O sistema Make

Video desta aula

O make é um programa utilitário Unix (criado em 1976) responsável por organizar o processo de compilação e ligação do código-fonte de um projeto, para a geração do programa executável.

O Make é uma ferramenta essencial em grandes projetos, quando há muitos arquivos envolvidos, ou quando a compilação usa muitos *flags* e opções.

O makefile

Para compilar um projeto, o programa make lê um arquivo Makefile ou makefile, que contém as definições de comandos de compilação/ligação e as regras de dependência entre os diversos arquivos do projeto.

Um arquivo Makefile é composto de uma ou mais regras, cuja estrutura básica é a seguinte:

```
# comentário
alvo: dependência dependência dependência ...
    receita
    receita
    ...
```

Onde:

- **alvo**: nome do objeto a ser construído (geralmente um nome de arquivo ou uma ação sobre os arquivos)
- **dependência**: alvos (ou arquivos) dos quais este alvo depende para ser construído
- **receita**: comandos necessários para compilar/construir/gerar o arquivo alvo

A endentação das linhas de comando abaixo de cada regra deve ser feita com **tabulação** (TAB) e nunca com espaços em branco.

Exemplo

Considere um programa C composto pelos seguintes arquivos:

escreva.h

```
#ifndef __ESCREVA__
#define __ESCREVA__

void escreva (char *msg) ;

#endif
```

escreva.c

```c
#include <stdio.h>

void escreva (char *msg)
{
  printf ("%s", msg) ;
}
```

hello.c

```c
#include "escreva.h"

int main ()
{
  escreva ("Hello, world!\n") ;
  return (0) ;
}
```

O arquivo a seguir define uma regra mínima para compilar o programa e gerar o executável `hello`:

Makefile

```
hello: hello.c escreva.c escreva.h
    gcc -Wall hello.c escreva.c -o hello
```

A regra acima significa, literalmente:

- `hello` depende de `hello.c`, `escreva.c` e `escreva.h`: se algum destes tiver sido modificado, o alvo `hello` deve ser recompilado;
- para (re)compilar o alvo `hello`, deve-se executar o comando `gcc -Wall hello.c escreva.c -o hello`.

Ao executar o comando `make hello` será gerado o executável `hello`:

```
$ ls
escreva.c  escreva.h  hello.c  Makefile

$ make
gcc hello.c escreva.c -o hello -Wall

$ ls
escreva.c  escreva.h  hello  hello.c  Makefile
```

Caso nenhum alvo seja indicado na linha de comando do `make`, por default o **primeiro alvo** definido no arquivo `Makefile` é ativado. Assim, `make` e `make hello` irão gerar o mesmo resultado, no exemplo anterior.

Uma característica essencial do sistema `make` é a verificação automática da necessidade de reconstruir os alvos, através da análise das datas dos arquivos declarados como dependências: um alvo só é reconstruído se alguma de suas dependências tiver sido modificada (ou seja, se a data de algum dos arquivos correspondentes for mais recente que a data do arquivo-alvo).

Regras de compilação e de ligação

Podem ser definidas regras separadas para compilação e ligação, agilizando a construção de sistemas com muitos arquivos:

Makefile

```
# regras de ligação
hello: hello.o escreva.o
    gcc hello.o escreva.o -o hello

# regras de compilação
hello.o: hello.c escreva.h
    gcc -c hello.c -Wall

escreva.o: escreva.c escreva.h
    gcc -c escreva.c -Wall
```

Assim, como as regras do arquivo `Makefile` acima definem que `hello` depende de `hello.o` e `escreva.o`; assim a receita da regra `hello` só será ativada caso os arquivos `hello.o` ou `escreva.o` sejam **mais recentes** que o arquivo `hello` (ou não existam). O mesmo ocorre entre `hello.o` e `hello.c`, e assim por diante.

Exemplo (o comando `touch` atualiza a data de um arquivo):

```
$ make
make: Nada a ser feito para `hello'.

$ touch hello.c
$ make
gcc -c hello.c -Wall
gcc hello.o escreva.o -o hello -Wall

$ touch escreva.c
$ make
gcc -c escreva.c -Wall
gcc hello.o escreva.o -o hello -Wall

$ touch escreva.h
$ make
gcc -c hello.c -Wall
gcc -c escreva.c -Wall
gcc hello.o escreva.o -o hello -Wall
```

Regras usuais

As seguintes regras são usualmente encontradas em arquivos `Makefile`:

- `all`: constrói todos os alvos definidos no `Makefile`; costuma ser a primeira regra, ativada por default.
- `clean`: remove arquivos temporários como os arquivos objeto (`.o`), backups de editor, etc.
- `purge`: remove os arquivos temporários e os alvos construídos, preservando somente os arquivos-fonte do projeto (também são usados os nomes `mrproper` e `distclean`).

