Projecto de Sistemas Operativos 2017-18
`heatSim`
Exercício 0 (parte 2)

LEIC-A / LEIC-T / LETI
IST

Abstract

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.

1 Introdução

Assume-se que os alunos já completaram o Exercício 0, parte 1, tendo realizado a versão sequencial do simulador `heatSim`. Se não for esse o caso, os alunos podem ainda dispender algum tempo a completar o simulador ou podem recorrer à solução `heatSim_ex01_solucao.zip` disponível na página da disciplina.

Neste guia são sugeridos vários exercícios para os alunos se familiarizarem com duas ferramentas muito úteis no desenvolvimento de aplicações: make e gdb.

2 Utilização da ferramenta `make`

A documentação completa da ferramenta make pode ser consultada em:

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

O make pega num ficheiro, habitualmente chamado Makefile, que descreve alvos (normalmente são ficheiros que se pretendem gerar) e respectivas dependências (habitualmente são ficheiros fonte, de que os alvos dependem). Notar 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. As receitas têm, obrigatoriamente, de estar numa linha que começa com um `tab`.

O make é frequentemente utilizado para construir software (ver Figura 1), pois permite especificar, por exemplo, como "os ficheiros objecto dependem dos respectivos ficheiros fonte" e como "um executável depende dos respectivos ficheiros objecto e eventuais bibliotecas".

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 actualizado, basta executar make para que todos os passos necessários até à geração do executável sejam realizados.

1. Recupere o trabalho da aula anterior ou, caso não o tenha, descarregue da página da cadeira o arquivo `heatSim_ex01_solucao.zip` .

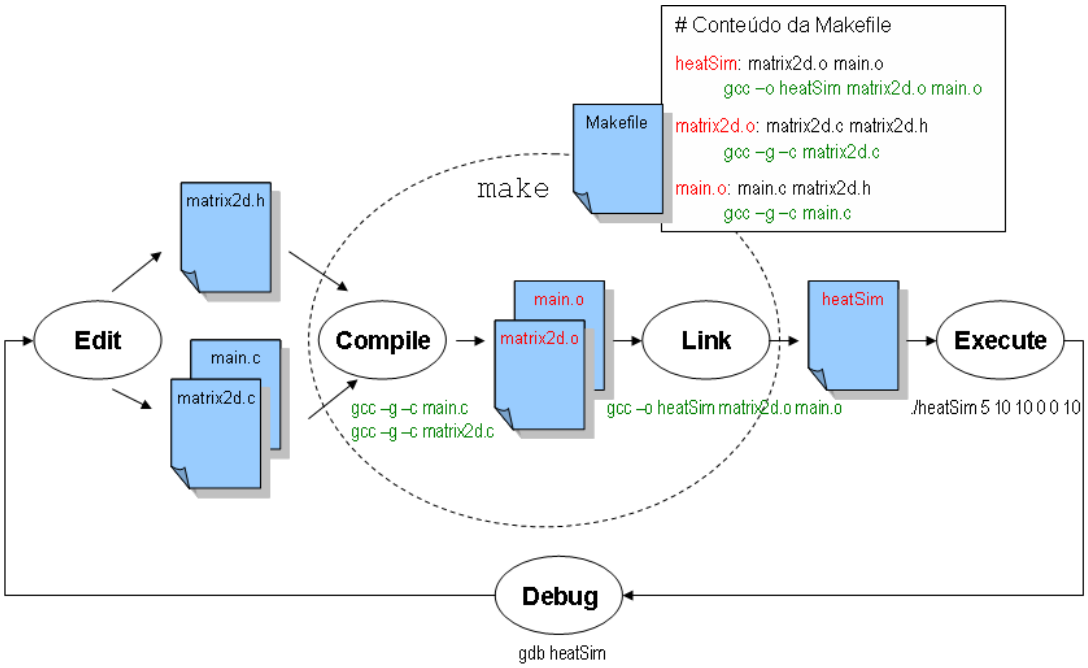

Figure 1: Representação do ciclo de desenvolvimento da aplicação heatSim

2. O arquivo indicado contém o ficheiro Makefile. Assegure-se que esse ficheiro fica na mesma directoria em que se encontram os ficheiros main.c, matrix2d.c e matrix2d.h.

   Use o comando `ls -l` para verificar que ficheiros existem na directoria e qual a data em que foram modificados. Em seguida, execute:

   ```
   make
   ```

   Volte a executar `ls -l` e verifique o que aconteceu.

3. Apague o ficheiro matrix2d.o com o comando rm matrix2d.o. Verifique se o ficheiro foi eliminado usando o comando ls.

   Re-execute make e interprete o sucedido.

4. Utilize o comando `ls -l` para verificar a data do ficheiro main.c.

   Simule uma alteração ao ficheiro main.c com o comando touch (lembrar que pode fazer `man touch` para obter informação).

   Verifique de novo a data de main.c e re-execute make. Interprete o resultado.

5. Simule a alteração do ficheiro matrix2d.h e execute make.

   Porque razão todos os ficheiros foram gerados?

6. Simule a alteração do ficheiro matrix2d.o .

   O que acontece quando faz make matrix2d.o ?

   E se agora fizer make ?

7. Retire a dependência do ficheiro matrix2d.h da regra matrix2d.o da makefile.

   Repita o procedimento do ponto 5. Explique a diferença no resultado.

