Sistemas Operativos 2018-19
Guião da 2^a aula prática

LEIC-A / LEIC-T / LETI
IST

Este documento pretende guiar os alunos no contacto com duas ferramentas de grande importância no contexto do desenvolvimento de aplicações em ambiente UNIX – make e gdb.

Também continuaremos a explorar alguns mecanismos de base de programação de *shell*.

Este exercício **não será avaliado**.

Assume-se que os alunos já completaram o Guião da primeira aula prática. Se não for esse o caso, os alunos podem ainda despender algum tempo a completar as modificações pedidas. O material para este guião é o mesmo do do guião anterior, disponível no Fénix sob a secção "Laboratórios".

1 Utilização da ferramenta `make`

O make é uma ferramenta frequentemente utilizada para compilar *software*. Em projetos de qualquer dimensão, a utilização do make oferece uma forma unificada e conhecida de compilar o *software* em questão.

A documentação completa da ferramenta make pode ser consultada em:

`http://www.gnu.org/software/make/manual/make.html`

O make necessita de um ficheiro, habitualmente chamado Makefile, que descreve uma série de alvos (*targets*) e respetivas dependências. Normalmente, tanto os alvos como as dependências são ficheiros: os alvos são ficheiros que se pretendem gerar, enquanto que as dependências são ficheiros de código-fonte. Note que um alvo pode ser uma dependência de outro alvo, sendo estes casos resolvidos automaticamente.

Para além dos alvos e das suas dependências, o ficheiro Makefile deve incluir também os comandos (*receitas*) que permitem gerar os alvos a partir das dependências. Os comandos das receitas têm, obrigatoriamente, de estar numa linha que começa com um `tab`. Ao conjunto de alvo, dependências e receita chama-se regra, e a sua estrutura geral é:

```
alvo1: dependencia1 dependencia2 dependencia3...
        comando1
        comando2
        ...
```

O make é, em particular, útil para compilar *software* pois é capaz de, a partir das dependências descritas na Makefile, determinar quais os ficheiros (alvo) que estão desatualizados e precisam de ser recompilados. Isto torna-se muito vantajoso durante o desenvolvimento de um projeto de dimensão considerável pois poupa no tempo de compilação.

Quando as dependências são mais recentes do que os alvos, ou quando os alvos não existem, o make volta a executar as receitas. Deste modo, quando um ficheiro fonte é atualizado, basta executar make para que todos os passos necessários até à geração do executável sejam realizados.

A Figura 1 ilustra a geração do programa MyProg a partir do ficheiro MyProg.c e dos ficheiros util.c e util.h.

