# The Unix and Internet Fundamentals HOWTO

# Contents

**Abstract**

Questo documento descrive il funzionamento di base dei computer di classe PC, dei sistemi operativi di tipo Unix e Internet in un linguaggio non-tecnico. Basato sulla traduzione di Mirko Nasato, aggiornamento a cura di Hugh Hartmann (hhartmann at fastwebnet.it), revisione a cura di Vieri Giugni (v dot giugni at gmail dot com).

# 1 Introduzione

## 1.1 Scopo di questo documento

Questo documento ha lo scopo di aiutare gli utenti di Linux e di Internet che stanno imparando attraverso la pratica. Mentre questo è un ottimo metodo per acquisire competenze specifiche, qualche volta lascia determinate lacune nella conoscenza dei concetti di base, che possono rendere difficile pensare in modo creativo o risolvere efficacemente dei problemi, a causa della mancanza di un chiaro modello mentale relativo a cosa stia realmente accadendo.

Proverò a descrivere, con un linguaggio chiaro e semplice, come funziona il tutto. La presentazione sarà orientata a coloro che usano Unix o Linux su macchine di classe PC. Tuttavia, in questo testo abitualmente mi riferirò semplicemente a 'Unix', dato che la maggior parte di ciò che descriverò è valido anche per macchine differenti e per altre varianti di Unix.

Assumerò che si stia usando un PC Intel. I dettagli differiscono leggermente se si lavora su di un PowerPC o qualche altro tipo di computer, ma i concetti fondamentali sono gli stessi.

Non ripeterò le cose, quindi si dovrà prestare attenzione, ma questo significa anche che si imparerà da ogni parola che si legge. È una buona idea limitarsi a dare una rapida lettura la prima volta; poi si dovrà tornare indietro e rileggere alcune volte finché si avrà assimilato quello che si è imparato.

Questo è un documento in evoluzione. Intendo continuare ad aggiungere sezioni in risposta agli stimoli dei lettori, così, periodicamente, si dovrebbe tornare a rivederlo.

## 1.2 Nuove versioni di questo documento

Nuove versioni dello Unix and Internet Fundamentals HOWTO verranno periodicamente postate su , comp.os.linux.help e comp.os.linux.announce e news.answers. Saranno anche caricate su vari siti WWW e FTP dedicati a Linux, inclusa la pagina principale di LDP (Linux Documentation Project).

È possibile visionare l'ultima versione sul World Wide Web attraverso l'URL http:http://www.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/index.html.

Questo testo è stato tradotto nelle seguenti lingue: Farsi, Polacco Spagnolo Turco

## 1.3 Suggerimenti e correzioni

Se si hanno domande o commenti riguardo questo documento ci si senta liberi di contattare Eric S. Raymond all'indirizzo esr@thyrsus.com. Ogni suggerimento o critica sarà benvenuto. Accoglierò specilmente link a spiegazioni più dettagliate sui singoli concetti. Se, in questo documento si trova qualche errore per favore fatemelo sapere, in modo che lo possa correggere nella prossima versione. Grazie.

## 1.4 Risorse relative

Se si sta leggendo questo testo per imparare come fare hacking, si dovrebbe leggere anche How To Become A Hacker FAQ. In questo testo ci sono dei link ad alcune altre risorse utili.

# 2 Anatomia di base del proprio computer

Il proprio computer contiene al suo interno un chip processore che compie l'elaborazione vera e propria. Ha una memoria interna (quella che la gente DOS/Windows chiama "RAM" e la gente Unix spesso chiama "core"; il termine Unix rappresenta una memoria popolare da quando la RAM era costituita da un nucleo di anellini di ferrite). Il processore e la memoria risiedono sulla *scheda madre*, che è il cuore del proprio computer.

Il proprio computer ha uno schermo e una tastiera. Ha dischi rigidi e un lettore ottico di CD-ROM (o potrebbe essere un lettore DVD) e potrebbe esserci un lettore di dischi floppy. Alcuni di questi dispositivi funzionano grazie a *schede controller* che si

connettono sulla scheda madre e aiutano il computer a gestirli; altri sono chipset specializzati implementati direttamente sulla scheda madre che compiono interamente la medesima funzione di una scheda controller. La tastiera è troppo semplice per aver la necessità di una scheda separata; il controller è costruito all'interno della tastiera stessa.

Più avanti entreremo in alcuni dettagli relativi al funzionamento di questi dispositivi. Per ora, ecco alcuni concetti di base da tenere a mente su come esse funzionino insieme:

Tutte le parti interne del proprio computer sono collegate tramite un *bus*. Fisicamente, il bus è quello dove si inseriscono le schede controller (la scheda video, il controller del disco, una scheda audio se è presente). Il bus è l'autostrada dei dati tra il processore, lo schermo, il disco e tutto il resto.

(Se nella connessione con i PC sono stati visti riferimenti a 'ISA', 'PCI', e 'PCMCIA' e non si è capito cosa significhi, questi sono tipi di bus. ISA è, eccetto qualche dettaglio minore, lo stesso bus che era usato sui PC IBM originali nel 1980; ora non viene più usato. PCI, che sta per Peripheral Component Interconnection (interconnessione tra componenti periferici), è il bus usato nella maggior parte dei PC e dei Macintosh moderni. PCMCIA è una variante di ISA con connettori fisici più piccoli usata sui computer laptop.)

Il processore, che fa funzionare tutto il resto, in realtà non è in grado di vedere direttamente nessuna delle altre parti; deve comunicare con loro attraverso il bus. L'altro sottosistema al quale ha accesso veramente rapido, immediato, è la memoria (core). Perché i programmi siano eseguiti, dunque, devono risiedere *nel core* (in memoria).

Quando il proprio computer legge un programma o dei dati dal disco in effetti accade che il processore usa il bus per inviare una richiesta di lettura disco al controller del disco. Dopo un po' di tempo il controller del disco usa il bus per segnalare al computer che ha letto i dati e li ha messi in una certa locazione di memoria. Il processore può allora usare il bus per guardare questi dati.

Anche la tastiera e lo schermo comunicano con il processore attraverso il bus, ma in modi più semplici. Ne discuteremo più avanti. Per ora, si conosce abbastanza per capire cosa succede quando si accende il proprio computer.

## 3 Che cosa succede quando si accende un computer?

Un computer senza un programma in esecuzione è soltanto un ammasso inerte di componenti elettronici. La prima cosa che un computer deve fare quando viene acceso è far partire un programma speciale chiamato *sistema operativo*. Il compito del sistema operativo è quello di aiutare gli altri programmi del computer a funzionare, gestendo gli intricati dettagli relativi al controllo dell'hardware del computer.

Il processo di avvio del sistema operativo è chiamato *booting* (in origine era *bootstrapping* e alludeva al processo di tirarsi su da solo, "attraverso il proprio bootstraps"). Il proprio computer sa come avviarsi perché le istruzioni per il boot sono incorporate in uno dei suoi chip, il BIOS (Basic Input/Output System).

Il chip BIOS gli comunica di cercare in un posto predefinito del disco fisso con numero più basso (il *disco di avvio*) uno speciale programma chiamato *boot loader* (sotto Linux il boot loader è chiamato Grub o LILO) che si trova. Il boot loader è caricato in memoria e avviato. Il compito del boot loader è quello di avviare il sistema operativo vero e proprio

Per compiere quest'ultima operazione il loader cerca un *kernel*, lo carica in memoria e lo avvia. Quando si avvia Linux e si vede "LILO" sullo schermo seguito da una riga di puntini, vuol dire che sta caricando il kernel. (Ogni puntino significa che ha caricato un altro *blocco del disco* di codice del kernel).

(Ci si potrebbe meravigliare del perchè il BIOS non carichi direttamente il kernel — perchè questo processo a due stadi con il boot loader? Bene, il BIOS non è molto intelligente. In effetti è proprio stupido, e Linux non lo usa più dopo la fase di avvio. Fu scritto in origine per i primi PC a 8 bit con dischi poco capienti e proprio non riesce ad accedere a una parte abbastanza grande del disco per caricare direttamente il kernel. La fase del boot loader consente anche di far partire diversi sistemi operativi da posti diversi del disco, nella improbabile eventualità che Unix non vi soddisfi a sufficienza.)

Una volta avviato, il kernel si guarda in giro, trova il resto dell'hardware e si prepara ad eseguire i programmi. Fa tutto questo guardando non nelle ordinarie locazioni di memoria ma piuttosto alle *porte I/O*; speciali indirizzi del bus che probabilmente hanno schede controller dei dispositivi che sono in ascolto in attesa di comandi. Il kernel non cerca a caso; ha molta conoscenza al suo interno su cosa sia probabile trovare e dove, e su come i controller rispondano se sono presenti. Questo processo è chiamato *autorilevamento*.

Si potrebbe essere o non essere capaci di vedere ogniuno di questi messaggi Una volta, quando i sistemi Unix usavano le console di testo, si sarebbero visti i messaggi di boot scorrere sul proprio schermo nel momento dell'avvio del sistema. Al giorno d'oggi

spesso gli Unix nascondono i messaggi d'avvio dietro una schermata grafica sullo schermo. È possibile vedere i messaggi d'avvio passando a una console di testo con la combinazione di tasti Ctrl-Shift-F1. Se questo funziona, si dovrebbe essere capaci di tornare indieto allo schermo grafico d'avvio con una sequenza differente dei tasti Ctrl-Shift; provare F7, F8, e F9.

