Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Ambienti Operativi: Make

Dario Gallucci, Ricercatore, DTI / ISIN (Bachelor Gestionale)
Amos Brocco, Ricercatore, DTI / ISIN (Bachelor Informatica)

Che cosa è make?

- L'utilità `make` è stata realizzata con lo scopo di automatizzare la compilazione di programmi che dipendono da più di un file sorgente.

- Con il comando make, fornito delle istruzioni necessarie, è possibile automatizzare qualunque progetto complesso che possa essere scomposto in comandi di shell.

- *Perché non realizzare uno script bash?*
  - Il comando make risolve autonomamente le dipendenze tra i vari componenti.
  - Verifica lo stato di aggiornamento dei file da produrre ed esegue i comandi indicati solo quando necessario
  - Include regole predefinite per produrre programmi nei linguaggi di programmazione più usati

Make e Makefile

- Il comando make legge le istruzioni da un file di nome `Makefile` che contiene la lista delle istruzioni.
- Le istruzioni sono contenute nel makefile sotto la seguente forma:

```
obiettivo: requisito_1 requisito_2 … requisito_n
    comando_1
    comando_2
    …
    comando_n
```

- Ognuno di questi blocchi può essere visto come una ricetta di cucina:

```
pietanza: ingrediente_1 ingrediente_2 … ingrediente_n
    ricetta
    …
    ricetta
```

- In genere, obiettivi e requisiti sono file. I comandi che seguono, invece, indicano la procedura per creare il file obiettivo a partire dai requisiti.

Esempio di Makefile

- Per compilare un programma scritto in C, ad esempio, è necessario eseguire il compilatore sul sorgente ed indicare il nome del file risultante:

```
hello: hello.c
    cc hello.c -o hello
```

- Dato che make contiene già le istruzioni per compilare un programma in C, lo stesso risultato può essere ottenuto con il seguente makefile:

```
hello: hello.c
```

- Un makefile può essere anche usato per elaborare altri tipi di file, come ad esempio un file audio:

```
audio.ogg: audio.wav
    oggenc -Q audio.wav -o audio.ogg
```

Invocazione di make

- Per eseguire make è sufficiente eseguire il programma make all'interno della directory in cui si trova il Makefile

```
$ make
cc hello.c -o hello
```

- È possibile specificare un diverso Makefile utilizzando l'opzione -f

```
$ make -f audio.mk
oggenc -Q audio.wav -o audio.ogg
```

- Make normalmente esegue il primo obiettivo specificato nel Makefile. Per far eseguire un obiettivo diverso è possibile esplicitarlo nella riga di comando

```
$ make hello
cc hello.c -o hello
```

Wildcards (caratteri jolly)

- Un makefile può contenere sia nell'obiettivo sia nei requisiti gli stessi caratteri speciali che usava bash. In particolare, come nella shell, possono essere usati per espandere un numero qualsiasi di caratteri (`*`), un solo carattere (`?`), la direcory dell'utente (`~`), una famiglia di caratteri (`[…]`) o la sua negazione (`[^…]`).
- Ad esempio la seguente istruzione per make genera un programma usando tutti i file che terminano con .c all'interno della directory corrente:

```
prog: *.c
    cc -o $@ $^
```

- È anche possibile usare le wildcards all'interno dell'obiettivo. In questo caso, ricordatevi che **l'espansione avviene al momento dell'esecuzione** dell'istruzione, per cui un'istruzione come questa non funzioni come atteso:

```
*.o: constants.h
```

Questa istruzione associa a tutti i file **presenti nella directory corrente** che terminano con .o la dipendenza dal file constants.h. Tuttavia i file .o vengono generati di solito come passo intermedio della compilazione, per cui non saranno presenti in una directory di sorgenti mai usata prima.
Vedremo più avanti come definire delle wildcards che utilizzano i nomi dei prerequisiti per generare la regola corretta.

