella figura 8.2 impostiamo la tipologia di tastiera utilizzata.

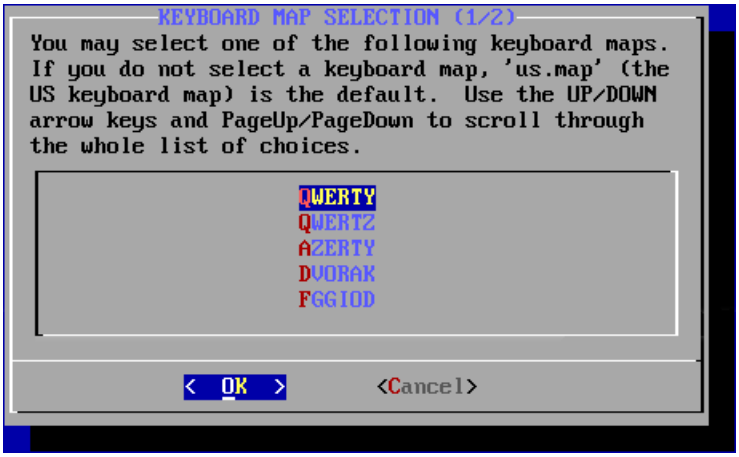


Figura 8.2: Mappatura della tastiera - 1

Mentre nella figura 8.3 l'utente sceglierà la mappatura della tastiera identificativa del proprio paese – ad esempio, `it` sta per italia.

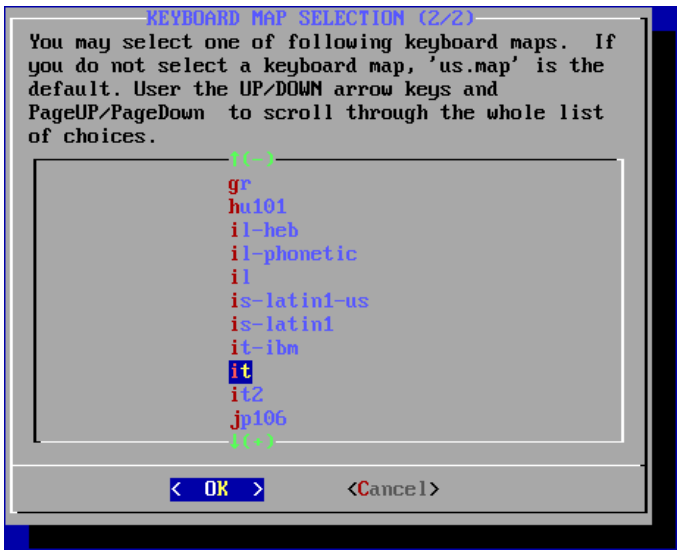


Figura 8.3: Mappatura della tastiera - 2

A questo punto, se l'utente ha usato l'opzione `single_user` si otterrà una shell; in alternativa inizierà la vera procedura d'installazione.

Nella figura 8.4 verrà proposta laa lista di dischi rigidi riconosciuti, sui quali è possibile installare la distribuzione.

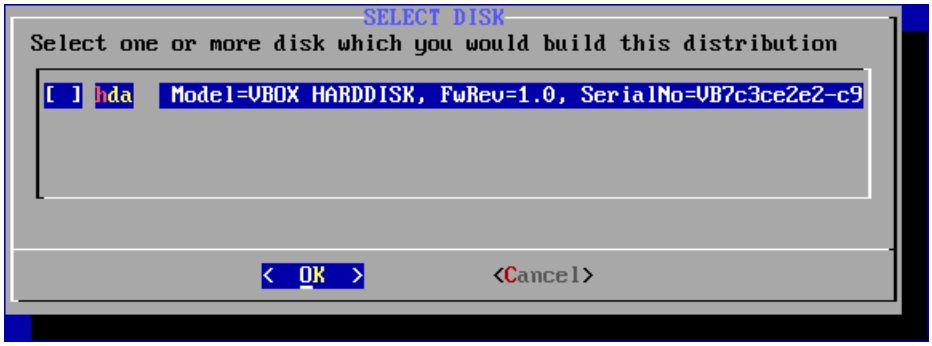


Figura 8.4: Lista hard disk

Pertanto, come si vede nella figura 8.5, l'utente selezionerà uno o più dischi rigidi.

