

Note esercitazioni Assembler

Raffaele Zippo

30 settembre 2021

Indice

1	Introduzione ambiente di sviluppo	2
2	Ambiente DOS	3
2.1	Problemi noti	3
3	Ambiente Linux	5
4	Differenze tra gli ambienti	7
4.1	File utility.s	7
4.2	Meccanismi di protezione e segmentation fault	7
4.3	Differenze con l'ambiente degli anni precedenti	7
5	Debugging: utilizzo di gdb	8
5.1	Avvio di gdb	8
5.2	Comandi per il controllo dell'esecuzione	8
5.2.1	Comandi aggiuntivi	9
5.3	Comandi per ispezionare lo stato del programma	9

1 Introduzione ambiente di sviluppo

In questo corso, programmeremo assembly usando la sintassi GAS (anche nota come AT&T). Per assemblare e debuggare questi programmi, utilizzeremo degli script `assemble` e `debug`. Questi script non fanno che invocare, con gli opportuni parametri, rispettivamente `gcc` e `gdb`.




L'ambiente è fornito in due versioni, DOS e Linux. *Normalmente*, questi sistemi operativi sono molto diversi e programmi scritti per uno non funzionano per l'altro. *In questo caso*, i file forniti si occupano di rendere omogeneo il comportamento in modo che siano intercambiabili.

Quindi, quale che sia l'ambiente utilizzato per esercitarsi il codice da scrivere sarà lo stesso.

2 Ambiente DOS

Questo ambiente si basa su DOSBox, un emulatore del sistema operativo DOS. L'ambiente include i binari per DOS di gcc e gdb, e i binari DOSBox per Windows.

L'ambiente è fornito in un file .zip che, estratto nella posizione che si preferisce, avrà la seguente struttura:

-  files
 -  DOSBox
 -  GAS
 - utility.s
- ASSEMBLE.BAT
- DEBUG.BAT
- runDosBox.bat
- CWSDPMI.EXE

Fare doppio click su runDosBox.bat (o lanciarlo da terminale) per avviare DOSBox. Per scrivere il codice, si può usare un qualunque editor di testo. Durante le esercitazioni, verrà mostrato Visual Studio Code. I sorgenti possono essere salvati sia nella root dell'ambiente che in sottocartelle, ma è bene *evitare* di spostarsi (comando cd) dalla root del progetto.

Gli script forniti sono

- ASSEMBLE.BAT, da eseguire in DOS per assemblare il proprio codice. Esempio:

```
C:\> ASSEMBLE.BAT FILE.S
```

Questo produrrà, se non ci sono errori, l'eseguibile FILE.EXE e l'output dell'assemblatore in LISTATO.EXE

- DEBUG.BAT, da eseguire in DOS per debuggare l'eseguibile. Esempio:

```
C:\> DEBUG.BAT FILE.EXE
```

Questo avvierà gdb.

2.1 Problemi noti

Per via delle versioni datate dei software per DOS, può capitare che l'assemblatore corrompa il file sorgente. Si consiglia di stare attenti a come viene lasciato il file .S dopo aver chiamato ASSEMBLE.BAT, e tenere lasciato aperto l'editor di testo per poter annullare le eventuali modifiche indesiderate.

Sempre per via della versione datata, il debugger gdb può non eseguire correttamente tutti i comandi discussi più avanti, segnalando errori interni per i quali suggerisce di terminare il debugging. In tal caso, si può *provare* a continuare, ma non è possibile dire quanto questi

errori influiscano sulla normale esecuzione del programma che si sta debuggando.

Per quanto riguarda la facilità d'uso, si ricorda che in DOS i nomi di cartelle sono limitati a 8 caratteri, mentre quelli di file a 8 caratteri più 3 di estensione. DOSBox aiuta molto in questo senso, sfruttando il carattere ~. Basterà usare l'autocompletamento (tasto TAB della tastiera) per ottenere un nome valido per file con più di 8 caratteri.

Infine, la schermata mostrata nella finestra DOSBox sarà difatti l'output video del sistema DOS, non è un semplice terminale. Ci sarà infatti impossibile visualizzare più caratteri a schermo, scorrere le righe per vedere l'output precedente, copiare o incollare del testo.

In gdb poi non è presente una cronologia dei comandi, che ci permette (su Linux) di ripetere i comandi precedenti usando i tasti freccia invece di riscriverli daccapo.


3 Ambiente Linux

Questo ambiente si basa direttamente su Linux. Può essere utilizzato direttamente su una macchina con Linux, o con una macchina virtuale. Per le esercitazioni, verrà utilizzato Ubuntu 20.04 in WSL2.