La maggior parte dei messaggi emessi al momento dell'avvio sono del kernel che effettua l'autorilevamento del proprio hardware attraverso le porte I/O, riconosce cosa ha a sua disposizione e si adatta al computer. Il kernel di Linux è estremamente preciso in questo, meglio della maggior parte degli altri Unix e *molto* meglio del DOS o di Windows. Infatti, molti utenti di Linux di vecchia data pensano che l'intelligenza del rilevamento all'avvio di Linux (che lo rende relativamente facile da installare) sia stata una delle principali ragioni che lo hanno fatto sfondare rispetto a tanti esperimenti di Unix liberi, attraendo una massa critica di utenti.

Ma avere il kernel completamente caricato e funzionante non è la fine del processo di avvio; ne è solo il primo stadio (a volte chiamato *run level 1*, livello di esecuzione 1). Dopo questo primo stadio, il kernel passa il controllo ad un processo speciale chiamato 'init' che genera diversi processi residenti comuni. (Alcuni Linux recenti usano un programma differente chiamato 'upstart' che svolge compiti simili)

Il primo compito del processo init è abitualmente quello di verificare che i dischi siano a posto. I file system dei dischi sono cose fragili; se vengono danneggiati da un malfunzionamento hardware o da un'improvvisa mancanza di elettricità, ci sono buoni motivi per compiere alcuni passaggi di recupero prima che Unix sia perfettamente funzionante. Approfondiremo ciò più avanti, parlando di come un file systems può essere danneggiato.

Il passo successivo di init è quello di avviare diversi *daemon*. Un demone (o daemon) è un programma come uno spooler di stampa, un programma che attende di ricevere posta in arrivo o un server WWW che rimane latente in sottofondo, aspettando qualcosa da fare. Questi programmi speciali spesso devono coordinare diverse richieste che potrebbero entrare in conflitto. Sono demoni perché spesso è più facile scrivere un programma che gira costantemente e viene a conoscenza di tutte le richieste piuttosto che cercare di assicurarsi che un gruppo di copie (che girano tutte contemporaneamente, con ciascuna che processa una richiesta) non si ostacolino a vicenda. La particolare serie di demoni che il vostro sistema fa partire può variare, ma quasi certamente include uno spooler di stampa (un demone che fa da portinaio per la propria stampante).

Il prossimo passo è quello di prepararsi per gli utenti. Init avvia una copia di un programma chiamato **getty** per controllare lo schermo e la tastiera (e forse altre copie per controllare le porte seriali dial-in). Attualmente, al giorno d'oggi init avvia copie multiple di **getty** così da avere diverse console virtuali (solitamente 7 o 8), con schermo e tastiera connessi ad una di esse in ogni momento. Ma, probabilmente, non si vorrà vedere nessuna di queste, dato che una delle proprie console sarà occupata dal server X (che approfondiremo tra poco).

Non abbiamo ancora finito. Il passo successivo è quello di avviare vari daemon che supportano la rete e altri servizi. Il più importante di questi è il server X. X è un daemon che gestisce lo schermo, la tastiera e il mouse. Il suo compito principale è quello di produrre pixel grafici a colori che si possono vedere normalmente sullo schermo.

Quando il server X si avvia, durante l'ultima parte del processo di avvio del proprio computer, esso prende in consegna efficacemente l'hardware da qualsiasi console virtuale stesse controllando in precedenza. Ciò succede quando si vedrà uno schermo di login grafico, prodotto da un programma chiamato *gestore di display*.

## 4 Cosa succede quando si fa il log in?

Quando si fa il log in, ci si identifica sul computer. Sui moderni Unix abitualmente il log in si farà tramite un gestore di display grafico. Ma è anche possibile passare alle console virtuali tramite una sequenza dei tasti Crtl-Shift e fare il login testuale. In questo caso si passa attraverso l'istanza del comando **getty** che controlla da quale console chiamare il programma **login**.

Ci si identifica al gestore di display o con il **login** con un nome e una password di login. Questo nome di login è ricercato in un file chiamato /etc/passwd, che è composto da una sequenza di linee ogniuna delle quali descrive un account utente.

Uno di questi campi è una versione criptata della password dell'account (qualche volta i campi criptati sono abitualmente riposti in un secondo file /etc/shadow con permessi limitati; questo rende più difficile effettuare il cracking della password). Quello che viene inserito come password di account viene criptato esattamente allo stesso modo e il programma **login** controlla se essi corrispondono. La sicurezza di questo metodo dipende dal fatto che, mentre è facile passare dalla propria password in chiaro alla versione criptata (cifrata), l'inverso è molto più difficile. Per cui, anche se qualcuno potesse vedere la versione cifrata della vostra password non può usare il vostro account. (Significa anche che se si dimentica la propria password, non c'è alcun modo di recuperarla, ma solamente di cambiarla in un'altra di propria scelta).

Una volta effettuato il log in con successo, si ottengono tutti i privilegi associati al singolo account che si sta utilizzando. Si può anche essere riconosciuti come appartenenti a un *gruppo*. Si chiama gruppo un insieme di utenti impostato dall'amministratore di sistema. I gruppi possono avere privilegi indipendentemente dai privilegi dei loro membri. Un utente può essere un membro di gruppi multipli. (Per maggiori dettagli su come funzionano i privilegi in Unix, vedere la sezione seguente sui permessi.)

(Si noti che, sebbene si faraà normalmente riferimento agli utenti e ai gruppi mediante il nome, essi sono solitamente immagazzinati internamente come ID numerici. Il file password associa il proprio nome di account a un ID di utente; il file `/etc/group` associa i nomi del gruppo agli ID numerici del gruppo. I comandi relativi agli account e ai gruppi effettuano automaticamente la conversione).

La propria voce di account contiene anche la propria *home directory*, il luogo nel file system Unix dove sono contenuti i propri file personali. Infine, la voce dell'account imposta anche la propria *shell*, l'interprete dei comandi che **login** avvierà per accettare i propri comandi.

Che cosa succede dopo aver effettuato il login con successo dipende da come lo si è fatto. Su una console di testo, **login** lancierà una shell e si sarà in pista. Se si è fatto il login attraverso un gestore di display, il server X farà partire il proprio desktop grafico e si potranno eseguire programmi da esso; sia attraverso i menu, che attraverso le icone del desktop, o attraverso un *emulatore di terminale* che esegue una *shell*.

## 5 Cosa succede quando si eseguono i programmi dopo la fase di avvio?

Dopo la fase di avvio, e prima che sia eseguito un programma, si può pensare al proprio computer come ad un contenitore di molti processi che stanno tutti aspettando qualcosa da fare. Stanno tutti aspettando degli *eventi*. Un evento può essere la pressione di un tasto o il movimento di un mouse. Oppure, se il proprio computer è collegato a una rete, un evento può essere un pacchetto di dati che arriva tramite quella rete.

Il kernel è uno di questi processi. È un processo speciale, perché controlla quando gli altri*processi utente* possono essere eseguiti, ed è normalmente l'unico processo con accesso diretto all'hardware del computer. Infatti, i processi utente devono fare richiesta al kernel quando vogliono ottenere un input dalla tastiera, scrivere sullo schermo, leggere o scrivere sul disco, o fare solo qualsiasi altra cosa che non sia macinare bit in memoria. Queste richieste sono conosciute come *chiamate di sistema*.

Normalmente tutto l'I/O passa attraverso il kernel, così quest'ultimo può organizzare le operazioni e impedire che i processi si ostacolino a vicenda. Alcuni processi utente speciali hanno il permesso di aggirare il kernel, di solito per ottenere accesso diretto alle porte I/O. I server X sono gli esempi più comuni di questo fatto.

Si potranno eseguire i programmi in uno dei due modi: attraverso il proprio server X o attraverso una shell. Spesso, abitualmente si farà da entrambi, dato che si avvierà un emulatore di terminale che imita una console testuale vecchio stile, fornendo una shell per eseguire i programmi da essa. Descriverò che cosa succede quando si fa questo, poi ritornerò a cosa succede quando si esegue un programma attraverso un menu di X o una icona del desktop.

La shell è così chiamata perché essa avvolge completamente e nasconde il kernel del sistema operativo. È una importante caratteristica di Unix che la shell e il kernel siano programmi separati che comunicano attraverso un piccolo insieme di chiamate di sistema. Questo rende possibile che ci siano shell multiple, che si adattano a differenti gusti nelle interfacce.

La shell normale fornisce il prompt '$' che si vede dopo il login (a meno che non lo abbiate personalizzato per essere qualcosa d'altro). Non parleremo della sintassi della shell e delle cose semplici che si possono vedere sullo schermo; daremo piuttosto uno sguardo dietro le quinte a quello che succede dal punto di vista del computer.

La shell è solo un processo utente, e neppure uno tanto speciale. Attende che si digiti qualcosa, ascoltando (tramite il kernel) sulle porte I/O della tastiera. Come il kernel vede che è stato digitato qualcosa lo visualizza sulla propria console virtuale o emulatore di terminale di X. Quando il kernel vede un 'Invio' passa la propria linea di testo alla shell. La shell tenta di interpretare la pressione di questi tasti come dei comandi.

Supponiamo che si digiti 'ls' e Invio per invocare il programma Unix che elenca le directory. La shell applica le sue regole incorporate per cercare di capire che si vuole eseguire il comando eseguibile nel file `/bin/ls`. Essa fa una chiamata di sistema chiedendo al kernel di avviare /bin/ls come un nuovo *processo figlio* e gli fornisce un accesso allo schermo e alla tastiera tramite il kernel. Poi la shell riposa, aspettando che ls finisca.

Quando **/bin/ls** ha finito, comunica al kernel che ha terminato emettendo una chiamata di sistema *exit*. Il kernel allora attiva la shell e le comunica che può continuare a funzionare. La shell emette un altro prompt e attende un'altra linea di input.

