The Linux GCC HOWTO

Daniel Barlow (dan@detached.demon.co.uk)

v1.17, 28 Febbraio 1996

Questo documento tratta la configurazione del compilatore C GNU e delle librerie di sviluppo sotto Linux, e fornisce una panoramica sulla compilazione, il link, l'esecuzione e la correzione (debug) dei programmi. La maggior parte del materiale contenuto nel documento è stato preso dal GCC-FAQ di Mitch D'Souza, che sostituisce, o dall' *ELF-HOWTO* (per la traduzione in italiano vedi *[1]*). Quella che state leggendo è la prima versione rilasciata pubblicamente (nonostante il numero). Ogni feedback è benvenuto. Revisione e manutenzione della traduzione italiana: *Andrea Girotto (andrea.girotto@usa.net)*

Indice

1 Preliminari

1.1 Confronto tra ELF e a.out

Lo sviluppo di Linux si trova attualmente in uno stato di continua evoluzione. Brevemente, esistono due formati di file binari che Linux è in grado di eseguire, ed a seconda di come è stato costruito, un sistema potrebbe usare uno o l'altro dei due. Leggendo questo HOWTO si scoprirà quale.

Per riconoscere il tipo di un binario è possibile utilizzare l'utility `file` (ad esempio: `file /bin/bash`). Per un programma ELF, darà una risposta contenente `ELF`; per un programma `a.out` la risposta sarà qualcosa del tipo `Linux/i386`.

Le differenze tra ELF e a.out sono descritte ampiamente in questo documento. ELF è il formato più recente, generalmente ritenuto il migliore.

1.2 Questioni amministrative

Informazioni riguardo al copyright si trovano alla fine di questo documento, insieme a dovute avvertenze relative a certe stupide domande poste su Usenet, che, segnalando problemi inesistenti, rivelano un'ignoranza del linguaggio C.

1.3 Tipografia

La versione in formato Postscipt, dvi, o html, di questo documento ha una maggiore varietà tipografica rispetto a quella in solo testo. In particolare, i nomi di file, i comandi, l'output dei comandi e gli stralci di codice sorgente sono impostati con un carattere tipo `macchina da scrivere`, come pure sono state messe in evidenza le "variabili" ed altre parti che dovevano essere **enfatizzate**.

Inoltre, è presente un utile indice. In dvi o postscript, la numerazione dell'indice corrisponde a quella dei paragrafi. In HTML si tratta di numeri assegnati sequenzialmente che rimandano ad altre parti del testo. Nella versione a solo testo, si tratta solo di numeri. Si consiglia una versione avanzata piuttosto che quella in modalità testo.

Negli esempi viene utilizzata la sintassi dell'interprete dei comandi (shell) Bourne (al posto di quella C). Gli utenti di C-shell potranno utilizzare il comando:

```
% setenv FOO bar
```

al posto di:

```
$ FOO=bar; export FOO
```

Se il prompt mostrato è `#` invece di `$`, il comando indicato probabilmente funzionerà solo se digitato come **root**. Naturalmente, non si assumere alcuna responsabilità per qualsiasi cosa accada al sistema nell'utilizzo di questi esempi.

2 Dove ottenere quello che serve.

2.1 Questo documento.

Questo documento fa parte della serie dei Linux HOWTO, pertanto si può ottenere da tutti gli archivi di Linux HOWTO, come *http://sunsite.unc.edu/pub/linux/docs/HOWTO/* (le traduzioni italiane sono disponibili presso *[2]*). La versione HTML in inglese può anche essere trovata (probabilmente leggermente più aggiornata) su *http://ftp.linux.org.uk/~barlow/howto/gcc-howto.html*

(traduzione italiana *[3]*).

2.2 Ulteriore documentazione.

La documentazione ufficiale di gcc si trova nella distribuzione sorgente (si veda sotto) sotto forma di file texinfo, e come file `.info`. Se si dispone di una connessione di rete veloce, un cd-rom o una buona dose di pazienza, è possibile eseguirne l'`untar` e copiare le parti rilevanti in `/usr/info`. In caso contrario, possono essere trovati presso *ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/* , ma non necessariamente si tratterà della versione più recente.

Esistono due sorgenti di documentazione per libc. La GNU libc contiene dei file info che descrivono piuttosto accuratamente la libc Linux, fatta eccezione per stdio. Inoltre, le pagine di manuale dell'archivio *ftp://sunsite.unc.edu/pub/Linux/docs/* sono state scritte per Linux e descrivono numerose chiamate di sistema (sezione 2) e funzioni libc (sezione 3).

2.3 GCC

La distribuzione ufficiale del GCC Linux può sempre essere trovata in formato binario (già compilato) all'indirizzo *ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/* . Nel momento in cui viene scritto questo documento, la versione più recente è la *2.7.2* (`gcc-2.7.2.bin.tar.gz`).

È possibile ottenere la distribuzione sorgente più recente di GCC fornita dalla Free Software Foundation dagli archivi *ftp://prep.ai.mit.edu/pub/gnu/* . I gestori del GCC Linux hanno reso molto semplice la compilazione dell'ultima versione - il programma di configurazione (`configure`) dovrebbe impostare tutto da solo. Si verifichi anche l'eventuale presenza di patch da applicare presso: *ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/* .

Per compilare qualcosa di non banale (ma anche alcune cose banali) sarà necessario possedere anche quanto descritto nel paragrafo che segue.

2.4 Libreria C e header file

Quello che si desidera a questo punto dipende da due fattori:

1. se il proprio sistema è ELF oppure a.out

2. quale dei due si desidera avere.

Se si sta passando da libc 4 a libc 5, si raccomanda di leggere l' *ELF-HOWTO* , rintracciabile più o meno nello stesso luogo in cui è stato trovato questo documento (traduzione italiana *[1]*).

Da *ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/*

sono disponibili:

libc-5.2.18.bin.tar.gz

Immagini di librerie condivise ELF, librerie statiche e file include per le librerie C e matematiche.

libc-5.2.18.tar.gz

Sorgenti per quanto descritto sopra. Sarà necessario anche il pacchetto `.bin.` per gli header file. Se si è indecisi tra compilarsi in proprio la libreria C o utilizzare il formato binario, la scelta migliore consiste nell'usare il codice già compilato. Nel caso in cui si desideri il supporto NIS o per le shadow password di dovrà comunque gestirli in prima persona.

libc-4.7.5.bin.tar.gz

Immagini di libreria condivisa `a.out` e librerie statiche per la versione 4.7.5 della libreria C e simili. Questo è stato studiato per coesistere con il pacchetto libc 5 sopra menzionato, ma è necessario solo se si desidera continuare a utilizzare o sviluppare programmi in formato a.out.

2.5 Strumenti associati (as, ld, ar, strings, ecc.)

Si possono trovare su *ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/* , come ogni altra cosa sinora descritta. La versione corrente è `binutils-2.6.0.2.bin.tar.gz`.

Le binutils sono disponibili unicamente in ELF, la versione corrente di libc si trova in ELF ed inoltre è meglio utilizzare la versione a.out di libc in congiunzione con una ELF. Lo sviluppo della libreria C si sta muovendo verso ELF: se non c'è un particolare motivo per l'utilizzo di a.out, si consiglia di fare altrettanto.