Obiettivi virtuali (Phony targets)

- Normalmente make assume che gli obiettivi indicati sono file, tuttavia è utile a volte creare delle istruzioni che non producono un file come risultato.
- In molti file trovate ad esempio l'obiettivo clean. Questo di solito contiene le istruzioni per cancellare i file che vengono prodotti da make in modo da poter ricominciare tutto da capo nel caso fosse necessario.

```
clean:
    rm -f *.o prog
```

- Se tuttavia nella directory viene creato un file clean, questa istruzione non verrà mai eseguita. Per ovviare a questo problema è possibile esplicitamente indicarlo come obbiettivo virtuale inserendolo tra le dipendenze dell'obiettivo `.PHONY` in questo modo:

```
.PHONY: clean
clean:
        rm -f *.o prog
```

Obiettivi speciali

- Oltre a .PHONY, make supporta alcuni obiettivi speciali:

Obiettivo	Descrizione
`.INTERMEDIATE`	Make considera tutti i suoi requisiti come file temporanei: questi file, se creati durante l'esecuzione di make, saranno cancellati alla fine della sua esecuzione
`.SECONDARY`	Indica i file temporanei che non devono essere cancellati automaticamente
`.PRECIOUS`	I suoi requisiti non sono cancellati anche se make si interrompe per un errore durante la loro generazione
`.DELETE_ON_ERROR`	Cancella tutti i suoi requisiti nel caso make si interrompa per colpa di un errore

Obiettivi virtuali di uso comune

- Tra gli obiettivi virtuali usati nei Makefile, ne esistono alcuni che usano un nome specifico per convenzione, anche se non sono definiti a priori. Li trovate indicati in questa tabella:

Obiettivo	Descrizione
`all`	Effettua tutte le operazioni richieste per generare il programma
`install`	Installa l'applicazione creata nei percorsi predefiniti (Unix)
`clean`	Cancella i file binari prodotti durante il make
`distclean`	Cancella tutti i file che non facevano parte dei sorgenti originali
`check`	Esegue i test contenuti nei sorgenti dell'applicazione

Obiettivi virtuali II (Empty targets)

- Gli obiettivi viruali sono eseguiti sempre, anche se nessuno dei file è stato aggiornato. Come creiamo un obiettivo virtuale che non viene eseguito quando nessuno dei requisiti è cambiato?
- Utilizzando per target un file vuoto ed il comando touch per crearlo o aggiornare la sua data di ultima modifica in questo modo:

```
hello: size hello.o
    ld hello.o -o hello
size: hello.o
    size hello.o
    touch size
```

- Come è possibile intuire l'obiettivo size viene eseguito solo quando il file size è più vecchio del file hello.o

Variabili

- Finora abbiamo visto solo Makefile che contengono informazioni fisse. Per rendere il comando make più potente è necessario poter eseguire operazioni che dipendono, ad esempio, dalle variabili della shell.
- Le variabili in make si scrivono in modo simile a quello di bash: le variabili sono precedute dal simbolo $ e sono racchiuse in parentesi tonde, in questo modo:

```
$(variabile)
```

- Per comodità make include alcune variabili predefinite, tra cui:

Variabile	Descrizione
$(CC)	Contiene il path del comando per compilare i file .c
$(INSTALL)	Contiene il path del comando install, usato per istallare i file sui sistemi Unix
$(LD)	Contiene il path del comando per produrre un file eseguibile a partire dai file intermedi .o
$(MAKE)	Contiene il path del comando make

Definizione di variabili

- In make ci sono due tipi di variabili, quelle ad espansione immediata e quelle ad espansione ritardata. Le prime vengono interpretate non appena vengono lette da make, le seconde invece nel momento in cui devono essere usate.

- Ad esempio nel caso:

```
B = "ciao"
A = $(B)
B = "mondo"
```

  Se stamperemo A e B troveremo che entrambe contengono la stringa “mondo”. Questo perché normalmente in make le variabili sono ad espansione ritardata.

- Per utililzzare variabili ad espansione immediata dovremmo usare invece:

```
B := "ciao"
A := $(B)
B := "mondo"
```

  In questo caso A contterrà la stringa “ciao”, mentre B la stringa “mondo”.

Altre assegnazioni di variabili

- Il comando make offre altri due operatori per definire e modificare le variabili.

- Il primo (`?=`) è detto assegnazione condizionata: il valore viene assegnato alla variabile solo se la variabile non è già stata definita precedentemente. La variabile creata è ad espansione ritardata.

  ```
  OUTPUT_DIR ?= $(PROJECT_DIR)/out
  ```

  In questo modo se la variabile `OUTPUT_DIR` è già stata definita precedentemente, non sarà modificata

- Il secondo comando (`+=`) serve per aggiungere ulteriore contenuto ad una variabile già presente. Il tipo di variabile non viene modificato se già presente, altrimenti viene creata una variabile ad espansione ritardata.

  ```
  OBJECTS += test.o
  ```

Variabili automatiche

- Oltre alle comuni variabili in make esistono alcune variabili che sono generate automaticamente durante l'esecuzione di un obiettivo.
- Le variabili sono le seguenti:

Variabile	Descrizione
$@	Contiene il nome dell'obiettivo
$<	Contiene il nome del file del primo requisito
$?	Contiene i nomi dei requisiti che sono più nuovi dell'obiettivo, separati da spazi
$^	Contiene il nome di tutti i requisiti, separati da spazi. Se uno o più requisiti sono duplicati, saranno indicati solo una volta.
$+	Come $^, ma non rimuove i duplicati.
$*	Contiene il nome dell'obiettivo, privo del suffisso. Il suo uso è sconsigliato (vedi regole di sostituzione).

