L' ENTRY POINT DI UN ESEGUIBILE E IL LINKING CON LIBRERIE DI DEFAULT. CHIAMATA AD INTERRUPT SOFTWARE E CHIAMATA A SYSTEM CALLS.

Prof. Vittorio Ghini, Appunti delle lezioni di Sistemi Operativi,
Corso di Laurea in Ingegneria e Scienze Informatiche, Università di Bologna

0. IL FORMATO DEL MODULO OGGETTO E DELL'ESEGUIBILE E IL PROCESSORE SU CUI PUO' ESEGUIRE.

Usando il comando 'objdump -f nomeeseguibile.exe' oppure 'objdump nomefileoggetto.o' posso vedere il formato dell'eseguibile e il processore per cui è stato creato.

Con il comando 'ld -V' mi faccio dire dal linker quali formati di eseguibile è in grado di creare.

1. INIZIO ESECUZIONE DI UN ESEGUIBILE

Quando io lancio l'eseguibile per farlo eseguire, il loader crea il processo (riempiendo tutte le necessarie tabelle di sistema) e carica in memoria l'immagine del processo,
poi passa il controllo al processo caricato e ne fa eseguire la prima istruzione macchina.
Nell'eseguibile, il formato **ELF** prevede una sezione header in cui e' contenuto un campo "**Entry point address**" il cui valore e' **l'indirizzo in cui si trovera' l'istruzione iniziale da eseguire dopo che il processo sara' stato caricato in memoria dal loader**.
Si può lanciare il comando *readelf -h nomefileseguibile* per vedere questo campo.

Nel codice assembly, **la prima istruzione macchina da eseguire è etichettata con la label _start**, che ne rappresenta l'indirizzo.
Quando il linker genera l'eseguibile, cerca l'istruzione etichettata con **_start** e ne pone l'indirizzo nel campo "**Entry point address**" della sezione header.
Se il linker non trova questa label, produce un errore e non genera l'eseguibile.

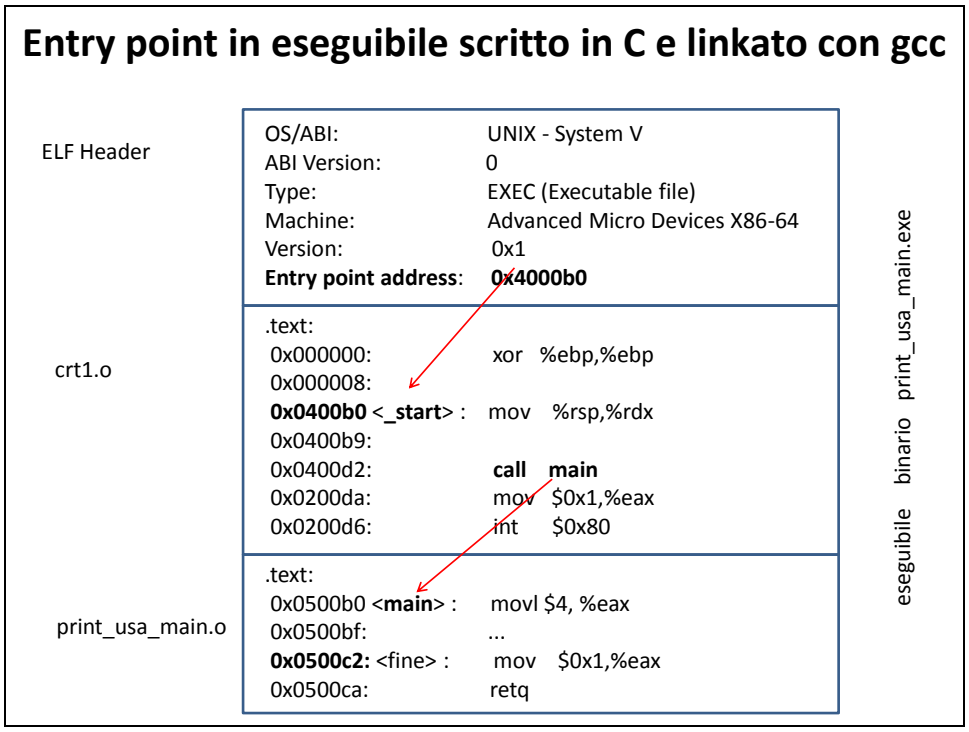
2. LINKING PER CREAZIONE di ESEGUIBILE scritto in LINGUAGGIO C CON gcc

- Quando io scrivo un programma codice in C, il punto di inizio del mio codice e' la funzione main.
- Ma **la prima istruzione che viene eseguita** su ordine del loader, dopo il caricamento in memoria del programma, **non e' quella del main**.