3 Installazione e impostazione di GCC

3.1 Versioni GCC

È possibile scoprire quale versione GCC si sta eseguendo digitando `gcc -v` alla richiesta dell'interprete comandi. Si tratta di un metodo piuttosto affidabile anche per scoprire se la propria impostazione è ELF o a.out. Il risultato potrebbe essere:

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
```

Le cose da notare sono:

i486

Indica che il gcc è stato compilato per un processore 486 - altre possibilità sono 386 o 586. Tutti i processori possono eseguire il codice compilato per ciascuna delle altre versioni; la differenza consiste nell'ottimizzazione che non ha effetti sulle prestazioni in un 386, ma rende il codice eseguibile leggermente più grande.

box

Non è affatto importante, potrebbe esserci qualcosa di diverso (come `slackware` o `debian`) o anche nulla (in tal caso, il nome di directory completa sarà `i486-linux`). Se si esegue personalmente la compilazione di gcc, è possibile impostare questa caratteristica al momento della compilazione, a fini puramente estetici.

linux

In alternativa, potrebbe essere `linuxelf` o `linuxaout`, il significato varia a seconda della versione utilizzata, fatto che potrebbe confondere le idee.

linux

significa ELF se la versione è la 2.7.0 o seguente, altrimenti significa a.out.

linuxaout

significa a.out. È stato introdotto quando la definizione di `linux` è stata modificata da a.out in ELF, in modo che non sia possibile vedere un gcc `linuxaout` più vecchio della versione 2.7.0.

linuxelf

è obsoleto. Si tratta generalmente di una versione di gcc 2.6.3 impostata per produrre eseguibili ELF. Si noti che il gcc 2.6.3 contiene degli errori nella produzione di codice per ELF - si consiglia un aggiornamento.

2.7.2

numero della versione.

In conclusione, si tratta di gcc 2.7.2 che produce del codice ELF.

3.2 Dov'è finito?

Se gcc è stato installato senza prestare attenzione, oppure se è stato ottenuto come parte di una distribuzione, potrebbe essere necessario scoprire dove si trova all'interno del filesystem. Le parti chiave sono:

- `/usr/lib/gcc-lib/`**destinazione**`/`**versione**`/` (e sottodirectory) posto in cui si trova la maggior parte del compilatore, i programmi eseguibili che effettuano la compilazione, alcune librerie e file include specifici per la versione che si sta utilizzando.
- `/usr/bin/gcc` driver del compilatore - la parte che si esegue dalla riga di comando. Può essere utilizzato con diverse versioni di gcc, a patto che siano state installate più directory di compilatore (come descritto sopra). Per scoprire la versione predefinita, digitare `gcc -v`. Per forzare un'altra versione, digitare `gcc -V` **versione**. Ad esempio:

```
# gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
# gcc -V 2.6.3 -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.6.3/specs
gcc driver version 2.7.2 executing gcc version 2.6.3
```

- `/usr/`**destinazione**`/(bin|lib|include)/`. Se sono stati installati più destinazioni (target) (ad esempio, a.out ed elf, o un cross compilatore di qualche tipo), le librerie, le binutils (come `as`, `ld` o altri) e gli header file per destinazioni non native possono essere trovati in questa posizione. Se è stato installato un solo tipo di gcc, diverse parti di esso sono poste o in questa directory o, alternativamente, in `/usr/(bin|lib|include)`.

- /lib/, /usr/lib e altre sono directory di libreria per il sistema nativo. Sarà anche necessaria la directory /lib/cpp per molte applicazioni (ad esempio X ne fa un grande utilizzo) - è possibile copiarla da /usr/lib/gcc-lib/destinazione/versione/ o creare un collegamento (link) simbolico.

3.3 Dove si trovano gli header file?

Indipendentemente dalla posizione in cui sono stati installati i propri, sotto /usr/local/include, esistono tre sorgenti principali degli header file in Linux:

- la maggior parte di /usr/include/ e relative subdirectory sono sostituiti con il pacchetto binario di libc da H. J. Lu. In tali posizioni si potrebbero anche trovare file da altri sorgenti (librerie **curses** e **dbm**, ad esempio) specialmente se si sta utilizzando la distribuzione libc più recente (che non contiene curses o dbm, come invece accadeva nelle più vecchie).
- /usr/include/linux e /usr/include/asm (per i file <linux/*.h> e <asm/*.h>) dovrebbero essere link simbolici alle directory linux/include/linux e linux/include/asm nella distribuzione sorgente del kernel. È necessario installarli se si pensa di eseguire lo sviluppo di un qualsiasi programma non banale; non servono solo per la compilazione del kernel. Potrebbe essere anche necessario eseguire il make config nelle directory del kernel dopo aver scaricato i sorgenti. Molti file dipendono da <linux/autoconf.h> che potrebbe non esistere, e in alcune versioni di kernel, asm è un link simbolico che viene creato al momento del make config. Pertanto, se si scaricano i sorgenti del kernel sotto /usr/src/linux:

```
$ cd /usr/src/linux
$ su
# make config
[rispondere alle domande. A meno che non si abbia intenzione di
 proseguire con la compilazione del kernel, la risposta non ha
 molta importanza]
# cd /usr/include
# ln -s ../src/linux/include/linux .
# ln -s ../src/linux/include/asm .
```

- File come <float.h>, <limits.h>, <varargs.h>, <stdarg.h> e <stddef.h> variano a seconda della versione del compilatore, pertanto si trovano in /usr/lib/gcc-lib/i486-box-linux/2.7.2/include/ e posizioni simili.

3.4 Compilazione di cross compilatori

3.4.1 Linux come piattaforma di destinazione

Supponendo di aver ottenuto il codice sorgente per gcc, solitamente è sufficiente seguire le istruzioni contenute nel file INSTALL di GCC. Un comando configure -target=i486-linux -host=XXX sulla piattaforma XXX seguito da un make dovrebbe funzionare. Si noti che saranno necessari gli include di Linux, gli include del kernel e sarà necessario eseguire anche la compilazione del cross assembler e del cross linker dai sorgenti in *ftp://tsx-11.mit.edu/pub/linux/packages/GCC/* .

Linux come piattaforma sorgente, MSDOS come destinazione Apparentemente dovrebbe essere possibile utilizzando il pacchetto emx o l'extender go. Si invita a dare un'occhiata a *ftp://sunsite.unc.edu/pub/Linux/devel/msdos* .

L'autore, non ha ancora verificato le funzionalità e pertanto non può dare alcuna garanzia in merito.

4 Il porting e la compilazione

4.1 Simboli definiti automaticamente

È possibile elencare i simboli definiti automaticamente dalla propria versione di gcc eseguendolo con l'opzione `-v`. Ad esempio:

```
$ echo 'main(){printf("hello world\n");}' | gcc -E -v -
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
/usr/lib/gcc-lib/i486-box-linux/2.7.2/cpp -lang-c -v -undef
-D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux
-D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386
-D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386)
-Amachine(i386) -D__i486__ -
```

Se si sta scrivendo del codice che utilizza delle caratteristiche specifiche per Linux, è una buona idea includere le parti non portabili in

```
#ifdef __linux__
/* ... altro codice ... */
#endif /* linux */
```

Si utilizzi `__linux__` per questo scopo, **non** semplicemente `linux`. Sebbene anche il secondo sia definito, non è conforme a POSIX.

4.2 Chiamata del compilatore

La documentazione per le opzioni del compilatore è rappresentata dalla info page di gcc (in Emacs, si utilizzi `C-h i` quindi si selezioni la voce 'gcc'). Il proprio distributore potrebbe non aver incluso questa parte nel sistema, o la versione che si possiede potrebbe essere vecchia; la cosa migliore da fare in questo caso consiste nello scaricare l'archivio sorgente gcc da *ftp://prep.ai.mit.edu/pub/gnu* o da uno dei suoi siti mirror.

La pagina di manuale gcc (`gcc.1`) di solito è obsoleta. Tentando di leggerla si troverà questo avvertimento.

4.2.1 Opzioni del compilatore

L'output di gcc può essere ottimizzato aggiungendo `-On` alla sua riga di comando, dove **n** rappresenta un intero opzionale. I valori significativi di **n**, ed il loro esatto effetto, variano a seconda della versione; tipicamente vanno da 0 (nessuna ottimizzazione) a 2 (molte ottimizzazioni) a 3 (moltissime ottimizzazioni).

Internamente, gcc le traduce in una serie di opzioni `-f` e `-m`. È possibile vedere esattamente quale livello `-O` si lega a ogni opzione eseguendo `gcc -v -Q` (`Q` è un'opzione non documentata). Ad esempio, -O2 sul sistema dell'autore produce questo risultato:

```
enabled:        -fdefer-pop -fcse-follow-jumps -fcse-skip-blocks
            -fexpensive-optimizations
            -fthread-jumps -fpeephole -fforce-mem -ffunction-cse -finline
            -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop
            -fcommon -fgnu-linker -m80387 -mhard-float -mno-soft-float
            -mno-386 -m486 -mieee-fp -mfp-ret-in-387
```

L'utilizzo di un livello di ottimizzazione maggiore di quanto previsto per il proprio compilatore (ad esempio, -O6) avrà lo stesso risultato che utilizzare il livello più alto che è in grado di supportare. Tuttavia, la distribuzione di codice impostato per la compilazione in questo modo non è una buona idea - se in versioni future saranno incorporate ulteriori ottimizzazioni, potrebbe accadere che interrompano il proprio codice.

Gli utenti di gcc dalla versione 2.7.0 fino alla 2.7.2 dovrebbero notare che esiste un errore in -O2. In particolare, '**-fstrenght-reduction**' non funziona. È disponibile una patch per risolvere questo problema, che necessita della ricompilazione del gcc, altrimenti è sufficiente assicurarsi di usare sempre l'opzione `-fno-strength-reduce`.

Opzioni specifiche per il processore Esistono altre opzioni `-m` che non sono attivate da nessun `-O`, e si dimostrano utili in molti casi. Le principali sono `-m386` e `-m486`, che indicano a gcc di favorire rispettivamente 386 o 486. Il codice compilato con una di queste due opzioni funzionerà anche su macchine dell'altro tipo; il codice 486 è più voluminoso, ma non è più lento se eseguito su 386.

Attualmente non esiste un'opzione `-mpentium` o `-m586`. Linus suggerisce di utilizzare `-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2`, per ottenere l'ottimizzazione del codice 486 ma senza i salti necessari per l'allineamento (di cui il pentium non ha bisogno). Michael Meissner (di Cygnus) dice:

> La mia impressione è che `-mno-strength-reduce` produca anche un codice più veloce sull'x86 (si noti che non mi sto riferendo all'errore relativo all'opzione '-fstrength-reduction': altra questione). Ciò è dovuto alla carenza di registri dell'x86 (ed il metodo di GCC di raggruppare i registri in registri sparsi piuttosto che in altri registri non aiuta molto). '`strenght-reduce`' ha come effetto l'utilizzo di registri addizionali per sostituire moltiplicazioni con addizioni. Sospetto anche che `-fcaller-saves` possa causare una perdita di prestazioni.
>
> Un'altra considerazione consiste nel fatto che `-fomit-frame-pointer` possa o meno rappresentare un vantaggio. Da un lato, rende disponibile un altro registro per l'uso; tuttavia, il modo in cui x86 codifica il suo set di istruzioni comporta che gli indirizzi relativi di stack occupino più spazio rispetto agli indirizzi relativi di frame; di conseguenza, sarà disponibile ai programmi una quantità (leggermente) inferiore di Icache. Inoltre, `-fomit-frame-pointer` implica il continuo aggiustamento del puntatore allo stack da parte del compilatore, dopo ogni chiamata, quando, con un frame, può lasciare che lo stack esegua un accumulo per alcune chiamate.

L'ultima parola su questo argomento viene ancora da Linus:

> Se si desidera ottenere prestazioni ottimali, non credete alle mie parole, effettuate delle prove. Esistono molte opzioni nel compilatore gcc, e può darsi che un insieme particolare dia l'ottimizzazione migliore per la propria impostazione.

4.2.2 Internal compiler error: cc1 got fatal signal 11

(Ovvero: Errore interno del compilatore: cc1 ha ricevuto il segnale fatale 11).

Il segnale 11 è SIGSEGV, o 'segmentation violation'. Solitamente significa che il programma ha confuso i puntatori e ha tentato di scrivere su una porzione di memoria che non possedeva. Potrebbe trattarsi di un errore di gcc.

Tuttavia, gcc è ben verificato ed affidabile, nella maggior parte dei casi. Utilizza anche un gran numero di strutture dati complesse, e una grande quantità di puntatori. In breve, è il miglior tester di RAM tra quelli disponibili comunemente. Se non è possibile replicare l'errore - il sistema non si ferma nello stesso punto quando si riavvia la compilazione - molto probabilmente si tratta di un problema legato all'hardware (CPU, memoria, scheda madre o cache). Non lo si deve considerare un errore del compilatore solo perché il proprio computer supera i controlli di avvio del sistema o è in grado di eseguire Windows o qualunque altro programma; questi 'test' sono comunemente ritenuti, a ragione, di nessun valore. Comunque è sbagliato ritenere che si tratti di un errore, perché una compilazione del kernel si blocca sempre durante '`make zImage`' - è logico che ciò accada: '`make zImage`' probabilmente compila più di 200 file. In genere si ricerca un bug in un insieme più **piccolo** di così.

Se è possibile duplicare l'errore, e (ancora meglio) produrre un breve programma che lo dimostra, è possibile inviarlo come report di errori all'FSF, o alla mailing list di `linux-gcc`. Si faccia riferimento alla documentazione di gcc per i dettagli relativi alle informazioni effettivamente necessarie.

4.3 Portabilità

È stato detto che, in questi giorni, se non può essere portato a Linux, allora è qualcosa che non vale la pena di possedere. :-)

In generale, sono necessarie solo piccole modifiche per ottenere la conformità POSIX al 100% di Linux. Sarebbe anche molto utile restituire agli autori ogni modifica al codice in modo che, in futuro, si possa ottenere un eseguibile funzionante tramite il solo comando '`make`'.

4.3.1 BSD (inclusi bsd_ioctl, demone e <sgtty.h>)

È possibile compilare il proprio programma con `-I/usr/include/bsd` ed eseguire il link con `-lbsd` (ossia, aggiungere `-I/usr/include/bsd` a `CFLAGS` e `-lbsd` alla riga `LDFLAGS` nel proprio `Makefile`). Non è più necessario aggiungere `-D__USE_BSD_SIGNAL` se si desidera un comportamento dei segnali di tipo BSD, dal momento che questa caratteristica viene ottenuta automaticamente quando si ha `-I/usr/include/bsd` e l'include `<signal.h>`.

4.3.2 Segnali 'mancanti' (`SIGBUS`, `SIGEMT`, `SIGIOT`, `SIGTRAP`, `SIGSYS` ecc)

Linux è conforme a POSIX. Quelli elencati nel seguito sono segnali non definiti in POSIX - ISO/IEC 9945-1:1990 (IEEE Std 1003.1-1990), paragrafo B.3.3.1.1:

> I segnali SIGBUS, SIGEMT, SIGIOT, SIGTRAP, e SIGSYS sono stati omessi da POSIX.1 perché il loro comportamento dipende dall'implementazione e non è stato possibile classificarlo adeguatamente. Implementazioni conformi potranno contenere questi segnali, ma devono documentare le circostanze in cui sono rilasciati ed elencare ogni restrizione riguardante il loro rilascio.

Il modo più economico e scadente di gestire la cosa consiste nel ridefinire questi segnali in `SIGUNUSED`. Il modo **corretto** consiste nell'inserire il codice che li gestisce in appropriati `#ifdef`

```
#ifdef SIGSYS
/* ... codice SIGSYS non-posix .... */
#endif
```

4.3.3 Codice K & R

GCC è un compilatore ANSI; una grande quantità di codice esistente non è ANSI. Non c'è molto da fare per risolvere questo problema, oltre all'aggiungere `-traditional` alle opzioni del compilatore. Si invita a consultare l'info page di gcc.

Si noti che `-traditional` ha altri effetti oltre a cambiare il linguaggio accettato da gcc. Ad esempio, attiva `-fwritable-strings`, che sposta le stringhe costanti nello spazio dati (dallo spazio di codice, dove non possono essere scritte). Questo aumenta l'occupazione di memoria del programma.

4.3.4 Conflitto dei simboli di preprocessore con prototipi nel codice

Uno dei problemi più frequenti consiste nel fatto che alcune funzioni comuni sono definite come macro negli header file di Linux e il preprocessore si rifiuterà di eseguire il parsing di definizioni prototipo simili. I più comuni sono `atoi()` e `atol()`.

4.3.5 `sprintf()`

Soprattutto quando si esegue il porting da SunOS, è necessario essere consapevoli del fatto che `sprintf(string, fmt, ...)` restituisce un puntatore a stringhe, mentre Linux (seguendo ANSI) restituisce il numero di caratteri che sono stati messi nella stringa.

4.3.6 `fcntl` e affini. Quali sono le definizioni di `FD_*`?

Le definizioni si trovano in `<sys/time.h>`. Se si sta utilizzando `fcntl`, probabilmente si vorrà includere anche `<unistd.h>`.

In genere, la pagina di manuale di una funzione elenca gli `#include` necessarie nella sua sezione SYNOPSIS.

4.3.7 Il timeout di `select()`. Programmi in busy-waiting

Una volta, il parametro timeout di `select()` era utilizzato a sola lettura. Già allora, le pagine di manuale avvertivano:

> select() dovrebbe probabilmente restituire il tempo rimanente dal timeout originale, se esistente, modificando il valore del tempo. Questo potrà essere implementato in versioni future del sistema. Pertanto, sarebbe sbagliato ritenere che il puntatore al timeout non sarà modificato dalla chiamata select().

Ora il futuro è arrivato. Di ritorno da una `select()`, l'argomento di timeout sarà impostato al tempo rimanente che si sarebbe atteso se i dati non fossero arrivati. Se non è arrivato alcun dato, il valore sarà zero, e le chiamate future utilizzando la stessa struttura timeout eseguiranno immediatamente il return.

Per rimediare, in caso di errore, si metta il valore di timeout nella struttura ogni volta che si chiama `select()`. Si modifichi il codice come:

```
struct timeval timeout;
timeout.tv_sec = 1; timeout.tv_usec = 0;
while (some_condition)
        select(n,readfds,writefds,exceptfds,&timeout);
```

in

```
struct timeval timeout;
while (some_condition) {
        timeout.tv_sec = 1; timeout.tv_usec = 0;
        select(n,readfds,writefds,exceptfds,&timeout);
}
```

Alcune versioni di Mosaic evidenziavano questo problema. La velocità dell'animazione del globo rotante era inversamente correlata alla velocità con cui i dati giungevano dalla rete!

4.3.8 Chiamate di sistema interrotte

Sintomo Quando un programma viene interrotto utilizzando `Ctrl-Z` e poi viene riavviato - o in altre situazioni che generano dei segnali: interruzione con `Ctrl-C`, terminazione di un processo figlio ecc. - si ottengono messaggi del tipo interrupted system call o write: unknown error.

Problema I sistemi POSIX controllano la presenza di segnali più frequentemente rispetto a sistemi UNIX più vecchi. Linux può eseguire dei gestori di segnali:

- in modo asincrono (tramite un timer)
- al ritorno da ogni chiamata di sistema
- durante l'esecuzione delle seguenti chiamate di sistema: `select()`, `pause()`, `connect()`, `accept()`, `read()` su terminali, su socket, su pipe o su file in `/proc`, `write()` su terminali, su socket, su pipe o sulla line printer; `open()` su FIFO, su PTY o su linee seriali, `ioctl()` su terminali; `fcntl()` con il comando `F_SETLKW`; `wait4()`, `syslog()`, ogni operazione TCP o NFS.

Per altri sistemi operativi potrebbe essere necessario includere in questa lista le chiamate di sistema `creat()`, `close()`, `getmsg()`, `putmsg()`, `msgrcv()`, `msgsnd()`, `recv()`, `send()`, `wait()`, `waitpid()`, `wait3()`, `tcdrain()`, `sigpause()`, `semop()`.

Se un segnale (per il quale il programma ha installato un gestore) avviene durante una chiamata di sistema, viene chiamato il gestore. Quando il gestore restituisce il controllo (alla chiamata di sistema), essa rileva che è stata interrotta e restituisce immediatamente -1 e `errno = EINTR`. Il programma non si aspetta che questo accada, pertanto si blocca.

È possibile scegliere tra due alternative, per rimediare.

- Per ogni gestore di segnali che viene installato, aggiungere `SA_RESTART` ai flag `sigaction`. Per esempio, modificare

```
signal (sig_nr, my_signal_handler);
```

in

```
signal (sig_nr, my_signal_handler);
{ struct sigaction sa;
  sigaction (sig_nr, (struct sigaction *)0, &sa);
#ifdef SA_RESTART
  sa.sa_flags |= SA_RESTART;
#endif
#ifdef SA_INTERRUPT
  sa.sa_flags &= ~ SA_INTERRUPT;
#endif
  sigaction (sig_nr, &sa, (struct sigaction *)0);
}
```

Si noti che mentre questo viene applicato alla maggior parte delle chiamate di sistema, è necessario controllare EINTR personalmente riguardo a `read()`, `write()`, `ioctl()`, `select()`, `pause()` e `connect()` . Si veda sotto.

- Controllare EINTR esplicitamente.

Seguono due esempi per `read()` e `ioctl()`.

Una parte di codice originale utilizzante `read()`

```
int result;
while (len > 0) {
        result = read(fd,buffer,len);
        if (result < 0) break;
        buffer += result; len -= result;
}
```

diventa

```
int result;
while (len > 0) {
        result = read(fd,buffer,len);
        if (result < 0) { if (errno != EINTR) break; }
        else { buffer += result; len -= result; }
}
```

e una parte di codice utilizzante `ioctl()`

```
    int result;
    result = ioctl(fd,cmd,addr);
```

diventa

```
    int result;
    do { result = ioctl(fd,cmd,addr); }
    while ((result == -1) && (errno == EINTR));
```

Si noti che in alcune versioni di Unix BSD il comportamento predefinito consiste nel riavviare le chiamate di sistema. Per ottenere l'interruzione delle chiamate di sistema è necessario utilizzare i flag SV_INTERRUPT o SA_INTERRUPT.

4.3.9 Stringhe scrivibili (il programma genera 'segmentation fault' in modo casuale)

GCC ha una visione ottimistica dei suoi utenti, considerando le costanti stringa esattamente quello che sono - delle costanti. Pertanto, le memorizza nell'area del codice, dove possono essere inserite ed estratte dall'immagine di disco del programma (invece di occupare uno swapspace). Ne consegue che ogni tentativo di riscriverle causerà 'segmentation fault'.

Questo può causare dei problemi a vecchi programmi che, per esempio eseguono una chiamata `mktemp()` con una stringa costante come argomento. `mktemp()` tenta di riscrivere il suo argomento.

Per correggere,

- compilare con l'opzione `-fwritable-strings`, per fare in modo che gcc posizioni le costanti nello spazio dati, oppure
- riscrivere le parti che causano dei problemi per allocare una stringa non costante ed eseguire la `strcpy` dei dati in essa prima della chiamata.

4.3.10 Perché la chiamata `execl()` fallisce?

Probabilmente accade perché viene eseguita in modo errato. Il primo argomento per `execl` è il nome del programma da eseguire. Il secondo e i successivi diventano l'array `argv` del programma che si sta chiamando. Ricordare che: `argv[0]` viene impostato anche per un programma eseguito senza argomenti. Pertanto si dovrebbe scrivere