Potrebbero succedere altre cose mentre il proprio comando 'ls' è in esecuzione, tuttavia (supponiamo che si stia elencando una directory molto lunga). Per esempio, si potrebbe passare ad un'altra console virtuale, fare il log in da questa console, e iniziare un gioco di Quake. Oppure si supponga di essere collegati a Internet. Il proprio computer potrebbe spedire o ricevere posta mentre **/bin/ls** è in esecuzione.

Quando si eseguono programmi attraverso il server X invece che da una shell (cioè, scegliendo una applicazione da un menu a scomparsa, o facendo doppio click (con il mouse) su di una icona del desktop), ognuno dei diversi programmi associati con il proprio server X può comportarsi come una shell e lanciare il programma. Vorrei sorvolare sui dettagli dato che sono sia variabili che poco importanti. Il punto fondamentale è che il server X, diversamente da una normale shell, non va a dormire mentre il programma client è in esecuzione — invece, si posiziona tra voi e il client, passando i propri click del mouse e la pressione dei tasti ad esso e soddisfando le sue richieste di puntamento di pixel sul proprio schermo.

## 6 Come funzionano i dispositivi di input e gli interrupt?

La propria tastiera è un dispositivo di input molto semplice; semplice perché genera piccole quantità di dati molto lentamente (per gli standard di un computer). Quando si preme o si rilascia un tasto, questo evento viene segnalato attraverso il cavo della tastiera per far scattare un *interrupt hardware*.

È compito del sistema operativo fare attenzione a questi interrupt. Per ogni possibile tipo di interrupt, ci sarà un *gestore di interrupt*, una parte del sistema operativo che immagazzina ogni dato ad esso associato (come il valore del proprio premere/rilasciare il tasto) affinché possa essere processato.

Quello che effettivamente fa il gestore dell'interrupt della vostra tastiera è mettere il valore del tasto in un'area di sistema vicino al fondo della memoria. Là rimarrà disponibile per ispezione quando il sistema operativo passa il controllo a qualsiasi programma che ritiene stia attualmente leggendo dalla tastiera.

Dispositivi di input più complessi come i dischi o le schede di rete funzionano in modo simile. Precedentemente, mi sono riferito a un controller del disco che usa il bus per segnalare che una richiesta disco è stata completata. Quello che in realtà succede è che il disco fa scattare un interrupt. Il gestore dell'interrupt del disco poi copia in memoria i dati ottenuti, a uso successivo da parte del programma che aveva fatto la richiesta.

Ad ogni specie di interrupt è associato un *livello di priorità*. Gli interrupt con priorità più bassa (come gli eventi della tastiera) devono dare la precedenza agli interrupt con priorità più alta (come i tick dell'orologio o gli eventi del disco). Unix è progettato per dare alta priorità al tipo di eventi che hanno bisogno di essere processati rapidamente, in modo da mantenere fluida la risposta del computer.

Nei messaggi d'avvio del proprio sistema operativo, si possono vedere dei riferimenti a numeri di *IRQ*. Forse si sa, senza capirne esattamente il perché, che uno dei modi più comuni di configurare male l'hardware è avere due dispositivi diversi che cercano di usare lo stesso IRQ.

Ecco la spiegazione. IRQ è l'abbreviazione di "Interrupt Request" (richiesta di interrupt). Il sistema operativo ha bisogno di sapere al momento dell'avvio quali interrupt numerati verranno usati da ciascun dispositivo hardware, in modo da poter associare a ciascuno il gestore appropriato. Se due dispositivi diversi cercano di usare lo stesso IRQ a volte gli interrupt verranno notificati al gestore sbagliato. Questo di solito provocherà quantomeno il blocco del dispositivo, ma può a volte confondere il SO a tal punto da farlo diventare instabile oppure mandarlo in crash.

## 7 Come fa il computer a svolgere diverse cose contemporaneamente?

Non lo fa, in realtà. I computer possono svolgere soltanto un compito (o *processo*) alla volta. Ma un computer può cambiare compito molto rapidamente e indurre i lenti esseri umani a pensare che stia facendo diverse cose contemporaneamente. Questo viene chiamato *timesharing*.

Uno dei compiti del kernel è gestire il timesharing. Possiede una parte chiamata *scheduler* (pianificatore) che contiene informazioni relative a tutti gli altri processi (a parte il kernel) del vostro repertorio. Ogni sessantesimo di secondo nel kernel fa scattare un timer e viene generato un clock di interrupt. Lo scheduler ferma qualunque processo sia attualmente in esecuzione, lo sospende sul posto e passa il controllo a un altro processo.

Un sessantesimo di secondo può non sembrare una grande quantità di tempo. Ma per i microprocessori odierni è sufficiente per eseguire decine di migliaia di istruzioni macchina, che si possono tradurre in una gran mole di lavoro. Quindi anche se ci sono molti processi, ciascuno di essi può fare molte cose nella porzione di tempo a sua disposizione.

In pratica, non sempre un programma ottiene la sua intera porzione di tempo. Se scatta un interrupt da un dispositivo I/O, il kernel ferma effettivamente il compito corrente, esegue il gestore dell'interrupt e poi ritorna al compito corrente. Una tempesta di interrupt ad alta priorità può scombinare il normale funzionamento dei processi; questo fenomeno viene chiamato *thrashing* e per fortuna è molto difficile da provocare negli Unix moderni.

Infatti, la velocità dei programmi solo molto raramente è limitata dalla quantità di tempo macchina a loro disposizione (ci sono alcune eccezioni a questa regola, quali il suono o la generazione di grafica 3D). Molto più spesso dei ritardi si generano quando il programma deve attendere dei dati da un disco o da una connessione di rete.

Un sistema operativo che può di norma gestire più processi simultaneamente è detto "multitasking". La famiglia di sistemi operativi Unix è stata progettata fin dall'inizio per il multitasking e lo fa molto bene; in modo molto più efficace rispetto a Windows o al Mac OS ai quali il multitasking è stato appiccicato a posteriori in seguito a un ripensamento e lo fanno in modo piuttosto povero. Il multitasking efficiente e affidabile costituisce buona parte di ciò che rende Linux superiore per le applicazioni di rete, le comunicazioni e i servizi Web.

# 8 Come fa il computer a evitare che i processi si intralcino tra loro?

Lo scheduler del kernel si prende cura di dividere i processi nel tempo. Il proprio sistema operativo deve dividerli anche riguardo lo spazio, per evitare che non sconfinino oltre la porzione di memoria di lavoro loro assegnata. Anche se si assume che tutti i programmi provino ad essere cooperativi, non si vorrebbe che un bug in uno di loro sia capace di corrompere anche gli altri. Le operazioni compiute dal proprio sistema operativo per risolvere questo problema sono chiamate *gestione della memoria*.

Ogni processo del vostro repertorio ha la propria area di memoria, come luogo dal quale eseguire il proprio codice e dove immagazzinare le variabili e i risultati. Potete pensare a questo insieme come formato da un *segmento di codice*, di sola lettura (che contiene le istruzioni del processo), e da un *segmento dati* scrivibile (che contiene tutte le variabili immagazzinate dal processo). Il segmento dati è sempre unico per ogni processo, mentre nel caso due processi usino lo stesso codice Unix automaticamente fa in modo che condividano un unico segmento codice, come misura di efficienza.

## 8.1 Memoria virtuale: la versione semplice

L'efficienza è importante, perché la memoria è costosa. A volte non ce n'è abbastanza per contenere per intero tutti i programmi che il computer sta eseguendo, specialmente se si usa un grosso programma come un server X. Per ovviare a questo problema, Unix usa una tecnica chiamata *memoria virtuale*. Non cerca di tenere in memoria tutto il codice e i dati di un processo. Tiene piuttosto caricato solo un *working set* relativamente piccolo; il resto dello stato del processo viene lasciato in una speciale area di *spazio swap*sul proprio disco fisso.

Notare che in passato quel "A volte" dell'ultimo paragrafo era un "Quasi sempre", perchè la dimensione della memoria era tipicamente ridotta rispetto alla dimensione dei programmi in esecuzione, quindi il ricorso allo swap era frequente. Oggi la memoria è molto meno costosa e persino i computer di fascia bassa ne hanno molta. Sui moderni computer monoutente con 64MB di memoria e oltre è possibile eseguire X e un insieme tipico di programmi senza neppure ricorrere allo swap dopo che sono stati inizialmente caricati nel core.

## 8.2 Memoria virtuale: la versione dettagliata

In realtà, nella precedente sezione le cose sono state un po' semplificate. Certo, i programmi vedono la maggior parte della propria memoria come un unico grande banco opaco di indirizzi più grande della memoria fisica, e lo swap su disco è usato per mantenere questa illusione. Ma il proprio hardware in realtà contiene almeno cinque tipi differenti di memoria, e le differenze tra loro possono avere una grande importanza quando i programmi devono essere ottimizzati per la massima velocità. Per capire realmente quello che succede all'interno del proprio computer, si dovrebbe imparare come funzionano tutte queste memorie.

I cinque tipi di memoria sono questi: registri del processore, cache interna (o su chip), cache esterna (o fuori dal chip), memoria principale, e disco. Il motivo per cui ci sono così tanti tipi di memoria è semplice; la velocità costa denaro. Ho elencato questi tipi di memoria in ordine crescente rispetto al tempo di accesso e ordine decrescente di costo. La memoria del registro è la più

veloce e la più costosa e può effetuare circa un bilione di accessi casuali al secondo, mentre il disco è la memoria più lenta e di bassa qualità e può effettuare 100 accessi casuali al secondo.

Di seguito c'è un elenco completo che riflette le velocità per un computer desktop tipico precedente agli anni 2000. Mentre la velocità e la capacità aumenteranno e il prezzo diminuirà, ci si può aspettare che questi rapporti rimangano abbastanza costanti — e sono questi rapporti che formano la gerarchia della memoria.

