Sistemas Operativos

Guia de compilação de programas em C

O presente guia é uma pequena introdução ao uso da *Makefile*, para compilar programas em C. O guia destina-se aos alunos da UC de Sistemas Operativos do ISCTE–IUL, e não dispensa a consulta da documentação própria dos programas (*make*, e *gcc*).

Compilação de programas e módulos em C

A compilação de um programa em C envolve 4 etapas:

- pré-processamento – para expansão de macros;
- compilação – tradução de código fonte em linguagem assembly;
- geração de código máquina (assembler) – tradução do assembly em linguagem máquina;
- ligação (link) – criação final do ficheiro executável.

Cada uma das etapas pode ser feita individualmente recorrendo aos programas `cpp`, `gcc -S`, `as` e `ld`, respectivamente. No entanto, todos os passos podem ser realizados de uma só vez recorrendo ao compilador `gcc`:

```
gcc programa.c –o programa
```

O compilador `gcc` vai processar o ficheiro `programa.c` (executando as quatro etapas) e gerar o ficheiro executável `programa`. O ficheiro executável é aquele que o computador vai "correr" e depende do sistema operativo onde é compilado. Assim, um programa compilado em *Windows* funcionará noutros computadores com o mesmo sistema operativo instalado, mas não em *OS X* ou *Linux*.

O compilador, para além do ficheiro de entrada e do ficheiro de saída, aceita várias opções. As opções são precedidas do carácter `-`. Para compilar um módulo, é necessário usar a opção `-c`, e não é necessário indicar o ficheiro de saída (ele assume por defeito o nome do ficheiro sucedido de `.o`). Por exemplo, se quisermos compilar um módulo cujo código fonte se encontra no ficheiro `modulo.c`, devemos chamar o compilador com os seguintes argumentos:

```
gcc –c modulo.c
```

O compilador irá compilar o ficheiro `modulo.c` e gerar o ficheiro `modulo.o`. Se o nosso programa estiver dividido em vários módulos, por exemplo um ficheiro principal `programa.c` e um módulo `modulo.c`, podemos compilar cada um em separado e no fim juntar:

```
gcc –c modulo.c
gcc –c programa.c
gcc programa.o modulo.o –o programa
```

ou

```
gcc –c modulo.c
gcc programa.c modulo.o –o programa
```

A ordem pela qual são executados os comandos é importante. Pode existir uma dependência na compilação de diferentes ficheiros. No último exemplo, o ficheiro `programa.c` depende do

módulo `modulo.o`. Se alterarmos o ficheiro `programa.c`, é necessário compilá-lo para gerar um novo executável. Se por outro lado alterarmos o ficheiro `modulo.c`, será necessário compilar o módulo e o programa.

Se quisermos usar um "debugger", um programa que permite parar a execução do nosso programa em sítios pré-definidos, executar o programa instrução a instrução, verificar o conteúdo das variáveis, devemos acrescentar a opção `-g`. O debugger por defeito no linux é o *gdb*. A utilização do *gdb* recomenda-se, mas a sua documentação está fora do âmbito do presente guia.

Para terminar, se o nosso programa usar mais do que um módulo, devemos compilar primeiro os módulos, tendo em atenção possíveis dependências entre eles, e por último compilar o programa usando os módulos já compilados:

```
gcc –c modulo1.c
gcc –c modulo2.c
gcc –c modulo3.c
gcc programa.c modulo1.o modulo2.o modulo3.o –o programa
```

Makefile

O exemplo anterior ilustra um caso de um programa que depende de três módulos. Cada vez que fizermos uma alteração a um dos módulos, devemos re-compilar o respectivo módulo e todos os programas ou módulos que dependam dele. Uma solução pode passar por compilar sempre tudo. No entanto, à medida que os programas aumentam em termos de complexidade e tamanho, esta opção é fortemente desaconselhada.

O programa *make* serve precisamente para automaticamente determinar que partes de um programa precisam de ser compiladas ou re-compiladas, e executar os comandos específicos para cada um deles. Para usar o programa *make*, é necessário criar um ficheiro denominado *Makefile*, que descreve as relações entre os programas e módulos, e fornece as instruções para actualizar cada ficheiro em particular. Uma vez criado este ficheiro, a execução do comando,

```
make
```

é suficiente para efectuar todas as compilações e/ou re-compilações necessárias. O programa *make* usa a data e hora dos ficheiros para determinar quais os programas e/ou módulos que precisam de ser actualizados.

Estrutura da Makefile

Uma *Makefile* simples consiste num conjunto de regras com a seguinte forma:

```
1 target... : prerequisites ...
2     command
3     command
4     ...
```

O `target` é usualmente um nome de um ficheiro que é gerado por um programa. Como exemplos de `targets` temos os ficheiros executáveis e os módulos. Um `target` pode ser também o nome de uma acção, por exemplo, "`clean`".