```
execl("/bin/ls","ls",NULL);
```

e non solo

```
execl("/bin/ls", NULL);
```

L'esecuzione del programma senza nessun argomento è interpretata come un invito a stampare le sue dipendenze a librerie dinamiche, almeno utilizzando `a.out`. ELF si comporta diversamente.

(Se si desidera questa informazione di libreria, esistono interfacce più semplici; si veda il paragrafo relativo al caricamento dinamico, o la pagina di manuale per `ldd`).

5 Debugging e Profiling

5.1 Manutenzione preventiva (lint)

Non esiste un `lint` ampiamente utilizzato per Linux, dal momento che la maggior parte della gente si accontenta degli avvertimenti che gcc può generare. Probabilmente, quello più utile è il `-Wall` opzione che significa 'Warnings, all' ma probabilmente ha un maggior valore mnemonico, se pensato come qualcosa contro cui si sbatte la testa (NdT. wall in inglese significa muro).

Esiste un lint di dominio pubblico disponibile su *ftp://larch.lcs.mit.edu/pub/Larch/lclint* . Putroppo non sono in grado di giudicare la sua validità.

5.2 Debugging

5.2.1 Come si ottengono le informazioni di debug in un programma?

È necessario compilare ed eseguire il link di tutte le sue parti con l'opzione `-g`, e senza `-fomit-frame-pointer`. Non è necessario ricompilarlo interamente, solo le parti di cui si esegue il debug.

Nelle configurazioni a.out le librerie condivise sono compilate con `-fomit-frame-pointer`, con la quale gdb non funzionerà. Questo perché l'opzione `-g` implica un link statico.

Se il linker va in errore fornendo un messaggio che indica l'impossibilità di trovare `libg.a`, significa che non si possiede `/usr/lib/libg.a`, che consiste nella speciale libreria C abilitata al debugging. Tale libreria può essere fornita nel pacchetto binario libc, oppure (in versioni di libreria C più recenti) può essere necessario ottenere il codice sorgente libc ed eseguire personalmente la compilazione. Tuttavia, è possibile ottenere sufficienti informazioni per la maggior parte degli scopi semplicemente sostituendola con un collegamento simbolico con `/usr/lib/libc.a`.

Come eliminarle? Molto software GNU è impostato affinché la compilazione e il link siano eseguite con l'opzione `-g`, che determina la generazione di eseguibili molto voluminosi (e spesso statici). Questa non è una buona idea.

Se il programma possiede uno script di configurazione '`autoconf`', è possibile disabilitare le informazioni di debugging controllando il `Makefile`. Naturalmente, se si sta utilizzando ELF, il link del programma viene eseguito in modo dinamico indipendentemente dall'impostazione `-g`, pertanto si può semplicemente usare `strip`.

5.2.2 Software disponibile

Molte persone utilizzano gdb, che può essere ottenuto in forma sorgente dai siti di archivio GNU *ftp://prep.ai.mit.edu/pub/gnu* , oppure in formato binario da *tsx-11 (ftp://tsx-11.mit.edu/pub/linux/packages/GCC)* o da sunsite. `xxgdb` è un debugger X basato su gdb (ossia, è necessario che sia installato gdb). È possibile trovare i sorgenti presso *ftp://ftp.x.org/contrib/xxgdb-1.08.tar.gz* .

Rick Sladkey ha eseguito il porting del debugger UPS. Può essere eseguito anche sotto X, ma a differenza di `xxgdb`, non è un semplice front end X per un debugger basato su testo. Possiede diverse caratteristiche molto interessanti, e se si ha la necessità di eseguire diversi debug, è il caso di provarlo. La versione precompilata per Linux e i patch per i sorgenti UPS possono essere trovati in *ftp://sunsite.unc.edu/pub/Linux/devel/debuggers/* , mentre i sorgenti originali in *ftp://ftp.x.org/contrib/ups-2.45.2.tar.Z* .

Un altro strumento utile per il debug è '`strace`', che visualizza le chiamate di sistema fatte da un processo. Questo strumento possiede anche molti altri utilizzi, inclusa la possibilità di sapere i nomi di file compilati, di cui non si possiede il sorgente; di esasperare i race condition in programmi che si sospettano contenerle, e in generale di imparare come funzionano le cose. La versione più recente di `strace` (attualmente la 3.0.8) può essere trovata in *ftp://ftp.std.com/pub/jrs/* .

5.2.3 Programmi background (demone)

I programmi demone tipicamente eseguono presto la `fork()`, e terminano il programma chiamante. Questo rende la sessione di debug molto breve.

Il modo più semplice per aggirare questo ostacolo consiste nell'impostare un breakpoint per `fork`, e quando il programma si blocca, forzare la restituzione di 0.

```
(gdb) list
1               #include <stdio.h>
2
3               main()
4               {
5               if(fork()==0) printf("child\n");
6               else printf("parent\n");
7               }
(gdb) break fork
Breakpoint 1 at 0x80003b8
(gdb) run
Starting program: /home/dan/src/hello/./fork
Breakpoint 1 at 0x400177c4

Breakpoint 1, 0x400177c4 in fork ()
(gdb) return 0
Make selected stack frame return now? (y or n) y
```

```
#0  0x80004a8 in main ()
at fork.c:5
5               if(fork()==0) printf("child\n");
(gdb) next
Step singolo fino all'uscita dalla funzione fork,
che non contiene informazioni sul numero di riga.
child
7               }
```

5.2.4 Core file

Quando Linux viene avviato, solitamente è configurato per non produrre core file. Se si desidera, è possibile utilizzare il comando interno dell'interprete comandi per riabilitarli: per *shell* compatibili C (come `tcsh`) corrisponde al comando

```
% limit core unlimited
```

mentre per interpreti comandi di tipo Bourne (`sh`, `bash`, `zsh`, `pdksh`) utilizzare

```
$ ulimit -c unlimited
```

Se si desidera una maggiore flessibilità nell'assegnazione dei nomi ai core file (nel caso, per esempio, di analisi post-mortem con un debugger che contiene errori) è possibile eseguire una piccola modifica al proprio kernel. Cercare in `fs/binfmt_aout.c` e `fs/binfmt_elf.c` il codice tipo:

```
  memcpy(corefile,"core.",5);
#if 0
  memcpy(corefile+5,current->comm,sizeof(current->comm));
#else
  corefile[4] = '\0';
#endif
```

e modificare gli `0` in `1`.

5.3 Profiling

Il profiling rappresenta un modo per esaminare le parti di un programma richiamate più frequentemente o eseguite più a lungo. È buona norma ottimizzare il codice e vedere dove viene perso del tempo. È necessario compilare tutti i file oggetto sui quali si desiderano informazioni sul tempo di esecuzione con l'opzione `-p`, e per interpretare il file di output sarà necessario anche `gprof` (dal pacchetto binutils). Si veda la pagina di manuale `gprof` per ulteriori dettagli.

6 Linking

Questo paragrafo è piuttosto complicato a causa dei due formati binari incompatibili, la distinzione tra libreria statica e condivisa, e del sovraccarico del verbo 'link' che significa sia 'cosa accade dopo la compilazione', sia 'cosa accade quando viene richiamato un programma compilato' (e, di fatto, il sovraccarico del verbo 'load' in un senso simile ma opposto).

Per ridurre in qualche modo la confusione, si userà il termine 'caricamento dinamico' (dynamic loading) per quello che accade durante l'esecuzione, argomento descritto nel prossimo paragrafo. Potrebbe anche accadere di trovare il termine 'collegamento dinamico' (dynamic linking) con lo stesso significato, ma non in questo documento. Questo paragrafo, quindi, tratta esclusivamente il tipo di link che avviene alla fine di una compilazione.

6.1 Librerie condivise e statiche

L'ultima fase della compilazione di un programma consiste nel 'collegamento' (link), ossia nell'unire tutte le parti e vedere se manca qualcosa. Ovviamente esistono alcune cose che molti programmi hanno in comune - ad esempio, aprire file, ed il codice in grado di fare queste cose sono fornite sotto forma di librerie. Nella maggior parte dei sistemi Linux si trovano in `/lib` e `/usr/lib/`.

Quando si utilizza una libreria statica, il linker trova le parti necessarie ai moduli di programma e le copia fisicamente nel file di output eseguibile che viene generato. Al contrario, questo non avviene per le librerie condivise - in questo caso nell'output viene inserita una nota del tipo 'quando viene eseguito questo programma, è necessario caricare questa libreria'. Ovviamente, le librerie condivise tendono a creare eseguibili di dimensioni minori; inoltre utilizzano una quantità inferiore di memoria e viene utilizzato meno spazio su disco. Il comportamento predefinito di Linux consiste nell'eseguire il collegamento di librerie condivise se esistono, altrimenti vengono utilizzate quelle statiche. Se si ottengono dei binari statici quando, al contrario, si vogliono quelli condivisi, controllare che i file di libreria condivisa (`*.sa` per a.out, `*.so` per ELF) si trovino dove dovrebbero essere e siano leggibili.

Su Linux, le librerie statiche hanno nomi come `libname.a`, mentre le librerie condivise sono denominate `libname.so.x.y.z` dove `x.y.z` rappresenta il numero di versione. Le librerie condivise, inoltre, contengono spesso dei collegamenti che puntano ad esse, che risultano essere molto importanti, e (in configurazioni a.out) contengono dei file `.sa` associati. Le librerie standard sono disponibili sia in formato condiviso, sia in formato statico.

È possibile sapere quali librerie condivise sono richieste da un programma utilizzando `ldd` (List Dynamic Dependencies)

```
$ ldd /usr/bin/lynx
        libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
        libc.so.5 => /lib/libc.so.5.2.18
```

Questo esempio mostra che il browser WWW 'lynx' dipende dalla presenza di `libc.so.5` (la libreria C) e `libncurses.so.1` (utilizzata per il controllo del terminale). Se un programma non ha dipendenze, `ldd` risponderà 'statically linked' o 'statically linked (ELF)'.

6.2 Interrogazione delle librerie ('In quale libreria si trova sin()?')

`nm` **nome_libreria** dovrebbe listare tutti i simboli per i quali esistono dei riferimenti in **nome_libreria**. Il comando funziona sia per librerie statiche che dinamiche. Si supponga di voler saper dov'è definita `tcgetattr()`: si potrebbe utilizzare