Infatti, prima di poter cominciare ad eseguire il codice del main, occorre svolgere alcune operazioni preliminari, atte a garantire il successivo corretto funzionamento della libc (la libreria standard del C), quali ad esempio impostare il vettore degli interrupt per il processo, inizializzare parte dello stack e altre ancora.
Queste operazioni non sono implementate all'interno del main ma vengono svolte da codice aggiunto autonomamente dal gcc quando effettua l'operazione di linking di moduli scritti in linguaggio C.
Questo codice aggiunto autonomamente dal gcc durante il linking fa parte delle librerie standard del C ed e' "stranamente" contenuto in dei moduli oggetto, ad esempio

/usr/lib/x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/5/crtend.o

- E' dentro questo codice macchina, aggiunto dal gcc al momento del linking, che si trova l'istruzione macchina iniziale etichettata con _start.



Solo dopo avere eseguito il codice aggiunto che inizia in _start il codice stesso passerà il controllo al main effettuando un salto al codice di inizio del main stesso.

Core OS C Runtime Objects (**CRT**)

The `crt1.o`, `crti.o`, and `crtn.o` objects comprise the core CRT (C RunTime) objects required to enable basic C programs to start and run. CRT objects are typically added by compiler drivers when building executables and shared objects.

`crt1.o` provides the `_start` symbol that the runtime linker, `ld.so.1`, jumps to in order to pass control to the executable, and is responsible for providing ABI mandated symbols and other process initialization, for calling `main()`, and ultimately, `exit()`. `crti.o` and `crtn.o` provide prologue and epilogue `.init` and `.fini` sections to encapsulate ELF init and fini code.

`crt1.o` is only used when building executables. `crti.o` and `crtn.o` are used by executables and shared objects.

These CRT objects are compatible with position independent (PIC), and position dependent (non-PIC) code, including both normal and position independent executables (PIE).

Nota bene: il gcc, quando deve eseguire il linking, in realtà lancia ld passandogli alcuni argomenti che indicano quale codice aggiungere e altre informazioni.
Ad esempio, in un sistema Linux su processore x64 o amd, se voglio linkare il mio modulo print_usa_main.o il gcc lancia il linker ld e gli passa i seguenti argomenti:

/usr/bin/ld -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-

fresolution=/tmp/cc7yRp5X.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed **-dynamic-linker /lib64/ld-linux-x86-64.so.2** -z relro -o print_usa_main.exe **/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o** -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/5/../../.. **print_usa_main.o** -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed **/usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crtn.o**

Se guardate li' in mezzo, vedete elencati anche i moduli standard aggiunti dal gcc che vi ho citato prima.

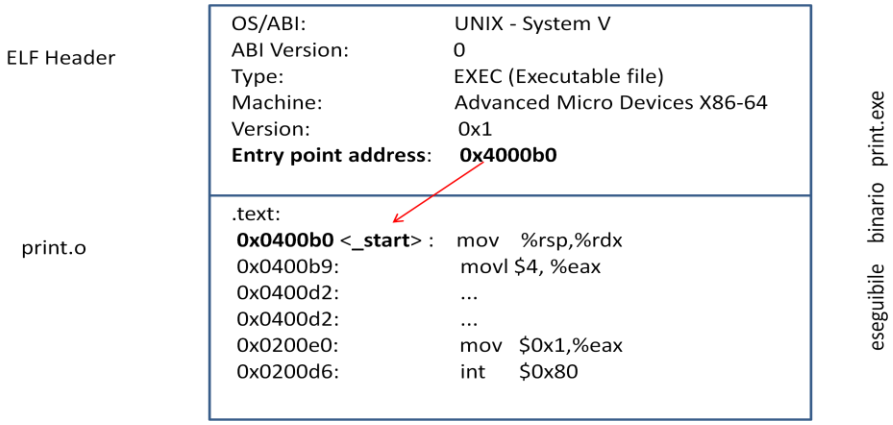
3. LINKING PER CREAZIONE di ESEGUIBILE scritto in ASSEMBLY CON ld

Se invece io scrivo in assembly il codice del mio modulo principale, sono io che devo indicare quale e' la istruzione iniziale etichettata con _start che deve essere eseguita per prima. Sono io che scrivo tutto cio' che deve essere fatto dall'inizio, **il linker non aggiunge niente.**
Nel mio codice assembly quindi **esiste la istruzione iniziale** ed e' indicata dalla etichetta/indirizzo **_start**.