L'ambiente non include tutti i binari. Su Ubuntu 20.04, si dovranno installare:

- build-essential
- gcc-multilib
- musl-dev
- gdb
- powershell

L'ambiente è fornito in un file .zip che, estratto nella posizione che si preferisce, avrà la seguente struttura:

-  files
 - gdb_startup
 - main.c
 - utility.s
- assemble.ps1
- debug.ps1
- test.ps1

Per utilizzare l'ambiente basterà aprire un terminale Powershell¹ nella cartella root dell'ambiente. Per scrivere il codice, si può usare un qualunque editor di testo. Durante le esercitazioni, verrà mostrato Visual Studio Code. I sorgenti possono essere salvati sia nella root dell'ambiente che in sottocartelle, ma è bene *evitare* di spostarsi (comando cd) dalla root del progetto.

Gli script forniti sono

- assemble.ps1, da eseguire per assemblare il proprio codice. Esempio:

```
> ./assemble.ps1 esercizio.s
```

Questo produrrà, se non ci sono errori, l'eseguibile esercizio e l'output dell'assemblatore in esercizio.lst

- debug.ps1, da eseguire per debuggare l'eseguibile. Esempio:

```
> ./debug.ps1 esercizio
```

Questo avvierà gdb.

- test.ps1, di utilità per fare test con file di input e (opzionalmente) salvare l'output su file

¹Per cambiare shell, basta eseguire il comando pwsh senza argomenti

```
> ./test.ps1 esercizio input.txt
```

```
> ./test.ps1 esercizio input.txt output.txt
```

Gli script controllano che i file passati siano testo od eseguibili, per evitare errori comuni con l'autocompletamento.

```
> ./assemble.ps1 esercizio
```

```
The file is an executable
```

```
> ./debug.ps1 esercizio.s
```

```
The file is not an executable
```

4 Differenze tra gli ambienti

Le differenze principali riguardano l'usabilità, per il quale si lascia alla scelta personale, soprattutto per le esercitazioni a casa. Dal punto di vista del codice da scrivere, gli ambienti si comportano allo stesso modo: si utilizzano le stesse sintassi, le stesse primitive di input/output, gli stessi caratteri di terminazione riga¹.

4.1 File `utility.s`

Entrambi gli ambienti sono *portabili*, cioè i rispettivi file possono essere salvati dove si preferisce. Per utilizzare i sottoprogrammi di ingresso/uscita, si utilizzerà

```
.INCLUDE "../files/utility.s"
```

Il percorso è relativo alla *root* dell'ambiente, cioè la cartella che conterrà gli script assemble e debug. È importante che sia da questa cartella che si chiamino gli script.

4.2 Meccanismi di protezione e `segmentation fault`

Marcare le sezioni `.data` e `.text` potrebbe risultare opzionale in DOS, dato che la distinzione non viene applicata dal sistema. Nell'ambiente Linux la distinzione è invece applicata: se non si dichiarano opportunamente le sezioni, il programma potrebbe essere terminato con `segmentation fault`.

4.3 Differenze con l'ambiente degli anni precedenti

Le uniche cose che cambiano rispetto all'ambiente degli anni precedenti sono

- Il percorso del file di utility, che ora va incluso con `.INCLUDE "../files/utility.s"`
- Le sezioni `.data` e `.text`, che, se non dichiarate, possono causare terminazione su Linux

Questa classe di errori non verrà considerata ai fini della valutazione.

¹In particolare, l'ambiente fa sì che Linux si comporti allo stesso modo di DOS.

5 Debugging: utilizzo di gdb

Il *debugging* è il processo di ricerca e rimozione degli errori (bug) di un programma. In linguaggi ad alto livello, il primo strumento utilizzato è la stampa su terminale o su file di log, per individuare rapidamente i punti d'errore. Qui invece questo non è altrettanto facile, e dobbiamo affidarci ad un debugger completo, per l'appunto gdb.

Il principale scopo del debugger è far eseguire il programma *un passo alla volta*, e permetterci di osservare lo stato dei registri e della memoria a ciascuno di questi passi. Quali sono questi "passi" su cui soffermarsi lo decidiamo noi, definendo dei *breakpoints*: quando il programma giunge ad un breakpoint, il debugger mette in pausa l'esecuzione e lascia a noi il controllo. Possiamo allora, tramite specifici comandi, stampare informazioni sullo stato, proseguire un'istruzione alla volta, far continuare fino al prossimo breakpoint, etc.

5.1 Avvio di gdb

La sintassi più semplice è `gdb percorso_eseguibile`.

Lo script di debug nell'ambiente del corso fa dei passi in più, che semplificano l'utilizzo: definisce i comandi `rr` e `qq` (sezione successiva) ed esegue i comandi per mettere un breakpoint a `_main` ed avviare l'esecuzione del programma.

