

x86 ASM Programming for Linux – THE.UNIX.WORLD

Pubblicato da *mammon_* il 11/1998

Livello *base*

Introduzione

/* Questo articolo e' stato tratto dall'Assembly Programming Journal numero 1 e tradotto da Little-John */

```

::/ \:.....
:/___\:.....
/|   \:.....
:|   _/\:.....
:|   _|\ \:.....
::\___\:.....THE.UNIX.WORLD
    
```

Iniziamo

Essenzialmente questo articolo e' una scusa per conciliare i miei due interessi favoriti di coding: il sistema operativo Linux e la programmazione in linguaggio assembly. Tutti e due gli argomenti non necessitano (meglio, non dovrebbero) di una introduzione; come l'assembly Win32, assembly per Linux e' eseguito in protected mode 32-bit... comunque ha il netto vantaggio di permetterti di chiamare le funzioni delle librerie standard C come ogni altra funzione delle normali librerie Linux "condivise". Ho cominciato con una breve introduzione sulla compilazione dei programmi in assembly language per Linux; per una migliore leggibilita' potresti bypassarla e andare direttamente alla sezione su "Le Basi".

Compiling e Linking

I due assemblers principali per Linux sono Nasm, l'Assembler (gratis) di Netwide, e GAS, l'Assembler (pure gratis) di Gnu, integrato in GCC. Mi concentrero' su Nasm in questo articolo, lasciando GAS per un altro giorno dal momento che usa la sintassi AT&T e cio' richiederebbe una introduzione piu' prolissa. Nasm dovrebbe essere azionato con l'opzione di formato ELF ("nasm -f elf hello.asm"); l'object che ne deriva e' poi linkato con GCC ("gcc hello.o") per creare il binario ELF finale. Lo script seguente puo' essere usato per compilare moduli ASM; l'ho scritto in modo che sia molto semplice, quindi tutto cio' che fa e' prendere il primo filename passatogli (io consiglio di chiamarlo con una estensione ".asm"), lo compila con nasm, e lo linka con gcc.

```

#!/bin/sh
# assemble.sh =====
outfile=${1%.*}
tempfile=asmtemp.o
nasm -o $tempfile -f elf $1
gcc $tempfile -o $outfile
rm $tempfile -f
#EOF =====
    
```

Le Basi

La cosa migliore per partire, naturalmente, e' un esempio, prima di immergerci nei dettagli dell'OS. Ecco qui un programma "hello-world" davvero semplice:

```

; asmhello.asm =====
global main
extern printf

section .data
msg db "Helloooooo, nurse!", 0Dh, 0Ah, 0

section .text
main:
    push dword msg
    call printf
    pop eax
    ret
; EOF =====
    
```

Una spiegazione veloce: il "global main" deve essere dichiarato global—e dal momento che stiamo usando il linker GCC, l'entrypoint deve essere chiamato "main"—per il loader dell'OS.

L'"extern printf" e' semplicemente una dichiarazione per la call successiva nel programma; nota che questo e' tutto il necessario; non e' necessario dichiarare le dimensioni dei parametri. Ho diviso questo esempio nelle sezioni standard .data e .text, sebbene cio' non sia strettamente necessario —chiunque potrebbe svignarsela con il solo segmento .text, proprio come in DOS.

Nel corpo del codice, nota che devi pushare i parametri alla call, e in Nasm devi dichiarare la dimensione di tutti i dati ambigui (p.e. non-register): di qui il qualificatore "dword". Nota che come in altri assembleri, Nasm assume che ogni reference memory/label e' volta a significare l'indirizzo della locazione di memoria o della label, non il loro contenuto. Percio', per specificare l'indirizzo della stringa 'msg' scriveresti 'push dword msg', mentre per specificare il contenuto della stringa 'msg' scriveresti 'push dword [msg]' (nota che questo conterra' solo i primi 4 bytes di 'msg'). Dal momento che printf richiede un pointer alla string, specificheremo l'indirizzo di 'msg'.

La call a printf e' abbastanza lineare. Considera che pulire lo stack dopo ogni call che esegui (vedi sotto); quindi, avendo PUSHato una dword, POPpiamo una dword dallo stack in un registro "da cestinare". I programmi Linux si chiudono semplicemente con una RET all'OS, dato che ogni processo e' aperto dalla shell (o PID 1 ;) e finisce restituendogli il controllo.

Nota che in Linux fai uso delle librerie standard condivise fornite con l'OS in luogo di una "API" di degli Interrupt Services. Tutte le reference esterne saranno risolte dal linker GCC, in modo da alleggerire buona parte del carico di lavoro del programmatore asm. Una volta che ti sei abituato alle stranezze di base, il coding in assembler in Linux e' davvero piu' semplice di quello su una macchina DOS-based!