**Disco** Dimensione: 13000MB Accesso: 100KB/sec

**Memoria principale** Dimensione: 256MB Accesso: 100M/sec

**Cache esterna** Dimensione: 512KB Accesso: 250M/sec

**Cache interna** Dimensione: 32KB Accesso: 500M/sec

**Processore** Dimensione: 28 bytes Accesso: 1000M/sec

Non possiamo produrre tutto con i tipi di memoria più veloci. Potrebbe essere un modo troppo dispendioso; e anche se si potesse, la memoria veloce è volatile. Ciò significa che abbandona i propri dati quando si spegne il computer. Così i computer devono avere i dischi rigidi o altri tipi di memoria non volatile che mantengono i dati quando si spegne l'alimentazione. E qui c'è un enorme disparità tra la velocità del processore e la velocità dei dischi. Nel mezzo della gerachia dei tre livelli di memoria (*cache interna*, *cache esterna*, e memoria principale) fondamentalmente esiste per superare questo dislivello.

Linux e gli altri Unix hanno una caratteristica chiamata *memoria virtuale*. Questo significa che il sistema operativo si comporta come se disponesse di molta più memoria principale ripetto a quella che ha realmente. La propria memoria fisica principale reale si comporta come un insieme impostato di finestre o cache su di uno spazio molto ampio di memoria "virtuale", la maggior parte della quale in qualsiasi momento è attualmente immagazzinata sul disco in una zona speciale chiamata l'*area di swap*. Non visibile dai programmi utente, il SO muove blocchi di dati (chiamati "pagine") tra la memoria e il disco per mantenere questa illusione. Il risultato finale è che la propria memoria virtuale è più grande ma non così più lenta rispetto alla memoria reale.

La maggiore o minore lentezza della memoria virtuale rispetto a quella fisica dipende da come gli algoritmi dello swapping del sistema operativo corrispondono al modo in cui i programmi usano la memoria virtuale. Fortunatamente, le letture e scritture in memoria che sono vicine nel tempo tendono anche a raggrupparsi nello spazio di memoria. Questa tendenza è chiamata *località*, o più formalmente *località di riferimento* — ed è una cosa buona. Se i riferimenti di memoria fossero allocati nello spazio virtuale in modo casuale, si dovrebbe effettuare una'operazione di lettura e scrittura sul disco per ogni nuovo riferimento e la memoria virtuale potrebbe essere lenta come una memoria su disco. Ma siccome i programmi hanno effettivamente una forte località, il sistema operativo può svolgere relativamente pochi swap per ogni riferimento.

È stato trovato per esperienza che il metodo più efficace per un'ampia classe di modelli per l'uso della memoria è molto semplice; è chiamato algoritmo LRU o "least recently used (usato meno di recente) ". Il sistema della memoria virtuale prende i blocchi del disco dentro i suoi *working set* secondo le sue necessità. Quando esso è eseguito fuori dalla memoria fisica per il working set, scarica il blocco usato meno di recente. Tutti gli Unix, e la maggior parte degli altri sistemi operativi aventi la memoria virtuale, usano variazioni minori sull'LRU.

La memoria virtuale è il primo anello di congiunzione tra la velocità del disco e quella del processore. Essa è gestita esplicitamente dal SO. Ma c'è ancora un divario considerevole tra la velocità della memoria fisica principale e la velocità a cui un processore può accedere ai suo registro di memoria. La cache esterna e quella interna si occupano di questo, usando una tecnica simile a quella della memoria virtuale come ho descritto precedentemente.

Così come la memoria fisica principale si comporta come un insieme di finestre o cache sull'area swap del disco, allo stesso modo la cache esterna agisce come delle finestre sulla memoria principale. La cache esterna è veloce (250M accessi per sec, invece di 100M) e più piccola. L'hardware (specificatamente, il controller della memoria del proprio computer) fa l'operazione LRU nella cache esterna su blocchi di dati scaricati dalla memoria principale. Per ragioni storiche, l'unità della cache swapping è chiamata *linea* invece che pagina.

Ma questo non è tutto. La cache interna dà l'incremento definitivo della velocità effettiva attraverso porzioni di memoria della cache esterna. Essa è ancora più rapida e piccola — infatti, risiede sul chip del processore.

Se si vogliono creare dei programmi realmente veloci, è utile conoscere questi dettagli. I propri programmi diventano più veloci quando hanno una forte località di referenza dato che questo permette un miglior funzionamento della cache. Tuttavia il modo più facile di rendere i programmi più veloci è quello di renderli più piccoli. Se un programma non è rallentato da una gran quantità di I/O del disco o aspetta degli eventi della rete, normalmente verrà eseguito alla velocità della cache più piccola nella quale il programma può risiedere.

Se non è possibile ridimensionare il programma, qualche sforzo per adattare le porzioni che dipendono criticamente dalla velocità in modo che abbiano una forte località può dare buoni risultati. I dettagli sulle tecniche per realizzare questo adattamento vanno oltre lo scopo di questo tutorial; nel momento in cui se ne avrà bisogno, si avrà preso abbastanza confidenza con qualche compilatore per scoprirne molti da soli.

### 8.3 Unità della gestione della memoria

Anche quando si dispone di una memoria fisica abbastanza grande da evitare lo swapping, la parte del sistema operativo chiamata *gestore della memoria* mantiene un importante ruolo da svolgere. Deve garantire che i programmi possano modificare soltanto il proprio segmento dati; deve cioè impedire che del codice difettoso o malizioso in un programma rovini i dati di altri programmi. A questo scopo tiene una tabella dei segmenti di dati e di codice. La tabella è aggiornata non appena un processo richiede più memoria oppure libera memoria (quest'ultimo caso si verifica di solito all'uscita dal programma).

Questa tabella è usata per passare comandi a una parte specializzata dell'hardware sottostante chiamata *MMU* o *unità di gestione della memoria*. I chip dei processori moderni hanno MMU incorporate. La MMU ha la capacità speciale di porre dei delimitatori attorno alle aree di memoria, in modo che un riferimento che sconfina venga rifiutato e faccia scattare uno speciale interrupt.

Se non si ha mai visto un messaggio di Unix del tipo "Segmentation fault", "core dumped" o qualcosa di simile, questo è esattamente quello che è successo; un tentativo da parte del programma in esecuzione di accedere alla memoria (core) al di fuori dal proprio segmento ha fatto scattare un interrupt fatale. Questo rivela un bug nel codice del programma; il *core dump* si trascina dietro costituisce una informazione diagnostica che ha lo scopo di aiutare il programmatore nell'individuazione del problema.

C'è un altro modo per proteggere i processi l'uno dall'altro, oltre alla limitazione della memoria a cui possono accedere. Si dovrebbe anche poter controllare il loro accesso ai file in modo che un programma difettoso o maligno non possa rovinare parti critiche del sistema. È per questo motivo che Unix ha i permessi sui file che vedremo in dettaglio in seguito.

## 9 Come fa il computer a immagazzinare le cose in memoria?

Probabilmente si saprà che ogni cosa in un computer viene immagazzinata come stringhe di bit (cifra binaria; si possono immaginare come molti piccoli interruttori on-off). Ora spiegeremo come questi bit vengono impiegati per rappresentare le lettere e i numeri che il computer manipola.

Prima di poter affrontare questo argomento, è necessario comprendere la *dimensione di parola* del proprio computer. Si tratta della dimensione preferita dal computer per spostare unità di informazioni; tecnicamente è l'ampiezza dei *registri* del processore, ovvero le aree che il processore utilizza per compiere calcoli logici e aritmetici. Quando leggiamo che i computer hanno dimensione in bit (per esempio "32-bit" o "64-bit") questo è ciò che significa.

La maggior parte dei computer ha una dimensione di parola di 64 bit. Nel passato recente (primi del 2000) molti pc avevano parole da 32 bit. Le vecchie macchine 286 nel 1980 avevano una dimensione di parola di 16. I calcolatori centrali vecchio stile spesso avevano parole di 36 bit.

Il computer vede la propria memoria come una sequenza di parole numerate da zero in avanti, fino a valori molto grandi dipendenti della dimensione della memoria. Tale valore è limitato dalla dimensione della parola, motivo per cui le vecchie macchine come i 286 dovevano svolgere complicati contorsionismi per indirizzare grandi quantità di memoria. Non li descriverò qui; procurano ancora degli incubi ai vecchi programmatori.

### 9.1 Numeri

I numeri interi sono rappresentati come parole o coppie di parole, a seconda della dimensione di parola del processore. Su macchine a 64 bit, la parola è la dimensione più comune.

L'aritmetica dei numeri interi è simile ma non è esattamente identica alla matematica in base due. Il bit di ordine più basso è 1, il successivo 2, poi 4 e così via come nella notazione binaria pura. Ma i numeri dotati di segno sono rappresentati in notazione *complemento a due*. Il bit di ordine più alto è un *bit di segno* che rende negativa la quantità rappresentata, mentre ogni numero negativo può essere ottenuto dal valore positivo corrispondente invertendo tutti i bit e aggiungendo uno. È per questo motivo che i numeri interi su una macchina a 64 bit devono essere compresi nell'intervallo tra $-2^{63}$ a $2^{63} - 1$. Quel 64-esimo bit è usato per il segno; 0 significa un numero positivo o zero, 1 un numero negativo.

Alcuni linguaggi di programmazione forniscono accesso a una *aritmetica senza segno* ovvero una artimetica in base 2 con solo lo zero e i numeri positivi.

