

Assembly do IA-32 em ambiente Linux

TPC6 e Guião laboratorial

Alberto José Proença

Objectivo e notas

A lista de exercícios/tarefas propostos no TPC6 / Guião laboratorial analisa o **suporte a estruturas de controlo e a funções em C**, no IA-32, com recurso a um depurador (*debugger*). Relembra-se que devem usar o mesmo servidor remoto que foi usado em sessões laboratoriais anteriores, para se garantir coerência na análise e discussão dos resultados.

O texto de “**Introdução ao GDB debugger**”, no fim deste guião, contém informação pertinente ao funcionamento desta sessão laboratorial, e é uma sinopse ultra-compacta do manual; a versão integral está disponível no site da GNU, e recomenda-se ainda a consulta dos documentos disponibilizados nas notas de apoio da disciplina (na Web), por se referirem a versões mais compatíveis com as ferramentas instaladas no servidor.

Este guião permite a resolução do trabalho de modo autónomo e a sua **entrega é obrigatória**, na plataforma de *e-learning* da UC.

Ciclo *While*

1. a) **Coloque** a seguinte função em C num ficheiro com o nome `while_loop.c`, e execute a sua compilação para *assembly*, usando o comando `gcc -O2 -S while_loop.c`.

```
1 int while_loop(int x, int y, int n)
2 {
3     while ((n > 0) && (y < 16*n)) {
4         x += n;
5         y *= n;
6         n--;
7     }
8     return x;
9 }
```

- b) (A) Considerando que os argumentos passados para a função `x`, `y`, e `n`, se encontram respetivamente à distância 8, 12 e 16 do endereço especificado em `%ebp`, **preencha a tabela de utilização de registos** (semelhante ao exemplo da série Fibonacci); considere também a utilização de registos para variáveis temporárias (não visíveis no código C).

Variável	Registo	Atribuição inicial
x		
y		
n		

c) **Confirme esta utilização dos registos**, directamente no servidor.

Sugestão de resolução:

- (i) Escrever o código do `main`, inserindo-o no ficheiro que contém a função.
- (ii) Compilar e procurar no *assembly* as instruções que alterem registos pela 1ª vez.
- (iii) Inserir pontos de paragem logo a seguir a essas instruções em *assembly*.
- (iv) Executar o código de modo a parar nesses locais, e confirmar os valores nos registos.

Detalhes destas 4 tarefas:

- i. (A) **Complete o ficheiro** `while_loop.c` com um programa simples (`main`) que chame a função `while_loop`, passando como argumentos para a função os valores 4, 2 e 3, respetivamente.
- ii. (A) **Crie um executável** para ser depurado, com o comando `gcc -Wall -O2 -g`, **desmonte** o executável com o comando `objdump -d`, e **encontre a localização** em `while_loop`, no código *assembly*, da **1ª instrução logo a seguir** à:
 - (i) leitura de cada um dos argumentos da *stack* ;
 Nota: se o código gerado pelo compilador efectuar esta leitura em 3 instruções consecutivas, basta então identificar apenas a instrução que se segue à última leitura; e
 - (ii) utilização pela 1ª vez de cada um dos restantes registos; **escreva aqui** essas instruções em *assembly* e sua **localização em memória** (endereços de memória):
- iii. (A) Invocando o *debugger* (com `gdb <nome_fich_executável>`), **insira pontos de paragem** (*breakpoints*) nesses endereços, antes da execução das instruções; **explícite** os comandos usados e registre o nº de *breakpoint* atribuído a cada endereço:
- iv. (A) Estime os valores atribuídos aos registos, **preenchendo esta tabela sem executar qualquer código** (apenas com base na análise do código *assembly*).
 Depois, **confirme esses valores** executando o programa dentro do *debugger*: após cada paragem num *breakpoint*, **visualize** o conteúdo dos registos com `print $reg` ou com `info registers` e preencha a tabela em baixo com os valores lidos:

Variável	Registo	Break1	Break_	Break_	Break_	Break_
x						
y						
n						

d) (R/B) Com base nos argumentos passados para a função `while_loop` (no `main`), é possível estimar quantas vezes o *loop* é executado na função. Para confirmar esse valor, uma técnica é introduzir um *breakpoint* na instrução de salto condicional de regresso ao início do *loop*. Indique o que fazer depois para **confirmar o nº de execuções** do *loop*.

Anexo: Introdução ao GNU *debugger*

O GNU *debugger* GDB disponibiliza um conjunto de funcionalidades úteis na análise e avaliação do funcionamento de programas em linguagem máquina, durante a sua execução; permite ainda a execução controlada de um programa, com indicação explícita de quando interromper essa execução – através de *breakpoints*, ou em execução passo-a-passo – e possibilitando a análise do conteúdo de registos e de posições de memória, após cada interrupção.

Use o GDB para confirmar as tabelas de utilização de registos e o valor dos argumentos nos exercícios. Nota: utilize primeiro `objdump` para obter uma versão “desmontada” do programa.

A tabela/figura seguinte (de CSAPP) ilustra a utilização de alguns dos comandos mais comuns para o IA-32.

