Assembly HOWTO

François-René Rideau `rideau@ens.fr` v0.4l, 16 Novembre 1997

Questo è il Linux Assembly HOWTO. Questo documento descrive come programmare in assembly utilizzando strumenti di programmazione *LIBERI*, concentrandosi sullo sviluppo per o sul sistema operativo Linux su piattaforme i386. Il materiale incluso potrebbe essere o meno applicabile ad altre piattaforme hardware e/o software. Contributi riguardo a queste sarebbero ben accetti. *parole chiave*: assembly, assembler, libero, macro, preprocessore, asm, inline, 32-bit, x86, i386, gas, as86, nasm. Traduzione di Matteo De Luigi (giotto@maya.dei.unipd.it).

Indice

1 INTRODUZIONE

1.1 Legal Blurp

1.2 NOTA IMPORTANTE

C'è da aspettarsi che questo sia l'ultimo rilascio di questo documento da parte mia.

C'è un candidato al ruolo di curatore, ma finché l'HOWTO non diventa ufficialmente suo, accetterò suggerimenti e critiche.

Siete in particolare invitati a porre domande, a rispondere alle domande, a correggere le risposte date, ad aggiungere nuove risposte alle FAQ, a fornire riferimenti ad altro software, ad indicare errori o lacune nelle pagine al responsabile attuale. Se qualcuno di voi è motivato, potrebbe perfino *DIVENTARE IL RESPONSABILE DELLE FAQ*. In una parola, contribuite!

Per contribuire, per favore contattate chiunque sembri curare l'Assembly-HOWTO. I curatori attuali sono:

François-René Rideau `mailto:rideau@clipper.ens.fr`

ed ora

Paul Anderson `mailto:paul@geeky1.ebtech.net` .

1.3 Prima di cominciare

Questo documento intende rispondere alle domande più frequenti delle persone che programmano o vogliono programmare in assembly a 32 bit per x86 utilizzando assemblatori *liberi*, specialmente sotto il sistema operativo Linux. Potrebbe inoltre rimandare ad altri documenti circa assemblatori non liberi, non per x86 o non a 32 bit, anche se questo non è il suo scopo principale.

Poiché l'interesse principale della programmazione in assembly consiste nel realizzare le viscere dei sistemi operativi, degli interpreti, dei compilatori e dei giochi, laddove un compilatore C non riesce a fornire l'espressività richiesta (è abbastanza raro che si tratti di una questione di prestazioni), ci interesseremo principalmente di questo tipo di software.

1.3.1 Come usare questo documento

Questo documento contiene risposte ad alcune domande poste di frequente. In molte occasioni, vengono forniti URL di alcuni archivi di software o documentazione. Per favore, notate che gli archivi di maggiore utilità hanno dei mirror e che accedendo ad un mirror più vicino da un lato evitate ad Internet traffico non necessario e dall'altro risparmiate tempo prezioso. In particolare, ci sono dei grandi depositi sparsi per tutto il mondo che fanno il mirror anche di altri archivi di pubblico interesse.

Dovreste imparare ad annotarvi quali sono questi posti vicino a voi (dal punto di vista della rete).

Talvolta, la lista dei mirror si trova in un file o in un messaggio di login. Siete pregati di seguire i consigli. Altrimenti, dovreste interrogare archie circa il software di cui siete alla ricerca...

La versione più recente di questi documenti risiede ad

`http://www.eleves.ens.fr:8080/home/rideau/Assembly-HOWTO`

oppure

`http://www.eleves.ens.fr:8080/home/rideau/Assembly-HOWTO.sgml`

ma anche ciò che si trova negli archivi degli HOWTO di Linux *dovrebbe* essere abbastanza aggiornato (io non ho modo di saperlo):

`ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO/` (?)

Una traduzione in francese di questo HOWTO può essere trovata dalle parti di:

`ftp://ftp.ibp.fr/pub/linux/french/HOWTO/`

1.3.2 Documenti correlati

- Se non sapete cos'è il software *libero*, siete pregati di leggere *attentamente* la Licenza Pubblica Generale GNU, che viene usata in tantissimo software libero ed è un modello per la maggior parte delle licenze per questo tipo di programmi. Si trova di solito in un file chiamato `COPYING`, con una versione per le librerie in un file chiamato `COPYING.LIB`. Anche qualche pubblicazione della FSF (free software foundation) potrebbe esservi d'aiuto.
- In particolare, il software libero più interessante è disponibile con sorgenti che possono essere consultati e corretti. Talvolta è persino possibile prendere in prestito del codice. Leggete con cura le licenze specifiche ed agite in conformità ad esse.
- C'è una FAQ per comp.lang.asm.x86 che risponde a domande generiche circa la programmazione in assembly su x86 e a domande circa alcuni assemblatori commerciali in ambiente DOS a 16 bit. Alcune riguardano la programmazione libera in asm a 32 bit, così potreste essere interessati a leggere queste FAQ...

  `http://www2.dgsys.com/~raymoon/faq/asmfaq.zip`
- Esistono FAQ e documentazione riguardanti la programmazione sulla vostra piattaforma preferita, qualunque essa sia, che dovreste consultare per questioni specifiche alla piattaforma stessa non direttamente correlate alla programmazione in assembler.

1.4 Storia

Ogni versione contiene alcune correzioni e rettifiche di poco conto che non è necessario menzionare ogni volta.

Versione 0.1 23 apr 1996

Francois-Rene «Faré» Rideau <rideau@ens.fr> crea e pubblica il primo mini-HOWTO, perché «Non ne posso più di rispondere sempre alle stesse domande su comp.lang.asm.x86»

Versione 0.2 4 mag 1996

*

Versione 0.3c 15 giu 1996

*

Versione 0.3f 17 ott 1996

Trovata l'opzione -fasm per abilitare l'assemblatore inline di GCC senza le ottimizzazioni -O

Versione 0.3g 2 nov 1996

Creata la storia del documento. Aggiunti i riferimenti nella sezione sulla compilazione incrociata. Aggiunta la sezione circa la programmazione dell'I/O sotto Linux (video in particolare).

Versione 0.3h 6 nov 1996

maggiori informazioni sulla compilazione incrociata. Vedere devel/msdos su sunsite.

Versione 0.3i 16 nov 1996

NASM sta diventando molto affidabile.

Versione 0.3j 24 nov 1996

Riferimento alla traduzione in francese.

Versione 0.3k 19 dic 1996

Cosa? Mi ero dimenticato di fare riferimento a Terse???

Versione 0.3l 11 gen 1997

*

Versione 0.4pre1 13 gen 1997

Il mini-HOWTO in formato testo viene trasformato in un completo HOWTO linuxdoc-sgml, per vedere come sono gli SGML tools.

Versione 0.4 20 gen 1997

Primo rilascio dell'HOWTO come tale.

Versione 0.4a 20 gen 1997

Aggiunta la sezione «ringraziamenti».

Versione 0.4b 3 feb 1997

Spostato NASM: ora è prima di AS86

Versione 0.4c 9 feb 1997

Aggiunta la sezione «avete bisogno dell'assembly?»

Versione 0.4d 28 feb 1997

Annuncio prematuro di un nuovo responsabile dell'Assembly-HOWTO.

Versione 0.4e 13 mar 1997

Rilascio per DrLinux

Versione 0.4f 20 mar 1997

*

Versione 0.4g 30 mar 1997

*

Versione 0.4h 19 giu 1997

ancora aggiunte a «come NON usare l'assembly»; aggiornamenti su NASM, GAS.

Versione 0.4i 17 luglio 1997

informazioni sull'accesso al modo a 16 bit da Linux.

Versione 0.4j 7 settembre 1997

*

Versione 0.4k 19 ottobre 1997

*

Versione 0.4l 16 novembre 1997

rilascio per LSL, sesta edizione.