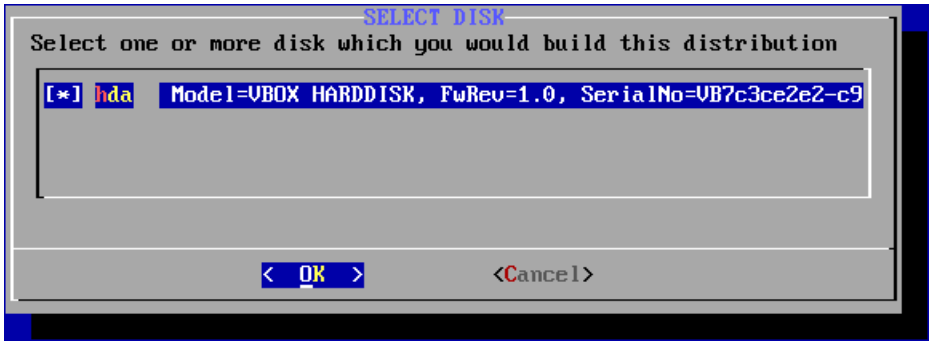


Figura 8.5: Selezione hard disk

A questo punto l'utente dovrà scegliere se partizionare i dischi selezionati oppure no – come si vede dalla figura 8.6.

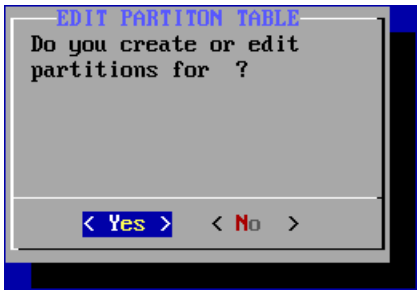


Figura 8.6: Partizionamento dischi - 1

Qualora l'utente vuole cambiare la tabella delle partizioni verrà chiamato il programma `cfdisk`, il quale presenta un'interfaccia come la figura 8.7.

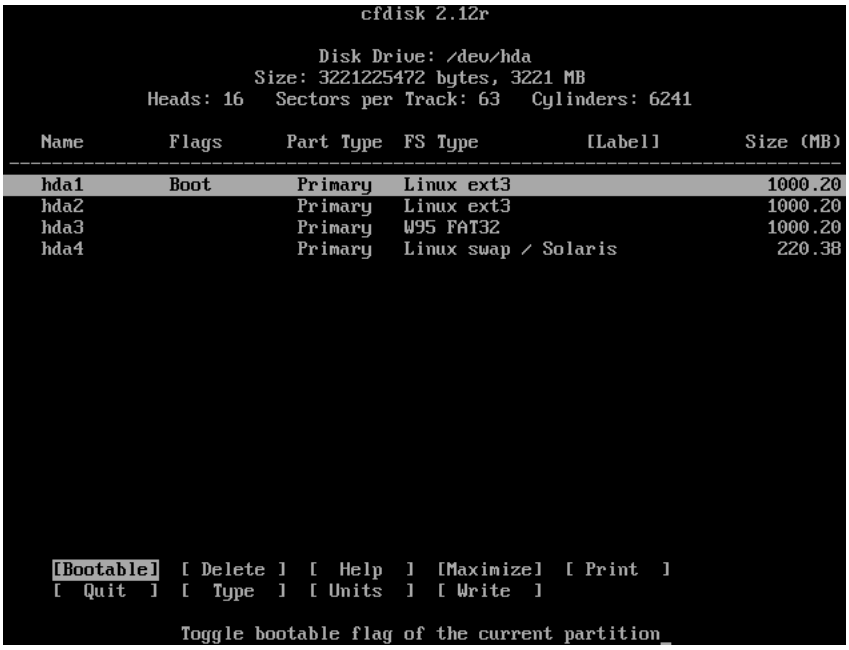


Figura 8.7: Partizionamento dischi - 2

Nella figura 8.8, l'utente seglierà se effettuare il controllo sul filesystem.

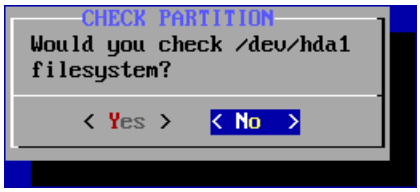


Figura 8.8: Check del filesystem

Mentre, nella figura 8.9, sceglierà se formattare la partizione.