8. Adicione a regra seguinte no final do ficheiro (recorde que a segunda linha tem de começar com um `tab`).

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

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

9. Execute `make clean`. O que aconteceu? Notar que o comando é executado sempre que esta regra é invocada explicitamente.

10. Observe que as receitas contêm *flags* repetidas. Isso pode ser evitado utilizando variáveis.

    Analise o ficheiro Makefile_v2 existente no arquivo `heatSim_ex01_solucao.zip` e estude o uso de variáveis que permitem explicitar qual a ferramenta que será usada nas receitas (gcc) e quais as *flags* utilizadas. Deste modo, evitam-se repetições e fica facilitada a realização de alterações.

3 Utilização do *debugger* gdb

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

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

O objectivo de um *debugger* é permitir analisar o que está a acontecer dentro de um programa quando 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 efectuar.

1. Execute o programa `heatSim` no gdb:

   ```
   gdb --args ./heatSim 5 10 10 0 0 10
   ```

2. Utilize o comando break (abreviado b) para colocar um *breakpoint* na primeira instrução da função dm2dNew. Um breakpoint pode ser colocado indicando uma função ou uma linha de um ficheiro:

   ```
   (gdb) b dm2dNew
   ```
   ou
   ```
   (gdb) b matrix2d.c:18
   ```

3. Execute a aplicação usando o comando run (abreviado r):

   ```
   (gdb) r
   ```

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

   ```
   (gdb) l
   ```

5. Pode agora ver o valor das variáveis que estão no *scope* da função usando o comando print (abreviado p):

   ```
   (gdb) p lines
   (gdb) p matrix
   ```

6. Defina novo *breakpoint* na linha 35 (do ficheiro actual), continue a execução até esse *breakpoint* (comando continue ou apenas c) e faça o print das variáveis que se indicam:

   ```
   (gdb) b 35
   (gdb) c
   (gdb) p matrix->data
   (gdb) p *matrix->data
   (gdb) p matrix->data[5]
   ```

   Qual a diferença entre os vários comandos print anteriores?

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. O programa pode ser executado passo a passo usando o comando step (abreviado s), o qual permite entrar nas funções por onde passa. Execute:

   ```
   (gdb) s
   (gdb) s
   ```

   Regressou ao ficheiro main.c, linha 112.

9. Também é possível executar o programa passo a passo usando o comando next (abreviado n), o qual salta as funções pode onde passa (na realidade, entra na função, executa-a na totalidade e retorna, passando à instrução seguinte).

   Execute os comandos:

   ```
   (gdb) l
   (gdb) p matrix
   (gdb) p matrix_aux
   (gdb) n
   (gdb) p matrix_aux
   (gdb) p *matrix_aux
   ```

   Notar que executou a função dm2dNew e que a matriz criada foi atribuída a matrix_aux. Os dois últimos comandos print mostram o valor do apontador e o conteúdo da estrutura.

10. Saia do gdb com quit 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, proceder do seguinte modo:

1. Copie o ficheiro bug.c (presente em heatSim_ex01_solucao.zip) para main.c, gere novo programa e execute-o:

   ```
   cp bug.c main.c
   make clean
   make
   ./heatSim 5 10 10 0 0 10
   ```

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

   Se executar o comando ls verá que existe um ficheiro core na directoria actual. Se não for esse o caso, execute o comando seguinte para permitir que sejam gerados ficheiros core com dimensão até 10MB.

   ```
   ulimit -c 10000000
   ```

   Execute de novo o programa para gerar o ficheiro core.

   Nota: Em certos sistemas o ficheiro core não é gerado na directoria actual, sendo os *coredumps* geridos por um programa chamado systemd. O acesso aos *coredumps* é feito assim: coredumpctl gdb

2. Use o gdb para saber onde ocorreu o erro, fazendo:

   ```
   gdb heatSim core
   ```

3. Para saber qual a instrução que originou o erro execute o comando *backtrace* (abreviado bt):

   ```
   (gdb) bt
   ```

   O comando *backtrace* mostra, de baixo para cima, a lista de funções que foram executadas até ao ponto em que o programa terminou. Por exemplo:

   ```
   #0 0x000055d14d104a14 in simul (matrix=0x55d14ea91010,
       matrix_aux=0x55d14ea911c0, linhas=7, colunas=7, numIteracoes=10)
       at main.c:36
   #1 0x000055d14d104f2b in main (argc=7, argv=0x7ffcebad8ca8) at main.c:156
   ```

   O primeiro número em cada linha indica o nível em que essa função está, começando pela função onde o programa rebentou (neste caso, simul). Notar 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.

   Para observar as variáveis que estão no nível da nossa última função que foi executada usar o comando frame seguido do nível correspondente. No presente caso, a função simul encontra-se no nível 0 pelo que se deverá executar:

   ```
   (gdb) frame 0
   ```

   Neste momento pode consultar o conteúdo das variáveis que estão no *scope* da função simul. Adicionalmente aparece a indicação:

   ```
   #0 0x000055d14d104a14 in simul (matrix=0x55d14ea91010,
       matrix_aux=0x55d14ea911c0, linhas=7, colunas=7, numIteracoes=10)
       at main.c:36
   36      dm2dSetEntry(aux, i, j, value);
   ```

que assinala que o problema ocorreu na linha 36, onde se encontra a chamada dm2dSetEntry(aux, i, j, value); .

Ao consultar as variáveis envolvidas é fácil constatar que o apontador aux tem o valor NULL (0) e foi isso que originou o problema.

```
(gdb) p aux
```

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.

4. Corrija o programa.