La maggior parte dei processori e alcuni linguaggi possono eseguire operazioni con numeri a *virgola mobile* (funzionalità incorporata nel chip di tutti i processori recenti). I numeri a virgola mobile forniscono un intervallo più ampio degli interi e consentono di esprimere le frazioni. I modi in cui questo avviene sono diversi e un po' troppo complicati per essere affrontati in dettaglio qui, ma l'idea generale è molto simile alla cosiddetta 'notazione scientifica', dove si può scrivere (per esempio) 1.234 * $10^{23}$; la codifica del numero viene divisa in una *mantissa* (1.234) e in una parte esponenziale (23) che indica le potenze di dieci (significa che il numero moltiplicato dovrebbe avere 20 zeri su di esso, 23 meno i tre posti decimali).

### 9.2 Caratteri

I caratteri sono normalmente rappresentati come stringhe di sette bit, in una codifica chiamata ASCII (American Standard Code for Information Interchange). Sulle macchine moderne, ciascuno dei 128 caratteri ASCII è dato dai sette bit più bassi di un *ottetto* o byte a 8 bit; gli ottetti sono riuniti in parole di memoria in modo che (per esempio) una stringa di sei caratteri occupi solamente una parola di memoria a 64 bit. Per vedere una mappa dei caratteri ASCII, scrivere 'man 7 ascii' al prompt di Unix.

Il paragrafo precedente, però, non è completamente corretto, per due ragioni. Quella secondaria è che il termine 'ottetto' è formalmente corretto ma in realtà raramente usato; la maggior parte delle persone si riferisce a un ottetto come a un *byte* e ritiene che i byte siano lunghi otto bit. Per essere corretti, il termine 'byte' è più generale; per esempio, ci sono state macchine a 36 bit con byte di 9 bit (anche se probabilmente non capiterà più in futuro).

La ragione principale è, invece, che non tutto il mondo usa i codici ASCII. Di fatto, molti paesi non possono usarli. I codici ASCII, anche se funzionano bene per l'inglese americano, non contengono molte lettere accentate e altri caratteri speciali necessari per le altre lingue. Anche l'inglese britannico ha il problema della mancanza di un segno per la sterlina.

Ci sono stati diversi tentativi di risolvere questo problema. Tutti fanno uso dell'ottavo bit non usato dai codici ASCII, che in questo modo risultano la metà inferiore di un insieme di 256 caratteri. Quello più largamente utilizzato è l'insieme di caratteri 'Latin-1' (o più formalmente ISO 8859-1). Si tratta dell'insieme di caratteri predefinito per Linux, vecchie versioni di HTML, e X. Microsoft Windows usa una versione mutante di Latin-1 che aggiunge alcuni caratteri come le virgolette destre e sinistre, in posizioni lasciate libere da Latin-1 per ragioni storiche (per un serio resoconto dei problemi che ha provocato, vedere la pagina demoroniser).

Latin-1 gestisce le principali lingue europee, incluse l'inglese, francese, tedesco, spagnolo, italiano, olandese, norvegese, svedese, danese e islandese. Tuttavia, non è ancora sufficiente, per cui esiste una serie completa di insiemi di caratteri da Latin-2 a -9 per rappresentare il greco, l'arabo, l'ebraico, l'esperanto e il serbo-croato. Per maggiori dettagli vedere la pagina ISO alphabet soup.

La soluzione definitiva è uno standard enorme chiamato Unicode (e il suo gemello identico ISO/IEC 10646-1:1993). Unicode è identico a Latin-1 nella 256 posizioni più basse. Nello spazio successivo dei 16 bit comprende greco, cirillico, armeno, ebraico, arabo, devanagarico, bengalese, gurmukhi, gujarati, oriya, tamil, telugu, kannada, malese, tailandese, lao, georgiano, tibetano, giapponese kana, l'insieme completo del coreano hangul moderno e un insieme unificato di ideogrammi cinesi/giapponesi/coreani (CJK). Per maggiori dettagli vedere la Unicode Home Page. L'XML el'XHTML usano questo insieme di caratteri.

Versioni di Linux recenti usano una codifica di Unicode chiamata UTF-8. In UTF, i caratteri 0-127 sono ASCII. I caratteri 128-255 sono usati solo nelle sequenze da 2 a 4 byte che identificano caratteri non-ASCII.

## 10 Come fa il mio computer a immagazzinare le cose sul disco?

Quando si legge un disco fisso su Unix, si vede un albero di nomi di file e directory. Normalmente non sarà necessario vedere oltre, ma può essere utile avere maggiori dettagli se capita un crash del disco e si ha la necessità di provare a salvare dei file. Sfortunatamente, non c'è un buon modo per descrivere l'organizzazione del disco dal livello dei file in giù, quindi dovrò partire dall'hardware e risalire.

### 10.1 Struttura di basso livello del disco e del file system

La superficie del disco, dove vengono immagazzinati i dati, è divisa in qualcosa simile a un bersaglio per il tiro a freccette: in piste circolari che sono poi divise in settori. Dal momento che le piste vicino al bordo esterno hanno area maggiore di quelle vicino

al centro, le piste esterne hanno più settori rispetto a quelle interne. Ogni settore (o *blocco del disco*) ha la stessa dimensione, che sui moderni sistemi Unix è generalmente pari a 1 K binario (1024 byte da 8 bit). Ogni blocco del disco è individuato da un indirizzo univoco o *numero di blocco del disco*.

Unix divide il disco in *partizioni del disco*. Ogni partizione è formata da una serie continua di blocchi che vengono usati separatamente da quelli delle altre partizioni, come file system oppure come spazio di swap. In origine lo scopo delle partizioni aveva a che fare col ripristino dopo un crash quando i dischi erano più lenti e maggiormante inclini ad errori; le delimitazioni che le separano riducono la frazione del disco che probablmente diventerebbe inaccessibile o difettosa per un errore casuale sul disco. Oggigiorno, è più importante che le partizioni possano essere dichiarate di sola lettura (impedendo a un intruso di modificare file di sistema critici) o condivisi in una rete in diversi modi che qui non verranno trattati. La partizione con numero più basso viene spesso trattata in modo speciale, come una *partizione di avvio* dove si può mettere un kernel per essere avviato.

Ogni partizione può essere sia uno *spazio di swap* (usato per implementare la memoria virtuale) sia un *file system* usato per contenere i file. Le partizioni swap sono trattate proprio come una sequenza lineare di blocchi. I file system, invece, hanno bisogno di un modo per associare i nomi dei file alle sequenze di blocchi del disco. Dal momento che la dimensione dei file aumenta, diminuisce, si modifica nel tempo, i blocchi-dati di un file non saranno una sequenza lineare ma potranno essere disseminati su tutta la sua partizione (dipende da dove il sistema operativo riesce a trovare un blocco libero quando gliene serve uno). Questo effetto dispersivo è chiamato *frammentazione*.

## 10.2 Nomi dei file e directory

All'interno di ciascun file system la corrispondenza tra i nomi e i blocchi viene gestita attraverso una struttura chiamata *i-node*. C'è un gruppo di questi elementi vicino al "fondo" (i blocchi a numerazione più bassa) di ciascun file system (quelli più bassi in assoluto sono usati per scopi di manutenzione e di etichettatura, non saranno descritti qui). Ogni i-node individua un file. I blocchi-dati dei file (incluse le directory) si trovano sotto gli i-node (nei blocchi a numerazione più alta).

Ogni i-node contiene una lista dei numeri di blocco-disco relativi al file che individua. (Questa è una mezza verità, corretta solo per i file piccoli, ma il resto dei dettagli non è importante qui.) Notare che l'i-node *non* contiene il nome del file.

I nomi dei file si trovano nelle *strutture delle directory*. Una struttura della directory associa solo i nomi ai numeri i-node. Ecco perché, su Unix, un file può avere più nomi reali (o *collegamenti fisici*); sono soltanto voci di directory multiple che puntano allo stesso i-node.

## 10.3 Punti di Mount

Nel caso più semplice, tutto il proprio file system Unix si trova su di una sola partizione del disco. Anche se questa situazione si ritrova in qualche piccolo sistema Unix personale, è inusuale. Più generalmente il file system è distribuito tra pi` partizioni-disco, possibilmente su diversi dischi fisici. Così, per esempio, il proprio sistema può avere una piccola partizione dove risiede il kernel, una un po' più grande dove si trovano i programmi di utilità del SO e una molto più grande dove ci sono le directory personali degli utenti.

La sola partizione alla quale si avrà accesso subito dopo l'avvio del sistema è la propria *partizione di root*, che è (quasi sempre) quella dalla quale si è si è avviato il sistema. Essa contiene la "root directory" (directory radice) del file system, il nodo superiore dal quale dipende tutto il resto.

Le altre partizioni del sistema devono essere collegate a questa directory "root" affinché tutto il proprio file system multipartizione sia accessibile. Circa a metà del processo di avvio, il sistema Unix renderà accessibili queste partizioni non root. Esso dovrà *montare* ognuna di esse su di una directory della partizione root.

Per esempio, se si ha una directory Unix chiamata `/usr`, si tratta probabilmente di un punto di mount per una partizione che contiene molti programmi installati col proprio sistema Unix ma che non sono necessari durante l'avvio iniziale.

## 10.4 Come un file viene visto

Ora possiamo guardare al file system dall'alto al basso. Ecco cosa succede quando si apre un file (quale, ad esempio, `/home/esr/WWW/ldp/fundamentals.sgml`):

Il kernel si avvia alla radice del file system Unix (dalla partizione root). Cerca una directory chiamata 'home'. Di solito 'home' è un punto di mount per una partizione utente di grandi dimensioni che si trova da qualche altra parte, così andrà lì. Nella

struttura della directory di livello più alto di quella partizione utente cerca poi una voce chiamata ‘esr’ e ne estrae un numero di i-node. Va a quell’i-node, vede che si tratta di una struttura di directory e cerca ‘WWW’. Estraendo *quell*’i-node, va alla corrispondente sottodirectory e cerca ‘ldp’. Questo lo porta a un altro i-node di directory ancora. Aprendolo, trova il numero i-node di ‘fundamentals.xml’. Questo i-node non è una directory, ma contiene invece l’elenco dei blocchi-disco associati al file.