Se scrivo codice assembly, allora per linkare il mio codice macchina devo usare il linker **ld** (usato anche dal gcc stesso) che costruisce l'eseguibile usando il mio codice macchina ma non aggiunge altro codice in autonomia.
Ad esempio, per linkare il mio modulo assembly print.o lancero' il linker ld con questi parametri:

```
ld -o print.exe print.o
```

Entry point in eseguibile scritto in assembly e linkato con ld



4. PROBLEMA nel LINKING con gcc PER CREAZIONE di ESEGUIBILE scritto in ASSEMBLY

Se invece io scrivo in assembly il codice del mio modulo principale, indicando l'istruzione iniziale etichettata con _start e poi linko il mio modulo oggetto chiamando il gcc senza parametri speciali, **il gcc** normalmente **aggiunge dei moduli** aggiuntivi **che contengono** essi stessi **una istruzione iniziale** etichettata con **_start**.

- In tal modo, **durante il linking saranno presenti due istruzioni etichettate con _start ed il linker non sa quale usare come istruzione iniziale da far eseguire**.
  Percio' il linker provoca un errore e non genera l'eseguibile.
  L'errore viene indicato cosi': "multiple definition of `_start"

Inoltre, **nel codice aggiunto dal gcc c'e' un salto alla istruzione del main, ma nel mio codice assembly non ho messo una label main, perche' sono partito da _start.**

- Quindi **il linker non riesce a trovare una etichetta main e percio' non riesce a risolvere il simbolo main che manca**.
  Il linker quindi provoca un altro errore e non genera l'eseguibile.
  L'errore viene indicato cosi': "In function `_start': undefined reference to `main'"

4. SOLUZIONI del PROBLEMA nel LINKING con gcc PER CREAZIONE di ESEGUIBILE scritto in ASSEMBLY

Esistono due modi diversi per risolvere il problema che impedirebbe di linkare con gcc dei moduli scritti in assembly.

4.1 MODO 1 - LINKING con gcc, SENZA LIBRERIE STANDARD, di ESEGUIBILE scritto in ASSEMBLY

Ordino al gcc di fare il linking SENZA USARE LE SUE LIBRERIE STANDARD, che tanto non mi servono perche' io uso solo istruzioni assembly.
Per fare cio' AGGIUNGO NEL COMANDO DI LINKING ordinato al gcc il flag **-nostdlib** che ordina di non usare le librerie standard del C.

```
gcc -nostdlib -o print.exe print.o
```

In tal modo il gcc non aggiunge i moduli aggiuntivi iniziali e quindi non aggiunge l'istruzione etichettata con _start e nemmeno aggiunge il salto verso l'etichetta main.

Risolto.
Nasce il problema che non posso usare le librerie standard del C all'interno del mio codice assembly.

4.2 MODO 2 - LINKING con gcc, CON LIBRERIE STANDARD, di ESEGUIBILE scritto in ASSEMBLY

Modifico il mio codice assembly NON FACENDOGLI FARE la parte iniziale di esecuzione.
Cioè faccio partire il mio codice assembly non da _start bensi' da main, come se fosse un programma scritto in C.

In pratica, nel mio codice assembly principale, al posto delle due righe di codice seguenti:

```
.globl _start
_start:
```

devo sostituire le due righe seguenti:

```
.globl main
main:
```

Poi assemblero' il sorgente assembly con l'assemblatore as o col gcc

```
as -o print.o --gstabs print.s  # assemblaggio diretto con as
```

oppure con

```
gcc -c -g -o print.o print.s    # assemblaggio tramite gcc che chiama as
```

A questo punto posso usare il gcc per linkare normalmente senza dover chiedere di escludere le librerie di default.

```
gcc -o print.exe print.o
```

Infatti, in questo modo, il gcc linker aggiunge il suo codice macchina (che parte da _start e che poi salta al main) e trova nel modulo oggetto originato dall'assembly il simbolo main che indica da dove far partire il codice utente.

In questo modo, nel mio eseguibile mi ritrovo anche le librerie standard e quindi, volendo, potrei richiamare da assembly anche le funzioni C presenti nelle librerie standard, purche' sia io da assembly a preparare correttamente lo stack per la chiamata a funzione C, come farebbe il compilatore, e analogamente a recuperare il risultato e ripulire lo stack dopo la fine della chiamata a funzione.