Esempi di uso delle variabili

- Prendiamo l'esempio della slide 9, possiamo ridefinirlo utilizzando delle variabili:

```
hello: size hello.o
   $(LD) -o $@ $^
size: hello.o
   size $?
   touch size
```

- In questo modo, se supponiamo che il programma “hello” utilizzi un ulteriore file chiamato print.o ci basterà aggiungerlo ai prerequisiti senza dover modificare anche le istruzioni, in questo modo:

```
hello: size hello.o print.o
   $(LD) -o $@ $^
size: hello.o print.o
   size $?
   touch size
```

Regole generali

- Le regole che abbiamo visto fin'ora sono valide per eseguire delle azioni su file il cui nome è noto al momento della compilazione. Tuttavia è comodo poter generare un Makefile che agisce su file che vengono generati durante l'esecuzione di make.
- Prendiamo la regola per comprimere l'audio che avevamo scritto in precedenza e rendiamola generale:

```
%.ogg: %.wav
    oggenc -Q $^ -o $@
```

- Questa regola prende ogni singolo file il cui nome termina in .wav e lo converte nel file con il medesimo nome ma terminante in .ogg utilizzando il comando oggenc.
  - Notate che a differenza dell'asterisco, utilizzando il %, i file .wav vengono considerati singolarmente.
- In alcuni vecchi Makefile potreste trovare una forma antica di regola generale:

```
.SUFFIXES: .wav .ogg
.wav.ogg:
        oggenc -Q $^ -o $@
```

Sebbene questa forma sia equivalente, questo formato è meno chiaro e ne è sconsigliato l'utilizzo.

Path di ricerca

- Supponiamo di tenere i file audio del primo esempio in una directory diversa da quella in cui creeremo i file compressi con make.
- Per risolvere il problema possiamo specificare manualmente il path accanto ad ognuno dei requisiti, tuttavia questo non copre il caso di più directory che contengono i file richiesti né il caso in cui ci siano più directory annidate.
- Per facilitare questo lavoro make possiede il comando vpath:

```
vpath pattern lista-directory
```

- Se ad esempio nel caso precedente i file non compressi fossero stati nella directory “originale” avremmo dovuto aggiungere prima dell'obiettivo:

```
vpath %.wav originale/
```

Funzioni

- Aumentando la complessità degli script diventa necessario ripetere alcune operazioni per file di tipo diverso. Make permette di utilizzare delle funzioni a tale scopo.

- La sintassi per richiamare una funzione è la seguente:

```
$(nome-funzione arg1[,…,argn])
```

- *Attenzione a non lasciare spazi prima e dopo le virgole* perché make li considera parte dei parametri.

- Make offre diverse funzioni già disponibili, che possono essere suddivise nelle seguenti categorie:
  - Funzioni per manipolare le stringhe
  - Funzioni per manipolare percorsi e nomi di file
  - Funzioni varie

Funzioni personalizzate

- Make permette anche di definire delle funzioni personalizzate.
- Per chiamare una funzione si possono usare i seguenti metodi:
  - `$(nome-funzione)` se non devono essere passati parametri.
  - `$(call nome-funzione arg1[,…,argn])` per passare i parametri.
- Una funzione può essere definita invece con:

```
define nome-funzione
   istruzione_1
   …
   istruzione_n
endef
```

- I parametri sono utilizzabili attraverso i riferimenti $1, $2, …, $n.

Principali funzioni per manipolare le stringhe

- Tra le funzioni che make offre per manipolare le stringhe ci sono:

Funzione	Descrizione
$(filter pattern,testo)	Tratta il testo come una lista di parole separati da spazi e ritorna la lista di parole che corrispondono al pattern. **Un solo** carattere % può essere usato come wildcard.
$(filter-out pattern,testo)	Come filter, ma ritorna le parole che non corrispondono al pattern.
$(findstring stringa,testo)	Trova la stringa nel testo, se la trova ritorna stringa. Non permette l'uso di wildcards.
$(subst stringa, rimpiazzo,testo)	Trova la stringa nel testo e la sostituisce con rimpiazzo. Non permette l'uso di wildcards.
$(patsubst pattern, rimpiazzo,testo)	Trova il pattern nel testo e lo sostituisce con rimpiazzo, se il rimpiazzo contiene una wildcard, questa sarà sostituita dalla parte di testo che corrisponde al pattern.