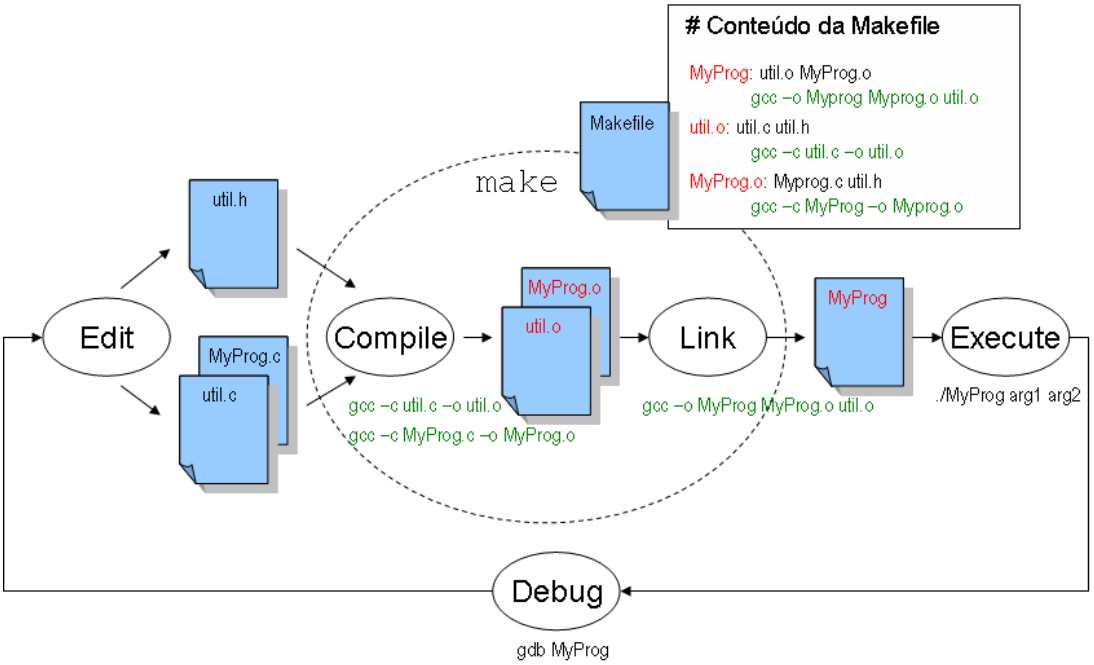

Figura 1: Representação genérica do ciclo de desenvolvimento de uma aplicação

1. Recupere o trabalho da aula anterior ou, caso não o tenha, descarregue da página da cadeira o arquivo circuitrouter_seqsolver_ex01.zip e extraia o seu conteúdo usando o comando `unzip`.

2. Aceda à diretoria para onde extraiu o código-fonte. Utilize o comando `ls -l` para verificar a data do ficheiro coordinate.c.

3. Em vez dos comandos repetitivos utilizados na aula anterior para compilar o programa, vamos escrever uma Makefile para automatizar o processo. Comece por criar um ficheiro chamado Makefile com o seu editor favorito. Adicione a seguinte regra à Makefile:

   ```
   coordinate.o: coordinate.c
           gcc -c coordinate.c -o coordinate.o
   ```

   Guarde o ficheiro e execute o comando `make`. O que aconteceu? Execute de novo o comando `make`. O que mudou entre as duas execuções?

4. Simule uma alteração ao ficheiro coordinate.c com o comando touch coordinate.c (pode saber mais sobre o comando lendo a sua página do manual com `man touch`). Verifique de novo a data de coordinate.c e re-execute `make`. Interprete o resultado.

5. Simule a alteração do ficheiro coordinate.h e execute `make`. Dado que o ficheiro coordinate.c inclui o ficheiro coordinate.h, porque é que o ficheiro coordinate.o não foi gerado de novo? O que necessita de adicionar à regra para resolver o problema?

6. Verifique que ficheiros são incluídos por coordinate.c. Provavelmente a lista de dependências de coordinate.o ainda não está completa. O que falta acrescentar?

7. Adicione ao ficheiro Makefile a seguinte regra

```
grid.o: grid.c grid.h coordinate.h lib/types.h lib/vector.h
	gcc -c grid.c -o grid.o
```

Execute `make`. O que aconteceu? E se fizer `make grid.o`? Note que o comando `make` executado sem argumentos apenas constrói a *primeira regra* (e respectivas dependências e sub-dependências à medida da necessidade).

8. Adicione as restantes regras à Makefile. Tenha em conta que, para a simples invocação do `make` gerar o executável CircuitRouter-SeqSolver, a primeira regra presente na Makefile deve ser aquela que tem como alvo o ficheiro executável. Relembre o exemplo apresentado na Figura 1.

9. Adicione a seguinte regra no final do ficheiro:

```
clean:
	rm -f *.o lib/*.o CircuitRouter-SeqSolver
```

   O que descreve esta regra? Identifique o alvo, as dependências e a receita (comando).

10. Execute `make clean`. O que aconteceu? Note que o comando é executado sempre que esta regra é invocada explicitamente. Isto acontece porque o alvo da regra (clean) nunca chega a ser construído intencionalmente.

    (Sugestão: experimente criar um ficheiro chamado clean (`touch clean`) e correr `make clean`. Interprete o resultado.)