5. CHIAMATA AD INTERRUPT SOFTWARE

Un programma in assembly, che opera su un processore Intel, può invocare l'esecuzione di un interrupt software chiamando l'istruzione ' **int**' e passandole come argomento un numero che identifica l'interrupt software da eseguire. Ad esempio la seguente istruzione invoca l'interrupt numero 8016 (80 in esadecimale).

int $0x80

La routine di gestione di quell'interrupt può essere quella originariamente contenuta nel BIOS della CPU oppure può essere stata sostituita, dopo il boot, da una diversa routine di gestione stabilita e implementata dal sistema operativo.

Solitamente, se il programma assembly deve passare dei parametri ad un interrupt, i parametri vengono messi in alcuni registri PRIMA di chiamare l'interrupt, e poi la routine di gestione dell'interrupt controlla il contenuto di quei registri e agisce di conseguenza. Ad esempio, nel seguente codice, prima di chiamare l'interrupt 0x80 viene collocato il valore 1 nel registro eax e questo dice all'interrupt che deve eseguire un particolare servizio, cioè la terminazione del processo stesso.

```
movl $1, %eax          /* servizio da ottenere: terminazione processo */
int $0x80              /* ordine di eseguire l'interrupt numero  0x80 */
```

In altri casi, i parametri vengono collocati in un'area di memoria e poi l'indirizzo di inizio viene messo in alcuni registri. Ad esempio, nel seguente codice, prima di chiamare l'interrupt 0x80 viene collocata in un'area di memoria una stringa da stampare e poi l'indirizzo di inizio di questa stringa viene copiata nel registro ecx, mentre la lunghezza della stringa da stampare viene collocata nel registro edx. Solo dopo questo, viene invocato l'interrupt, il quale scatena la routine di gestione che controlla il contenuto di questi registri e si comporta in funzione di quello che trova in questi registri.

```
.data
miastringa: .string "STRUNZ\n" # stringa da stampare con a capo finale
  ...
movl $4, %eax          /* servizio da ottenere: stampa */
movl $1, %ebx          /* sottoservizio da ottenere: stampa da indirizzo */
movl $miastringa, %ecx /*indirizzo di inizio stringa in registro ecx*/
movl $7, %edx          /* lunghezza stringa in registro edx */
int  $0x80             /* ordine di eseguire interrupt */
```

6. CHIAMATA A SYSTEM CALL

Un programma in assembly, che opera su un processore Intel, tipicamente invoca le system call:

- **chiamando l'esecuzione dell'interrupt 0x80**
- e mettendo, prima, nel registro **eax** il numero identificatore della system call da eseguire
- e mettendo in altri registri i parametri necessari a quella system call.

Ad esempio, per stampare a video una stringa a partire da un indirizzo, il programma assembly può invocare l'esecuzione della system call 4 mettendo il valore 4 nel registro EAX (specifica il servizio) ed il valore 1 nel registro EBX (specifica il sottoservizio) poi chiamando l'interrutpt 0x80. Inoltre, prima della chiamata ad interrupt, il programma assembly mette nel registro ECX l'indirizzo di inizio della stringa da stampare e nel registro EDX il numero di byte da stampare. Vedere esempio sopra.

7. LIMITI dell' ISTRUZIONE 'int 0x80' (usare system call) IN PROCESSORI A 64 bit.

L'istruzione 'int 0x80' serve per invocare le system call quando eseguo codice in un processore Intel a 32 bit. Questa istruzione prende argomenti in registri a 32 bit. Ad esempio eax ed ebx per stabilire il servizio da utilizzare, edx per la lunghezza della stringa da stampare ed ecx per l'indirizzo della stringa da stampare.

Se io modifico il codice per eseguirlo su processore a 64 bit, utilizzando i registri a 64 bit rax rbx rcx ed rdx invece dei registri eax ebx ecx ed edx per passare i parametri all'istruzione 'int 0x80' allora possono accadere errori a run-time per due motivi:

a) **l'istruzione 'int 0x80' NON LEGGE TUTTO IL CONTENUTO DEI REGISTRI a 64 bit ma solo i 32 bit meno significativi di quei registri**, cioè la parte che equivale al corrispondente registro a 32 bit. I 32 bit più significativi vengono ignorati. Ciò implica che perdo parte del valore inserito nel registro. Se il registro conteneva un indirizzo a 64 bit, perdo parte dell'indirizzo e quando l'istruzione 'int 0x80' cerca di accedere all'indirizzo sbagliato può provocare un segmentation fault e far killare il processo.

a.1) Come caso particolare del precedente, se l'istruzione 'int 0x80' effettua operazioni di accesso allo stack posso avere problemi legati all'indirizzo della locazione nello stack.

