Sistemi Operativi

Laboratorio – linea 2

7

Lezione 7:

Unix software factory

Sistemi Operativi

Laboratorio – linea 2

7

UNIX SOFTWARE FACTORY

- UNIX nasce come sistema *per i programmatori* (l'unica tipologia di utente all'inizio degli anni '70...)
- progettato insieme ad un linguaggio di programmazione (C)
- la 'filosofia di UNIX' (piccoli programmi che fanno molto bene una sola cosa su file) si adatta perfettamente al paradigma di sviluppo edit-compile-debug
- tool all'avanguardia nell'elaborazione di *file di testo* (per lo piú organizzati per "righe") e per la scrittura dei programmi di elaborazione stessi (lex, yacc,...)

Edit / Compile

- Editor: `ed`, `vi`, `emacs` manipolano arbitrariamente i byte di un file, generalmente interpretandoli come caratteri stampabili (testo)
- Compilatore: `cc` (`gcc`)
  1. `cc` sorgente (`.c`) $\rightsquigarrow$ assembly (`.s`)
  2. `as` assembly $\rightsquigarrow$ *oggetto* (`.o`)
  3. (`ar` archivia diversi oggetti in una *libreria* (`.a`)
  4. `ld` *oggetti* e *librerie* $\rightsquigarrow$ eseguibile (`a.out`) (il formato storico è COFF, oggi ELF)

Si noti che a sua volta anche la compilazione vera e propria è fatta da due passi (pre-processore `cpp` e compilazione `cc1`).

Make

Stuart Feldman, 1977 at Bell Labs.
Permette di specificare dipendenze fra processi di generazione.
Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
helloworld.o: helloworld.c
        cc -c -o helloworld helloworld.c

helloworld: helloworld.o
        cc -o $@ $<

.PHONY: clean
clean:
        rm helloworld.o helloworld
```

Come funziona il file Makefile ?

Example (A simple Makefile)

```
Hello:  Hello.c
        clang -Wall -o Hello Hello.c
```

- First Line: Dependency Tree
  - Target and Sources
  - Target: the module to be built (e.g. `Hello`)
  - Sources: pre-requisites (e.g. `Hello.c`)

Make rules :

Example (A simple Makefile)

```
Hello:  Hello.c
        clang -Wall -o Hello Hello.c
```

- Second Line: Make rule
  - command to execute
    - `clang -Wall -o Hello Hello.c`
  - requires a tab character (not spaces) for indentation

Target multipli :

Example (Makefile for compiling multiple Modules)

```
Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -o module2.o module2.c
```

- Default Target: first target (`Program`)
  - link two object files (`module1.o` and `module2.o`) into one program (`Program`)

Target multipli (2) :

Example (Makefile for compiling multiple Modules)

```
Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -o module2.o module2.c
```

- Second Target: `module1.o`
  - rule to compile object file `module1.o` from `module1.c`
  - `clang -c` compiles a single module (not a full executable)

Target multipli (3) :

Example (Makefile for compiling multiple Modules)

```
Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module1.o:  module1.c
        clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -o module2.o module2.c
```

- Third Target: `module2.o`
  - compile `module2.o` from source `module2.c`
  - also depends on `module2.h` (header file)

Più di un programma ...

Example (Makefile for compiling multiple Programs)

```
all:  Program1 Program2

Program1:  module1.o
        clang -o Program module1.o

Program2:  module2.o module3.o
        clang -o Program module2.o module3.o

module1.o:  module1.c
        clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
        clang -c -Wall -o module2.o module2.c

module3.o:  module3.c module3.h
        clang -c -Wall -o module3.o module3.c
```

- 'all' target:
  - compiles all programs (`Program1` and `Program2`)

Regole generiche

Evitano la necessità di scrivere molte volte nel temrinale comandi di utilizzo “generale” ed utilizzati molto frequentemente.

Incremento consistenza: è il modo più sicuro di cambiare il modo in cui invochiamo abitualmente compilatore/i

Utilizzo di liste di suffissi per “trasformare” (compilare) un tipo di file in un altro:

.c.o per compilare un file .c in un file .o

Regole generiche: esempio 1

Example (Makefile containing a generic rule)

```
.c.o:
        clang -c -Wall -o $*.o $*.c

Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module2.o:  module2.c module2.h
```

- `.c.o:`
  - how to compile a `.c` into a `.o` file
  - `$*` gets replaced by the file name (without extension)

Regole generiche: esempio 2

Example (Makefile containing a generic rule)

```
.c.o:
        clang -c -Wall -o $*.o $*.c

Program:  module1.o module2.o
        clang -o Program module1.o module2.o

module2.o:  module2.c module2.h
```

- No need for a `module1.o:` rule!
  - compiler already knows how to compile `.c` into `.o`
  - But: `module2.o` needs a rule (also depends on `.h`)

Variabili Make

- Allow more flexible make files
  - "what if the compiler is not called `clang`?"
- Variables allow assigning a value, e.g:
  - `CC=gcc`
- Varables can be used using `$(`*variable*`)`, e.g.:
  - `$(CC) -c -Wall -o $*.o $*.c`
  - will replace `$(CC)` with `gcc`

```
.c.o:
        clang -c -Wall -o $*.o $*.c
```

Regole per linguaggi diversi da C

- The `make` utility by default only knows about C
  - "what if I want to compile a different language?"
- Suffixes can be specified
  - using the `.SUFFIXES:` command, e.g.:
  - `.SUFFIXES: .o .m`
    - "a `.o` file can also be compiled from a `.m` (Objective-C) file"

Regole per linguaggi diversi da C : C / Objective-C

Example (Makefile for a mixed C/Objective-C program)