- debug: mesmo que `all`, mas incluindo *flags* de depuração, como `-g` e `-DDEBUG`, por exemplo.
- `test`: compila e executa procedimentos de teste.

```
# regra default (1a)
all: hello

...

# remove arquivos temporários
clean:
    -rm -f *~ *.o

# remove tudo o que não for o código-fonte original
purge: clean
    -rm -f hello
```

Por default, o `make` interrompe a execução quando uma receita resulta em erro; para evitar esse comportamento, pode-se adicionar o caractere "-" no início da receita cujo erro se deseja ignorar (como nas receitas das regras `clean` e `purge` acima).

Variáveis

Variáveis podem ser criadas dentro do `Makefile` para facilitar a escrita de regras envolvendo muitos arquivos. O exemplo a seguir ilustra a sintaxe de criação e uso de variáveis:

Makefile

```
# Makefile de exemplo (Manual do GNU Make)

CFLAGS = -Wall     # flags de compilacao
LDLIBS = -lm       # bibliotecas a ligar

# arquivos-objeto
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects) $(LDLIBS)
main.o : main.c defs.h
    cc -c main.c $(CFLAGS)
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c $(CFLAGS)
command.o : command.c defs.h command.h
    cc -c command.c $(CFLAGS)
display.o : display.c defs.h buffer.h
    cc -c display.c $(CFLAGS)
insert.o : insert.c defs.h buffer.h
    cc -c insert.c $(CFLAGS)
search.o : search.c defs.h buffer.h
    cc -c search.c $(CFLAGS)
files.o : files.c defs.h buffer.h command.h
    cc -c files.c $(CFLAGS)
```

```
utils.o : utils.c defs.h
    cc -c utils.c $(CFLAGS)
clean :
    -rm -f edit $(objects)
```

Na declaração da variável `objects`, observe como quebrar linhas muito longas!

Regras podem ser usadas para alterar o valor de variáveis. No exemplo abaixo, a regra debug adiciona alguns flags à variável CFLAGS e em seguida chama a regra `all`:

```
# compila com flags de depuração
debug: CFLAGS += -DDEBUG -g
debug: all
```

Regras implícitas

O sistema Make possui um conjunto de receitas e regras implícitas para as operações mais usuais. Receitas e regras implícitas não precisam ser declaradas, o que simplifica o arquivo `Makefile`.

A geração de arquivo-objeto (`.o`) a partir do código-fonte C correspondente (`.c`) usa esta receita implícita:

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c fonte.c
```

A ligação de arquivos-objeto (`.o`) para a geração do executável usa esta receita implícita:

```
$(CC) $(LDFLAGS) arq1.o arq2.o ... $(LDLIBS) -o executavel
```

As variáveis usadas nessas regras implícitas têm os seguintes significados:

variável	**significado**	**exemplo**
CC	compilador a ser usado (por default cc)	`CC = clang`
CPPFLAGS	opções para o preprocessador	`CPPFLAGS = -DDEBUG -I/opt/opencv/include`
CFLAGS	opções para o compilador	`CFLAGS = -std=c99`
LDFLAGS	opções para o ligador	`LDFLAGS = -static -L/opt/opencv/lib`
LDLIBS	bibliotecas a ligar ao código gerado	`LDLIBS = -lpthreads -lm`

Eis como ficaria nosso `Makefile` com variáveis e receitas implícitas:

Makefile

```
CFLAGS = -Wall -g  # gerar "warnings" detalhados e infos de depuração

objs = hello.o escreva.o

# regra default (primeira regra)
all: hello

# regras de ligacao
hello: $(objs)

# regras de compilação
```

```
hello.o:   hello.c escreva.h
escreva.o: escreva.c escreva.h

# remove arquivos temporários
clean:
    -rm -f $(objs) *~

# remove tudo o que não for o código-fonte
purge: clean
    -rm -f hello
```

Uma lista de receitas e regras implícitas para várias linguagens pode ser encontrada nesta página.

Tópicos avançados

Este pequeno tutorial apenas "arranha a superfície" do sistema Make, mas é suficiente para o uso em projetos mais simples. Contudo, existem muitos tópicos avançados não tratados aqui, como:

- Variáveis automáticas
- Regras com padrões
- Condicionais
- *Wildcards*
- Makefiles recursivos
- Avaliação de regras em paralelo
- Geração automática de dependências
- ...

Esses e outros tópicos podem ser estudados em maior profundidade no Manual do GNU Make.