Infatti, l'istruzione 'int 0x80' legge solo indirizzi a 32 bit anche se io li metto in registri a 64 bit. Se la locazione di memoria a cui voglio accedere si trova più in basso dell'indirizzo (2^32) allora accedo correttamente. Se invece si trova nei 2^32 byte dall'indirizzo 2^32 compreso in su allora il tentativo di accesso fallisce e provoca un segmentation fault. Purtroppo, nelle architetture a 64 bit, lo stack è tipicamente collocato nella parte alta della memoria, sopra i 2^32 byte. Quindi se la system call chiamata dall'istruzione 'int 0x80' cerca di accedere allo stack, probabilmente causerà un segmentation fault. Se invece la system call chiamata dall'istruzione 'int 0x80' cerca di accedere solo alla sezione 'data' e alla sezione 'text' allora probabilmente tutto andrà a buon fine perché si accede a parti di memoria indirizzabili con i soli 32 bit meno significativi dei registri.
.

3. Invocare le system call in ambiente a 64 bit sostituendo l'istruzione 'int 0x80' con l'istruzione 'syscall'.

Se devo invocare una system call a cui devo passare un indirizzo a 64 bit, non posso usare l'istruzione 'int 0x80' che esegue l'interrupt che attiva le system call.
Per invocare la system call devo fare 3 modifiche al codice:

A. Usare l'istruzione 'syscall' invece che 'int 0x80'.
B. Passare gli argomenti alla syscall nei registri giusti (diversi da quelli usati nella istruzione 'int').
C. Passare il giusto numero che identifica ciascuna system call da eseguire (diverso da quello usato nella istruzione 'int').

Vediamo qualche dettaglio:

A. Usare l'istruzione 'syscall' invece che 'int 0x80'.

B. usare, per passare gli argomenti alla syscall, dei registri diversi da quelli che avrei usato per passare gli argomenti all'istruzione 'int 0x80'.
   Posso trovare informazioni sulle system call usate a 64 bit nel file . In particolare si vede quali registri sono genericamente usati:

```
/*
* System call entry. Upto 6 arguments in registers are supported.
* SYSCALL does not save anything on the stack and
```

```
* does not change the stack pointer.
* Register setup:
* rax  system call number
* rdi  arg0
* rcx  return address for syscall/sysret, C arg3
* rsi  arg1
* rdx  arg2
* r10  arg3    (--> moved to rcx for C)
* r8   arg4
* r9   arg5
* r11  eflags for syscall/sysret, temporary for C
* r12-r15,rbp,rbx saved by C code, not touched.
*/
```

C. **specificare** correttamente il **numero che identifica la system call**. **Questi numeri sono cambiati nel passaggio dalle architetture a 32 alle architetture a 64 bit**.
   Guardare nel file 'unistd_64.h' o ' /usr/include/asm-generic/unistd.h' per vedere l'elenco delle system calls ed i loro numeri identificativi.
   Ad esempio, chiamando l'istruzione 'int 0x80', per ottenere la system call 'write' devo passare come numero di system call nel registro 'eax' il valore '4'.
   Invece, chiamando l'istruzione '**syscall**', per ottenere la system call 'write' devo passare come numero di system call nel registro 'rax' il valore '**1**'.
   Analogamente, la terminazione si ottiene col valore 1 usando l'interrupt 'int 0x80' mentre si ottiene con la system call numero 60 usando l'istruzione syscall.

Ad esempio, il codice a 32 bit per invocare la scrittura a video, di una stringa collocata nella parte dati, potrebbe essere questo:

```
movl $4, %eax              /* servizio da ottenere: write, stampa, in eax, è 4 */
movl $1, %ebx              /* primo arg, sottoservizio da ottenere, in ebx */
movl $miastringa, %ecx     /* secondo arg, indirizzo di inizio stringa, in ecx */
movl $7, %edx              /* terzo arg, lunghezza stringa da stampare in edx */
int  $0x80
```

Invece, il codice a 64 bit per invocare la scrittura a video di una stringa collocata nella parte dati sarà questo:

```
mov $1, %rax               /* servizio da ottenere: write, stampa, in rax , ora è 1*/
mov $1, %rdi               /* primo arg, sottoservizio da ottenere, in rdi non in rbx */
mov $miastringa, %rsi      /* secondo arg, indirizzo inizio stringa, in rsi non in ecx */
```

```
mov $7, %rdx                /* terzo arg, lunghezza stringa da stampare, in rdx */
syscall                     /* chiamo syscall  non  int 0x80 */
```