Um `prerequisite` é um ficheiros que é usado como entrada para criar o ficheiro `target`. Um `target` normalmente depende de vários ficheiros.

O `command` é a acção (ou acções) que o programa *make* executa na respectiva regra. Cada linha que represente uma acção deve começar obrigatoriamente pelo carácter *tab* (e não espaços!).

Por defeito o programa *make* executa a primeira regra existente na *Makefile*. Geralmente esta primeira regra costuma ser denominada `all`. Assim, para compilarmos o nosso programa devemos chamar

```
make all
```

ou simplesmente

```
make
```

É possível definir variáveis na *Makefile*, tornado-a mais flexível. Uma variável define-se por um nome, sucedido de `=`, e do conteúdo da variável. Para usar a variável, devemos usar `$()`, com o nome da variável dentro dos parentesis.

Exemplo de Makefile

No exemplo que se segue pretende-se fazer um programa que faz uso de duas operações: uma multiplicação e uma soma. A multiplicação, por outro lado, faz uso também da operação soma. Para ilustrar o processo de compilação, vão-se organizar as operações soma e multiplicação em dois ficheiros C separados, denominados `operacoes1.c` e `operacoes2.c`. Os ficheiros C são os seguintes:

- `programa.c` – código do programa principal;
- `operacoes1.c` – módulo com a implementação da operação multiplicação;
- `operacoes2.c` – módulo com a implementação da operação soma.

As dependências entre os ficheiros são facilmente identificadas: o programa `programa.c` depende do módulo `operacoes1.o` e `operacoes2.o`; o ficheiro `operacoes1.o` depende do ficheiro `operacoes2.o`. Assim, e acrescentando uma regra `clean` para apagar todos os módulos e executáveis criados, a *Makefile* terá o seguinte aspecto:

```
1  all: programa
2
3  operacoes2.o: operacoes2.c
4      gcc -c operacoes2.c
5
6  operacoes1.o: operacoes1.c operacoes2.o
7      gcc -c operacoes.c
8
9  programa: programa.c operacoes1.o
10     gcc programa.c operacoes1.o operacoes2.o -o programa
11
12 clean:
13     rm -f programa *.o
```

Podemos tornar a *Makefile* mais flexível usando variáveis. Isso permite alterar rapidamente as opções de compilação, o próprio compilador, etc., sem ter de rescrever toda a *Makefile*. O ficheiro terá o seguinte aspecto:

```
CC = gcc
FLAGS = -g -Wall

all: programa

operacoes2.o: operacoes2.c
    $(CC) $(FLAGS) -c operacoes.c

operacoes1.o: operacoes1.c operacoes2.o
    $(CC) $(FLAGS) -c operacoes.c

programa: programa.c operacoes1.o
    $(CC) $(FLAGS) programa.c operacoes1.o operacoes2.o -o programa

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

Código exemplo

programa.c

```
#include <stdio.h>
#include "operacoes1.h"

int main(){
    printf("A multiplicacao de 2 por 3 é %d\n", multiplicacao(2,3));
    return 0;
}
```

operacoes1.h

```
#ifndef OPERACOES1_H
#define OPERACOES1_H

int multiplicacao(int, int);

#endif
```

operacoes1.c

```
#include "operacoes2.h"

int multiplicacao(int a, int b){
    int i,c;
    for(i = 0, c = 0; i < b; i++)
        c = soma(c, a);
    return c;
}
```

operacoes2.h

```
#ifndef OPERACOES2_H
#define OPERACOES2_H

int soma(int, int);

#endif
```

operacoes2.c

```
int soma(int a, int b){
    return a + b;
}
```

Regras implícitas

Não é necessário estar sempre a indicar as regras de compilação para cada ficheiro individualmente: o programa *make* depreende pelo tipo de ficheiro o que é necessário fazer. Assim, para actualizar um ficheiro `.o` a partir de um ficheiro `.c`, o programa *make* invocará o comando `cc -c`. Desta forma podemos omitir as regras de compilação dos objectos. A *makefile* terá então o seguinte aspecto mais compacto:

```
CC = gcc
FLAGS = -g -Wall
objectos = operacoes1.o operacoes2.o

all: programa

programa: programa.c $(objectos)
    $(CC) $(FLAGS) -o programa programa.c $(objectos)

operacoes2.o: operacoes2.c

operacoes1.o: operacoes1.c operacoes2.o

clean:
    rm -f programa $(objectos)
```

Super Makefile

O exemplo seguinte mostra o código de uma *Makefile* que compila todos os ficheiros C, sem dependências, que houver na directoria actual, e inclui uma regra `clean` para apagar todos os ficheiros executáveis que resultarem da compilação dos programas C:

```
CC = gcc
CFLAGS = -Wall -g

files = $(patsubst %.c,%,$(wildcard *.c))

all: $(files)

clean:
    rm -f $(files)
```