```
$ nm libncurses.so.1 |grep tcget
         U tcgetattr
```

'`U`' significa 'undefined' - mostra che la libreria `ncurses` la utilizza ma non la definisce. In alternativa, si potrebbe usare

```
$ nm libc.so.5 | grep tcget
00010fe8 T __tcgetattr
00010fe8 W tcgetattr
00068718 T tcgetpgrp
```

'W' significa 'weak', ossia che il simbolo è definito, ma in modo tale da poter essere sostituito da un'altra definizione in una libreria diversa. Una definizione 'normale' (come quella per `tcgetpgrp`) è indicata con una 'T'.

Comunque la risposta breve alla domanda del titolo, consiste in `libm.(so|a)`. Tutte le funzioni definite in `<math.h>` sono tenute nella libreria `math`; ne consegue che sarà necessario eseguire il collegamento con l'opzione `-lm` quando si utilizza una di esse.

6.3 Ricerca dei file

`ld: Output file requires shared library 'libfoo.so.1'`
(Ovvero: ld: Il file di output richiede la libreria condivisa 'libfoo.so.1')

La strategia di ricerca di un file per ld e simili varia a seconda della versione, ma l'unico punto che si può ritenere predefinito è `/usr/lib`. Se si vuole che le librerie vengano cercate in altre locazioni, è necessario specificare le loro directory tramite l'opzione `-L` in gcc o ld.

Se non dovesse funzionare, controllare che i file necessari si trovino effettivamente in quelle directory. Per a.out, il collegamento con `-lfoo` fa in modo che ld cerchi `libfoo.sa` (condivisa) e, in caso di insuccesso, `libfoo.a` (statica). Per ELF, verrà eseguita la ricerca di `libfoo.so`, quindi di `libfoo.a`. `libfoo.so` è generalmente un collegamento simbolico a `libfoo.so.x`.

6.4 Compilazione delle proprie librerie

6.4.1 Controllo della versione

Come qualunque altro programma, le librerie possono contenere errori che vengono riscontrati e risolti nel tempo. Inoltre, le librerie possono introdurre nuove caratteristiche, modificare l'effetto di altre esistenti, o rimuovere quelle vecchie. Questo potrebbe costituire un problema per i programmi che le utilizzano.

Pertanto si è introdutto il concetto di versione della libreria. Tutte le modifiche che possono essere fatte a una libreria sono catalogate in 'minori' o 'maggiori', dove una modifica 'minore' non interrompe il funzionamento dei vecchi programmi che la utilizzano. La versione di una libreria può essere dedotta dal suo nome di file (di fatto, questo non è vero per quanto riguarda ELF; nel seguito viene spiegato il motivo): `libfoo.so.1.2` ha '1' come versione maggiore, e '2' come minore. Il numero di versione minore può essere svariato - libc inserisce in esso il 'livello di patch', assegnando alle librerie nomi del tipo `libc.so.5.2.18`, e spesso sono utilizzate lettere, underscore, o quasi ogni altro carattere ASCII.

Una delle differenze principali tra il formato ELF e a.out consiste nel modo in cui viene eseguita la compilazione delle librerie condivise. Per prima cosa viene descritto ELF, dal momento che è più semplice.

6.4.2 ELF. Di cosa si tratta?

ELF (Executable and Linking Format) è un formato binario originariamente sviluppato da USL (UNIX System Laboratories) e attualmente utilizzato in Solaris e System V Release 4. A seguito della sua aumentata flessibilità rispetto al più vecchio formato a.out utilizzato da Linux, gli sviluppatori di librerie GCC e C hanno deciso lo scorso anno di utilizzare ELF come formato binario standard per Linux.

Uteriori dettagli Questo paragrafo è tratto dal documento '/news-archives/comp.sys.sun.misc'.

> ELF (Executable Linking Format) è il nuovo e migliorato formato di file oggetto introdotto in SVR4. ELF è molto più potente di COFF, nel fatto di essere estendibile dall'utente. ELF vede un file oggetto come una lista di sezioni arbitrariamente lunga (piuttosto che come un array di entità a lunghezza fissa), tali sezioni, a differenza di quanto accade in COFF, non si devono trovare in un luogo specifico e non devono essere in un ordine specifico ecc. Gli utenti possono aggiungere nuove sezioni ai file oggetto, se desiderano avere a disposizione nuovi dati. ELF, inoltre, possiede un formato di debugging molto più potente denominato DWARF (Debugging With Attribute Record Format) - attualmente non supportato completamente su Linux (ma ci si sta lavorando). Una lista linkata di DIE (o Debugging Information Entries) di DWARF costituisce la sezione `.debug` di ELF. Invece di essere un insieme di piccoli record a dimensione fissa, ogni DIE di DWARF contiene una lista di lunghezza arbitraria di attributi complessi ed è scritto come un albero di dati di programma. I DIE sono in grado di includere una quantità di informazioni di molto maggiore rispetto alla sezione `.debug` di COFF (come grafi di eredità del C++ ecc).
>
> L'accesso ai file ELF avviene tramite la libreria di accesso ELF SVR4 (Solaris 2.0 ?), che fornisce un'interfaccia semplice e rapida alle parti più complicate di ELF. Uno dei vantaggi principali derivanti dall'utilizzo della libreria di accesso ELF consiste nel fatto che non sarà mai necessario vedere un file ELF come file Unix, è possibile eseguire l'accesso come file Elf *, dopo una chiamata `elf_open()` si eseguono chiamate `elf_foobar()` sulle sue componenti invece di occuparsi della sua immagine effettiva su disco (cosa che con COFF si faceva impunemente).

I vantaggi e gli svantaggi di ELF, e le evoluzioni necessarie per eseguire l'upgrade di un sistema a.out per supportarlo, sono descritti nell'ELF-HOWTO e non ho intenzione di ripeterli qui. L'HOWTO dovrebbe essere disponibile nello stesso luogo in cui è stato trovato questo documento.

Librerie condivise di ELF Per eseguire la compilazione di `libfoo.so` come libreria condivisa, i passi di base hanno la seguente forma:

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ LD_LIBRARY_PATH='pwd':$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

Questi comandi genererano una libreria condivisa denominata `libfoo.so.1.0`, i collegamenti appropriati per ld (`libfoo.so`) e il caricamento dinamico (`libfoo.so.1`) per trovarla. Per eseguire un collaudo, si aggiunge la directory corrente a `LD_LIBRARY_PATH`.

Quando si è sicuri che la libreria funziona, deve essere spostata, ad esempio, in `/usr/local/lib`, e devono essere creati appropriati collegamenti. Il collegamento da `libfoo.so.1` a `libfoo.so.1.0` è mantenuto aggiornato da `ldconfig`, che nella maggior parte dei sistemi viene eseguito come parte del processo di avviamento. Il collegamento `libfoo.so` deve essere aggiornato manualmente. Se si è scrupolosi nell'eseguire l'aggiornamento di tutte le parti di una libreria (ossia degli header file) contemporaneamente, la cosa più semplice da fare consiste nel rendere `libfoo.so -> libfoo.so.1`, in modo che ldconfig mantenga correnti entrambi i collegamenti. In caso contrario, potrebbe in seguito verificarsi ogni genere di stranezza.