## 10.5 Possesso dei file, permessi e sicurezza

Per impedire ai programmi di intervenire accidentalmente o malignamente su dati su cui non dovrebbero intervenire, Unix ha la funzionalità dei *permessi*. Questi vennero originariamente pensati per il "timesharing" (suddivisione di tempo), proteggendo gli uni dagli altri utenti diversi sulla stessa macchina, quando ancora Unix veniva usato su costosi minicomputer condivisi.

Per comprendere i permessi sui file, occorre rivedere la descrizione di utenti e gruppi nella sezione Che cosa accade quando si fa il log in?. Ciascun file ha un utente proprietario e un gruppo proprietario. Inizialmente sono quelli del creatore del file; possono essere poi modificati con i programmi chown(1) e chgrp(1).

I permessi fondamentali che possono essere associati a un file sono ‘read’ (permesso di leggere i dati contenuti), ‘write’ (permesso di modificarli) ed ‘execute’ (permesso di eseguirli come programma). Ogni file ha tre impostazioni di permessi; uno per l’utente proprietario, uno per tutti gli utenti nel gruppo proprietario e uno per tutti gli altri. I ‘privilegi’ che si ottengono al momento del log in sono solo la possibilità di leggere, modificare ed eseguire quei file i cui bit dei permessi corrispondono con il proprio ID utente o quello di un gruppo a cui si appartiene, o i file che sono stati resi accessibili a tutti.

Per vedere come questi possono interagire e come le visualizza Unix, osserviamo alcuni elenchi di file su un sistema Unix ipotetico. Ecco un esempio:

```
 snark:~$ ls -l notes
-rw-r--r--   1 esr      users         2993 Jun 17 11:00 notes
```

Si tratta di un file di dati ordinario. Il listato ci informa che il proprietario è l’utente ‘esr’, creato con il gruppo proprietario ‘users’. Probabilmente la macchina su cui si trova mette per definizione tutti gli utenti ordinari in questo gruppo; altri gruppi che si vedranno comunemente su macchine con timesharing sono ‘staff’, ‘admin’, o ‘wheel’ (per ovvie ragioni, i gruppi non sono molto importanti su workstation a singolo utente o PC). Il sistema Unix in uso potrebbe usare un gruppo predefinito differente, magari derivato dal proprio ID utente.

La stringa ‘-rw-r--r--’ rappresenta i bit di permessi per il file. Il primo trattino è la posizione del bit directory; se il file fosse stato una directory il bit sarebbe stato ‘d’, o ‘l’ se il file fosse stato un collegamento simbolico. Dopo di questo, le prime tre posizioni successive sono i permessi utente, le seconde tre i permessi del gruppo e le terze tre i permessi per gli altri (spesso chiamate autorizzazioni ‘world’). Su questo file l’utente proprietario ‘esr’ può leggere e modificare il file, gli altri appartenenti al gruppo ‘users’ possono leggerlo e così tutti gli altri utenti. Si tratta di un insieme di permessi piuttosto tipici per un file di dati ordinario.

Ora osserviamo un file con permessi molto diversi. Tale file è GCC, il compilatore C GNU.

```
snark:~$ ls -l /usr/bin/gcc
-rwxr-xr-x   3 root     bin         64796 Mar 21 16:41 /usr/bin/gcc
```

Questo file appartiene a un utente chiamato ‘root’ e a un gruppo chiamato ‘bin’; può essere modificato solo da root, ma letto ed eseguito da tutti. Si tratta di un proprietario e di un insieme di permessi tipici per un comando di sistema pre-installato. Il gruppo ‘bin’ esiste su alcuni sistemi Unix per raggruppare i comandi di sistema (il nome è una reliquia storica, abbreviazione di ‘binary’). Il sistema Unix in uso potrebbe usare invece un gruppo ‘root’ (non esattamente la stessa cosa dell’utente ‘root’!).

L’utente ‘root’ è il nome convenzionale per utente con ID uguale a 0, un account speciale privilegiato che può scavalcare tutti i privilegi. L’accesso come root è utile ma pericoloso; un errore di battitura quando si è collegati come root potrebbe rovinare file critici del sistema, cosa che non può avvenire con un account di utente ordinario.

Poiché l’account root è così potente, l’accesso a questo account dovrebbe essere sorvegliato attentamente. La propria password di root è la parte più critica nelle informazioni di sicurezza del proprio sistema, ed essa è ciò che ogni "cracker" e ogni intruso che verranno dopo cercheranno di ottenere.

Per quanto riguarda le password: non la si scriva su qualche parte; e non si scelgano password che possano essere facilmente indovinate, come il nome della propria ragazza/sposa o del proprio ragazzo/sposo. Questa è una cattiva pratica sorprendentemente comune che aiuta continuamente i "cracker". In generale, non si scelga alcuna parola nel dizionario; esistono dei programmi chiamati *dictionary crackers* che cercano le possibili password scorrendo liste di parole di uso comune. Una tecnica valida è

quella di scegliere una combinazione consistente di una parola, un numero, e un'altra parola, come ad esempio: 'shark6cider' o 'jump3joy'; che renderà il campo di ricerca troppo grande per un "dictionary cracker". Non si usino questi esempi, però; i "cracker" se lo potrebbero aspettare dopo la lettura di questo testo e mettere questi esempi nei loro dizionari.

Osserviamo ora un terzo caso:

```
snark:~$ ls -ld ~
drwxr-xr-x  89 esr      users          9216 Jun 27 11:29 /home2/esr
snark:~$
```

Questo file è una directory (notare la 'd' in prima posizione nella stringa dei permessi). Vediamo che può essere modificata solo da esr, ma letta ed eseguita da tutti gli altri.

Il permesso in lettura fornisce la possibilità di listare la directory, cioè di vedere i nomi dei file e delle directory che essa contiene. Il permesso in scrittura dà la possibilità di creare e cancellare file nella directory. Se si ricorda che una directory include una lista dei nomi di file e delle sottodirectory che contiene, questa regola ha una sua logica

Il permesso di esecuzione su di una directory significa che si può esaminare la directory per aprire i file e le directory sotto di essa. Di fatto, dà il permesso di accedere agli i-node della directory. Una directory con il permesso di esecuzione completamente mancante potrebbe essere inusabile

Occasionalmente si vedrà una directory che è eseguibile da tutti ma non leggibile da tutti; questo significa che un utente qualunque può accedere ai file e alle directory al suo interno, ma solamente se ne conosce il nome esatto (la directory non può essere listata).

È importante ricordare che i permessi di lettura, scrittura o esecuzione su una directory sono indipendenti dai permessi sui file e le directory al suo interno. In particolare, l'accesso in scrittura su una directory significa che si possono creare nuovi file o cancellare file esistenti al suo interno, ma non dà automaticamente l'accesso in scrittura ai file esistenti.

Infine, osserviamo i permessi dello stesso programma login.

```
snark:~$ ls -l /bin/login
-rwsr-xr-x   1 root     bin         20164 Apr 17 12:57 /bin/login
```

Possiede i permessi che ci aspetteremmo per un comando di sistema; eccetto per la 's' dove dovrebbe esserci il bit del permesso di esecuzione del proprietario. Questa è la manifestazione visibile di un tipo speciale di permesso chiamato 'set-user-id' o *setuid bit*.

Il bit setuid è normalmente legato a programmi che necessitano di dare agli utenti ordinari i privilegi di root, ma in un modo controllato. Quando è impostato su un programma eseguibile, si acquistano i privilegi del proprietario di quel file di programma finché si esegue quel programma, sia che corrispondano a quelli dell'utente oppure no.

Come l'account root stesso, i programmi setuid sono utili ma pericolosi. Chiunque sia in grado di sovvertire o modificare un programma setuid che ha root come proprietario, può usarlo per accedere alla shell con privilegi di root. Per questa ragione, sulla maggior parte dei sistemi Unix, aprendo un file in scrittura il suo bit setuid viene disattivato automaticamente. Molti attacchi alla sicurezza su Unix tentano di scoprire bug nei programmi setuid, con lo scopo di sovvertirli. Gli amministratori di sistema attenti alla sicurezza sono quindi molto prudenti con questi programmi e riluttanti a installarne di nuovi.

Ci sono un paio di dettagli importanti che abbiamo sorvolato durante la precedente spiegazione dei permessi; in particolare, come vengono assegnati il gruppo proprietario e i permessi quando viene creato un file o una directory per la prima volta. Il gruppo è un problema poiché gli utenti possono essere membri di più gruppi, ma uno di essi (specificato nella voce dell'utente in `/etc/passwd`) è il *gruppo predefinito* dell'utente e normalmente sarà proprietario dei file creati dall'utente.

La storia con i bit iniziali dei permessi è un po' più complicata. Un programma che crea un file normalmente specifica i permessi coi quali dovrà iniziare. Ma questi verranno modificati da una variabile nell'ambiente dell'utente chiamata *umask*. Umask speciica quali bit dei permessi *disattivare* quando si crea un file; il valore più comune, quello predefinito sulla maggior parte dei sistemi, è -------w- o 002, che disattiva il bit di scrittura per tutti gli utenti. Vedere la documentazione per il comando umask nella pagina di manuale della shell per maggiori dettagli.

Anche il gruppo iniziale di directory è un po' complicato. Su alcuni sistemi Unix una nuova directory ottiene il gruppo predefinito dell'utente che l'ha creata (questo nella convenzione del System V); su altri sistemi ottiene il gruppo proprietario della directory genitrice in cui essa viene creata (questa è la convenzione di BSD). Su alcuni Unix moderni, incluso Linux, quest'ultimo comportamento può essere selezionato impostando il set-group-ID sulla directory (chmod g+s).