Questo è ancora un altro ultimo-rilascio-di-Faré-prima-che-un-altro-curatore-gli-subentri (?)

1.5 Ringraziamenti

Vorrei ringraziare le seguenti persone, in ordine di apparizione:

- *Linus Torvalds* `mailto:sepolto.vivo@nella.posta`

  per Linux

- *Bruce Evans* `mailto:bde@zeta.org.au`

  per bcc da cui è estratto as86

- *Simon Tatham* `mailto:anakin@poboxes.com` e

  Julian Hall `mailto:jules@earthcorp.com`

  per NASM

- *Jim Neil* `mailto:jim-neil@digital.net`

  per Terse

- *Tim Bynum* `mailto:linux-howto@sunsite.unc.edu`

  perché mantiene gli HOWTO

- *Raymond Moon* `mailto:raymoon@moonware.dgsys.com`

  per le sue FAQ

- *Eric Dumas* `mailto:dumas@excalibur.ibp.fr`

  per la sua traduzione in francese del mini-HOWTO (è una cosa triste per l'autore originale essere francese e scrivere in inglese)

- *Paul Anderson* `mailto:paul@geeky1.ebtech.net`

  e *Rahim Azizarab* `mailto:rahim@megsinet.net`

  per l'aiuto, se non per aver rilevato l'HOWTO.

- Tutte le persone che hanno dato il loro contributo con idee, commenti e supporto morale.

2 AVETE BISOGNO DELL'ASSEMBLY?

Beh, non vorrei interferire con ciò che state facendo, ma ecco alcuni consigli derivanti da una esperienza ottenuta faticosamente.

2.1 Pro e contro

2.1.1 I vantaggi dell'assembly

L'assembly può esprimere cose molto a basso livello:

- potete accedere a registri e ad I/O dipendenti dalla macchina.
- potete controllare l'esatto comportamento di codice in sezioni critiche che potrebbe comportare il bloccarsi di hardware o I/O.
- potete trasgredire le convenzioni del vostro compilatore abituale, il che potrebbe permettere alcune ottimizzazioni (come ad esempio violare temporaneamente le regole per il garbage collecting, threading, ecc).
- ottenere accesso a modi di programmazione insoliti del vostro processore (ad esempio codice a 16 bit per l'avvio o l'interfaccia BIOS sui PC Intel).
- potete costruire interfacce tra frammenti di codice che usano convenzioni incompatibili (ad esempio prodotti da compilatori diversi o separati da una interfaccia a basso livello).
- potete produrre codice ragionevolmente veloce per cicli stretti per far fronte ad un compilatore non-ottimizzante di qualità scadente (ma dopotutto sono disponibili compilatori ottimizzanti liberi!).
- potete produrre codice ottimizzato a mano che risulta ottimo per la vostra particolare configurazione hardware, ma non per quella di chiunque altro.
- potete scrivere del codice per il compilatore ottimizzante del vostro nuovo linguaggio (è qualcosa che poche persone fanno, ed anche loro non lo fanno molto spesso).

2.1.2 Gli svantaggi dell'assembly

L'assembly è un linguaggio molto a basso livello (più in basso c'è solo la codifica a mano delle istruzioni in codice binario).

Ciò significa:

- All'inizio è lungo e tedioso da scrivere.
- È notevolmente soggetto ad errori.
- Gli errori saranno molto difficili da scovare.
- È molto difficile da comprendere e modificare, in altre parole da mantenere.
- Il risultato è decisamente non portabile verso altre architetture, esistenti o future.
- Il vostro codice verrà ottimizzato solo per una certa implementazione di una stessa architettura: ad esempio, tra le piattaforme compatibili Intel, avere CPU diverse o differenti configurazioni (ampiezza del bus, velocità e dimensioni relative di CPU/cache/RAM/bus/dischi, presenza di FPU, estensioni MMX, ecc.) può richiedere tecniche di ottimizzazione radicalmente diverse. I tipi di CPU comprendono

già Intel 386, 486, Pentium, PPro, Pentium II; Cyrix 5x86, 6x86; AMD K5, K6. Inoltre, continuano ad apparire nuovi tipi, perciò non aspettatevi che questo elenco o il vostro codice siano aggiornati.

- Il vostro codice potrebbe inoltre non essere portabile verso piattaforme con sistemi operativi differenti ma con la stessa architettura per la mancanza di strumenti adeguati (beh, GAS pare funzionare su tutte le piattaforme; a quanto sembra, NASM funziona o può essere reso funzionante su tutte le piattaforme Intel).
- Perdete più tempo su pochi dettagli e non potete concentrarvi sulla progettazione algoritmica su piccola e grande scala che, come è noto, porta il maggior contributo alla velocità del programma. Per esempio, potresete perdere del tempo per scrivere in assembly delle primitive molto veloci per la manipolazione di liste o matrici, quando sarebbe bastato utilizzare una tabella hash per accelerare molto di più il vostro programma. Magari, in un altro contesto, sarebbe servito un albero binario, o qualche struttura ad alto livello distribuita su un cluster di CPU.
- Un piccolo cambiamento nell'impostazione algoritmica potrebbe far perdere ogni validità a tutto il codice assembly già esistente. Perciò o siete pronti a (ed in grado di) riscriverlo tutto, oppure siete vincolati ad una particolare impostazione algoritmica.
- Per quanto riguarda il codice che non si scosta troppo da quello che è presente nei benchmark convenzionali, i compilatori ottimizzanti commerciali permettono di ottenere prestazioni migliori rispetto all'«assembly manuale» (beh, ciò è meno vero sulle architetture x86 rispetto alle architetture RISC e forse meno vero per compilatori largamente disponibili/liberi; comunque, per codice C tipico, GCC se la cava discretamente).
- E in ogni caso, come dice il moderatore John Levine su comp.compilers, «i compilatori rendono decisamente più facile utilizzare strutture dati complesse, non si stancano a metà strada e ci si può aspettare che generino codice abbastanza buono». Inoltre provvederanno a propagare *correttamente* trasformazioni di codice attraverso tutto il (lunghissimo) programma quando si tratterà di ottimizzare codice tra i confini delle procedure e dei moduli.

2.1.3 Valutazioni

Tutto sommato, potreste notare che nonostante l'uso dell'assembly sia talvolta necessario (o semplicemente utile, in alcuni casi), sarà il caso che:

- minimizziate l'uso del codice assembly;
- incapsuliate questo codice in interfacce ben definite;
- facciate generare automaticamente il vostro codice assembly da strutture espresse in un linguaggio a più alto livello rispetto all'assembly stesso (ad esempio le macro dell'assembly inline di GCC);
- facciate tradurre in assembly questi programmi da strumenti automatici;
- facciate ottimizzare questo codice, se possibile;
- tutti i punti di cui sopra, cioè scriviate (un'estensione ad) un backend per un compilatore ottimizzante.

Anche nei casi in cui l'assembly è necessario (ad esempio, nello sviluppo di sistemi operativi), scoprirete che non ne serve poi molto e che i principi precedenti continuano a valere.

A questo riguardo, date un'occhiata ai sorgenti del kernel di Linux: poco assembly, giusto lo stretto necessario, il che ha come risultato un sistema operativo veloce, affidabile, portabile e mantenibile. Anche un gioco di successo come DOOM è stato scritto quasi completamente in C, con solo una minuscola parte scritta in assembly per renderlo più veloce.

2.2 Come NON usare l'assembly

2.2.1 Procedura generale per ottenere codice efficiente

Come dice Charles Fiterman su comp.compilers circa il confronto tra codice assembly generato a mano o automaticamente,

«L'uomo dovrebbe sempre vincere, ed eccone i motivi:

- Primo passo: l'uomo scrive il tutto in un linguaggio ad alto livello.
- Secondo passo: provvede ad un profiling per trovare i punti in cui si perde più tempo.
- Terzo passo: fa produrre al compilatore codice assembly per quelle piccole sezioni di codice.
- Quarto passo: le perfeziona a mano cercando piccoli miglioramenti rispetto al codice generato dalla macchina.