Attenzione a quello che `gdb` stampa all'avvio: se la terzultima riga è la seguente conviene fermarsi subito.

```
warning: Source file is more recent than executable.
```

Come dice l'errore, non abbiamo riassemblato il programma dopo le ultime modifiche.

5.2 Comandi per il controllo dell'esecuzione

Per guidare l'esecuzione del programma si usano i seguenti comandi¹:

- **frame**
Mostra la posizione attuale del programma, ossia l'istruzione che sta per essere eseguita e la riga a cui corrisponde nel file sorgente.
- **break label**
Inserisce un breakpoint alla posizione indicata da *label*. L'esecuzione del programma verrà quindi messa in pausa prima di eseguire l'istruzione associata a *label*.
- **continue**
Prosegue l'esecuzione fino al prossimo breakpoint.
- **step**
Esegue una singola istruzione.
Attenzione se questa è una *CALL*: il debugger si fermerà *dentro* il sottoprogramma chiamato.

¹I caratteri evidenziati in **grassetto** sono il minimo necessario perché `gdb` riconosca il comando. Non c'è quindi bisogno di scriverli per intero.

- **finish**
Prosegue l'esecuzione del sottoprogramma corrente (sia *f*) finché non termina (RET). L'esecuzione del programma verrà quindi messa in pausa prima di eseguire l'istruzione successiva all'istruzione CALL *f*.
- **next**
Continua l'esecuzione fino alla successiva istruzione di questo file. Questo significa che si comporta come *step*, tranne quando l'istruzione da eseguire è una CALL. In tal caso, il sottoprogramma viene eseguito senza pause.
- **run**
Avvia l'esecuzione del programma. Se il programma è già in esecuzione, dopo aver chiesto conferma all'utente, riavvia il programma dall'inizio.
- **quit**
Termina il debugger. Se il programma è ancora in esecuzione, chiede prima conferma all'utente.

5.2.1 Comandi aggiuntivi

I seguenti comandi non fanno parte dei comandi standard di `gdb`. Sono invece comandi personalizzati definiti all'avvio dallo script di debug fornito nell'ambiente, aggiunti allo scopo di semplificare i casi d'uso più comuni.

- **rrun**
Riavvia il programma, senza chiedere conferma.
- **qquit**
Termina il debugger, senza chiedere conferma.

5.3 Comandi per ispezionare lo stato del programma

Oltre a controllare il percorso seguito dal programma, è ovviamente utile controllare lo stato di registri e memoria prima e dopo l'esecuzione delle istruzioni di interesse. Per far questo useremo i seguenti comandi:

- **info register *registro***
L'argomento *registro* è opzionale, e deve essere in minuscolo e senza caratteri preposti: *eax, bx, cl*.
Se specificato, mostra il contenuto del registro, prima in esadecimale e poi in un formato che dipende dal tipo di registro:
 - decimale per registri accumulatori
 - label+offset per *eip*
 - lista dei flag a 1 per *eflags*

Se non specificato, farà quanto sopra per tutti i registri.

- **x/NFU indirizzo**
La *x* sta per *examine memory*, in questo caso non è un'abbreviazione e non si può usare una versione più lunga.
Notiamo intanto che ci sono diversi argomenti: numero (*N*), formato (*F*), dimensione (*U*) e indirizzo. Sono tutti *opzionali*, perché questo è un comando *con memoria*. Verranno infatti ricordati gli ultimi argomenti ed utilizzati come valori di default, ad eccezione di *N* il cui default è sempre 1. Per evitare confusioni, si consiglia comunque di specificare sempre tutto.

L'argomento *indirizzo* indica la locazione da cui *iniziare*. Questo può essere indicato:

- in esadecimale: `x 0x56559066`

- tramite label preceduta da &: x &buffer
- tramite registro preceduto da \$: x \$esi

L'argomento *N* indica il numero di locazioni da accedere. Questo è un semplice numero decimale. Se negativo, le locazioni saranno mostrate andando all'indietro.

L'argomento *F* indica il formato² con cui interpretare il contenuto delle locazioni, e quindi come va stampato a schermo:

- x esadecimale
- d decimale
- c ASCII
- t binario

L'argomento *U* indica la dimensione³ delle locazioni da accedere.

- b 1 byte
- h word, 2 byte
- w long, 4 byte

²Molti formati sono stati qui omessi perché, a noi, poco utili. Usare il comando `help x` per una lista esaustiva

³Le sigle sono diverse dal GAS perché diversa è la definizione di "word": h sta per halfword (per noi word), mentre w sta per word (per noi long).