```
#
# A mixed makefile example for C and Objective-C on Mac OS X
#
CC=clang

.SUFFIXES: .o .c
.SUFFIXES: .o .m

.c.o:
        $(CC) -c -Wall -o $*.o $*.c

.m.o:
        $(CC) -c -Wall -o $*.o $*.m

Program:  cmodule.o objcmodule.o
        $(CC) -o Program cmodule.o objcmodule.o -framework Foundation

objcmodule.o:  objcmodule.m objcmodule.h
```

Regole per linguaggi diversi da C : C / C++

Example (Makefile for a mixed C/C++ program)

```
#
# A mixed makefile example for C and C++
#
CC=clang
CPLUS=g++

.SUFFIXES: .o .c
.SUFFIXES: .o .cc

.c.o:
        $(CC) -c -Wall -o $*.o $*.c

.cc.o:
        $(CPLUS) -c -Wall -o $*.o $*.cc

Program:  cmodule.o cppmodule.o
        $(CPLUS) -o Program cmodule.o cppmodule.o

cppmodule.o:  cppmodule.cc cppmodule.h
```

Debugger : concetti fondamentali

Breakpoint

Un punto del programma in cui l'esecuzione deve essere bloccata, tipicamente per esaminare lo stato in quell'istante.

Stepping

Eseguire il programma *passo a passo*. La granularità del passo può arrivare fino all'istruzione macchina.

Stato

Lo stato del programma può essere analizzato come:

- forma simbolica: secondo i simboli definiti nel linguaggio di alto livello e conservati come *simboli di debugging*
- memoria virtuale: stream di byte suddiviso in segmenti
  - Text: contiene le istruzioni (spesso read only)
  - Initialized Data Segment: variabili globali inizializzate
  - Uninitialized Data Segment (bss): variabili globali non inizializzate
  - Stack: collezione di *stack frame* per le chiamate di procedura. Cresce verso il basso.
  - Heap: Strutture dati create dinamicamente. Cresce verso l'alto tramite system call `brk` (API `malloc`).

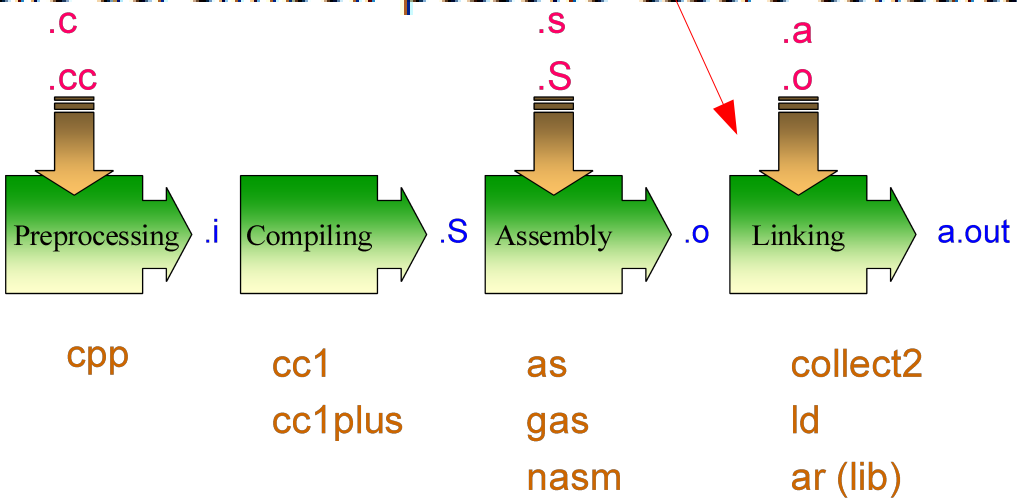
Uso del debugger

- `break ...` (un simbolo o un indirizzo `*0x...`)
- `run ...` (eventualmente con argv)
- `print ...` (`x`)
- `next` (`nexti`)
- `step` (`stepi`)
- `backtrace`

Simboli

La *symbol table* serve al *linker* per associare nomi simbolici e indirizzi prodotti dal compilatore:

- contenuta in tutti gli oggetti, generalmente viene lasciata anche negli eseguibili (ma può essere scartata con `strip`)
- una versione piú ricca viene detta "simboli di debug" (vari formati, p.es. DWARF)
- le tabelle dei simboli possono essere consultate con `nm`



Perchè capire le fasi ?

Per costruire sistemi operativi a volte serve alterare il flusso tradizionale

```
1 gcc -O -nostdinc -I. -c bootmain.c
2 gcc -nostdinc -I. -c bootasm.S
3 ld -m elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o
4 objdump -S bootblock.o > bootblock.asm
5 objcopy -S -O binary -j .text bootblock.o bootblock
```

bootmain.o

```
1 $ nm kernel | grep _start
2 8010b50c D _binary_entryother_start
3 8010b4e0 D _binary_initcode_start
4 0010000c T _start
```

Comandi GDB

Compilazione:

```
gcc [other flags] -g <source files> -o <output file>
```

Per avviare sessione GDB:

```
(gdb)
```

se non si specifica il file si può caricarlo dall'interno di GDB (comando: file) :

```
(gdb) file prog1.x
```

Per ottenere informazioni su un determinato comando:

```
(gdb) help [command]
```

Comandi GDB

Per eseguire il programma:

```
(gdb) run
```

Se tutto funziona procede fino al completamento. In caso contrario fornisce informazioni sull'errore. Ad esempio:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

Impostazione breakpoints :

E' possibile impostare dei breakpoint per specifiche coppie file/riga :

```
(gdb) break file1.c:6
```

Potete impostare quanti breakpoint volete. Ogni volta che uno di essi verrà raggiunto GDB fermerà l'esecuzione.

E' anche possibile interrompere l'esecuzione in corrispondenza di una particolare funzione. Supponiamo di avere, nei sorgenti, questa funzione :

```
int my_func(int a, char *b);
```

E' possibile impostare un breakpoint in corrispondenza di my_func in questo modo:

```
(gdb) break my_func
```

Controllo esecuzione :

Una volta impostato un breakpoint è possibile procedere con l'esecuzione utilizzando nuovamente il comando **run** . L'esecuzione verrà interrotta in corrispondenza del prossimo breakpoint incontrato (se non si verificano errori prima).

E' anche possibile passare “attraverso” il prossimo breakpoint, ingorandolo, utilizzando il comando **continue** :

```
(gdb) continue
```

E' possibile eseguire le istruzioni un “passo alla volta” facendo, dei singoli step di esecuzione:

```
(gdb) step
```

Questo fornisce un controllo estremamente granulare sull'esecuzione dle programma.

Indagare su altri aspetti dello stato di esecuzione del programma :

Fino a questo punto abbiamo visto come impostare dei breakpoint e come controllare l'esecuzione del programma. Ora vedremo come indagare sul valore delle variabili.

Il comando **print** stampa il valore di una variabile. **print/x** stampa il valore in esadecimale.

```
(gdb) print my_var
(gdb) print/x my_var
```

I breakpoint servono per controllare il flusso di esecuzione di un programma. Uno strumento correlato ai breakpoint ma utilizzato per indagare sulle variazioni di stato delle variabili è il **watchpoint** :

```
(gdb) watch my_var
```

Dopo l'utilizzo di questo comando ogni volta che la variabile **my_var** viene modificata GDB interrompe il flusso di esecuzione e mostra **il vecchio ed il nuovo valore** di my_var. **Pericolo** : problemi di scope. Se ci sono più variabili my_var GDB “seguirà” quella presente nello scope al momento dell'utilizzo del comando watch”

Altri comandi utili :

- `backtrace` - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)
- `where` - same as `backtrace`; you can think of this version as working even when you're still in the middle of the program
- `finish` - runs until the current function is finished
- `delete` - deletes a specified breakpoint
- `info breakpoints` - shows information about all declared breakpoints

Ricordatevi che è sempre possibile usare il comando **help** per avere informazioni sui comandi GDB. Altri comandi utili:

show args	mostra argomenti attuali
set args *argument*	imposta argomenti

Esempio di utilizzo di GDB

example.c

compilazione:

$ gcc -g example.c

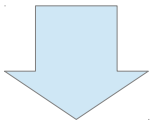
```
#include <stdio.h>


int main(void)
{

	const int data[5] = {1, 2, 3, 4, 5};
	int i, sum;

	for (i=0; i>=0; ++i) {
		sum += data[i];
	}

	printf("sum = %i \n", sum);

	return 0;
}
```

Esempio di utilizzo di GDB

1. compile: `gcc -g example.c`
2. try to run the program the regular way without gdb: `a.out`
3. it seg faults
4. start it with gdb: `gdb -tui ./a.out`
5. `list main`
6. `break main`
7. `run` (this starts running the program up to the break point at `main`)
8. `p data` (the debugger stops at the line preceding the current line of code shown, so when you try to print `data`, at that point, the array called `data` has not been initialized yet; in other words, the line of code that is highlighted is the line that *will be executed next*).
9. `next`
10. `p data` (now the values for `data` are there)
11. `next`
12. `next`
13. `next`
14. `p i`
15. `next`
16. `p i`
17. `next`
18. `p i` (see what's happening?)
19. `continue` the program continues until it gets to the next break, or, as in this case, the seg fault since no other break was set
20. this is what it shows:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000040052d in main () at example.c:11
```

sum += data[i];



Esempio di utilizzo di GDB

21. `list` this will list 10 lines near where the seg fault occurred (which is already in the upper box)
22. `p i` this will show what value `i` has at this point (what is `i` on your screen and why?)
23. `quit` will quit the gdb debugger and give you back a regular prompt

Ulteriori informazioni sui breakpoint :

I breakpoint sono noiosi … ci obbligano ad eseguire moltissime istruzioni step o next … Supponiamo di avere una idea precisa rispetto al problema che blocca il programma di cui stiamo effettuando il debug.

In questo caso sarebbe utile poter impostare dei **breakpoint condizionali**, che si attivano solamente al verificarsi di una data condizione.

I breakpoint condizionali funzionano esattamente come I breakpoint che abbiamo visto fin qui ma, a differenza di essi, si attivano solo in corrispondenza di un certo evento. Per impostarne uno possiamo fare così:

```
(gdb) break file1.c:6 if i >= ARRAYSIZE
```

Questo comando imposta un breakpoint alla riga 6 del file file1.c che si attiva se solo se il valore della variabile i è maggiore di ARRAYSIZE (il che è problematico se, in questa riga dei sorgenti, c'è una operazione del tipo arr[i]) .

L'utilizzo di questo tipo di breakpoint ci risparmia molte istruzioni step/next .

Puntatori :

Chi non si diverte a giocare con i puntatori? Supponiamo di avere nei nostri sorgenti una struct di questo tipo:

```
struct entry {
  int key;
  char *name;
  float price;
  long serial_number;
};
```

Essa potrebbe essere utilizzata in qualche sorta di hash table utilizzata nella realizzazione di un catalogo di prodotti.