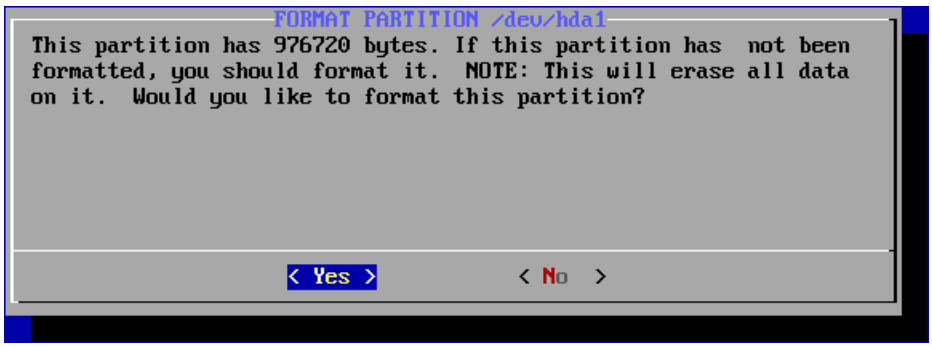


Figura 8.9: Formattazione partizione

Infine, l'utente imposterà il mount point della partizione – come visto in figura 8.10.

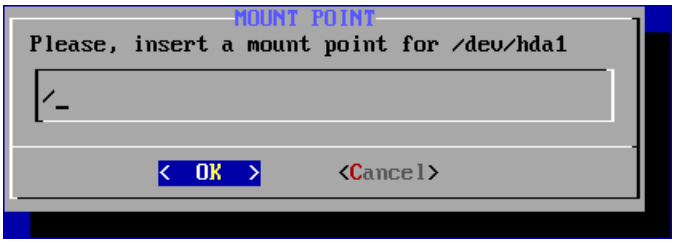


Figura 8.10: Mount point

Nella figura 8.11, l'utente sceglie tra i menu disponibili.

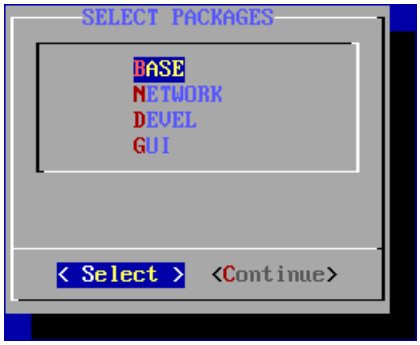


Figura 8.11: Selezione pacchetti - 1

Ogni menu contiene una serie di pacchetti software che l'utente può installare.

Nella figura 8.12, si vede una parte della lista dei pacchetti presente nel menu Base.

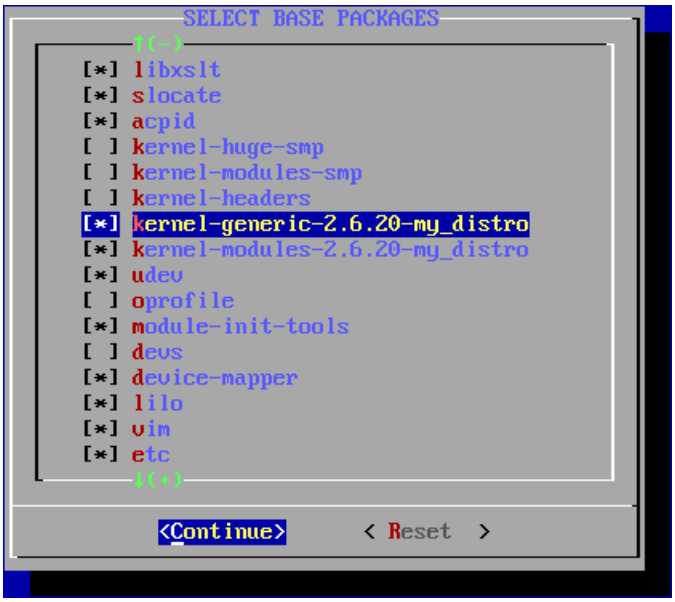


Figura 8.12: Selezione pacchetti - 2

Dopo aver configurato la lista di pacchetti che l'utente ha intenzione di installare, partirà l'installazione vera e propria.