L'uomo vince perché sa usare la macchina.»

2.2.2 Linguaggi con compilatori ottimizzanti

I linguaggi quali ObjectiveCAML, SML, CommonLISP, Scheme, ADA, Pascal, C, C++, tra gli altri, dispongono di compilatori ottimizzanti liberi che ottimizzeranno il grosso dei vostri programmi (e spesso otterranno risultati migliori rispetto all'assembly manuale anche per cicli stretti), permettendovi nel frattempo di concentrarvi su dettagli più ad alto livello, il tutto senza vietarvi di ottenere qualche punto percentuale di prestazioni in più nella maniera espressa sopra, una volta che il vostro progetto avrà raggiunto un'impostazione stabile.

Certo, ci sono anche compilatori ottimizzanti commerciali per la maggior parte di quei linguaggi!

Alcuni linguaggi hanno compilatori che producono codice C, che può essere ulteriormente ottimizzato da un compilatore C. LISP, Scheme, Perl e molti altri fanno parte di questa categoria. La velocità è abbastanza buona.

2.2.3 Procedura generale per accelerare il vostro codice

Per quanto riguarda l'accelerazione del vostro codice, dovreste restringerla alle parti di un programma che uno strumento di profiling ha decisamente identificato come un collo di bottiglia.

Perciò, se identificate qualche porzione di codice come troppo lenta, dovreste:

- prima di tutto provare ad usare un algoritmo migliore;
- poi provare a compilarla invece di interpretarla;
- poi provare ad abilitare e raffinare l'ottimizzazione per il vostro compilatore;
- poi dare al compilatore dei consigli su come ottimizzare (informazione sui tipi in LISP; uso di register con GCC; un mucchio di opzioni nella maggior parte dei compilatori, ecc.);
- infine, se è il caso, ripiegate sulla programmazione assembly.

Come ultima cosa, prima che vi riduciate a scrivere assembly, dovreste ispezionare il codice generato, per controllare che il problema risieda proprio nella cattiva generazione del codice, visto che potrebbe anche non essere così: il codice generato dal compilatore potrebbe essere migliore di quanto avreste potuto fare voi, specialmente sulle moderne architetture multi-pipelined! Le parti lente di un programma potrebbero essere intrinsecamente tali. I più grossi problemi sulle architetture moderne con processori veloci sono dovuti a ritardi di accesso alla memoria, cache-miss, TLB miss, e page fault; l'ottimizzazione sui registri diventa inutile, ed otterrete risultati migliori riprogettando le strutture dati ed il threading per ottenere una miglior località nell'accesso alla memoria. Potrebbe forse essere d'aiuto un approccio completamente diverso al problema.

2.2.4 Ispezione del codice generato dal compilatore

Ci sono molte ragioni per ispezionare il codice assembly generato dal compilatore. Ecco cosa potete fare con tale codice:

- controllate se il codice generato può essere migliorato in maniera ovvia con assembly manuale (o con le opportune opzioni per il compilatore).
- Quando è il caso, partite da codice generato e modificatelo, invece di ripartire da zero.
- Più in generale, utilizzate il codice generato come stub da modificare. In questo modo, almeno, viene gestito correttamente il modo in cui le vostre routine assembly si interfacciano col mondo esterno.
- Rintracciare dei bug nel vostro compilatore (raramente, si spera).

Il modo canonico per far generare codice assembly è invocare con il flag `-S` il vostro compilatore. Ciò funziona con la maggior parte dei compilatori UNIX, compreso il compilatore C di GNU (GCC), ma nel vostro caso le cose potrebbero andare diversamente. Nel caso di GCC, con l'opzione `-fverbose-asm` verrà prodotto codice assembly più comprensibile. Certo, se volete ottenere buon codice assembly, non dimenticate di dare i soliti consigli e le solite opzioni per l'ottimizzazione!

3 ASSEMBLATORI

3.1 Assembly inline di GCC

Il noto compilatore GNU C/C++ (GCC), un compilatore ottimizzante a 32-bit alla base del progetto GNU, supporta l'architettura x86 abbastanza bene, e fornisce la possibilità di inserire codice assembly nei programmi C, in modo tale che l'allocazione dei registri può essere o specificata o lasciata a GCC. GCC funziona sulla maggior parte delle piattaforme disponibili, tra le quali sono degne di nota Linux, *BSD, VSTa, OS/2, *DOS, Win*, ecc.

3.1.1 Dove trovare GCC

Il sito originale di GCC è il sito FTP di GNU

`ftp://prep.ai.mit.edu/pub/gnu/`

in cui si trova anche tutto il software applicativo del progetto GNU che è stato rilasciato.

Versioni configurate e precompilate per Linux possono essere trovate in

`ftp://sunsite.unc.edu/pub/Linux/GCC/` . Esistono un sacco di mirror FTP di entrambi i siti in tutte le parti del mondo, così come copie su CD-ROM.

Recentemente, lo sviluppo di GCC si è biforcato. Maggiori notizie sulla versione sperimentale, egcs, presso

`http://www.cygnus.com/egcs/` .

Dovreste trovare dei sorgenti adattati per il vostro sistema operativo preferito e dei binari precompilati ai soliti siti FTP.

La versione più comune per DOS si chiama DJGPP e può essere trovata nelle directory con questo nome nei siti FTP. Vedere:

`http://www.delorie.com/djgpp/`

C'è anche una versione di GCC per OS/2 chiamata EMX, che funziona anche sotto DOS ed include molte routine di libreria per l'emulazione di UNIX. Date un'occhiata dalle parti di:

`http://www.leo.org/pub/comp/os/os2/gnu/emx+gcc/`

`http://warp.eecs.berkeley.edu/os2/software/shareware/emx.html`

`ftp://ftp-os2.cdrom.com/pub/os2/emx09c/`

3.1.2 Dove trovare documentazione per l'assemblatore inline di GCC

La documentazione di GCC include file di documentazione in formato texinfo. Potete compilarli con TeX e poi stampare il risultato, oppure convertirli in .info e sfogliarli con emacs, oppure ancora convertirli in .html o (con gli strumenti appropriati) in tutto ciò che volete, oppure semplicemente leggerli così come sono.

Di solito i file .info si trovano in ogni buona installazione di GCC.

La sezione corretta da cercare è: `C Extensions::Extended Asm::`

La sezione `Invoking GCC::Submodel Options::i386 Options::` potrebbe anch'essa rivelarsi utile. In particolare, dà i vincoli sui nomi dei registri specifici per l'i386: abcdSDB corrispondono rispettivamente a: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp` (nessuna lettera per `%esp`).

La DJGPP Games resource (non solo per hacker dei giochi) ha questa pagina apposta per l'assembly:

`http://www.rt66.com/~brennan/djgpp/djgpp_asm.html`

Infine, c'è una pagina web chiamata «DJGPP Quick ASM Programming Guide» che tratta URL, FAQ, sintassi asm AT&T per x86, alcune informazioni sull'asm inline e la conversione dei file .obj/.lib:

`http://remus.rutgers.edu/~avly/djasm.html`

GCC dipende da GAS per l'assembling e segue la sua sintassi (vedere in seguito); se usate l'asm inline, badate bene: è necessario che i caratteri «percento» siano protetti dall'espansione per poter essere passati a GAS. Vedere la sezione su GAS in seguito.

Potete trovare *un sacco* di esempi utili nella sottodirectory `linux/include/asm-i386/` dei sorgenti del kernel di Linux.