Puntatori :

Ora supponiamo di essere in GDB e che l'esecuzione del programma sia a valle di una riga di codice di questo tipo:

```
struct entry * e1 = <something>;
```

Possiamo fare moltissime cose … le stesse cose che faremmo in C.

1) visualizzare il valore (indirizzo di memoria) del puntatore:

```
(gdb) print e1
```

2) Esaminare campi particolari della struttura referenziata dal puntatore:

```
(gdb) print e1->key
(gdb) print e1->name
(gdb) print e1->price
(gdb) print e1->serial_number
```

Puntatori :

E' anche possibile dereferenziare ed utilizzare l'operatore . Invece di utilizzare l'operatore -> :

```
(gdb) print (*e1).key
(gdb) print (*e1).name
(gdb) print (*e1).price
(gdb) print (*e1).serial_number
```

Vedere l'intero contenuto della struct referenziata dal puntatore (questo è difficile da ottere in C) :

```
(gdb) print *e1
```

Possiamo anche seguire il puntatore “iterativamente” come se si trattasse di una linked list:

```
(gdb) print list_prt->next->next->next->data
```

Esercizio aggiuntivo GDB:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i = 0;
    int table[10];
    int count = 0;
    int search = 1;
      int number = 5;

      table[0] = 5;

      count = 1;
    while (count < 10) {
           table[count] = number++ * 2;
       count++;
    }

    while (search = 1) {
        if ((table[i++] == 11) || (count < i)) {
           search = 0;
        }
    }

    if (i == count) {
       printf("The number 11 is not in the table.\n");
    }
    else {
       printf("The number 11 is in the table at location: %d\n", i-1);
    }

    return 0;
}
```

Questo programma (broken.c) crea un array di 10 interi e, verso la fine del programma, cerca di capire se il numero 11 è contenuto nell'array. Contiene un bug ...

GDB : Ulteriori informazioni su print

(gdb) print[/f] *express*

(gdb) print *variable@count*

(gdb) x/nfu *addr*

n : count

f : format, x c d f u o t a

u : size, b h w g

binary

pointer

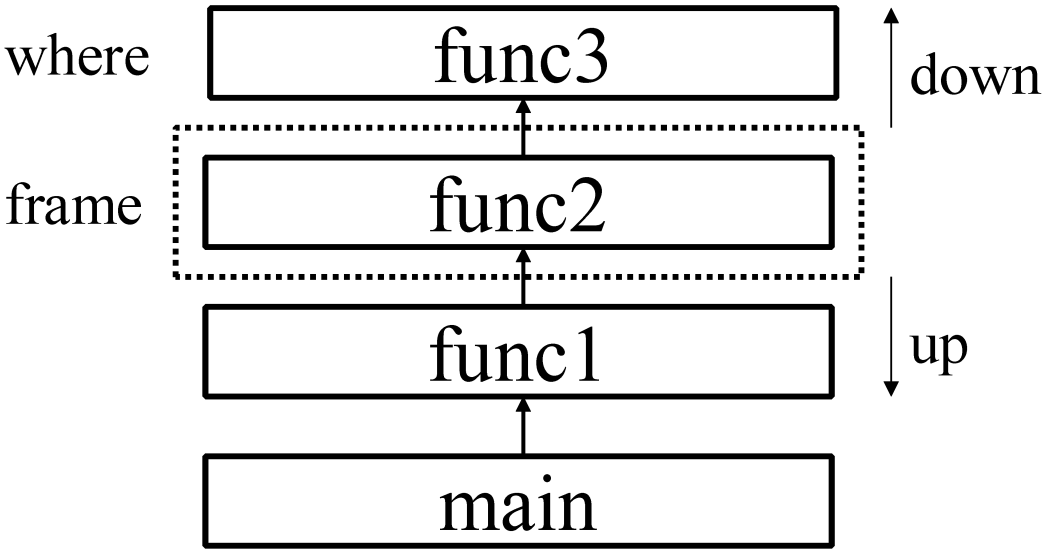
GDB : esaminare lo stack

(gdb) where / backtrace / info stack

(gdb) frame

(gdb) up *[n]*

(gdb) down *[n]*



Low level programming : mischiare Assembly e C :

In alcuni casi è comodo mischiare l'assembly al C (meno laborioso di organizzare il collegamento)

```
1 __asm__("nop");
2
3 __asm__("movl %eax, %ebx");
4 __asm__("xorl %ebx, %edx");
5 __asm__("movl $0, _booga");
6
7 __asm__("pushl %eax\n\t"
8         "movl $0, %eax\n\t"
9         "popl %eax");
```

Attenzione! Il compilatore C non "vede" l'effetto delle istruzioni assembly.

Low level programming : mischiare Assembly e C :

Si possono fare anche cose piú complicate, ma la sintassi è poco "amichevole"

```
1 __asm__("cld\n\t"
2         "rep\n\t"
3         "stosl"
4         : /* no output registers */
5         : "c" (count), "a" (fill_value), "D" (dest)
6         : "%ecx", "%edi" );
```

La sintassi è

```
1 __asm__( "statements" : output_registers : input_registers : clobbered_registers);
```

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

Basic Inline :

La sintassi di base è estremamente semplice:

```
asm("assembly code");
```

Ad esempio:

```
asm("movl %ecx %eax"); /* moves the contents of ecx to eax */
__asm__("movb %bh (%eax)"); /*moves the byte from bh to the memory pointed by eax */
```

Gcc riconosce sia la parola chiave asm che la parola chiave __asm__ (nel caso in cui asm dovesse collidere con alcune delle variabili C presenti nel sorgente).

Se abbiamo più di una istruzione le scriveremo tra apici doppi " e separeremo ogni istruzione utilizzando '\n' e '\t'. Il motivo è che gcc invia ogni istruzione a as (GAS) sottoforma di stringa e utilizzando newline\tab invieremo righe di codice formattate in modo corretto all'assemblatore.