Nella figura 8.13, si vede l'andamento dell'installazione.

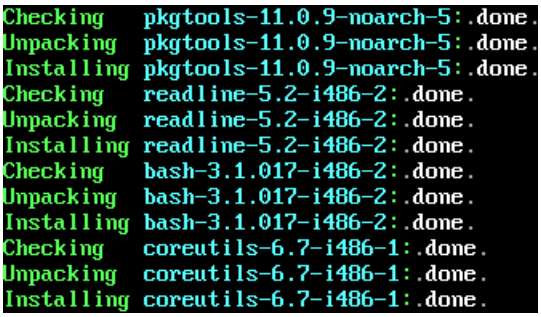


Figura 8.13: Installazione pacchetti

Una volta completata l'installazione, verrà chiesto all'utente di impostare la password per l'utente root e l'hostname della macchina appena installata – come si vede nella figura 8.14.

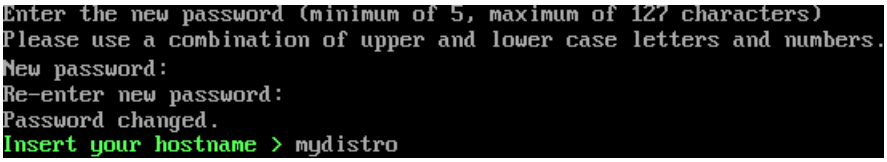


Figura 8.14: Impostazione della password di root e dell'hostname

Dopo aver finito la procedura automatica di installazione del bootloader e delle configurazioni del sistema di base (invisibili all'utente), verrà espulso il `CDROM` ed effettuata l'operazione di riavvio della macchina – come si vede in figura 8.15.

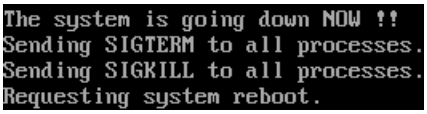


Figura 8.15: Riavvio

Bibliografia

[1] *http://syslinux.zytor.com*, Home page del progetto `isolinux`.

[2] *http://www.gnu.org/software/grub/*, Home page del progetto `GRUB`.

[3] *http://www.busybox.net*, Home page del progetto `BusyBox`.

[4] *http://www.gnu.org/software/libc/libc.html*, Home page del progetto `glibc`.

[5] *http://www.uclibc.org*, Home page del progetto `uClibc`.

[6] *http://lsb.freestandards.org*, Home page del progetto `Linux Standard Base`.

[7] *http://sourceware.org/autobook/*, GNU Autoconf, Automake and Libtool - 2006 - Gary V. Vaughan, Ben Elliston, Tom Tromey e Ian Lance Taylor

[8] *http://tldp.org/LDP/abs/*, Advanced Bash-Scripting Guide - 2007 - Mendel Cooper

[9] *http://www.linux.it/ rubini/docs/sysfs/sysfs.html*, Introduzione a sysfs - 2006 - Alessandro Rubini

[10] *http://www.ibm.com/developerworks/linux/library/l-initrd.html*, Linux initial RAM disk (initrd) overview - 2006 - M. Tim Jones

[11] *http://bravo.ce.uniroma2.it/didattica/la07/lez11-12.tar*, Inizializzazione del kernel - 2007 - Daniel P. Bovet

[12] *http://bravo.ce.uniroma2.it/kernelhacking2006/lkh06_2_p.pdf*, Compiling and installing the kernel - 2006 - G. Grilli

[13] *http://www.linuxjournal.com/article/2807*, Dialog: An Introductory Tutorial - 1994 - Jeff Tranter

[14] *http://www.busybox.net/downloads/BusyBox.html#item_init*, BusyBox's man page

[15] *http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=1750*, Information processing – Volume and file structure of CD-ROM for information interchange - 1988, International Organization for Standardization

[16] *http://www.phoenix.com/NR/rdonlyres/98D3219C-9CC9-4DF5-B496-A286D893E36A/0/specscdrom.pdf*, The "El Torito" Bootable CD-ROM Format Specification - 1995 - Curtis E. Stevens e Stan Merkin

[17] *http://developer.osdl.org/dev/DCL/PSDN/Testing_udev_notes.html*, Using UDEV to do Persistent storage device naming for large numbers of storage devices - 2004 - Open Source Development Lab