Eseguire system call -

passare dall'uso di interrupt 0x80 in codice a 32 bit

all'uso dell'istruzione syscall in codice a 64 bit.

0. Formato di modulo oggetto o eseguibile ed architettura del processore.

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. ASSEMBLARE COMPILARE e LINKARE a 32 bit o a 64 bit.

1.1 Assemblare compilare e linkare codice a 32 bit su architettura a 32 bit.

Se sono in un processore Intel i386, per **assemblare, compilare e linkare codice a 32 bit** devo usare dei flag specifici per l'assemblatore (as **--32**) e per il compilatore (gcc **-m32**), ma questi sono il default se lavoro su processore Intel i386. I moduli oggetto e l'eseguibile creati in tal modo saranno salvati in files aventi formato **elf_i386** e conterranno codice per l'architettura i386.
Se linko col gcc e chiedo di usare le libc standard, verranno aggiunti automaticamente i seguenti moduli:

/usr/lib/i386-linux-gnu/crt1.o
/usr/lib/i386-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/5/32/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/5/32/crtend.o
/usr/lib/i386-linux-gnu/crtn.o

Eseguendo il comando 'objdump -f nomeeseguibile.exe [oppure nomefileoggetto.o]' posso vedere il formato dell'eseguibile e il processore per cui è stato creato, nel caso di codice a 32 bit su architettura i386 sarà così:

print_usa_main.exe: file format **elf32-i386**
architecture: **i386**, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482e0

1.2 Assemblare compilare e linkare codice a 32 bit su architettura a 64 bit.

Per **assemblare, compilare e linkare codice a 32 bit** anche se sono in un processore Intel o AMD a **64 bit** (x86-64 architecture), devo usare dei flag specifici per l'assemblatore (as **--x32**) e per il compilatore (gcc **-mx32**). **Il codice utilizza puntatori a 32 bit ma può usare il set di istruzioni del processore**. Questo codice esegue anche in processori Intel o AMD a 64 bit. I moduli oggetto e l'eseguibile creati in tal modo saranno salvati in files aventi formato **elf32-x86-64** e conterranno codice per l'architettura x84-64 ma con puntatori a 32 bit.
Notare che per linkare questo codice con il gcc occorre avere il set di librerie definite '**gcc-multilib**' che contengono i moduli oggetto aggiuntivi adatti.
Infatti, se linko col gcc e chiedo di usare le libc standard, verranno aggiunti automaticamente i moduli seguenti:

/usr/libx32/crt1.o
/usr/libx32/crti.o
/usr/lib/gcc/x86_64-linux-gnu/5/x32/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/5/x32/crtend.o
/usr/libx32/crtn.o

Eseguendo objdump -f ottengo le seguenti informazioni sull'eseguibile:

print_usa_main.exe: file format **elf32-x86-64**
architecture: i386:**x64-32**, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00400074

1.3 Assemblare compilare e linkare codice a 64 bit su architettura a 64 bit.

Per assemblare, compilare e linkare codice a 64 bit, in un processore Intel o AMD a 64 bit, devo usare dei flag specifici per l'assemblatore (as --64) e per il compilatore (gcc -m64). Il codice utilizza puntatori a 64 bit e usa il set di istruzioni del processore. Questo codice ovviamente esegue nativamente sui processori Intel o AMD a 64 bit. I moduli oggetto e l'eseguibile creati in tal modo saranno salvati in files aventi formato **elf64-x86-64**.
In questo caso, se linko col gcc e chiedo di usare le libc standard, verranno aggiunti automaticamente i moduli seguenti:

/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

Eseguendo objdump -f ottengo le seguenti informazioni sull'eseguibile:

print64_usa_printf.exe: file format **elf64-x86-64**
architecture: i386:**x86-64**, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400430

2. 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 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:

1) **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.

2) 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 */
```

4. Appendice: Caratteristiche architetture x64

- Registri 64 bit: tutti i registri general purpose sono estesi da 32 a 64 bit. Si passa inoltre da 8 a 16 registri.
- ALU a 64: operazioni aritmetiche intere e operazioni logiche sono eseguite su ALU 64 bit.
- Gestione della memoria:
    - IA-32 prevede puntatori di 32 bit e quindi uno spazio di memoria indirizzabile di 4 GB.
    - Gli indirizzi logici (puntatori) in x64 sono a 64 bit e quindi lo spazio virtuale indirizzabile è di 16 EB (ExaByte).
    - La memoria fisica indirizzabile in CPU x64 dipende dal numero di linee del bus indirizzi (possibili fino a 52) per un totale di 4 PB (PetaByte).
- Accesso ai dati relativo a IP: le istruzioni possono referenziare i dati con indirizzi relativi all'Instruction Pointer. Questo rende il codice position independent e facilmente rilocabile.

5. Appendice: Registri architetture x64.