```
__asm__ ("movl %eax, %ebx\n\t"
         "movl $56, %esi\n\t"
         "movl %ecx, $label(%edx,%ebx,$4)\n\t"
         "movb %ah, (%ebx)");
```

Assembly esteso :

Se nel nostro codice modifichiamo qualcosa (ad es. il contenuto dei registri) e ritorniamo renza riportare i registri allo stato precedente si possono creare delle inconsistenze a causa del fatto che GCC non percepisce in nessun modo gli effetti del codice assembly che abbiamo utilizzato.
Quello che possiamo fare è :

- utilizzare istruzioni che non hanno effetto (non modificano nulla)
- riportare tutto allo stato precedente all'esecuzione del codice assembly

Assembly esteso fornisce alcune caratteristiche utili per gestire queste situazioni. Nell'inline assembly di base abbiamo solo istruzioni. Nell'assembly **esteso** possiamo anche specificare degli **operandi**. Esso permette di specificare registri di input, di output e una lista di clobbered registers.

Il formato di base è il seguente:

```
asm ( assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

La lista di registri clobbered indica a GCC quali registri vengono **modificati** (e quindi potrebbero essere origine di inconsistenze). In questo modo GCC può scegliere di **farne una copia** prima di eseguire il codice assembly.

Assembly esteso :

```
asm ( assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

Il template assembler è costituito da **istruzioni assembly.** Ogni operando è descritto da una stringa seguita dal codice C racchiuso tra parentesi. Gli operandi sono separati da **:** . Gli operandi di ciascun gruppo sono separati da una virgola **,** .
Se esistono operandi di input ma non esistono operandi di output bisogna comunque inserire due simboli : consecutivi.

Esempio:

```
asm ("cld\n\t"
     "rep\n\t"
     "stosl"
     : /* no output registers */
     : "c" (count), "a" (fill_value), "D" (dest)
     : "%ecx", "%edi"
     );
```

Questo esempio scrive in fill_value un numero corrispondente a count volte il contenuto della locazione di memoria puntata dal registro edi. Inoltre dice a gcc che il contenuto **di eax e edi non è più valido**.

Assembly esteso : altro esempio

Somma di due numeri :