3.1.3 Invocare GCC per fargli trattare correttamente l'assembly inline

Assicuratevi di invocare GCC con il flag `-O` (oppure `-O2`, `-O3`, ecc.) per abilitare le ottimizzazioni e l'assembly inline. Se non lo fate, il vostro codice potrebbe venire compilato ma non essere eseguito correttamente!!! In realtà (lodi smisurate a Tim Potter, timbo@mohpit.air.net.au) è sufficiente utilizzare il flag `-fasm` (e forse `-finline-functions`) che attiva solo una parte di tutte le funzionalità abilitate da `-O`. Così, se avete problemi a causa di bug nelle ottimizzazioni della vostra particolare versione/implementazione di GCC, potete comunque usare l'asm inline. In maniera analoga, usate -fno-asm per disabilitare l'assembly inline (perché dovreste?).

Più in generale, buoni flag di compilazione per GCC sulla piattaforma x86 sono

```
gcc -O2 -fomit-frame-pointer -m386 -Wall
```

`-O2` è il giusto livello di ottimizzazione. Ottimizzare oltre produce codice che è parecchio più grande, ma solo di poco più veloce; tale sovraottimizzazione potrebbe essere utile solo per cicli stretti (se ce ne sono), che potreste comunque realizzare in assembly; se ne sentite la necessità, fatelo solo per le poche routine che ne hanno bisogno.

`-fomit-frame-pointer` consente al codice generato di evitare la stupida gestione del frame pointer, il che rende il codice più piccolo e veloce e libera un registro per ulteriori ottimizzazioni. Ciò preclude il comodo utilizzo degli strumenti per il debugging (`gdb`), ma nel momento in cui usate questi strumenti, non è che vi importi poi molto delle dimensioni e della velocità.

`-m386` produce codice più compatto, senza alcun rallentamento misurabile (notate che codice piccolo significa anche meno I/O per il disco ed esecuzione più rapida), ma forse sui suddetti cicli stretti potreste apprezzare `-mpentium` per il GCC speciale che ottimizza per pentium (sempre che abbiate come obiettivo proprio una piattaforma pentium).

`-Wall` abilita tutti gli avvisi e vi aiuta a scovare errori stupidi ed ovvii.

Per ottimizzare ancora di più, l'opzione `-mregparm=2` e/o il corrispondente attributo per le funzioni vi potrebbero essere utili<, ma potrebbero porre un sacco di problemi qualora doveste fare un link con codice estraneo...

Notate che potete rendere questi flag quelli predefiniti modificando il file `/usr/lib/gcc-lib/i486-linux/2.7.2.2/specs` o dovunque esso si trovi nel vostro sistema (meglio non aggiungere -Wall in quella sede, comunque).

3.2 GAS

GAS è l'assemblatore GNU, su cui fa affidamento GCC.

3.2.1 Dove trovarlo

Lo trovate nello stesso posto dove avete trovato GCC, in un pacchetto denominato binutils.

3.2.2 Cos'è la sintassi AT&T

Poiché GAS è stato concepito per supportare un compilatore UNIX a 32 bit, esso utilizza la notazione standard «AT&T», che assomiglia molto alla sintassi degli assemblatori standard per m68k ed è standard nel mondo UNIX. Questa sintassi non è né peggiore né migliore della sintassi «Intel». È semplicemente diversa. Una volta che ci si è abituati, la si trova molto più regolare della sintassi Intel, anche se un po' noiosa.

Ecco le cose a cui prestare maggiore attenzione quando si ha a che fare con la sintassi di GAS:

- I nomi dei registri hanno `%` come prefisso, cosicché i registri sono `%eax`, `%dl` e così via invece di solo `eax`, `dl`, ecc. Ciò fa sì che sia possibile includere simboli C esterni direttamente nel sorgente assembly, senza alcun rischio di confusione e senza alcun bisogno di orribili underscore anteposti.

- L'ordine degli operandi è la sorgente (o sorgenti) per prima e la destinazione per ultima, che è l'opposto della convenzione Intel di avere prima la destinazione e le sorgenti per ultime. Quindi, ciò che nella sintassi intel è `mov ax,dx` (carica il contenuto del registro `dx` nel registro `ax`) diventerà `mov %dx, %ax` nella sintassi AT&T.

- La lunghezza dell'operando è specificata tramite un suffisso al nome dell'istruzione. Il suffisso è `b` per byte (8 bit), `w` per word (16 bit) e `l` per long (32 bit). Ad esempio, la sintassi corretta per l'istruzione menzionata poco fa sarebbe stata `movw %dx,%ax`. Comunque, gas non richiede una sintassi AT&T rigorosa, quindi il suffisso è opzionale quando la lunghezza può essere ricavata dai registri usati come operandi, altrimenti viene posta a 32 bit per default (con un avviso).

- Gli operandi immediati sono indicati con il prefisso `$`, come in `addl $5,%eax` (somma il valore long 5 al registro `%eax`).
- L'assenza di prefisso in un operando indica che è un indirizzo di memoria; perciò `movl $pippo,%eax` mette l'*indirizzo* della variabile `pippo` nel registro `%eax`, mentre `movl pippo,%eax` mette il `contenuto` della variabile `pippo` nel registro `%eax`.
- L'indicizzazione o l'indirezione è ottenuta racchiudendo il registro indice o l'indirizzo della cella di memoria di indirezione tra parentesi, come in `testb $0x80,17(%ebp)` (esegue un test sul bit più alto del valore byte all'offset 17 dalla cella puntata da `%ebp`).

Esiste un programma per aiutarvi a convertire programmi dalla sintassi TASM alla sintassi AT&T. Date un'occhiata a

`ftp://x2ftp.oulu.fi/pub/msdos/programming/convert/ta2asv08.zip`

GAS ha una documentazione esauriente in formato TeXinfo, che trovate nella distribuzione dei sorgenti (e forse altrove). Potete sfogliare le pagine .info estratte con emacs o con ciò che più vi aggrada. C'era un file chiamato gas.doc o as.doc dalle parti del pacchetto sorgente di GAS, ma è stato incorporato nella documentazione in TeXinfo. Certo, in caso di dubbio, la documentazione definitiva sono i sorgenti stessi! Una sezione che vi interesserà particolarmente è `Machine Dependencies::i386-Dependent::`

Ancora, i sorgenti di Linux (il kernel del sistema operativo) si rivelano buoni esempi; date un'occhiata ai seguenti file sotto linux/arch/i386:

`kernel/*.S, boot/compressed/*.S, mathemu/*.S`

Se state scrivendo qualcosa tipo un linguaggio, un pacchetto per i thread, ecc. potreste anche guardare come si comportano altri linguaggi (OCaml, gforth, ecc.) o pacchetti per i thread (QuickThreads, MIT pthreads, LinuxThreads, etc), o quel che è.

Infine, limitarsi a compilare un programma C in assembly potrebbe mostrarvi la sintassi del genere di istruzioni che vi interessano. Vedere la precedente sezione 2 (Avete bisogno dell'assembly?).

3.2.3 Modo a 16 bit limitato

GAS è un assemblatore a 32 bit, il suo compito è quello di supportare un compilatore a 32 bit. Attualmente ha solo un supporto limitato per il modo a 16 bit, che consiste nell'anteporre i prefissi per i 32 bit alle istruzioni, cosicché scrivete codice a 32 bit che gira nel modo a 16 bit su una CPU a 32 bit. In entrambi i modi supporta l'uso dei registri a 16 bit, ma non l'indirizzamento a 16 bit.

Utilizzate le direttive `.code16` e `.code32` per passare da un modo all'altro. Notate che un'istruzione di assembly inline `asm(".code16\n")` consentirà a GCC di produrre codice a 32 bit che girerà in real mode!

Mi è stato detto che la maggior parte del codice necessario per supportare pienamente la programmazione nel modo a 16 bit è stata aggiunta a GAS da Bryan Ford (si prega di confermare), tuttavia non si trova in nessuna delle distribuzioni che ho provato, fino a binutils-2.8.1.x ... sarebbero gradite maggiori informazioni su questo argomento.