La sintassi di chiamata C

Linux usa la convenzione di chiamata C — cio' significa che gli argomenti sono pushati nello stack in ordine inverso (l'ultimo arg per primo), e che il caller deve pulire lo stack. Puoi far cio' o poppando i valori dallo stack:

```
push dword szText
call puts
pop ecx
```

o modificando direttamente ESP:

```
push dword szText
call puts
add esp, 4
```

I valori restituiti dalla call si trovano in eax o in edx:eax se il valore e' piu' grande di 32-bit. EBP, ESI, EDI, e EBX sono tutti salvati e ripristinati dal caller.

Nota che devi conservare tutti i registri che usi come illustra il codice seguente:

```
; loop.asm =====
global main
extern printf

section .text
msg db "HoodooVoodoo WeedooVoodoo", 0Dh, 0Ah, 0

main:
    mov ecx, 0Ah
    push dword msg
looper:
    call printf
    loop looper
    pop eax
    ret
; EOF =====
```

A primo acchito questo sembra molto semplice: dal momento che stai per usare la stringa nelle call 10 printf(), non hai bisogno di ripulire lo stack.

Tuttavia quando lo compili, il loop non si ferma mai. Perche'? Perche' da qualche parte nella call printf() ECX e' usato e non salvato. Quindi per far funzionare il tuo loop a dovere, devi salvare il valore del contatore in ECX prima della call e ripristinarlo dopo, cosi':

```
; loop.asm =====
```

```

global main
extern printf

section .text
msg db "HoodooVoodoo WeedooVoodoo", 0Dh, 0Ah, 0

main:
    mov ecx, 0Ah
looper:
    push ecx ;salva Count
    push dword msg
    call printf
    pop eax ;pulisce lo stack
    pop ecx ;ripristina Count
    loop looper
    ret
; EOF =====

```

Programmazione della Porta I/O

E per avere un accesso diretto all'hardware? In Linux hai bisogno di un driver kernel-mode per fare ogni cosa che sia davvero ingegnosa... cio' significa che il tuo programma finira' per essere di due parti, una kernel-mode che fornisce le funzionalita' direct-hardware, l'altra user-mode per una interface. La buona notizia e' che puoi ancora accedere alle porta usando i comandi IN/OUT da un programma user-mode.

L'accesso alle porte I/O al tuo programma deve essere concesso da un permesso dell'OS; per far cio', devi compiere una call ioperm(). Questa funzione puo' essere chiamata solo da un utente root, quindi devi o setuid() il programma come root oppure eseguire il programma da root. La ioperm() ha la sintassi seguente:

```
ioperm( long StartingPort#, long #Ports, BOOL ToggleOn-Off)
```

dove 'StartingPort#' specifica il numero della prima porta da accedere (0 is port 0h, 40h is port 40h, etc), '#Ports' specifica quante porte accedere (i.e., 'StartingPort# = 30h' e '#Ports = 10' concederebbero l'accesso alle porte 30h-39h), e 'ToggleOn-Off' consente l'accesso se TRUE (1) o lo disabilita se FALSE (0).

Una volta che la call a ioperm() e' compiuta, si puo' accedere alle porte richieste come normal. Il programma puo' chiamare ioperm() un qualsivoglia numero di volte e non ha bisogno di fare un successiva call ioperm() (anche se l'esempio sotto lo fa) [siccome l'OS si curera' di cio'].

```

; io.asm=====
BITS 32
GLOBAL szHello
GLOBAL main
EXTERN printf
EXTERN ioperm

SECTION .data
szText1 db 'Enabling I/O Port Access', 0Ah, 0Dh, 0
szText2 db 'Disabling I/O Port Acess', 0Ah, 0Dh, 0
szDone db 'Done!', 0Ah, 0Dh, 0
szError db 'Error in ioperm() call!', 0Ah, 0Dh, 0
szEqual db 'Output/Input bytes are equal.', 0Ah, 0Dh, 0
szChange db 'Output/Input bytes changed.', 0Ah, 0Dh, 0

SECTION .text

main:
    push dword szText1
    call printf
    pop ecx
enable_IO:
    push word 1 ; enable mode
    push dword 04h ; 4 porte
    push dword 40h ; inizia dalla porta 40
    call ioperm ; Deve essere SUID "root" per questa call!
    add ESP, 10 ; pulisci lo stack (metodo 1)
    cmp eax, 0 ; controlla i risultati di ioperm()
    jne Error

;-----Port Programming Part-----
SetControl:

```

```

    mov al, 96 ; R/W low byte di Counter2, mode 3
    out 43h, al ; porta 43h = control register
WritePort:
    mov bl, 0EEh ; valore da inviare allo speaker timer
    mov al, bl
    out 42h, al ; porta 42h = speaker timer
ReadPort:
    in al, 42h
    cmp al, bl ; il byte dovrebbe essere cambiato--questo E' un timer :)
    jne ByteChanged
BytesEqual:
    push dword szEqual
    call printf
    pop ecx
    jmp disable_IO
ByteChanged:
    push dword szChange
    call printf
    pop ecx
;-----End Port Programming Part-----

disable_IO:
    push dword szText2
    call printf
    pop ecx
    push word 0 ; disable mode
    push dword 04h ; 4 porte
    push dword 40h ; parte dalla porta 40h
    call ioperm
    pop ecx ;pulisci lo stack (metodo 2)
    pop ecx
    pop cx
    cmp eax, 0 ; controlla i risultati di ioperm()
    jne Error
    jmp Exit
Error:
    push dword szError
    call printf
    pop ecx
Exit:
    ret
; EOF =====

```

Usare gli Interrupts In Linux

Linux e' un ambiente shared-library in protected mode, il che significa che non ci sono i servizi interrupt. Giusto? Sbagliato. Ho notato una call a INT 80 sul codice di alcuni esempi GAS con il commento "sys_write(ebx, ecx, edx)". Questa funzione e' parte della syscall dell'interfaccia di Linux, e cioe' l'interrupt 80 deve essere un gate ai servizi di syscall. Girovagando nel codice sorgente di Linux (e ignorando gli avvisi di NON USARE MAI l'interface INT 80 siccome i numeri della funzione potrebbero essere cambiati all'improvviso), ho trovato i "system call numbers" —che indicano la funzione da passare a INT 80 per ogni routine di syscall— nel file UNISTD.H. Ce ne sono 189, quindi non li elencherò qui... ma se ti accingi a programmare in Linux assembly, fa' un favore a te stesso e stampa questo file.

Quando chiami INT 80h, eax deve contenere il numero della funzione desirata. Tutti i parametri alla routine syscall devono trovarsi nei seguenti registri in questo ordine:

```
ebx, ecx, edx, esi, edi
```

quindi il parametro uno si trova in ebx, il parametro 2 in ecx, ecc. Nota non si usa lo stack per passare i valori alla routine syscall. Il risultato della call sara' restituito in eax.

Inoltre, l'interfaccia INT 80 e' uguale ad una normale call (solo un po' piu' divertente ;). Il programma seguente dimostra una semplice call a INT 80h in cui il programma controlla e visualizza la sua PID. Nota l'uso del formato della stringa di printf() —e' meglio psuedocodarlo come una call C prima, poi rendere il formato della stringa DB e pushare ogni variabile passata (%s, %d, ecc). La struttura C per questa call sarebbe printf("%d\n", curr_PID); Nota anche che le sequenze di escape ("\n") non sono tutte davvero attendibili in assembly; ho dovuto usare i valori hex (0Ah,0Dh) per il CR\LF.

```
;pid.asm=====
```

```
BITS 32
```

```

GLOBAL main
EXTERN printf

SECTION .data
szText1 db 'Getting Current Process ID...',0Ah,0Dh,0
szDone db 'Done!',0Ah,0Dh,0
szError db 'Error in int 80!',0Ah,0Dh,0
szOutput db '%d',0Ah,0Dh,0 ;la strana formattazione e' per printf()

SECTION .text

main:
    push dword szText1 ;messaggio di apertura
    call printf
    pop ecx
GetPID:
    mov eax, dword 20 ; getpid() syscall
    int 80h ; syscall INT
    cmp eax, 0 ; non sara' mai PID 0 ! :)
    jb Error
    push eax ; passa il valore restituito a printf
    push dword szOutput ; passa il formato della stringa a printf
    call printf
    pop ecx ; pulisci lo stack
    pop ecx
    push dword szDone ; messaggio di chiusura
    call printf
    pop ecx
    jmp Exit
Error:
    push dword szError
    call printf
    pop ecx
Exit:
    ret
; EOF =====

```

Conclusioni

Ultime considerazioni

Il piu' dei problemi derivera' dall'abituarsi a Nasm stesso. Mentre nasm non e' fornito di una man page, non la installa per default, quindi devi spostarla (cp or mv) da /usr/local/bin/nasm-0.97/nasm.man in /usr/local/man/man1/nasm.man

La formattazione e' un po' incasinata, ma e' facilmente risistemata usando le direttive nroff. Non ti da' ancora tutta la documentazione di Nasm, comunque; per questo, copia nasmdoc.txt da /usr/local/bin/nasm-0.97/doc/nasmdoc.txt in /usr/local/man/man1/nasmdoc.man

Ora puoi chiamare man page di nasm con 'man nasm' e la documentazione di nasm con 'man nasmdoc'.

Per ulteriori informazioni, controlla i seguenti:

- Linux Assembly Language HOWTO
- Linux I/O Port Programming Mini-HOWTO
- Jan's Linux & Assembler HomePage (bewoner.dma.be/JanW/eng.html)

Devo anche dei ringraziamenti a Jeff Weeks alla code^x software (gameprog.com/codex) per avermi inviato un paio di hello-world di GAS nelle giornate nere, prima che trovassi la pagina di Jan.