Command	Effect
Starting and Stopping	
<code>quit</code>	Exit GDB
<code>run</code>	Run your program (give command line argum. here)
<code>kill</code>	Stop your program
Breakpoints	
<code>break sum</code>	Set breakpoint at entry to function <code>sum</code>
<code>break *0x80483c3</code>	Set breakpoint at address <code>0x80483c3</code>
<code>disable 3</code>	Disable breakpoint 3
<code>enable 2</code>	Enable breakpoint 2
<code>clear sum</code>	Clear any breakpoint at entry to function <code>sum</code>
<code>delete 1</code>	Delete breakpoint 1
<code>delete</code>	Delete all breakpoints
Execution	
<code>stepi</code>	Execute one instruction
<code>stepi 4</code>	Execute four instructions
<code>nexti</code>	Like <code>stepi</code> , but proceed through function calls
<code>continue</code>	Resume execution
<code>finish</code>	Run until current function returns
Examining code	
<code>disas</code>	Disassemble current function
<code>disas sum</code>	Disassemble function <code>sum</code>
<code>disas 0x80483b7</code>	Disassemble function around address <code>0x80483b7</code>
<code>disas 0x80483b7 0x80483c7</code>	Disassemble code within specified address range
<code>print /x \$eip</code>	Print program counter in hex
Examining data	
<code>print \$eax</code>	Print contents of <code>%eax</code> in decimal
<code>print /x \$eax</code>	Print contents of <code>%eax</code> in hex
<code>print /t \$eax</code>	Print contents of <code>%eax</code> in binary
<code>print 0x100</code>	Print decimal representation of <code>0x100</code>
<code>print /x 555</code>	Print hex representation of <code>555</code>
<code>print /x (\$ebp+8)</code>	Print contents of <code>%ebp</code> plus 8 in hex
<code>print *(int *) 0xbffff890</code>	Print integer at address <code>0xbffff890</code>
<code>print *(int *) (\$ebp+8)</code>	Print integer at address <code>%ebp + 8</code>
<code>x/2w 0xbffff890</code>	Examine 2(4-byte) words starting at addr <code>0xbffff890</code>
<code>x/20b sum</code>	Examine first 20 bytes of function <code>sum</code>
Useful information	
<code>info frame</code>	Information about current stack frame
<code>info registers</code>	Values of all the registers
<code>help</code>	Get information about GDB

Figure 3.27: **Example GDB Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

Nº	Nome:	Turma:
----	-------	--------

Resolução dos exercícios (deve ser redigido manualmente)**1. Código em *assembly***

Escreva aqui o código otimizado em *assembly* (tal como está no ficheiro `while_loop.s`) devidamente anotado, i.e., com comentários à frente de cada instrução, comentários esses que
(i) expliquem o que está a acontecer se for a fase de arranque ou término duma função e
(ii) mostrem que parte do código C essa instrução em *assembly* está a executar.
De seguida, analise o código em *assembly* e preencha a tabela.

Variável	Registo	Atribuição inicial
x		
y		
n		

Escreva aqui o código C de um programa simples (`main`) que usa a função `while_loop`:

Apresente aqui o código executável depois de desmontado com o comando `objdump -d`. Assinale neste pedaço de código as instruções que vão buscar à *stack* cada um dos 3 argumentos que foram passados para a função, para os colocar em registos. Registe (marque) os endereços das instruções imediatamente a seguir a essas, para que sejam os pontos de paragem a introduzir na execução do código. Uma vez parada a execução do código nesses endereços, vamos poder ver o conteúdo dos registos que receberam os argumentos, confirmando parte da tabela da página anterior.

Escreva aqui os endereços das instruções onde vai inserir pontos de paragem (*breakpoints*) quando usar o depurador `gdb`.

Antes de executar qualquer código, coloque aqui a sua estimativa do que irá estar nos registos após cada ponto de paragem:

Variável	Registo	Break1	Break_	Break_	Break_	Break_
x						
y						
n						

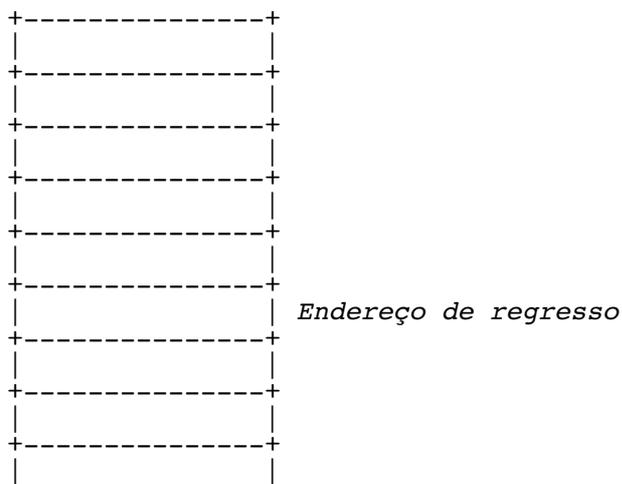
Após execução do código de modo controlado (dentro do depurador), preencha de novo essa tabela com os valores que efetivamente leu quanto ao conteúdo dos registos, após cada ponto de paragem.

Variável	Registo	Break1	Break_	Break_	Break_	Break_
x						
y						
n						

Preencha aqui os valores pedidos no enunciado relativamente à *stack frame* desta função, nomeadamente alguns endereços relevantes, o conteúdo de cada conjunto de 4 células de memória e uma interpretação de cada um desses conteúdos.

Os valores a colocar aqui deverão ser os valores lidos da memória do servidor no 2º ponto de paragem.

Endereço 1ª célula Conteúdo em hex Conteúdo comentado



Nota: neste diagrama, cada caixa representa um bloco de 32-bits em 4 células.