Una soluzione economica è quella di definire macro (vedere in seguito) che in qualche modo producono la codifica binaria (con `.byte`) solo per le istruzioni del modo a 16 bit di cui avete bisogno (quasi nessuna se usate il codice a 16 bit come sopra e se potete supporre con certezza che il codice girerà su una CPU x86 in grado di gestire i 32 bit). Per trovare la codifica corretta, potete ispirarvi ai sorgenti degli assemblatori in grado di gestire i 16 bit.

3.3 GASP

GASP (GAS Preprocessor) è il Preprocessore per GAS. Aggiunge macro e dei costrutti sintattici carini a GAS.

3.3.1 Dove trovare GASP

GASP si trova assieme a GAS nell'archivio binutils di GNU.

3.3.2 Come funziona

Funziona come un filtro, in modo molto simile a cpp e programmi analoghi. Non ho alcuna idea sui dettagli, ma assieme ad esso trovate documentazione relativa in texinfo, perciò limitatevi a sfogliarla (in .info), stamparla, sviscerarla. GAS con GASP mi sembra un comune assemblatore con macro.

3.4 NASM

Il progetto Netwide Assembler sta producendo un ulteriore assemblatore, scritto in C, che dovrebbe essere abbastanza modulare per supportare eventualmente tutte le sintassi ed i formati di oggetto conosciuti.

3.4.1 Dove trovare NASM

`http://www.cryogen.com/Nasm`

Le release binarie, nel vostro solito mirror di sunsite, sotto `devel/lang/asm/`. Dovrebbero inoltre essere disponibili come .rpm o .deb nei contrib delle vostre distribuzioni RedHat/Debian.

3.4.2 Cosa fa

Nel momento in cui questo HOWTO viene scritto, la versione corrente di NASM è 0.96.

La sintassi è in stile Intel. È integrato del supporto per le macro. I formati di file oggetto supportati sono `bin`, `aout`, `coff`, `elf`, `as86`, (DOS) `obj`, `win32`, `rdf` (il loro formato specifico).

NASM può essere usato come backend per il compilatore libero LCC (sono inclusi i file di supporto).

Di certo NASM si evolve troppo rapidamente perché questo HOWTO possa essere aggiornato. A meno che voi stiate usando BCC come compilatore a 16 bit (il che esula dagli scopi di questo HOWTO sulla programmazione a 32 bit), dovreste usare NASM invece di, ad esempio, ASM o MASM, perché è attivamente supportato online e gira su tutte le piattaforme.

Nota: con NASM trovate anche un disassemblatore, NDISASM.

Il suo parser scritto a mano lo rende molto più veloce di GAS anche se, ovviamente, non supporta tre fantastiliardi di architetture differenti. Se volete generare codice per x86, dovrebbe essere l'assemblatore da scegliere.

3.5 AS86

AS86 è un assemblatore 80x86 a 16 e 32 bit, parte del compilatore C di Bruce Evans (BCC). Segue fondamentalmente la sintassi Intel, anche se ne discosta leggermente per quanto riguarda le modalità di indirizzamento.

3.5.1 Dove procurarsi AS86

Una versione decisamente superata di AS86 è distribuita da HJLu semplicemente per compilare il kernel di Linux, in un pacchetto chiamato bin86 (versione corrente: 0.4), disponibile in ogni archivio di GCC per Linux. Tuttavia non consiglio a nessuno di usarlo per qualcosa che non sia compilare Linux. Questa versione supporta solo una versione modificata del formato per file oggetto di minix, che non è supportata dalle binutils GNU o altro e che ha qualche bug nel modo a 32 bit, quindi fareste proprio meglio a tenerla solo per compilare Linux.

Le versioni più recenti realizzate da Bruce Evans (bde@zeta.org.au) sono pubblicate assieme alla distribuzione FreeBSD. Beh, lo erano: non sono riuscito a trovare i sorgenti dalla distribuzione 2.1 in poi :(Quindi, metto i sorgenti da me:

`http:///www.eleves.ens.fr:8080/home/rideau/files/bcc-95.3.12.src.tgz`

Il progetto Linux/8086 (conosciuto anche come ELKS) sta in qualche modo mantenendo bcc (anche se non credo che abbiano incluso le patch per i 32 bit). Date un'occhiata dalle parti di

`http://www.linux.org.uk/Linux8086.html`

`ftp://linux.mit.edu/` .

Tra l'altro, queste versione più recenti, contrariamente a quelle di HJLu, supportano il formato GNU a.out per Linux, cosicché è possibile il linking tra il vostro codice ed i programmi Linux e/o l'utilizzo dei soliti strumenti dal pacchetto GNU binutil per manipolare i vostri dati. Questa versione può coesistere senza alcun danno con quella precedente (vedere la domanda relativa in seguito).

BCC, versione del 12 marzo 1995 e precedenti, esegue i push ed i pop di segmenti soltanto a 16 bit, il che è non poco seccante quando si programma nel modo a 32 bit.

Una patch è disponibile nel progetto Tunes

`http://www.eleves.ens.fr:8080/home/rideau/Tunes/`

alla sottopagina `files/tgz/tunes.0.0.0.25.src.tgz` nella directory decompressa `LLL/i386/` La patch dovrebbe anche essere disponibile direttamente da

`http://www.eleves.ens.fr:8080/home/rideau/files/as86.bcc.patch.gz`

Bruce Evans ha accettato questa patch, così se un giorno ci sarà da qualche parte una versione più recente di bcc, la patch dovrebbe essere stata inclusa.

3.5.2 Come invocare l'assemblatore?

Ecco la voce nel Makefile di GNU per usare bcc allo scopo di trasformare asm `.s` in oggetto `.o` ed ottenere un listato `.l` :

```
%.o %.l:        %.s
        bcc -3 -G -c -A-d -A-l -A$*.l -o $*.o $<
```

Togliete `%.l`, `-A-l` e `-A$*.l` se non volete alcun listato. Se volete qualcos'altro invece di un a.out GNU, potete consultare la documentazione di bcc circa gli altri formati supportati e/o utilizzare objcopy dal pacchetto delle GNU binutils.

3.5.3 Dove trovare documentazione

La documentazione è quella che è inclusa nel pacchetto bcc. Da qualche parte nel sito di FreeBSD sono inoltre disponibili le pagine di manuale. In caso di dubbi, i sorgenti stessi sono spesso una buona documentazione:

non è molto ben commentata, ma lo stile di programmazione è chiaro. Potreste provare a vedere come as86 è utilizzato in Tunes 0.0.0.25...

3.5.4 E se non riesco più a compilare Linux con questa nuova versione?

Linus è sepolto vivo nella posta e la mia patch per compilare Linux con as86 a.out per Linux non ce l'ha fatta ad arrivargli (!). Ora, questo non dovrebbe avere importanza: limitatevi a tenere il vostro as86 dal pacchetto bin86 in /usr/bin e lasciate che bcc installi l'as86 buono come /usr/local/libexec/i386/bcc/as dove dovrebbe risiedere. Non avrete mai bisogno di chiamare esplicitamente questo as86 «buono», perché bcc fa tutto come si deve, compresa la conversione dal formato a.out di Linux quando viene invocato con le opzioni corrette; limitatevi perciò ad assemblare usando bcc come frontend, non fatelo direttamente con as86.

3.6 ALTRI ASSEMBLATORI

Queste sono altre scelte possibili, non convenzionali, nel caso in cui le precedenti non vi abbiano soddisfatto (perché?). Non le consiglio nei casi comuni (?), ma potrebbero rivelarsi molto utili se l'assemblatore deve essere integrato nel software che state progettando (ad esempio un sistema operativo o un ambiente di sviluppo).