### 10.6 Come le cose possono andare male

Precedentemente è stato accennato che i file system possono essere delicati. Ora sappiamo che per raggiungere un file dobbiamo fare il gioco della campana attraverso quella che può essere una catena arbitrariamente lunga di directory e riferimenti i-node. Supponiamo ora che sul disco fisso si formi un punto danneggiato.

Se si è fortunati ciò verrà perso solo qualche file di dati. Se invece si è sfortunati, si potrebbe danneggiare una struttura di directory o un numero i-node e un intero sottoalbero del sistema potrebbe rimanere sospeso nel limbo. Oppure, peggio ancora, si potrebbe originare una struttura danneggiata che punta in più modi allo stesso blocco disco o i-node. Un danneggiamento di questo tipo si può propagare a partire da una normale operazione sui file, facendo perdere tutti i dati collegati al punto danneggiato di origine.

Fortunatamente, questo tipo di eventualità è divenuto abbastanza infrequente perché l'hardware dei dischi è più affidabile. Tuttavia, questo significa che il sistema Unix cercherà di controllare periodicamente l'integrità del file system per assicurarsi che non ci sia nulla fuori posto. I sistemi Unix moderni compiono un rapido controllo dell'integrità di ciascuna partizione nella fase di avvio, appena prima di montarle. Dopo un certo numero di riavvii fanno un controllo molto più approfondito che impiega qualche minuto in più.

Se tutto questo può far sembrare che Unix sia terribilmente complesso e incline a malfunzionamenti, può essere rassicurante sapere che questi controlli nella fase d'avvio tipicamente intercettano e correggono i problemi normali *prima* che diventino veramente disastrosi. Altri sistemi operativi non hanno questi strumenti, cosa che velocizza un po' l'avvio ma può mettere molto di più nei pasticci quando si cerca di fare un salvataggio manuale (e sempre assumendo che si abbia una copia delle Norton Utilities o simili, tanto per cominciare...).

Una delle tendenze attuali nella progettazione dei sistemi Unix sono i *file system con journaling*. Questi gestiscono il traffico al disco in modo da garantirgli di rimanere in uno stato coerente che può ripristinato quando il sistema ritorna a funzionare. Questo renderà molto più veloce la verifica dell'integrità all'avvio.

## 11 Come funzionano i linguaggi del computer?

Abbiamo già visto come vengono eseguiti i programmi. Ogni programma in definitiva deve eseguire un flusso di byte che sono istruzioni nel *linguaggio macchina* del computer. Ma gli esseri umani non si destreggiano molto bene con il linguaggio macchina; riuscirci è divenuta un'arte rara, una magia nera persino tra gli hacker.

Quasi tutto il codice Unix, ad eccezione di una piccola porzione relativa all'interfaccia diretta con l'hardware nel kernel stesso, viene oggi scritto in un *linguaggio ad alto livello* . (`Alto livello' in questa espressione è un residuo storico volto a distinguerlo dai *linguaggi assembler* di `basso livello', che sono fondamentalmente sottili involucri attorno al codice macchina.)

Ci sono diversi tipi di linguaggi di alto livello. Per affrontare l'argomento è utile tener presente che il *codice sorgente* di un programma (la versione creata dall'uomo, modificabile) deve passare attraverso un qualche tipo di traduzione in codice macchina che il computer può effettivamente eseguire.

### 11.1 Linguaggi compilati

Il tipo di linguaggio più convenzionale è il *linguaggio compilato*. I linguaggi compilati vengono tradotti in file eseguibili di codice macchina binario da uno speciale programma chiamato (ovviamente) un *compilatore*. Una volta che il codice binario è stato generato lo si può eseguire direttamente senza più guardare al codice sorgente. (La maggior parte del software viene distribuito come binari compilati a partire da codice che non si vede.)

I linguaggi compilati tendono a dare prestazioni eccellenti e hanno il più completo accesso al SO, ma tendono anche a essere difficili da programmare.

C, il linguaggio in cui Unix stesso è scritto, è di gran lunga il più importante tra questi (con la sua variante C++). FORTRAN è un altro linguaggio ancora usato tra gli ingegneri e gli scienziati ma più vecchio di diversi anni e molto più primitivo. Nel mondo Unix nessun altro linguaggio compilato è nell'uso dominante. Al di fuori di esso, il COBOL è molto usato per il software finanziario e commerciale.

Un tempo c'erano molti altri linguaggi compilati, ma la maggior parte di essi si sono estinti oppure sono strumenti strettamente di ricerca. Se si è un nuovo sviluppatore di Unix e si usa un linguaggio compilato è estremamente probabile che questo sia il C o il C++.

### 11.2 Linguaggi interpretati

Un *linguaggio interpretato* dipende da un programma interprete che legge il codice sorgente e lo traduce al volo in calcoli e chiamate di sistema. Il sorgente deve essere reinterpretato (e l'interprete deve essere presente) ogni volta che il codice viene eseguito.

I linguaggi interpretati tendono a essere più lenti dei linguaggi compilati e spesso hanno accesso limitato al sistema operativo e all'hardware sottostanti. Per contro, essi tendono a essere più facili da programmare e più indulgenti verso gli errori di codifica rispetto ai linguaggi compilati.

Molti programmi di utilità di Unix, inclusa la shell, bc(1), sed(1) e awk(1), sono in effetti piccoli linguaggi interpretati. I BASIC sono di solito interpretati. Così pure il Tcl. Storicamente, il più importante linguaggio interpretato è stato il LISP (un grande miglioramento in gran parte dei suoi successori). Oggi le shell Unix e il Lisp che vivono all'interno dell'editor Emacs sono probabilmente i linguaggi interprepati puri più importanti.

### 11.3 Linguaggi a codice P

Dal 1990 è andato assumendo importanza crescente un tipo di linguaggio ibridio che usa sia la compilazione che l'interpretazione. I linguaggi a codice P sono come i linguaggi compilati nel senso che il sorgente viene tradotto in una forma binaria compatta che è ciò che viene realmente eseguito, ma che non è esattamente codice macchina. Si tratta invece di *pseudocodice* (o *codice p*), che è solitamente molto più semplice ma più potente di un vero linguaggio macchina. Quando si esegue il programma, si interpreta il codice p.

Il codice P può girare velocemente quasi quanto un binario compilato (gli interpreti di codice P possono essere abbastanza semplici, leggeri e rapidi). Ma i linguaggi a codice P riescono a mantenere la flessibilità e la potenza di un buon interprete.

Importanti linguaggi a codice P includono Python e Java.

## 12 Come funziona internet?

Per aiutare a capire come funziona internet daremo uno sguardo alle cose che succedono quando si fa una tipica operazione su Internet: indirizzare un browser alla prima pagina di questo documento, sul sito web del Linux Documentation Project. Questo documento è

```
http://www.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/index.html
```

questo significa che si trova nel file HOWTO/Unix-and-Internet-Fundamentals-HOWTO/index.html sotto la directory di esportazione World Wide Web dell'host www.tldp.org.

### 12.1 Nomi e posizioni

La prima cosa che il browser deve fare è stabilire una connessione remota al computer dove si trova il documento. A tal fine deve prima trovare la posizione remota dell'*host* www.tldp.org (`host' è la forma breve di `computer host' o `host remoto'; www.tldp.org è un tipico *hostname*). La posizione corrispondente è in realtà un numero chiamato *indirizzo IP* (spiegheremo più avanti la parte `IP' di questa espressione).

A questo scopo il proprio browser interroga un programma chiamato *name server*. Il "name server" può trovarsi sul proprio computer, ma è più probabile che giri su un computer del fornitore col quale il proprio computer dialoga. Quando si sottoscrive un contratto con un ISP (fornitore di servizi internet) una parte della procedura di configurazione consiste quasi sicuramente nel comunicare al proprio software per internet qual ` l'indirizzo IP di un name server sulla rete dell'ISP.

I name server sui vari computer comunicano tra loro, scambiandosi e tenendo aggiornate tutte le informazioni necessarie per risolvere i nomi degli host (per tradurli in indirizzi IP). Il proprio name server può interrogare tre o quattro diversi siti sulla rete nel processo di risoluzione di www.tldp.org, ma di solito questo si verifica molto rapidamente (diciamo in meno di un secondo). Vedremo nella sezione successiva come funzionano i name server nel dettaglio

Il name server comunicherà al proprio browser che l'indirizzo IP di www.tldp.org è 152.19.254.81; conoscendo questo, la propria macchina sarà in grado di scambiare direttamente bit con www.tldp.org.

## 12.2 The Domain Name System

L'intera rete di programmi e database che cooperano alla traduzione degli hostname in indirizzi IP è chiamata 'DNS'(Domain Name System). Quando si vedono riferimenti a un 'DNS server', questo significa solo che viene chiamato un nameserver. Ora spiegherò come funziona l'intero sistema.

Gli hostname della rete sono composti di parti separate da punti. Un *dominio* è una collezione di macchine che condividono un suffisso di nome comune. I domini possono esistere all'interno di altri domini. Per esempio, la macchina www.tldp.org si trova nel sottodominio .tldp.org del dominio .org.

Ogni dominio è definito da un *name server autorevole* che conosce gli indirizzi IP delle altre macchine nel dominio. Il name server autorevole (o 'primario') può avere backup nel caso in cui andasse giù; se si vedono i riferimenti a un *name server secondario* o ('DNS secondario') si sta parlando di uno di questi. Tipicamente questi secondari aggiornano le loro informazioni dai loro primari ogni poche ore, così un cambiamento fatto all'hostname tradotto in indirizzo IP sul primario sarà propagato automaticamente.