11. O compilador GCC suporta um conjunto vasto de opções (ou *flags*):

    - -O0, -O1, -O2, ... permitem definir vários níveis de otimização a aplicar na geração do código;
    - -Wall e -pedantic permitem detetar mais erros (ou possíveis erros) sendo muito útil que o programador seja alertado para essas situações;
    - -g indica que o compilador deve acrescentar informação adicional ao executável para permitir a sua depuração (*debug*) com o gdb.

    (Nota: este resumo das opções não é exaustivo. Recomenda-se que os alunos consultem a descrição destas *flags* no manual do gcc.)

    A forma mais eficiente de indicar as *flags* desejadas (e mais tarde alterá-las facilmente) consiste no uso de variáveis.

    Adicione no início do ficheiro Makefile a seguinte linha:

```
CFLAGS= -g -Wall -pedantic
```

    Em seguida, sempre que aparece gcc no ficheiro, adicione-lhe imediatamente a seguir $(CFLAGS). Por exemplo:

```
grid.o: grid.c grid.h coordinate.h lib/types.h lib/vector.h
	gcc $(CFLAGS) -c grid.c -o grid.o
```

    Deste modo, evitam-se repetições e fica facilitada a realização de alterações, bastando alterar o conteúdo da declaração da variável.

2 Utilização do *debugger* gdb

A documentação completa da ferramenta de depuração gdb pode ser consultada em:

`http://www.gnu.org/software/gdb/documentation`

O objetivo de um *debugger* é permitir analisar o que está a acontecer dentro de um programa enquanto este está em execução. O gdb permite:

- Correr o programa que se quer analisar.
- Definir condições que permitem parar o programa (*breakpoints*).
- Examinar o que aconteceu quando o programa parou, fazer alterações (por exemplo, alterar o valor de variáveis) e continuar a execução do programa.

Para demonstrar as capacidades do gdb indicam-se em seguida um conjunto de procedimentos que deve efetuar. Não se esqueça de usar a *flag* -g na compilação para garantir que este inclui a devida informação simbólica (nomes de variáveis, funções, etc).

1. Carregue o programa CircuitRouter-SeqSolver no gdb:

   ```
   $ gdb CircuitRouter-SeqSolver
   ```

2. Utilize o comando break (abreviado b) para colocar um *breakpoint* na primeira instrução da função router_solve. Um *breakpoint* pode ser colocado indicando uma função ou uma linha de um ficheiro:

   ```
   (gdb) b router_solve
   ```

   ou

   ```
   (gdb) b router.c:297
   ```

3. Execute a aplicação usando o comando run (abreviado r), redirecionando o ficheiro inputs/random-x32-y32-z3-n64.txt para o *stdin* da aplicação:

   ```
   (gdb) r < inputs/random-x32-y32-z3-n64.txt
   ```

4. A aplicação é executada normalmente e, quando chega ao *breakpoint*, é interrompida pelo gdb. Pode ver onde o código parou utilizando o comando list (abreviado l):

   ```
   (gdb) l
   ```

5. Avance pelo código, linha a linha, utilizando o comando next (abreviado n), imprimindo as variáveis à medida que as vai definindo. Note que não pode observar o valor de variáveis que ainda não foram definidas, tendo de esperar até estar na linha seguinte à da definição para poder inspecionar o seu valor.

   ```
   (gdb) n
   (gdb) p argPtr
   (gdb) p *routerArgPtr
   (gdb) n
   (gdb) n
   (gdb) p *mazePtr
   ```

6. Defina um novo *breakpoint* na linha 333 do ficheiro atual com o comando breakpoint. Pode continuar a execução sem interrupções até ao próximo *breakpoint* utilizando o comando continue (abreviado c).

```
(gdb) b 333
(gdb) c
(gdb) p *srcPtr
(gdb) p *dstPtr
```

   É válido inspecionar a memória apontada por coordinatePairPtr? Porquê?

