Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

SISTEMAS OPERATIVOS I

Parte IV

Abril de 2006

Nuno Malheiro
Maria João Viamonte
Berta Batista
Luis Lino Ferreira

Sugestões e participações de erros para:
ntm@dei.isep.ipp.pt

BIBLIOGRAFIA

"Documentation for GDB version 6.3" de *the GDB developers*, http://sources.redhat.com/gdb/download/onlinedocs/

ÍNDICE

1 INTRODUÇÃO

O objectivo deste documento é a apresentação e interacção com o *debugger* da GNU (GDB). O processo de *debugging*, permite a execução de um programa de forma controlada, sendo fundamental para detecção e correcção de falhas de funcionamento. Um *debugger* permite a interrupção, alteração e verificação de um programa em execução, avaliando em cada momento o valor dos seus vários parâmetros (variáveis, código-fonte, stack, etc.), permitindo a detecção de eventuais erros.
Apesar de neste documento nos cingirmos à lingagem C, o GDB consegue ser igualmente eficaz com programas escritos em C, C++, Objective-C, Fortran, Java, Pascal, assembly, Modula-2 ou Ada.
Este documento assume que o leitor possui conhecimento da linguagem C.

2 Um pequeno exemplo de sessão de debug

2.1 A compilação para debug de um pequeno programa em C

Um exemplo possível será uma versão do "Olá Mundo" ligeiramente mais complexa. Esta é a nossa versão desse programa:

```
#include <stdio.h>

void ola (const char * nome)
{
  printf ("Ola, %s!\n", nome);
}

int main (void)
{
  ola ("Mundo");
  return 0;
}
```

Assumimos que o código fonte se encontra guardado num ficheiro chamado "ola.c". Para compilar o ficheiro "ola.c" para debug utilizando o GDB, usa-se o seguinte comando:

```
$ gcc -g ola.c -o ola
```

Note-se que na fase de debug, já deve ter sido efectuada no programa uma correcção de erros e verificação de avisos.
Este comando compila o ficheiro fonte para código máquina, guardando-o em seguida num ficheiro executável chamado "ola". Além do código necessário à execução, é guardada ainda informação para *debug* no formato nativo do sistema operativo.

2.2 Uso simples do GDB

Agora que o pequeno programa "Olá Mundo" está compilado e contém informação de debug, vamos verificar várias formas de o executar em ambiente de debug. A primeira tarefa é executar o GDB para efectuar o debug do programa "ola".

```
$ gdb ola
```

Após esta linha de comando, o gdb está pronto a aceitar comandos, como se pode ver pelo aparecimento da sua *prompt*:

```
(gdb)
```

Para executar directamente o programa da forma mais simples, no ambiente gdb usa-se o comando run, cuja execução causa:

```
(gdb) run
Starting program: /users/2b/ntm/tmp/ola
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000
Ola, Mundo!

Program exited normally.
(gdb)
```

Neste caso o programa executou como se fosse chamado da linha de comando. As vantagens da sua execução através do *debugger* são nulas neste caso.

2.3 Execução passo a passo

Imagine-se agora que se pretende uma execução instrução a instrução do programa "ola.c". Para isso seria necessário um *breakpoint* na função *main*, seguido de ordem de execução. As linhas seguintes ilustram esta funcionalidade:

```
(gdb) break main
Breakpoint 1 at 0x80483bf: file ola.c, line 10.
(gdb) run
Starting program: /users/2b/ntm/tmp/ola
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000

Breakpoint 1, main () at ola.c:10
10                ola ("Mundo");
(gdb)
```

O debugger iniciou a execução do processo "ola" e interrompeu-o na linha número 10, que é a primeira instrução da função main. A partir deste momento, poder-se-á efectuar uma execução passo a passo usando o comando **`step`** (ou apenas **`s`**). A linha imediatamente anterior ao prompt apresenta a instrução que será executada de seguida. Apresenta-se de seguida a execução completa comentada:

```
10                ola ("Mundo");
(gdb) s
ola (nome=0x80484e6 "Mundo") at ola.c:5
```

Neste passo entra-se na função "ola", sendo colocado na primeira instrução dessa função.

```
5                 printf ("Ola, %s!\n", nome);
(gdb) s
Ola, Mundo!
```

Neste passo executa-se a instrução de impressão, cujo resultado se pode ver no terminal.

```
6       }
(gdb) s
main () at ola.c:11
```

Neste passo executa-se a instrução de final de função, voltando a execução ao sitio de onde foi efectuada a chamada (função main).

```
11                return 0;
```

```
(gdb) s
```

Já de volta à função main, é executada a instrução de retorno da função.

```
12      }
(gdb) s
```

A finalização do programa envia-o de volta à função de entrada da biblioteca de C incluída pelo GCC. Esta biblioteca não pertence ao programa, logo não existe informação de *debug* tal como o número de linha de execução.

```
0xb7e30e40 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) s
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
(gdb)
```

Este último passo termina a execução do programa.