```
int main(void)
{
	int foo = 10, bar = 15;
		__asm__ __volatile__("addl  %%ebx,%%eax"
		                     :"=a"(foo)
		                     :"a"(foo), "b"(bar)
		                     );
	printf("foo+bar=%d\n", foo);
	return 0;
}
```

__**volatile**__ si utilizza quando vogliamo chiedere al compilatore gcc di eseguire il codice assembly esattamente **dove lo abbiamo scritto** e non, come potrebbe succedere ad esempio se sono attivi meccanismi di ottimizzazione, al di fuori di un loop in cui si trova.

http://www.cs.dartmouth.edu/~sergey/cs108/2009/gcc-inline-asm.pdf

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

diffutils : trovare differenze tra file

Con `cmp` è possibile controllare se due file sono identici.
Per i file di testo organizzato il righe esistono strumenti piú sofisticati:

- `diff` elenca le modifiche necessarie per trasformare un file in un altro (`diff3` si aiuta con un "antenato" comune, fondamentale per facilitare il *merge*)
- `diff` (e in maniera piú evoluta `diff3`) cerca di identificare le righe che *non sono cambiate*: le modifiche sono organizzate per hunk
- `patch` riapplica gli hunk di modifica al file originale (o versioni *leggermente* modificate dei medesimi)

Revision, Version, Configuration management

Dagli anni '80 sono stati proposti molti strumenti per trattare in modo efficiente:

- le successive revisioni di un file
- le versioni di un prodotto software
- le configurazioni che permettono di ottenere una specifica versione del prodotto

SCCS, RCS, CVS, SVN, `git`...

Si basano tutti sulla conservazione della "storia" dello sviluppo in un *repository*: per lavorare occorre fare *checkout* di un *artifact*, e poi chiedere il *commit* delle modifiche.

A proposito di git

- Created by Linus Torvalds, creator of Linux, in 2005
  - Came out of Linux development community
  - Designed to do version control on Linux kernel

- Goals of Git:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects efficiently

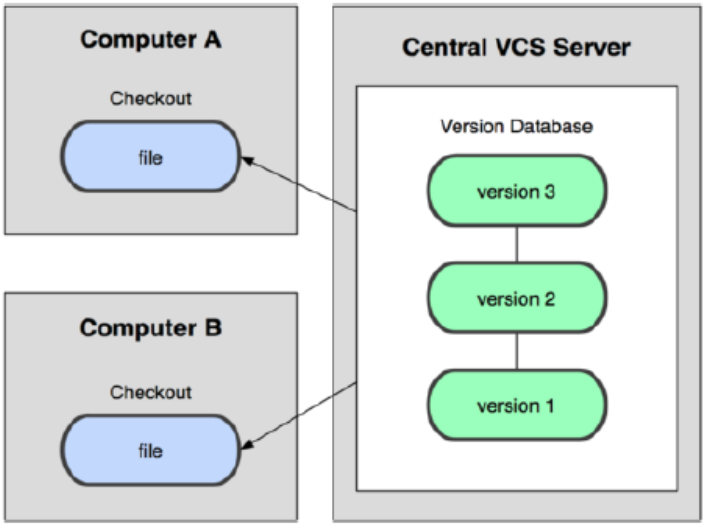
Reperire informazioni su git

- Git website: http://git-scm.com/
  - Free on-line book: http://git-scm.com/book
  - Reference page for Git: http://gitref.org/index.html
  - Git tutorial: http://schacon.github.com/git/gittutorial.html
  - Git for Computer Scientists:
    - http://eagain.net/articles/git-for-computer-scientists/

- At command line: *(where verb = config, add, commit, etc.)*
  - `git help verb`

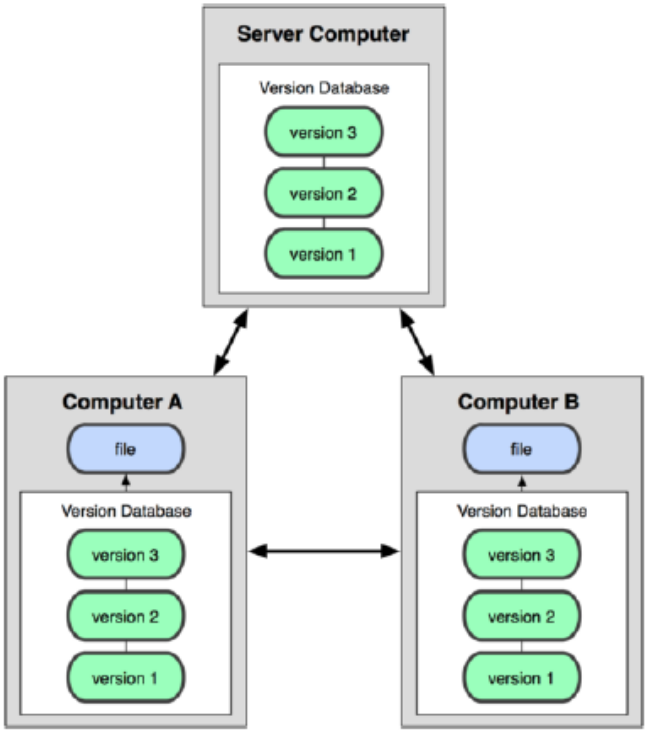
Sistemi per il controllo della versione (VCS) : modello centralizzato

- In Subversion, CVS, Perforce, etc.
  A central server repository (repo)
  holds the "official copy" of the code
  - the server maintains the sole version history of the repo
- You make "checkouts" of it to your local copy
  - you make local modifications
  - your changes are not versioned
- When you're done, you "check in" back to the server
  - your checkin increments the repo's version