```
$ su
# cp libfoo.so.1.0 /usr/local/lib
# /sbin/ldconfig
# ( cd /usr/local/lib ; ln -s libfoo.so.1 libfoo.so )
```

Numerazione delle versioni, soname e symlink Ogni libreria ha un **soname**. Quando il linker trova uno di questi in una libreria in cui sta eseguendo una ricerca, nel binario viene incluso il soname in luogo del nome di file effettivo ricercato. Durante l'esecuzione, il loader dinamico cercherà un file tramite il nome di soname, non con il nome di file/libreria. Pertanto, una libreria denominata `libfoo.so` potrebbe avere il soname `libbar.so`, di conseguenza tutti i programmi collegati ad essa, all'avvio, cercherebbero `libbar.so`.

Sembra essere una caratteristica di poca importanza, invece è la chiave per capire come su un sistema possono coesistere diverse versioni della stessa libreria. Di fatto, la denominazione standard delle librerie in Linux consiste nel chiamare la libreria, ad esempio, `libfoo.so.1.2`, e assegnare ad essa il soname `libfoo.so.1`. Se aggiunta in una directory di libreria 'standard' (ossia, `/usr/lib`), ldconfig creerà un collegamento simbolico `libfoo.so.1 -> libfoo.so.1.2` in modo che sia possibile trovare l'immagine appropriata durante l'esecuzione. È necessario anche un collegamento `libfoo.so -> libfoo.so.1` affinché ld possa trovare il soname corretto da utilizzare durante il link.

Pertanto, quando si risolve un errore nella libreria, o si aggiungono nuove funzioni (ogni modifica che non influenzi in modo negativo i programmi esistenti), si eseguirà nuovamente la compilazione mantenendo lo stesso soname, e modificando il nome di file. Quando si inseriscono nella libreria delle modifiche che causerebbero l'interruzione dei programmi esistenti, incrementare semplicemente il numero nel soname - in questo caso, rinominare la nuova versione `libfoo.so.2.0`, e assegnarle il soname `libfoo.so.2`. Quindi, convertire il collegamento a `libfoo.so` in modo che punti alla nuova versione e tutto è a posto.

Si noti che non è **necessario** dare un nome alle librerie, ma si tratta di una buona convenzione. ELF fornisce una flessibilità nel nominare le librerie in modi che potrebbero confondere, ma questo non significa che debba farlo per forza.

Riassumendo: se si suppone di osservare la tradizione secondo cui gli aggiornamenti maggiori potrebbero distruggere la compatibilità e che quelli minori non lo fanno, eseguire il collegamento con

```
gcc -shared -Wl,-soname,libfoo.so.major -o libfoo.so.major.minor
```

e tutto funzionerà alla perfezione.

6.4.3 a.out. Il formato tradizionale

La facilità con cui si esegue la compilazione di librerie condivise è uno dei motivi principali per passare a ELF. Detto questo, è ancora possibile utilizzare a.out. Si prenda *ftp://tsx-11.mit.edu/pub/linux/packages/GCC/src/tools-2.17.tar.gz*

e si legga il documento di 20 pagine.

ZMAGIC e QMAGIC QMAGIC è un formato eseguibile proprio come i vecchi binari a.out (conosciuti anche come ZMAGIC), ma che lascia la prima pagina non mappata. Questo consente che accada più facilmente un riferimento NULL dal momento che non esiste alcun mapping nel range 0-4096. Come effetto collaterale, i binari saranno di dimensioni inferiori (di circa 1 K).

I linker obsoleti supportano solamente ZMAGIC, quelli semi-obsoleti supportano entrambi i formati, mentre le versioni attuali supportano solo QMAGIC. In realtà questo non ha importanza, dal momento che il kernel riesce ad eseguire entrambi i formati.

Il proprio comando 'file' dovrebbe essere in grado di identificare se un programma è QMAGIC.

Posizione dei file Una libreria condivisa a.out (DLL) è formata da due file reali e da un collegamento simbolico. Per la libreria '`foo`' utilizzata come esempio, questi file sarebbero `libfoo.sa` e `libfoo.so.1.2`; il collegamento simbolico sarebbe `libfoo.so.1` e punterebbe all'ultimo di questi file. Ma a cosa servono?

Durante la compilazione, ld ricerca `libfoo.sa`. Si tratta del file 'matrice' della libreria, e contiene tutti i dati esportati e i puntatori alle funzioni richieste per il collegamento run time.

Durante l'esecuzione, il loader dinamico cerca `libfoo.so.1`. Si tratta di un collegamento simbolico anziché di un file reale in modo che le librerie possano essere aggiornate con versioni più recenti e corrette senza procurare danni a nessuna delle applicazioni utilizzanti la libreria in quel momento. Dopo che la nuova versione, ad esempio `libfoo.so.1.3` - è stata introdotta, l'esecuzione di ldconfig commuterà il collegamento affinché punti ad essa tramite un'operazione atomica, lasciando illeso ogni programma che stava utilizzando la vecchia versione.

Le librerie DLL appaiono spesso più grandi delle loro controparti statiche. Riservano spazio per future espansioni nella forma di 'buchi' che possono essere creati senza occupare spazio su disco. Una semplice chiamata `cp` o l'utilizzo del programma `makehole` raggiungerà questo scopo. Dopo la compilazione, è anche possibile rimuoverli, dal momento che gli indirizzi si trovano in locazioni fisse. Non tentare di rimuoverli dalle librerie ELF.

libc-lite? Un libc-lite è una versione ridotta della libreria libc per la quale è stata eseguita la compilazione in modo tale da stare su un floppy disk ed essere sufficiente per tutti i task Unix essenziali. Non include codice `curse`, `dbm`, `termcap` ecc. Se il proprio `/lib/libc.so.4` ha un collegamento con un lite lib, il sistema avvisa di sostituirlo con una versione completa.

6.4.4 Linking: problemi comuni

Inviatemi i problemi derivanti dal linking! Anche se non potrò fare niente per risolverli, ne scriverò un resoconto dettagliato.

Programmi eseguono il link statico anziché dinamico

Controllare di avere i collegamenti corretti affinché ld possa trovare tutte le librerie condivise. Per ELF questo significa un collegamento simbolico `libfoo.so` per l'immagine, in a.out un file `libfoo.sa`. Molte persone hanno riscontrato questo problema dopo il passaggio dai binutils ELF 2.5 ai 2.6 - la versione precedente ricercava le librerie condivise in un modo 'più intelligente' pertanto non avevano bisogno di creare tutti i collegamenti. Il comportamento intelligente è stato eliminato per compatibilità con altre architetture, e perché molto spesso le sue supposizioni erano sbagliate e causando più guai di quanti fossero i problemi risolti.

Lo strumento DLL 'mkimage' non riesce a trovare libgcc

Da `libc.so.4.5.x` e oltre, `libgcc` non è più condivisa. Pertanto, è necessario sostituire le occorrenze di '`-lgcc`' con '`gcc -print-libgcc-file-name`'.

Inoltre, bisogna cancellare tutti i file `/usr/lib/libgcc*`. Questo è molto importante.

Messaggi `__NEEDS_SHRLIB_libc_4 multiply defined`

Altra conseguenza del problema descritto al punto precedente.

Messaggio "Assertion failure" quando si ricompila una DLL?

Questo messaggio criptico molto probabilmente significa che uno degli slot della propria jump table è andato in overflow poiché è stato riservato troppo poco spazio nel file jump.vars originale. È possibile localizzare i colpevoli eseguendo il comando '`getsize`' fornito nel pacchetto `tools-2.17.tar.gz`. Tuttavia, probabilmente l'unica soluzione consiste nel sostituire il numero di versione maggiore della libreria, forzandolo affinché sia impossibile tornare indietro.

`ld: output file needs shared library libc.so.4`

Questo accade solitamente quando si sta eseguendo il collegamento con librerie diverse da libc (come le librerie X), e si utilizza l'opzione `-g` sulla riga di link utilizzando anche `-static`.

Gli stub `.sa` per le librerie condivise contengono solitamente un simbolo indefinito `_NEEDS_SHRLIB_libc_4` che viene risolto da `libc.sa`. Tuttavia, con `-g` si finisce di eseguire il collegamento con `libg.a` o `libc.a`, il simbolo non viene mai risolto, portando all'errore sopra descritto.

In conclusione, aggiungere `-static` quando si compila con l'opzione `-g`, oppure non eseguire il collegamento con `-g`. Molto spesso è possibile ottenere informazioni di debugging sufficienti compilando i file individuali con `-g`, ed eseguendo il collegamento senza questa opzione.

6.5 Loading dinamico

Questo paragrafo è per il momento piuttosto breve, verrà esteso in futuro.

6.5.1 Concetti

Linux possiede delle librerie condivise, come si è visto diverse molte volte nell'ultimo paragrafo. Gran parte del lavoro di associazione dei nomi a locazioni, che tradizionalmente era svolto al momento del link, deve essere ritardato al momento del load (caricamento).

6.5.2 Messaggi di errore

I lettori sono pregati di inviare i proprii errori di link all'autore, che anche se non potrà risolverli, comunque scriverà un resoconto dettagliato.

`can't load library: /lib/libxxx.so, Incompatible version`

(solo in a.out) Questo significa che non si possiede la versione maggiore aggiornata della libreria xxx. Non è possibile semplicemente creare un collegamento simbolico ad un'altra versione che si possiede; nella migliore delle ipotesi questo causerà un segfault nel proprio programma. Si consiglia di ottenere una nuova versione. Una situazione simile in ELF produrrà un messaggio del tipo