3.6.1 L'assemblatore Win32Forth

Win32Forth è un sistema ANS FORTH a 32 bit *libero* che gira sotto Win32s, Win95, Win NT. Include un assemblatore libero a 32 bit (con sintassi prefissa o postfissa) integrato nel linguaggio FORTH. La gestione delle macro è realizzata con la piena potenza del linguaggio FORTH; comunque, l'unico contesto di input e di output supportato è Win32Forth stesso (nessuna creazione di file .obj ; certo, potreste aggiungerla voi stessi). Lo trovate qui:

`ftp://ftp.forth.org/pub/Forth/win32for/`

3.6.2 Terse

Terse è uno strumento di programmazione con *LA* sintassi dell'assemblatore più compatta per la famiglia x86!

Vedere `http://www.terse.com` . Si dice che ce ne sia un clone libero da qualche parte. Sarebbe stato abbandonato in seguito a pretese infondate secondo le quali la sintassi apparterrebbe all'autore originale. Vi invito a continuarne lo sviluppo, nel caso la sintassi vi interessi.

3.6.3 Assemblatori non liberi e/o non a 32 bit.

Potete trovare più informazioni a riguardo, assieme alle basi della programmazione assembly per x86, nelle FAQ di Raymond Moon per comp.lang.asm.x86

`http://www2.dgsys.com/~raymoon/faq/asmfaq.zip`

Va notato che tutti gli assemblatori che si basano sul DOS dovrebbero funzionare sotto l'emulatore di DOS per Linux ed altri emulatori analoghi cosicché, se ne possedete già uno, potete continuare ad usarlo in un vero sistema operativo. Alcuni assemblatori recenti per DOS supportano anche COFF e/o altri formati di file oggetto che sono supportati dalla libreria GNU BFD, così potete usarli insieme ai vostri strumenti liberi a 32 bit, magari usando GNU objcopy (che fa parte delle binutils) come filtro di conversione.

4 METAPROGRAMMAZIONE E MACRO

La programmazione in assembly è una scocciatura, tranne che per sezioni critiche dei programmi.

Dovreste usare gli strumenti appropriati per il compito giusto, quindi non scegliete l'assembly quando non è adatto; C, OCAML, perl, Scheme potrebbero essere una scelta migliore per la maggior parte della vostra programmazione.

Comunque, ci sono casi in cui questi strumenti non consentono un controllo sufficientemente accurato della macchina e l'assembly risulta utile o addirittura necessario. In questi casi, vi sarà utile un sistema di macro e metaprogrammazione che permetterà di ridurre ogni struttura ricorrente in una definizione riutilizzabile molte volte, il che consente una programmazione più sicura, la propagazione automatica delle modifiche alla struttura stessa, ecc.

Un assemblatore «liscio» è spesso insufficiente, anche quando si sviluppano semplicemente piccole routine per un linking con il C.

4.1 Cosa è integrato nei pacchetti menzionati

Sì, lo so che questa sezione non contiene molte informazioni aggiornate. Sentitevi liberi di contribuire ciò che scoprite sulla vostra pelle...

4.1.1 GCC

GCC permette (e richiede) che voi specifichiate vincoli sui registri nel vostro codice assembly inline cosicché l'ottimizzatore ne è sempre al corrente; perciò il codice assembly inline, in realtà, non è per forza costituito da codice esatto, ma da strutture.

Potete poi inserire il vostro assembly in macro di CPP ed in funzioni C inline, così si può usare come una qualsiasi macro o funzione C. Le funzioni inline assomigliano molto alle macro, ma il loro uso risulta talvolta più pulito. Badate che in tutti questi casi si avrà una duplicazione di codice, quindi in questo codice asm dovrebbero essere definite solo etichette locali (del tipo `1:`). Comunque, una macro permetterebbe che il nome di una etichetta non definita localmente venga passato come parametro (oppure utilizzate metodi aggiuntivi di metaprogrammazione). Inoltre, propagare codice asm inline diffonderà potenziali bug presenti in esso, state quindi doppiamente attenti ai vincoli sui registri in situazioni simili.

Infine, il linguaggio C stesso può essere considerato una buona astrazione rispetto al linguaggio assembly, il che vi solleva dalla maggior parte dei problemi legati all'assembling.

Fate attenzione: alcune ottimizzazioni riguardanti il passaggio di argomenti a funzioni tramite registri possono rendere dette funzioni inadatte ad essere chiamate da routine esterne (ed in particolare da quelle scritte a mano in assembly) nel modo standard; l'attributo «asmlinkage» potrebbe impedire ad una routine di preoccuparsi di tale flag di ottimizzazione; guardatevi i sorgenti del kernel per gli esempi.

4.1.2 GAS

GAS fornisce del supporto per le macro, come è spiegato in dettaglio nella documentazione texinfo. GCC, oltra a riconoscere i file .s come assembly «grezzo» da inviare a GAS, riconosce anche i file .S come file da passare attraverso CPP prima di mandarli a GAS. Ancora una volta, guardate i sorgenti di Linux per gli esempi.

4.1.3 GASP

Aggiunge tutti i tipici trucchetti delle macro a GAS. Consultate la sua documentazione texinfo.

4.1.4 NASM

Anche NASM ha del supporto per le macro. Date un'occhiata alla documentazione relativa. Se avete qualche idea brillante, potreste voler contattare gli autori, dal momento che lo stanno sviluppando attivamente. Nel frattempo, date un'occhiata ai filtri esterni presentati in seguito.

4.1.5 AS86

Ha un po' di semplice supporto per le macro, ma non sono riuscito a trovare della documentazione. I sorgenti sono molto chiari perciò se siete interessati dovreste capirli facilmente. Se le capacità di base non vi bastano, dovreste usare un filtro esterno (vedere in seguito).

4.1.6 ALTRI ASSEMBLATORI

- Win32FORTH: CODE ed END-CODE sono delle macro che non commutano dal modo interpretazione al modo compilazione, cosicché avete pieno accesso alla potenza del FORTH durante l'assembling.
- TUNES: non funziona ancora, ma Scheme è un vero linguaggio ad alto livello che consente metaprogrammazione arbitraria.

4.2 Filtri esterni

Qualunque sia il supporto per le macro del vostro assemblatore, o qualunque linguaggio utilizziate (perfino il C!), se secondo voi il linguaggio non è sufficientemente espressivo, potete far passare i file attraverso un filtro esterno con una regola come questa nel Makefile:

```
%.s:    %.S altre_dipendenze
        $(FILTRO) $(OPZIONI_DEL_FILTRO) < $< > $@
```

4.2.1 CPP

CPP non è molto espressivo, ma è sufficiente per le cose facili, è standard ed è chiamato in modo trasparente da GCC.

Per fare un esempio delle sue limitazioni, non si possono dichiarare oggetti in modo tale che i distruttori vengano chiamati automaticamente al termine del blocco di dichiarazione; non avete diversion o regole di visibilità (scoping), ecc.

CPP si trova assieme ad ogni compilatore C. Se ve la siete cavata senza averne uno, non preoccupatevi di procurarvi GCC (anche se mi chiedo come abbiate fatto).

4.2.2 M4

M4 vi dà tutta la potenza delle macro, con un linguaggio Turing-equivalente, ricorsione, espressioni regolari, ecc. Potete farci tutto ciò che CPP non riesce a fare.

Date un'occhiata a macro4th/This4th da

ftp://ftp.forth.org/pub/Forth/ in Reviewed/ ANS/ (?), o le sorgenti di Tunes 0.0.0.25 come esempi di un utilizzo avanzato delle macro con m4. Comunque, il suo scomodo sistema di protezione dall'espansione vi obbliga ad utilizzare per le macro uno stile basato sulla ricorsione in coda con passaggio di continuazione esplicito (explicit continuation-passing) se volete una uso *avanzato* delle macro (il che ricorda TeX. A proposito, qualcuno ha provato ad usare TeX per le macro di qualcosa di diverso dalla composizione tipografica?). Questo comunque NON è peggio di CPP, il quale non permette né il quoting né la ricorsione. La versione giusta di m4 da procurarsi è GNU m4 1.4 (o successiva, se esiste), la quale ha il maggior numero di funzionalità ed il minor numero di bug o limitazioni rispetto alle altre. m4 è progettato per essere lento per tutto tranne gli utilizzi più semplici, il che potrebbe andare ancora bene per la maggior parte dei programmi assembly (non state scrivendo programmi in assembly da milioni di righe, vero?).