Sistemi per il controllo della versione (VCS) : modello distribuito (git)

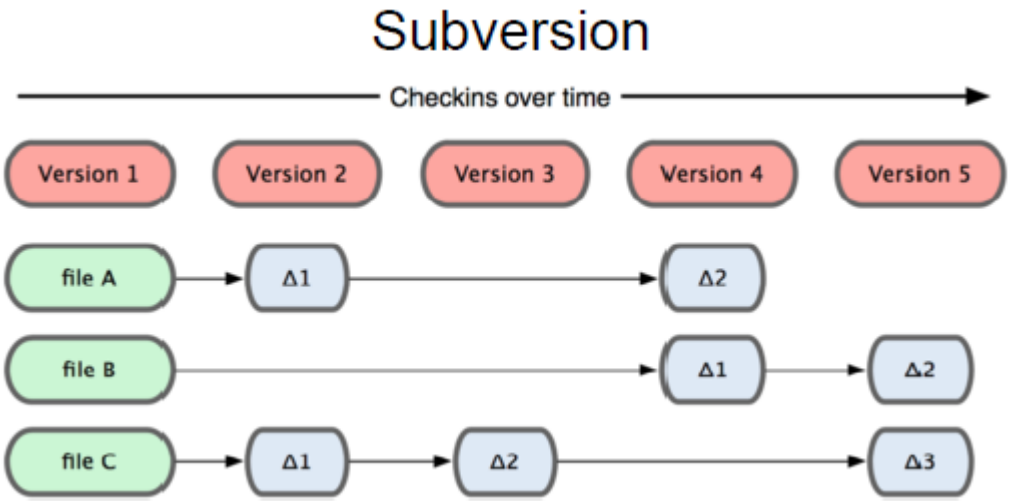
- In git, mercurial, etc., you don't "checkout" from a central repo
  - you "clone" it and "pull" changes from it
- Your local repo is a complete copy of everything on the remote server
  - yours is "just as good" as theirs
- Many operations are local:
  - check in/out from *local* repo
  - commit changes to *local* repo
  - local repo keeps version history



- When you're ready, you can "push" changes back to server

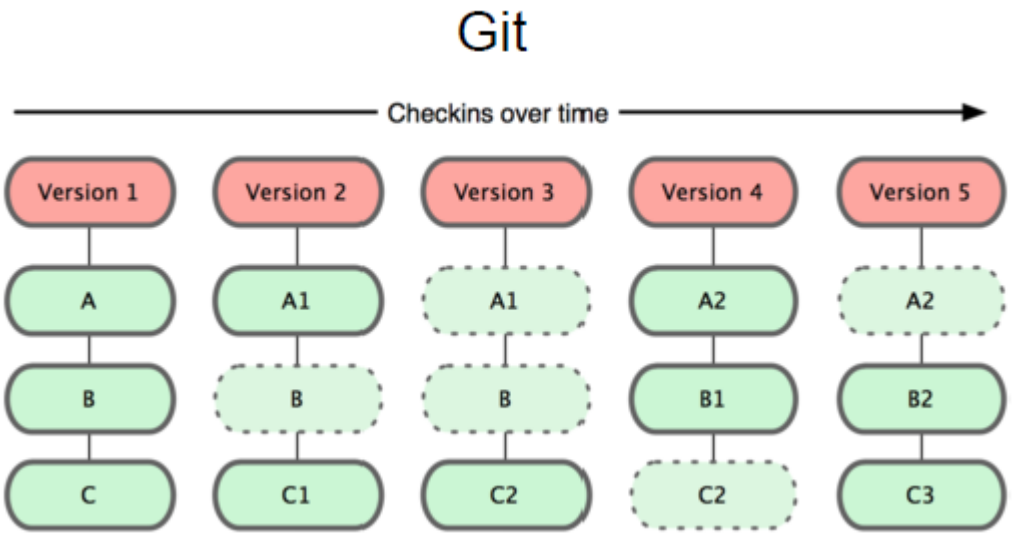
Sistemi per il controllo della versione (VCS)

Sistemi centralizzati (come subversion) mantengono la storia delle variazioni introdotte nei singoli file utilizzando come riferimento la versione del repository.



Sistemi per il controllo della versione (VCS): Git snapshots

Git, invece, tiene traccia delle variazioni sottoforma di snapshot (istantanee) dell'intero progetto.



Git

Nella nostra copia locale di git I file possono essere in alcunee aree funzionalmente diverse:

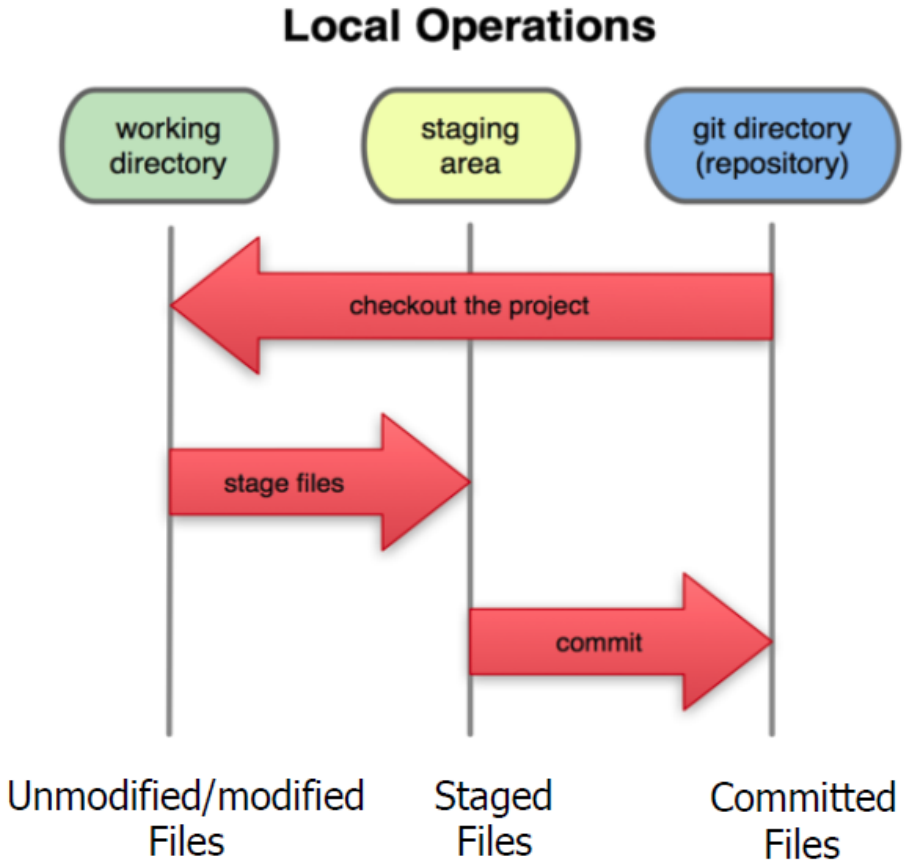
1) Nel repository locale:
COMMITTED

2) Modificati ma non ancora accettati/ammessi (COMMITTED)

Corrispondono a copie di lavoro dei file

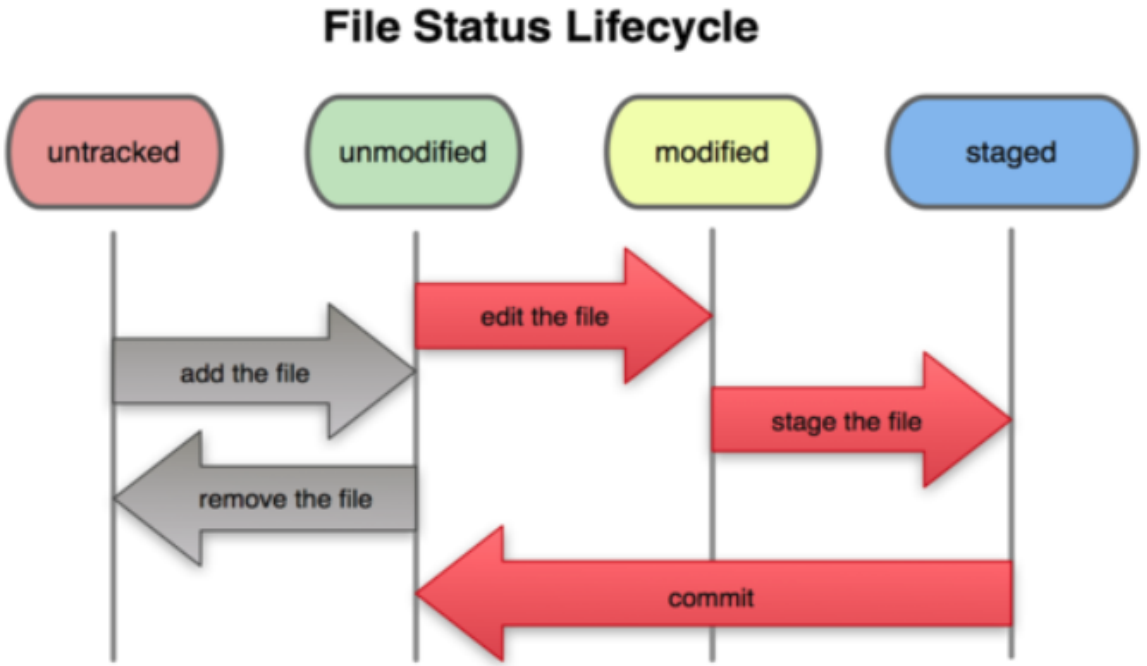
3) In un'area intermedia:
Area di “staging” . I file in quest'area sono pronti per essere ammessi ufficialmente nel repository locale. Sono in attesa di questo evento.

Un comando commit sposta tutti I file in quest'area nel repository locale



Schema di lavoro generale in Git

- **Modify** files in your working directory.
- **Stage** files, adding snapshots of them to your staging area.
- **Commit**, which takes the files in the staging area and stores that snapshot permanently to your Git directory.



Checksum dei commit in Git :

- In Subversion each modification to the central repo increments the version # of the overall repo.
  - In Git, each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server.
  - So Git generates a unique **SHA-1 hash** (40 character string of hex digits) for every commit.
  - Refers to commits by this ID rather than a version number.
  - Often we only see the first 7 characters:
    - `1677b2d Edited first line of readme`
    - `258efa7 Added line to readme`
    - `0e52da7 Initial commit`

Configurazione iniziale di un repository Git :

- Set the name and email for Git to use when you commit:
  - `git config --global user.name "Bugs Bunny"`
  - `git config --global user.email bugs@gmail.com`
  - You can call `git config -list` to verify these are set.
- Set the editor that is used for writing commit messages:
  - `git config --global core.editor nano`
    - (it is vim by default)

Creazione di un repository Git :

Due scenari comuni … utilizzare **solo uno** dei due metodi !

- To create a new **local Git repo** in your current directory:
  - `git init`