Esempi di funzioni per manipolare le stringhe (I)

- Vediamo come si comporta la funzione filter:

```
testo := sa casa sasso brasato casa%
trova-sa:
    @echo sa trova: $(filter sa, $(testo))
    @echo %sa trova: $(filter %sa, $(testo))
    @echo sa% trova: $(filter sa%, $(testo))
    @echo %sa% trova: $(filter %sa%, $(testo))
```

- Che produce come risultato:

```
sa trova: sa
%sa trova: sa casa
sa% trova: sa sasso
%sa% trova: casa%
```

- Osservate che non esiste modo per trovare “brasato”

Esempi di funzioni per manipolare le stringhe (II)

- La funzione findstring ritorna la stringa cercata, per esempio il seguente makefile:

```
testo := la casa ha due finestre
trova-stringa:
    @echo $(findstring casa,$(testo))
    @echo $(findstring minestre,$(testo))
    @echo $(findstring scuola,$(testo))
    @echo $(findstring tre,$(testo))
```

- prodice come risultato:

```
casa


tre
```

- La funzione patsubst è principalmente usata per modificare o eliminare parti non volute da un obiettivo oppure un requisito:

```
rimuovi-ultimo-slash = $(patsubst %/,%,$(directory))
```

Principali funzioni per manipolare i nomi di file

- Tra le funzioni che make offre per manipolare i nomi di file ci sono:

Funzione	Descrizione
$(wildcard pattern)	Espande i wildcard espressi in stile bash (*, ?, ~ e [...])
$(dir lista)	Ritorna la parte del path che rappresenta la directory in cui si trovano i file elencati in lista
$(notdir lista)	È il complementare di dir, ritorna i nomi dei file togliendo la directory.
$(join lista-directory, lista-filename)	Unisce i nomi di directory con i nomi di file, uno ad uno. Permette di riunire l'output di dir e notdir.
$(suffix lista)	Ritorna le estensioni dei nomi di file presenti in lista
$(basename lista)	Ritorna i nomi dei file in lista privi delle estensioni

Altre funzioni di make

- Esistono altre funzioni rilevanti per l'uso di make, tra cui:

Funzione	Descrizione
$(sort lista)	Mette in ordine le parole nella lista, rimuovendone i duplicati
$(shell command)	Esegue il comando in una sub-shell e restituisce lo standard output. Funziona come l'apice rovesciato (`) di bash.
$(strip testo)	Rimuove gli spazi iniziali e finali da un testo.
$(error testo)	Interrompe l'esecuzione di make stampando il testo preceduto dal nome dello script make e dal numero di riga in cui si è verificato l'errore.
$(warning testo)	Genera un messaggio analogo a quello di errore, senza però fermare l'esecuzione di make.

Funzioni per il controllo di flusso

- Make fornisce due funzioni per il controllo di flusso:

```
$(if condizione,parte-vera,parte-fals)
$(foreach variabile,lista,espressione)
```

- La prima permette di eseguire un espressione diversa se la condizione è vera o falsa. In make una espressione è considerata vera se contiene del testo (anche solo spazi), mentre è falsa solo se è una stringa vuota.
- La seconda esegue ripetutamente una espressione su ognuno dei termini di una lista di elementi, accessibile attraverso la variabile specificata come parametro.

Script shell multi-linea

- I comandi indicati all'interno di un makefile sono eseguiti ad uno ad uno dentro una shell separata. Questo vuol dire che ogni comando non influenza lo stato del successivo.
- Prendiamo il seguente esempio:

```
current-dir:
    @cd /
    @pwd
```

- Ci aspetteremmo che l'output sia la root, tuttavia ci ritornerà la directory di lavoro di make perchè il comando cd viene eseguito in una shell diversa dal successivo pwd
- Per risolvere il problema dobbiamo specificare che i comandi siano eseguiti in un unica shell. Lo script deve essere perciò definito nel seguente modo:

```
current-dir:
    @cd /;      \
    pwd
```

- Usando un backslash per fare l'escape dell'invio, si fa in modo che make assuma che i comandi siano eseguiti come un unico. Tuttavia per informare la shell che si tratta di due diversi comandi dobbiamo specificare il punto e virgola alla fine di ogni comando.