3 Tabela de comandos do GDB

A tabela seguinte mostra os vários comandos do GDB:

run *[argumentos]*	Inicia a execução como se tivesse passado os argumentos na linha de comando.
break *ponto*	Cria um breakpoint no ponto indicado. Ponto pode ser o nome de uma função ou um número de linha. Cada breakpoint é associado a um número identificativo.
delete *N*	Elimina o breakpoint com o número N. Se este número for omitido elimina todos os breakpoint activos.
info *algo*	Mostra informação sobre *algo.* Quando o parâmetro é omitido mostra todos os subcomandos
help *comando*	Dá informação sobre o comando indicado. Quando o parâmetro é omitido mostra todos os subcomandos
Step	Executa uma linha do comando e pára na linha Seguinte. Se a linha contiver uma chamada de função, a linha seguinte é a primeira instrução dessa chamada.
Next	Igual ao step mas se a linha corrente contiver uma chamada de função executa-a sem mostrar cada uma das suas instruções.
Finish	Equivalente a uma sucessão de next's até chegar ao final da função corrente.
Continue	Executa o programa sem debbuging até ao próximo breakpoint.
set *algo*	Efectua a atribuição de *algo*. Quando o parâmetro é omitido mostra todos os subcomandos de atribuição
Where	Faz uma lista das funções que trouxeram o programa até ao ponto em que está actualmente. Equivalente a "info stack"
print *variável*	Mostra o valor actual da variável, mas não o actualiza à medida que o programa vai decorrendo.
display *variável*	Mostra o valor actual da variável, actualizando-a à medida que o programa vai decorrendo.
Quit	Abandona o gdb.
List	Mostra a rotina em execução.

4 Execução e Interrupção

O objectivo principal do uso de um *debugger* é a capacidade de interrupção de um programa durante a sua execução para que se possam investigar as razões de eventuais problemas. Dentro do GDB existem várias formas de um programa poder ser parado e, para além disso, existem funcionalidades que permitem explorar o funcionamento do programa tais como variáveis, expressões, a *stack* de execução, etc. A linha de comando seguinte mostra informação sobre este assunto:

```
(gdb) help breakpoints
```

4.1 Pontos de quebra - Breakpoints

Um ponto de quebra corresponde a uma localização do programa onde se pretende parar a execução para que se possa verificar o funcionamento do programa. O ponto de quebra pode ser atribuído através de um nome de função ou número de linha.
A seguinte linha de comando cria dois pontos de quebra, um na função main e o outro na linha 22.

```
(gdb) break main
(gdb) break 22
```

Após a execução destes comandos, existem no programa dois breakpoints. A execução parará quando chegar a qualquer deste pontos, permitindo ao utilizador adquirir controlo sobre a execução do programa.
Se se pretender limpar os pontos de quebra, é possível através do comando clear, como mostram as próximas linhas de comando:

```
(gdb) clear main
(gdb) clear 22
```

Estas duas linhas apagaram os pontos de quebra criados anteriormente. A partir deste momento a execução do programa não será parada nesses pontos.

4.2 Pontos de vigia - Watchpoints

Os pontos de vigia são essencialmente semelhantes aos pontos de quebra, com a diferença que ao invés da quebra se dar numa localização específica, é permitida a definição de uma expressão que interrompe o programa quando for verdadeira. Assim são possíveis os seguintes pontos:

```
(gdb) watch i>8
(gdb) watch j=0
```

Os pontos de vigia são automaticamente apagados quando as variáveis deixam de existir. Se se estiver a vigiar uma variável de uma função, o ponto de vigia desaparece quando a função termina.

4.3 Execução em modo interrompido

A execução em modo interrompido pretende dar a quem está a fazer *debug* de uma aplicação a capacidade de ir executando um programa gradualmente, verificando o seu funcionamento. Vamos ilustrar esta secção com um programa de exemplo:

```
#include <stdio.h>

void fnc(char *n)
{
   printf("Valor de n: %s\n",n);
```

```
}
int main(void)
{
   fnc("SOP1");
   return 0;
}
```

Tal como já foi visto, é necessário compilar o programa incluindo informação de debug:

```
$ gcc -g teste.c -o teste
```

Falta depois chamar o GDB para iniciarmos uma sessão de debug do programa "teste".

```
$ gdb teste
```

Para executar um programa em modo interrompido será necessário criar inicialmente um ponto de quebra inicial. Para isso vamos criar o ponto de quebra mais comum, ou seja, uma quebra na função main:

```
(gdb) break main
```

Após esta definição pode iniciar-se a execução que parará quase imediatamente no início da primeira instrução da função main, mostrando o número e a linha de código onde se contra parado.

```
(gdb) run
```