7. Pode listar os *breakpoints* definidos (1 e 2, atualmente) e pode fazer o seu *disable*. Experimente:

```
(gdb) info b
(gdb) disable 1 2
(gdb) info b
```

8. Outra maneira de navegar pelo programa é usando o comando step (abreviado s), que funciona como o next só que entra dentro das funções que executa. Experimente entrar na função grid_copy, execute os *asserts* e observe as duas grelhas:

```
(gdb) s
(gdb) list
(gdb) n
(gdb) n
(gdb) n
(gdb) p *srcGridPtr
(gdb) p *dstGridPtr
```

9. Com o gdb é possível executar a função atual até à sua conclusão usando o comando finish, abreviado fin. Observe com list como retornou à função router_solve na instrução seguinte à chamada a grid_copy.

```
(gdb) fin
(gdb) l
```

10. Saia do gdb com quit (abreviado q) ou premindo Ctrl-D:

```
(gdb) q
```

O gdb é especialmente útil quando um programa rebenta (por exemplo, devido a *segmentation fault*) não dando nenhuma indicação onde ocorreu o erro. Nestes casos, o gdb pode ser utilizado para analisar o programa após este terminar, como se fosse uma autópsia. Para ilustrar esta capacidade, proceda do seguinte modo:

1. Abra o ficheiro grid.c, apague a implementação existente da função grid_print e coloque esta:

```
void grid_print(grid_t *gridPtr) {
    long i, j, k;
    for (i = 0; i < gridPtr->depth; ++i) {
        printf("[z␣=␣%ld]\n", i);
        for (j = 0; j < gridPtr->height; ++j) {
            for (k = 0; k < gridPtr->width; ++k)
                printf("%4ld", grid_getPoint(gridPtr, i, j, k));
            printf("\n");
        }
        printf("\n");
    }
}
```

2. Compile e execute o programa, passando a *flag* -p ao programa para imprimir as rotas finais. O que se sucede deve ser algo muito parecido ao seguinte:

```
$ make
$ ./CircuitRouter-SeqSolver -p <inputs/random-x32-x32-z3-n64.txt
Maze dimensions = 32 x 32 x 3
Paths to route  = 64
Paths routed    = 58
Elapsed time    = 0.020839 seconds

Routed Maze:
[z = 0]
Segmentation fault (core dumped)
```

   Irá ocorrer o erro *segmentation fault (core dumped)*.

3. Utilize o comando ls para averiguar a existência de um ficheiro core na diretoria atual. Se existir, pode lançar o gdb com esse ficheiro core:

```
$ gdb CircuitRouter-SeqSolver core
```

   Se não existir o ficheiro core, utilize o comando `ulimit -c` para consultar o tamanho máximo permitido para os ficheiros core. Caso seja 0, para permitir que sejam gerados ficheiros core com dimensão até 100MB use

```
$ ulimit -c 100000000
```

   Execute de novo o programa para gerar o ficheiro core. Se ainda não existir nenhum ficheiro core, é possível que a gestão desses ficheiros esteja a ser tratada por um programa chamado systemd. Pode lançar o depurador sobre o último core gerado com

```
$ coredumpctl gdb
```

4. Lançado o gdb, deve observar algo parecido com o seguinte. Note que os endereços variam consideravelmente de máquina para máquina, por isso os endereços mostrados neste guião são apenas um exemplo.

```
Core was generated by `./CircuitRouter-SeqSolver -p'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  grid_getPoint (gridPtr=0x562e3f5b3090, x=0, y=0, z=11) at grid.c:162
162         return *grid_getPointRef(gridPtr, x, y, z);
```

   Para saber qual o caminho percorrido pelo programa até chegar a esse ponto, use o comando *backtrace* (abreviado bt):

```
(gdb) bt
#0  grid_getPoint (gridPtr=0x562e3f5b3090, x=0, y=0, z=11) at grid.c:162
#1  0x0000562e3e577005 in grid_print (gridPtr=0x562e3f5b3090) at grid.c:238
#2  0x0000562e3e577ef2 in maze_checkPaths (mazePtr=0x562e3f5a1260,
    ↪ pathVectorListPtr=0x562e3f5a4550, doPrintPaths=1) at maze.c:356
#3  0x0000562e3e5766dc in main (argc=2, argv=0x7ffdca2fbb78) at CircuitRouter-
    ↪ SeqSolver.c:199
```