4.2.3 Macro con i vostri filtri

Potete scrivervi dei semplici filtri per l'espansione delle macro con i soliti strumenti: perl, awk, sed, ecc. Si fa in fretta ed avete il controllo su tutto. Ma ovviamente, la potenza nella gestione delle macro bisogna guadagnarsela con fatica.

4.2.4 Metaprogrammazione

Invece di usare un filtro esterno che espande le macro, un modo alternativo di procedere è quello di scrivere programmi che scrivono parti di altri programmi (o interi programmi).

Ad esempio, potreste utilizzare un programma che dia in uscita codice sorgente

- per generare tavole precalcolate di seno/coseno/quant'altro,
- per estrarre una rappresentazione in forma sorgente di un file binario,
- per inserire le vostre bitmap nel codice compilato di routine di visualizzazione rapide,
- per estrarre documentazione, codice di inizializzazione/conclusione, tabelle di descrizione così come codice ordinario dagli stessi file sorgenti,
- per avere codice assembly personalizzato, generato da uno script perl/shell/scheme che si occupa di elaborazione generica.
- per propagare dati definiti in un punto solo verso vari pezzi di codice e tabelle con riferimenti incrociati.
- ecc.

Pensateci!

Backend da compilatori esistenti Compilatori come SML/NJ, Objective CAML, MIT-Scheme, ecc. hanno il loro backend generico per gli assemblatori che potreste o meno voler usare se intendete generare semiautomaticamente del codice partendo dai linguaggi relativi.

Il New-Jersey Machine-Code Toolkit C'è un progetto per costruire, usando il linguaggio di programmazione Icon, una base per produrre codice che manipola l'assembly. Date un'occhiata dalle parti di

http://www.cs.virginia.edu/~nr/toolkit/

Tunes Il progetto Tunes OS sta sviluppando il suo assemblatore come un'estensione al linguaggio Scheme, come parte del suo processo di sviluppo. Al momento non è ancora in grado di girare, tuttavia ogni aiuto è bene accetto.

L'assemblatore manipola alberi di sintassi simbolici, così può ugualmente servire come base per un traduttore di sintassi assembly, un disassemblatore, un backend comune per assemblatore/compilatore, ecc. Inoltre tutta la potenza di un vero linguaggio, Scheme, lo rende insuperato per quanto riguarda le macro e la metaprogrammazione.

`http://www.eleves.ens.fr:8080/home/rideau/Tunes/`

5 CONVENZIONI DI CHIAMATA

5.1 Linux

5.1.1 Linking a GCC

È il modo preferito. Controllate la documentazione di GCC e gli esempi dai file `.S` del kernel di Linux che passano attraverso gas (non quelli che passano attraverso as86).

Gli argomenti a 32 bit vengono posti sullo stack (push) in ordine inverso rispetto alla sintassi (per cui vi si accede prelevandoli nell'ordine corretto) sopra l'indirizzo di ritorno a 32 bit. `%ebp`, `%esi`, `%edi`, `%ebx` sono salvati dalla funzione chiamata, gli altri dalla funzione chiamante; per contenere il risultato si usa `%eax`, oppure `%edx:%eax` per risultati a 64 bit.

Stack FP: non ne sono certo, ma penso che il risultato vada in `st(0)` e che l'intero stack sia salvato dalla funzione chiamante. Notate che GCC ha delle opzioni per modificare le convenzioni di chiamata riservando registri, per mettere argomenti nei registri, per non fare supposizioni sulla presenza dell'FPU, ecc. Controllate le pagine .info i386 .

Fate attenzione: in questo caso dovete dichiarare l'attributo `decl` per una funzione che seguirà le convenzioni di chiamata standard di GCC (non so cosa faccia con le convenzioni di chiamata modificate). Leggete le pagine info di GCC nella sezione: `C Extensions::Extended Asm::`

5.1.2 ELF ed a.out : problemi.

Alcuni compilatori C antepongono un underscore prima di ogni simbolo, mentre altri non lo fanno.

In particolare, GCC a.out per Linux effettua questa anteposizione, mentre GCC ELF per Linux no.

Se avete bisogno di gestire insieme entrambi i comportamenti, guardate come fanno i pacchetti esistenti. Ad esempio, procuratevi dei vecchi sorgenti di Linux, Elk, qthreads, o OCAML...

Potete inoltre far ignorare la rinominazione implicita C->asm inserendo istruzioni come

```
        void pippo asm("pluto") (void);
```

per essere sicuri che la funzione C «pippo» venga chiamata davvero «pluto» in assembly.

Notate che il programma di utilità `objcopy`, dal pacchetto `binutils`, dovrebbe permettervi di trasformare i vostri oggetti a.out in oggetti ELF e forse anche viceversa, in alcuni casi. Più in generale, effettuerà un gran numero di conversioni di formato dei file.

5.1.3 Linux: chiamate di sistema dirette

Ciò è espressamente *NON* consigliato, poiché le convenzioni cambiano di tanto in tanto o tra varianti del kernel (vedere L4Linux), inoltre si perde la portabilità, comporta un lavoro di scrittura massiccio, si ha ridondanza con gli sforzi di libc ED INOLTRE preclude estensioni e correzioni apportate a libc come, ad esempio, il pacchetto `zlibc`, che provvede ad una trasparente decompressione «al volo» di file compressi con gzip. Il metodo convenzionale e consigliato per chiamare i servizi di sistema di Linux è, e rimarrà, quello di passare attraverso la libc.

Gli oggetti condivisi dovrebbero mantenere la vostra roba entro dimensioni contenute. E se proprio volete binari più piccoli, utilizzate `#!` e scaricate sull'interprete il fardello che volete togliere dai vostri binari.

Ora, se per qualche ragione non volete fare un link alla libc, procuratevi la libc stessa e capite come funziona! Dopotutto, aspirate a sostituirla, no?

Potreste inoltre dare un'occhiata a come il mio

eforth 1.0c `ftp://ftp.forth.org/pub/Forth/Linux/linux-eforth-1.0c.tgz`

lo fa.

Anche i sorgenti di Linux tornano utili, in particolare l'header file asm/unistd.h, che descrive come effettuare le chiamate di sistema...

Fondamentalmente, generate un `int $0x80`, con il numero associato a `__NR_`nomedellasyscall (da `asm/unistd.h`) in `%eax`, ed i parametri (fino a cinque) in `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi` rispettivamente. Il risultato viene restituito in `%eax`, dove un numero negativo è un errore il cui opposto è ciò che libc metterebbe in errno. Lo stack utente non viene toccato, così non è necessario averne uno valido quando effettuate una chiamata di sistema.

5.1.4 I/O sotto Linux

Se volete effettuare dell'I/O sotto Linux, o si tratta di qualcosa di molto semplice che non richiede l'arbitraggio del sistema operativo, e allora dovreste consultare l'`IO-Port-Programming mini-HOWTO`, oppure ha bisogno di un driver di periferica nel kernel, nel qual caso dovreste provare ad approfondire le vostre conoscenze sull'hacking del kernel, sullo sviluppo di driver di periferica, sui moduli del kernel, ecc., per i quali ci sono altri eccellenti HOWTO e documenti del LDP.