    - This will create a `.git` directory in your current directory.
    - Then you can commit files in that directory into the repo.
  - `git add filename`
  - `git commit -m "commit message"`

- To **clone a remote repo** to your current directory:
  - `git clone url localDirectoryName`
    - This will create the given local directory, containing a working copy of the files from the repo, and a `.git` directory (used to hold the staging area and your actual local repo)

Comandi Git :

command	description
`git clone` ***`url [dir]`***	copy a Git repository so you can add to it
`git add` ***`file`***	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` ***`[command]`***	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

Aggiunta file e commit :

1) La prima volta che richiadiamo che un file venga incluso e ogni volta, prima di effettuare il commit di un file, dobbiamo inserirlo nella staging area :

- git add Hello1.c Hello2.c

Viene effettuato uno **snapshot** dei file e vengono inglusi nella staging area

Nei vecchi sistemi VCS “add” significa “inizia a monitorare la versione dei file”, In Git add significa “includi nella staging area” in modo da inserire nel repository al prossimo commit.

2) Per spostare il contenuto **della staging area nel repository** effettuiamo il commit:

- git commit -m “Fixing bug #22”

3) Per annullare alcune modifiche prima di effettuare il commit:

- git reset HEAD -- filename (rimuove filename dall'area di stage)
- git checkout – filename (annulla le ultime modifiche effettuate in filename)

Tutti questi comandi agiscono sulla versione **locale** del repository.

Visualizzare / Annullare le modifiche recenti :

1) Per ottenere informazioni sui file nell'area di lavoro o nell'area di stage:

- git status o git status -s (versione ridotta)

2) Per evidenziare ciò che è modificato **ma non** nell'area di stage

- git **diff**

3) Per visualizzare tutti I cambiamenti che **sono già** nell'area di stage

- giff diff --cached

4) Per visualizzare una lista di **tutti I cambiamenti** avvenuti nel repository locale:

- git log oppure git log --oneline (versione ridotta)

```
1677b2d Edited first line of readme
258efa7 Added line to readme
0e52da7 Initial commit
```

- git log -5 (mostrare solo I 5 update più recenti)

Versioning :

L'idea può essere incorporata a vari livelli: Emacs può "salvare" automaticamente le versioni precedenti dei file (generalmente una sola `*` , altrimenti `* 1` ...), oppure addirittura nel *file system*.
Git invece ricrea un suo "file system": blob e tree, ref.

- multi-phase commit: *working directory*, *stage* e *local repository*
- distribuito senza necessariamente server centralizzati: `pull` e `push`
- in un commit è conservato l'insieme delle modifiche (come 'diff') fatte ad un insieme (*change-set*) di file: perciò è associato a un *tree*
- una branch è semplicemente una *reference* mobile a una linea di sviluppo.