Ora c'è una parte importante. I nameserver per un dominio *non* hanno la conoscenza delle posizioni di tutte le macchine negli altri domini (includendo i loro propri sottodomini); hanno solamente la conoscenza della posizione dei nameserver. Nel nostro esempio, il name server autorevole per il dominio .org conosce l'indirizzo IP del nameserver per .tldp.org ma *non* gli indirizzi di tutte le altre macchine in .tldp.org.

I domini nel sistema DNS sono strutturati come un grande albero invertito. In alto ci sono i server di root. Ognuno conosce gli indirizzi IP dei server di root; sono collegati dentro al proprio software DNS. I server di root conoscono gli indirizzi IP dei nameserver per i domini di livello più alto come .com e .org, ma non gli indirizzi delle macchine all'interno di questi domini. Ogni server di dominio di alto livello conosce dove si trovano i nameserver per i domini direttamente al di sotto, e così via.

Il DNS è sviluppato con cura così che ogni macchina può andare avanti con la minima quantità di conoscenza di cui ha bisogno in relazione alla forma dell'albero, e le modifiche locali ai sottoalberi possono essere fatte semplicemente cambiando nella base dati del server autorevole le corrispondenze tra nome e indirizzo IP .

Quando si fa una richiesta per l'indirizzo IP di www.tldp.org, quel che realmente accade è questo: Prima, il proprio nameserver chiede a un server root di dirgli dove si può trovare un nameserver per .org. Una volta ottenuta la risposta, chiede poi al server .org di dirgli l'indirizzo IP di un nameserver .tldp.org. Una volta ottenuta la risposta, chiede al nameserver .tldp.org di dirgli l'indirizzo dell'host www.tldp.org.

La maggior parte delle volte, il proprio nameserver non deve realmente lavorare così tanto. I nameserver fanno molto uso della memoria cache; quando risolve un nome di host, tiene in memori aper qualche tempo l'associazione con l'indirizzo IP risultante. Questo è il motivo per cui, quando naviga in un nuovo sito, normalmente si vedrà nel browser un messaggio di ricerca dell'host solo per la prima pagina visitata. Alla fine l'associazione nome-indirizzo scade e il DNS dev'essere chiesto di nuovo. Questo è importante affinché la ricerca di informazioni non valide non debba durare in eterno quando un nome di host cambia indirizzo. L'indirizzo IP di un sito tenuto in memoria viene rifiutato anche quando l'host è irraggiungibile.

## 12.3 Pacchetti e instradatori

Quello che il browser vuol fare è inviare un comando al server web su www.tldp.org come il seguente:

```
GET /LDP/HOWTO/Fundamentals.html HTTP/1.0
```

Ecco cosa succede. Dal comando si costruisce un *pacchetto*, cioè un blocco di bit come un telegramma che è `impacchettato' con tre cose importanti: *indirizzo di provenienza* (l'indirizzo IP del proprio computer), l'*indirizzo di destinazione* (152.19.254.81), e un *numero di servizio* o *numero di porta* (in questo caso 80) che indica che si tratta di una richiesta World Wide Web.

Il proprio computer spedisce allora il pacchetto lungo il cavo (la connessione al proprio ISP o rete locale) finché arriva a un apparato specializzato chiamato *router*. Il router ha nella sua memoria una mappa di Internet; non sempre è una mappa completa, ma una che descrive completamente il proprio vicinato di rete e sa come raggiungere i router per altri circondari di Internet.

Il pacchetto potrebbe passare attraverso svariati router lungo il percorso per la sua destinazione. I router sono intelligenti. Guardano quanto tempo impiegano gli altri router per sapere se hanno ricevuto un pacchetto. Usano questa informazione anche per dirigere il traffico verso i collegamenti veloci. La usano per accorgersi se un altro router (o un cavo) sono fuori servizio o irraggiungibili e quindi, se possibile, ovviare al problema trovando un'altra strada.

C'è una leggenda metropolitana secondo la quale Internet è stata progettata per sopravvivere alla guerra nucleare. Questo non è vero, ma la struttura di Internet è estremamente adatta a ottenere prestazioni affidabili da hardware precarioin un mondo incerto. Questo deriva direttamente dal fatto che la sua intelligenza è distribuita attraverso migliaia di router piuttosto che concentrata in poche centrali enormi e vulnerabili (come la rete telefonica). Questo significa che i malfunzionamenti tendono ad essere ben localizzati e la rete può aggirarli.

Una volta che il pacchetto è giunto al computer di destinazione quest'ultimo usa il numero di servizio per inviare il pacchetto al server Web. Il server web può capire a chi rispondere guardando l'indirizzo IP di provenienza del pacchetto con il comando. Quando il server web restituisce questo documento lo suddividerà in un certo numero di pacchetti. La dimensione dei pacchetti varierà a seconda del mezzo di trasmissione sulla rete e del tipo di servizio.

## 12.4 TCP e IP

Per capire come vengono gestite le trasmissioni a pacchetti multipli, è necessario sapere che Internet in realtà usa due protocolli, uno sovrapposto all'altro.

Il livello più basso, l'*IP* (Internet Protocol), ha la responsabilità di etichettare singoli pacchetti con un indirizzo di provenienza e un indirizzo di destinazione di due computer che si scambiano informazioni su una rete. Per esempio, quando si accede a http://www.tldp.org, i pacchetti che si inviano avranno l'indirizzo IP del proprio computer, come 192.168.1.101, e l'indirizzo IP del computer www.tldp.org, 152.2.210.81. Questi indirizzi funzionano allo stesso modo del proprio indirizzo di casa quando qualcuno invia una lettera. L'ufficio postale può leggere gli indirizzi e determinare dove si abita e qual'è il miglior percorso perché la lettera arrivi, molto simile a quello che svolge un router per il traffico di internet.

Il livello superiore, *TCP* (Transmission Control Protocol), fornisce affidabilità. Questi due computer negoziano una connessione TCP (cosa che fanno usando l'IP), il ricevente sa che deve spedire al mittente un avviso di ricevimento dei pacchetti che legge. Se il mittente non vede un avviso di ricevimento per un pacchetto entro un certo periodo di tempo (timeout), rispedisce quel pacchetto. Inoltre, il mittente fornisce ogni pacchetto TCP di un numero di sequenza, che il ricevente può usare per riassemblare i pacchetti nel caso che risultino in disordine. (Questo può facilmente succedere se i collegamenti della rete vanno e vengono durante una connessione.)

I pacchetti TCP/IP contengono anche una somma di controllo (checksum) per consentire l'individuazione di dati rovinati da collegamenti difettosi (la somma di controllo è calcolata dal resto del pacchetto in modo tale che se il resto del pacchetto o la somma di controllo sono difettosi, il ricalcolo e il successivo confronto indichi con molta probabilità un errore). Così, dal punto di vista di chiunque usi il TCP/IP e i name server, sembra essere un modo affidabile per passare flussi di byte tra coppie di hostname/numero di servizio. Chi scrive i protocolli di rete non deve quasi mai pensare a tutte le operazioni di pacchettizzazione, riassemblaggio dei pacchetti, controllo degli errori, calcolo della somma di controllo e ritrasmissione che stanno al di sotto di quel livello.

## 12.5 HTTP, un protocollo applicativo

Torniamo ora al nostro esempio. I browser e i server web dialogano usando un *protocollo applicativo* che viene eseguito al disopra del TCP/IP, usandolo semplicemente come un modo per passare stringhe di byte avanti e indietro. Questo protocollo è chiamato *HTTP* (Hyper-Text Trasfer Protocol, protocollo per il trasferimento di ipertesti) e abbiamo è già visto un suo comando: il GET mostrato prima.

Quando il comando GET arriva al server www.tldp.org con numero di servizio 80, verrà notificato a un *server daemon* che è in attesa sulla porta 80. La maggior parte dei servizi Internet sono implementati da demoni server che si limitano ad ascoltare sulle porte, attendono ed eseguono i comandi in arrivo.

Se la struttura di Internet ha una regola generale, questa è che tutte le parti dovrebbero essere il più possibile semplici e accessibili per gli esseri umani. L'HTTP, e i suoi simili (come il Simple Mail Transfer Protocol, *SMTP*, che viene usato per trasferire la posta elettronica tra gli host) tende a usare comandi in semplice testo stampabile che terminano con un codice di carriage return/line feed.

Questo è leggermente inefficiente; in qualche circostanza si potrebbe ottenere una velocità maggiore usando un protocollo binario di stretta codifica. Ma l'esperienza ha dimostrato che i vantaggi di avere comandi facili da descrivere e comprendere per gli esseri umani supera qualsiasi guadagno marginale di efficienza che si possa ottenere al prezzo di rendere le cose oscure e complicate.

Di conseguenza, quello che il demone server rispedisce attraverso TCP/IP è anch'esso testo. L'inizio della risposta assomiglierà in qualche modo a questa (alcune intestazioni sono state omesse):

```
HTTP/1.1 200 OK
Date: Sat, 10 Oct 1998 18:43:35 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Thu, 27 Aug 1998 17:55:15 GMT
Content-Length: 2982
Content-Type: text/html
```

Queste intestazioni saranno seguite da una riga vuota e dal testo della pagina web (dopodich´ la connessione viene lasciata cadere). Il proprio browser si limita a visualizzare quella pagina. Le intestazioni servono a comunicargli come (in particolare, l'intestazione Content-Type gli comunica che i dati restituiti sono veramente HTML).

## 13 Per saperne di più

È disponibile un documento Reading List HOWTO che elenca i libri che si possono leggere per approfondire gli argomenti che sono stati affrontati in questo testo. Un altro documento di cui si consiglia la lettura è How To Become A Hacker.

# 14 Index