In particolare, se ciò che vi interessa è la programmazione della grafica, allora partecipate al progetto GGI:

`http://synergy.caltech.edu/~ggi/`

`http://sunserver1.rz.uni-duesseldorf.de/~becka/doc/scrdrv.html`

Comunque, in tutti questi casi, fareste meglio ad usare l'assembly inline di GCC con le macro da linux/asm/*.h piuttosto che scrivere file sorgenti completamente in assembly.

5.1.5 Accedere a driver a 16 bit da Linux/i386

Ciò è teoricamente possibile (dimostrazione: guardate come DOSEMU riesca a garantire ai programmi un accesso a porte hardware in modo selettivo), ed ho anche sentito voci secondo le quali qualcuno da qualche parte ci sarebbe di fatto riuscito (nel driver PCI? Roba per l'accesso VESA? ISA PnP? Non so). Se avete informazioni più precise a riguardo, siate i benvenuti. Comunque, buoni posti in cui cercare maggiori informazioni sono i sorgenti del kernel di Linux, i sorgenti di DOSEMU (ed altri programmi nel

DOSEMU repository `ftp://tsx-11.mit.edu/pub/linux/ALPHA/dosemu/`), e sorgenti di vari programmi a basso livello sotto Linux... (forse GGI se supporta VESA). Fondamentalmente, dovete usare la modalità protetta a 16 bit o il modo vm86.

Il primo è più semplice da mettere in piedi, ma funziona solamente con codice «educato» (well-behaved) che non utilizza l'aritmetica dei segmenti o indirizzamento assoluto degli stessi (segmento 0 in particolare), a meno che non ci si trovi nel caso in cui tutti i segmenti utilizzati possano essere preparati in anticipo nella LDT.

Il secondo permette più «compatibilità» con i comuni ambienti a 16 bit, ma richiede una gestione più complessa.

In entrambi i casi, prima di poter saltare al codice a 16 bit, dovete

- fare un mmap di tutti gli indirizzi assoluti utilizzati nel codice a 16 bit (come ROM, buffer video, zone su cui agirà il DMA ed I/O memory-mapped) da /dev/mem allo spazio indirizzi del vostro processo.
- preparare la LDT e/o il monitor per il modo vm86
- ottenere gli opportuni permessi di I/O dal kernel (vedere sopra)

Ancora una volta, leggete con cura i sorgenti dei contributi al DOSEMU repository menzionato sopra, in particolare quei mini-emulatori per far girare ELKS e/o semplici programmi .COM sotto Linux/i386.

5.2 DOS

La maggior parte dei DOS extender viene fornita con dell'interfacciamento a servizi DOS. Leggete la loro documentazione a riguardo, ma spesso si limitano a simulare `int $0x21` e simili, così vi comportate «come se» foste in modo reale (dubito che abbiano qualcosa di più di stub ed estensioni per lavorare con operandi a 32 bit; molto probabilmente si limiteranno a riportare l'interrupt nel gestore del modo reale o del vm86).

Documentazione su DPMI e affini (e molto più) può essere trovata in

`ftp://x2ftp.oulu.fi/pub/msdos/programming/`

Anche DJGPP viene fornito con il proprio derivato/sottoinsieme/sostituto di glibc.

È possibile la compilazione incrociata da Linux a DOS, guardate nella directory devel/msdos/ del vostro mirror locale dell'area FTP di sunsite.unc.edu Date anche un'occhiata al DOS extender «MOSS» dal progetto Flux in Utah.

Altri documenti e FAQ sono molto DOS-centrici. Noi non consigliamo di sviluppare per DOS.

5.3 Winzozz e compagnia bella

Ehi, questo documento riguarda solo il software libero. Telefonatemi quando Winzozz diventa libero, o ci si possono usare strumenti di sviluppo liberi!

Beh, dopotutto ci sono:

Cygnus Solutions `http://www.cygnus.com`

ha sviluppato la libreria cygwin32.dll per far girare i programmi GNU su piattaforme Micrashoft. Così potete usare GCC, GAS, tutti gli strumenti di GNU e molte altre applicazioni UNIX. Date un'occhiata alla loro homepage. Io (Faré) non intendo dilungarmi sulla programmazione di WinnaNanna ma sono certo che potete trovare un sacco di documentazione a riguardo praticamente ovunque...

5.4 Un sistema operativo tutto vostro.

Essendo il controllo ciò che attrae molti programmatori all'assembly, il desiderio di sviluppare sistemi operativi è spesso ciò che li porta all'(o deriva dall') hacking in assembly. Va notato che ogni sistema che permette lo sviluppo di se stesso potrebbe essere considerato un «sistema operativo», anche se magari gira «sopra» ad un sistema sottostante che fornisce multitasking o I/O (in modo molto simile a Linux sopra Mach o OpenGenera sopra UNIX), ecc.

Quindi, per rendere più facile il debugging, potreste voler sviluppare il vostro «sistema operativo» prima come processo che gira sopra a Linux (nonostante la lentezza), quindi usare il

Flux OS kit `http://ww.cs.utah.edu/projects/flux/`

(che consente l'utilizzo di driver di Linux e BSD nel vostro OS) per renderlo autonomo. Quando il vostro sistema è stabile, resta ancora del tempo per scrivere dei driver per l'hardware tutti vostri, se la cosa vi fa proprio piacere.

Questo HOWTO non si occuperà di argomenti quali il codice per il boot loader ed entrare nel modo a 32 bit, gestire gli interrupt, i fondamenti degli orrori intel «modo protetto» o «V86/R86», la definizione di un vostro formato per i file oggetto e le convenzioni di chiamata. Il luogo principale in cui trovare informazioni attendibili a riguardo sono i sorgenti di sistemi operativi o bootloader già esistenti. Ci sono un sacco di riferimenti in questa pagina WWW:

`http://www.eleves.ens.fr:8080/home/rideau/Tunes/Review/OSes.html`

6 COSE DA FARE E RIFERIMENTI

- riempire le sezioni incomplete.
- aggiungere ulteriori riferimenti a software e documentazione.
- aggiungere esempi facili dal mondo reale per illustrare la sintassi, la potenza e le limitazioni di ogni soluzione proposta.
- chiedere aiuto in giro per questo HOWTO.
- trovare qualcuno che ha del tempo per subentrarmi nel mantenimento.
- che sia il caso di spendere qualche parola sull'assembly su altre piattaforme?
- Alcuni riferimenti (in aggiunta a quelli già presenti nel resto dell'HOWTO)
  - *i manuali del pentium* `http://www.intel.com/design/pentium/manuals/`
  - *bug nelle CPU della famiglia x86* `http://www.xs4all.nl/~feldmann`
  - *hornet.eng.ufl.edu per programmatori assembly* `http://www.eng.ufl.edu/ftp`
  - *ftp.luth.se* `ftp://ftp.luth.se/pub/msdos/demos/code/`
  - *FAQ del modo protetto* `ftp://zfja-gate.fuw.edu.pl/cpu/protect.mod`
  - *Pagina dell'assembly 80x86* `http://www.fys.ruu.nl/~faber/Amain.html`
  - *Courseware* `http://www.cit.ac.nz/smac/csware.htm`
  - *programmazione di giochi* `http://www.ee.ucl.ac.uk/~phart/gameprog.html`
  - *esperimenti di programmazione solo in assembly sotto Linux* `http://bewoner.dma.be/JanW`
- E ovviamente, utilizzate i vostri strumenti di ricerca su Internet per cercare ulteriori informazioni e ragguagliatemi su qualunque cosa interessante troviate!

.sig dell'autore:

```
--    ,                                         ,           _ v    ~  ^  --
-- Fare -- rideau@clipper.ens.fr -- Francois-Rene Rideau -- +)ang-Vu Ban --
--                                      ,                   / .          --
Join the TUNES project for a computing system based on computing freedom !
                 TUNES is a Useful, Not Expedient System
WWW page at URL: http://www.eleves.ens.fr:8080/home/rideau/Tunes/
```