A instrução onde pára é a chamada à função "ciclo".

```
Breakpoint 1, main () at t.c:9
9          fnc("SOP1");
```

Neste ponto o utilizador pode decidir se entra para a função, executando-a passo a passo ou se, alternativamente executa toda a função como se se trata-se de uma só instrução.
Para executar passo a passo usa-se o comando **`step`**:

```
(gdb) s
```

O comando **`step`**, entra na função, passando a próxima instrução a ser a primeira instrução da função "ciclo". Usando este comando, a próxima instrução será:

```
fnc (n=0x80484ec "SOP1") at t.c:5
5          printf("Valor de n: %s\n",n);
```

Se, alternativamente for pretendida uma execução de toda a função é necessário o comando **`next`** (ou **`n`**):

```
(gdb) n
```

Neste caso, a função é executada e a próxima instrução a executar será a da linha seguinte:

```
Valor de n: SOP1
10         return 0;
```

Os dois comandos **`step`** e **`next`** são a base da execução passo a passo de um programa. A sua compreensão é necessária para se conseguir efectuar o mínimo numa sessão de *debug*.

5 Exemplo Elaborado

Nesta secção apresenta-se um exemplo mais elaborado da execução em modo de *debug* de um programa em C. Apesar do programa ser relativamente simples, conseguem ver-se várias das opções disponíveis para *debug*.
Considere o programa "teste_e.c" apresentado de seguida:

```
#include <stdio.h>

void ciclo(int n)
{
        int i =0;
        char c;  char *ptr=&c;
        for(i=1 ;i<n ;i++)
                printf("Valor de i: %i\n",i);
        ptr=NULL;
}
int main(void)
{
        ciclo(10);
        return 0;
}
```

A sua compilação e início de sessão de debug são:

```
$ gcc -g teste_e.c -o teste_e
$ gdb teste_e
```

A partir deste momento encontramo-nos no início da sessão de debug. Para começar a execução controlada, efectua-se um ponto de quebra na função main, seguida de um início de execução:

```
(gdb) break main
(gdb) run
```

Neste momento o GDB deverá mostrar a linha 13 a executar, que representa a instrução associada à função "ciclo":

```
Breakpoint 1, main () at teste_e.c:13
13          ciclo(10);
```

Para entrar na função deve usar-se o comando **`step`**, passando a execução para a primeira linha da função "ciclo":

```
(gdb) s
ciclo (n=10) at teste_e.c:5
5           int i =0;
```

Neste ponto podemos ver onde estamos na stack de execução usando o comando **`where`** ou, alternativamente o comando **`info stack`**:

```
(gdb) where
(gdb) info stack
```

Qualquer um destes comandos mostrará que estamos na função ciclo, chamada da função main:

```
#0  ciclo (n=10) at teste_e.c:5
#1  0x080483f3 in main () at teste_e.c:13
```

Neste momento continuamos o debug, executando a instrução seguinte que atribui a variável i a zero. O resultado será.

```
(gdb) s
6           char c;  char *ptr=&c;
```

Se quisermos verificar o valor da variável podemos recorrer ao comando **print**:

```
(gdb) print i
$1 = 0
```

No programa vai entrar-se num ciclo sobre a variável "i". Imaginando que queremos que este execute enquanto "i" seja menor do que 7, pode recorrer-se a um watchpoint:

```
(gdb) watch i<7
Hardware watchpoint 2: i < 7
```

Após esta definição sera necessário recomeçar o programa usando o comando **`continue`**:

```
(gdb) c
```

O programa continua até que a expressão altere o seu valor, altura em que devolve o comando:

```
Continuing.
Valor de i: 1
Valor de i: 2
Valor de i: 3
Valor de i: 4
Valor de i: 5
Valor de i: 6
Hardware watchpoint 2: i < 7

Old value = 1
New value = 0
0x080483bc in ciclo (n=10) at teste_e.c:7
7           for(i=1 ;i<n ;i++)
```

É ainda possível atribuir valores a variáveis em execução. Neste caso, o ciclo deveria decorrer até "i" atingir o valor 10. Podemos atribuir um novo valor a esta variável usando a instrução **`set`**:

```
(gdb) set var i=20
```

Dado que a próxima instrução vai ser a verificação da variável pelo ciclo, este vai terminar sem ser concluído por interferência do utilizador. Os valores entre 7 e 9 não serão impressos no terminal.
A próxima instrução sai do ciclo:

```
(gdb) s
9           ptr=NULL;
```

Usando a instrução **`continue`** pode ver-se que o watchpoint será retirado automaticamente e o programa finalizado:

```
(gdb) c
Continuing.
Hardware watchpoint 2 deleted because the program has left the block
in which its expression is valid.

Program exited normally.
```