   O primeiro número em cada linha indica o nível em que essa função está, começando pela função onde o programa rebentou (neste caso, grid_getPoint). Note que é frequente serem mostradas funções que são de sistema. Nesses casos, obviamente, o que interessa é a última função que correu do nosso programa, pois será aí que deverá estar o erro.

   Observe como o gdb mostra os argumentos passados às funções ao longo da pilha de chamadas (*call stack*). Compare os argumentos passados à função grid_getPoint com as dimensões da grelha:

```
(gdb) p *gridPtr
$1 = {width = 32, height = 32, depth = 3, points = 0x562e3f5ad080, points_unaligned
    ↪  = 0x562e3f5ad060}
```

   Pela pilha de chamadas podemos concluir que a função que chamou grid_getPoint com argumentos inválidos foi a função grid_print. Pode examinar a função imediatamente acima na pilha com o comando up ou saltar para uma posição arbitrária com o comando frame.

```
(gdb) frame 1
#1  0x0000562e3e577005 in grid_print (gridPtr=0x562e3f5b3090) at grid.c:238
238             printf("%4ld", grid_getPoint(gridPtr, i, j, k));
```

   Podemos ver que o problema aconteceu na linha 238 e que pelo menos o índice k não respeita os limites da grelha.

```
(gdb) p k
$5 = 11
(gdb) p gridPtr->depth
$6 = 3
```

   Num contexto mais realista, haveria agora que definir *breakpoints*, correr o programa de novo e seguir passo a passo a execução da função até identificar a origem do problema.

5. Corrija o programa.

3 Introdução à programação em *shell*

As *shells* representam uma das principais interfaces para os sistemas operativos. Ao longo do tempo têm sido desenvolvidas em ambientes UNIX/Linux várias *shells*, e.g. Bourne shell (sh), Bourne Again shell (bash), Korn shell (ksh), e C shell (csh), que oferecem diferentes linguagens de *scripting*. Neste guião vamos aprender alguns elementos básicos de programação para uma das *shells* mais populares, o *bash*. Em particular, vamos focar-nos na utilização de variáveis e de construções para suportar iterações.

3.1 Utilização interativa e através de programas (*scripts*)

O *bash* é a *shell* que é executada por omissão quando é aberto um *terminal* nos PCs dos laboratórios. Para verificar qual é a *shell* que está a ser utilizada podem inspecionar a variável `SHELL`:

```
$ echo $SHELL
/bin/bash
```

Caso o terminal esteja a usar uma *shell* diferente, podem forçar a utilização da *shell bash* através deste comando:

```
$ bash
```

Tal como dá para executar comandos interactivamente em Python e mais tarde juntá-los num *script*, também é possível fazer o mesmo em *shell*. Para isso, basta criar um ficheiro com os comandos e invocar a *shell*.

Crie o seguinte ficheiro usando o seu editor de texto favorito:

```
#!/bin/bash
echo Nesta aula vamos aprender a
echo desenvolver os nossos primeiros scripts em bash!
```

Guarde o ficheiro com nome meu_script.sh e torne-o executável usando o comando `chmod`:

```
$ chmod +x meu_script.sh
```

e execute-o:

```
$ ./meu_script.sh
Nesta aula vamos aprender a
desenvolver os nossos primeiros scripts em bash!
```

Notas importantes:

- A extensão ".sh" não é obrigatória, mas é uma convenção útil pois permite identificar facilmente *scripts* de *shell*.
- A primeira linha do *script* indica ao sistema operativo qual é o interpretador que deve ser utilizado para executar o *script*. Mais em detalhe, os primeiros dois caracteres (`#!`, pronunciados "she-bang" ou "hash-bang" em Inglês) são lidos pelo sistema operativo durante a chamada de sistema *execve*(), que será brevemente introduzida nas teóricas. Isto indica ao núcleo que se trata de um *script* cujo interpretador encontra-se no caminho absoluto especificado na restante parte da primeira linha do ficheiro (neste caso `/bin/bash`).
- Também é possível lançar um *shell script* usando esta sintaxe:

  ```
  $ bash meu_script.sh
  ```

Note que, neste caso, o ficheiro meu_script.sh não tem de ser executável.

- É possível executar um *script* em modalidade *debug* passando a *flag* `-x` ao interpretador bash, quer quando isto é invocado explicitamente pelo terminal:

```
$ bash -x meu_script.sh
```

quer quando o interpretador é especificado na primeira linha do *script*:

```
#!/bin/bash -x
echo Nesta aula vamos aprender
echo desenvolver os nossos primeiro scripts em bash!
```

3.2 Variáveis

Segue uma demonstração da utilização de duas variáveis, `a` e `b`.

```
$ a=10
$ echo $a
10
$ b=a
$ echo $b
a
$ b=$a
10
$ b=batata
$ echo Temos $a ${b}s
Temos 10 batatas
$ echo Temos $a $bs
Temos 10
```

Notas importantes:

- Não se usam espaços à volta do operador de atribuição.
- Para ler o valor contido numa variável usa-se um `$` imediatamente antes do nome.
- É boa prática, para evitar ambiguidades, usar chavetas (`{}`) à volta do nome da variável. Por exemplo, no último comando, ao usarmos `$bs` estaríamos a obter o valor da variável `bs`, o que não é o pretendido.

3.2.1 Expansão de comandos

Observe os exemplos reportados abaixo em que é usado o operador de *command expansion*, i.e., `$(command)`, para capturar o *output (stdout)* dum comando e atribuí-lo a uma variável.

```
$ echo batata doce >terra
$ cat terra
batata doce
$ v=$(cat terra)
$ echo $v
batata doce
```

Nota: Em alternativa à sintaxe `$(command)`, é possível usar também `` `command` `` (acento grave), como exemplificado de seguida.

```
$ myvar=`cat terra`
$ echo $myvar
batata doce
```

3.3 Iterações: ciclo *for*

O bash suporta vários operadores para implementar ciclos, e.g., *for*, *while* e *repeat*. Neste guião vamos introduzir apenas a sintaxe do operador *for*.

O ciclo *for* funciona com uma variável de iteração e uma lista de valores a iterar, tal como exemplificado por este *script*:

```
#!/bin/bash
echo Primeira demo do operador for
for i in 0 1 2
do
        echo $i
done

echo ------
echo Implementação equivalente usando expansão de comandos e o comando seq
echo Para compreender este exemplo, experimente executar seq 0 2 numa shell
echo Consulte o manual do comando seq (man seq)
for i in $(seq 0 2)
do
        echo $i
done

echo ---

echo O output do comando ls é expandido na variável de iteração f
for f in $(ls /usr/include/*.h)
do
        echo filename: $f
done

echo Neste exemplo vamos executar o programa CircuitRouter-SeqSolver
echo passando-lhe como input todos os ficheiros presentes na pasta inputs.
echo NOTA: Para que o script funcione, tem de ser executado na mesma pasta
echo onde se encontra o executável CircuitRouter-SeqSolver
for input in inputs/*.txt
do
   echo ==== ${input} ====
   ./CircuitRouter-SeqSolver -p <"$input"
 done
```

3.4 Exercício

Com base nas noções aprendidas neste guião (e no anterior), desenvolva um *script* que analisa todos os circuitos na pasta *input* do código base do projeto e que imprime, por cada circuito, o nome relativo do ficheiro, o número de linhas do ficheiro e o número de interconexões no circuito. (Sugestão: consulte os manuais dos comandos `wc` e `grep`.)