```
ftp: can't load library 'libreadline.so.2'
```

`warning using incompatible library version xxx`

(solo in a.out) Si possiede una versione minore della libreria più vecchia di quella posseduta dalla persona che ha compilato il programma in questione. Il programma funzionerà comunque. Tuttavia, un aggiornamento non sarebbe una cattiva idea.

6.5.3 Controllo delle operazioni del loader dinamico

Esistono diverse variabili di ambiente a che influenzano il comportamento del loader dinamico. La maggior parte di esse sono più utili a `ldd` di quanto non lo siano per l'utente medio, e possono essere impostate eseguendo ldd con diverse opzioni. Includono

- `LD_BIND_NOW` — normalmente, le funzioni non sono ricercate nelle librerie finché non vengono chiamate. L'impostazione di questa opzione attiva la ricerca all'atto del caricamento della libreria, determinando un tempo di avviamento maggiore. Può essere utile quando si vuole collaudare un programma per accertarsi che sia eseguito il link di tutte le parti.

- `LD_PRELOAD` — può essere impostato con un file contenente delle definizioni di funzioni da sovrapporre. Ad esempio, se si sta eseguendo un test delle strategie di allocazione della memoria, e si vuole sostituire 'malloc', è possibile scrivere la propria routine sostitutiva, compilarla come `malloc.o` e utilizzare i comandi

  ```
  $ LD_PRELOAD=malloc.o; export LD_PRELOAD
  $ programma_di_test
  ```

  `LD_ELF_PRELOAD` e `LD_AOUT_PRELOAD` sono simili, ma possono essere applicati solo al tipo binario appropriato. Se `LD_`**qualcosa**`_PRELOAD` e `LD_PRELOAD` sono entrambi impostati, verrà utilizzato quello più specifico.

- `LD_LIBRARY_PATH` — elenco, separato da virgole, di directory in cui ricercare le librerie condivise. **Non** ha effetti su ld, ma solo durante l'esecuzione. Inoltre, è disabilitato per programmi che eseguono setuid o setgid. `LD_ELF_LIBRARY_PATH` e `LD_AOUT_LIBRARY_PATH` possono anche essere utilizzati per impostare la ricerca in modo differente per diversi tipi di binari. `LD_LIBRARY_PATH` non dovrebbe essere necessario nelle operazioni normali; piuttosto aggiungere le directory a `/etc/ld.so.conf/` ed eseguire ldconfig.

- `LD_NOWARN` — si applica solo ad a.out. Quando impostato (ossia con `LD_NOWARN=true; export LD_NOWARN`) evita che il *loader* fornisca i warning non fatali (come i messaggi per incompatibilità di versione minore).

- `LD_WARN` — si applica solamente a ELF. Quando impostato, rende i messaggi, solitamente fatali, Can't find library dei semplici warning. Non è molto utilizzato nelle operazioni normali, ma è importante per ldd.

- `LD_TRACE_LOADED_OBJECTS` — si applica solamente a ELF, e fa in modo che i programmi credano di essere in esecuzione sotto ldd:

  ```
  $ LD_TRACE_LOADED_OBJECTS=true /usr/bin/lynx
          libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
          libc.so.5 => /lib/libc.so.5.2.18
  ```

6.5.4 Scrivere programmi con il loading dinamico

Questo è molto simile al funzionamento del supporto di loading dinamico di Solaris 2.x. L'argomento è trattato ampiamente nel documento di programmazione ELF di H J Lu e nella pagina di manuale `dlopen(3) manual page`, che può essere trovata nel pacchetto ld.so. Segue un semplice esempio: è necessario effettuare il link con `-ldl`

```
#include <dlfcn.h>
#include <stdio.h>

main()
{
    void *libc;
    void (*printf_call)();

    if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
    {
        printf_call=dlsym(libc,"printf");
        (*printf_call)("hello, world\n");
    }

}
```

7 Come contattare gli sviluppatori

7.1 Comunicazione degli errori

Per prima cosa è necessario cominciare a **circoscrivere il problema**. È specifico di Linux, oppure accade con gcc su altri sistemi? È specifico della versione di kernel? Della versione di libreria? Sparisce se si esegue il link statico? È possibile ritagliare una piccola parte del programma che dimostra l'errore?

Fatto questo, si è a conoscenza del/i programma/i in cui si trova l'errore. Per GCC, la procedura di comunicazione degli errori è spiegata nel file `info`. Per ld.so o per le librerie C o maths, inviare una mail a *linux-gcc@vger.rutgers.edu* . Se possibile, includere un breve e autonomo programma che possa dimostrare l'errore, e una descrizione sia di cosa si intendeva che facesse, sia di cosa fa effettivamente.

7.2 Aiuto nello sviluppo

Se si desidera aiutare lo sviluppo di GCC o della libreria C, la prima cosa da fare consiste nell'unirsi alla mailing list *linux-gcc@vger.rutgers.edu* . Se si vuole semplicemente dare un'occhiata alle discussioni, è possibile consultare *http://homer.ncm.com/linux-gcc/* . La seconda cosa e le successive dipendono solo dalle vostre intenzioni!

8 Ultime cose

8.1 Ringraziamenti

> Only presidents, editors, and people with tapeworms have the right to use the editorial we. (Mark Twain)

Ovvero

> Soli i presidenti, gli editori e la gente con il verme solitario hanno il diritto di usare il noi editoriale. (Mark Twain)

Questo HOWTO si basa molto strettamente al GCC-FAQ di Mitchum DSouza; la maggior parte delle informazioni (per non dire una grande quantità di testo) è derivato direttamente da quel documento. Esperienze riferite all'autore all'interno di questo HOWTO potrebbero riferirsi anche a Mitchum DSouza; generalmente le frasi del tipo 'Non si è mai collaudato questa parte; non maledite l'autore se il vostro disco/sistema si è abbrustolito' possono essere applicate a entrambi. Inoltre hanno contribuito a questo documento (in ordine ASCII per nome): Andrew Tefft, Axel Boldt, Bill Metzenthen, Bruce Evans, Bruno Haible, Daniel Barlow, Daniel Quinlan, David Engel, Dirk Hohndel, Eric Youngdale, Fergus Henderson, H.J. Lu, Jens Schweikhardt, Kai Petzke, Michael Meissner, Mitchum DSouza, Olaf Flebbe, Paul Gortmaker, Rik Faith, Steven S. Dick, Tuomas J Lukka, e naturalmente Linus Torvalds, senza il quale l'intero lavoro non avrebbe avuto senso. Vi prego di non sentirvi offesi se nella lista non compare il vostro nome e avete contribuito a questo documento (come HOWTO o come FAQ). Inviate mail ed sarà eseguita la correzione.

8.2 Traduzioni

Attualmente, non esistono traduzioni di questo documento Se desiderate farne una, siete liberi di farlo ma vi prego di farmelo sapere! Ci sono pochissime probabilità che io sia in grado di parlare la lingua in cui desiderate tradurre il documento, ma a parte questo sarei lieto di aiutarvi in qualsiasi modo.

8.3 Comunicazioni, opinioni, correzioni

Sono benvenute inviate email all'indirizzo *dan@detached.demon.co.uk* . La chiave pubblica PGP (ID 5F263625) è disponibile dalle pagine web dell'autore *http://ftp.linux.org.uk/~barlow/* , se ci sono necessità di riservatezza.

8.4 Informazioni legali

> L'unica licenza valida è quella originale in lingua inglese. Di seguito ne trovate una traduzione abbastanza fedele che però non ha alcun valore.

9 Riferimenti in italiano

Questa sezione riporta riferimenti a documenti italiani. Nel caso in cui un documento non sia ancora tradotto si rimanda al WEB server del Pluto.

- [1] *http://www.pluto.linux.it/ildp/HOWTO/ELF-HOWTO.html*
- [2] *http://www.pluto.linux.it/ildp/* WEB server del PLUTO, sono disponibili anche le traduzioni di altri HOWTO.
- [3] *http://www.pluto.linux.it/ildp/HOWTO/GCC-HOWTO.html*

10 Indice Analitico