

Assembly do IA-32 em ambiente Linux

TPC7 e Guião laboratorial

Alberto José Proença

Objectivo e notas

A lista de exercícios/tarefas propostos no TPC7 / Guião laboratorial analisa o **suporte a estruturas de controlo e a funções em C**, no IA-32, com recurso a um depurador (*debugger*). Os exercícios para serem resolvidos e entregues antes da aula TP estão assinalados com uma caixa cinza, e repetem-se na última folha. Recomenda-se o uso do mesmo servidor que foi usado na sessão laboratorial anterior, 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.

Ciclo *While*

- Coloque a seguinte função em C num ficheiro com o nome `while_loop.c`, e execute apenas 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 < n)) { /* Repare no uso do operador '&' */
4         x += n;
5         y *= n;
6         n--;
7     }
8     return x;
9 }
```

- ^(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).

Registo	Variável	Atribuição inicial
	x	
	y	
	n	

b) **Confirme esta utilização dos registos**, directamente no computador.

Sugestão de resolução: (i) escrever o código do `main`, (ii) inseri-lo no ficheiro que contém a função, (iii) procurar no código *assembly* as instruções que alterem registos pela 1ª vez, (iv) inserir pontos de paragem logo a seguir a elas, e (v) executar o código de modo a parar nesses locais e assim confirmar os valores nos registos. Detalhes destas 5 tarefas:

- i. (A) **Construa em C** um programa simples (`main`) que use a função `while_loop`, e que não faça mais do que inicializar numericamente um conjunto de valores que irá depois passar como argumento para a função (experimente 4, 2 e 3, respetivamente).
(Sugestão: use variáveis com designações diferentes das usadas na função)
- ii. (A) **Complete o ficheiro** `while_loop.c` com o programa `main` que elaborou e **crie um executável** pronto para ser depurado, usando o comando `gcc -Wall -O2 -g .`
- iii. (A) Desmonte o executável com o comando `objdump -d`, analise o código *assembly* e **identifique em `while_loop`, a 1ª instrução** (e respectiva **localização**) **logo a seguir** a: (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 registos de 8 bits (para quê?); **escreva aqui** essas instruções em *assembly* e sua **localização em memória** (lista de endereços de memória):
- iv. (A) Invocando o *debugger* (com `gdb <nome_fich_executável>`), **insira pontos de paragem** (*breakpoints*) nesses endereços, antes da execução das instruções; explicita aqui os comandos usados (e registe o nº de *breakpoint* atribuído a cada endereço):
- v. (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* e, após cada paragem num *breakpoint*, visualizando o conteúdo dos registos com `print $reg`, ou com `info registers` (nota1: o `gdb` apenas aceita especificação de registos de 32 bits; nota2: no IA-32 os registos de 8 bits são parte dos registos de 32 bits).

Registo	Variável	Break1	Break_	Break_	Break_	Break_
	x					
	y					
	n					

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 *) 0xbfff890</code>	Print integer at address <code>0xbfff890</code>
<code>print *(int *) (\$ebp+8)</code>	Print integer at address <code>%ebp + 8</code>
<code>x/2w 0xbfff890</code>	Examine 2(4-byte) words starting at addr <code>0xbfff890</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

1. ^(A)Análise do código em *assembly*

```

1 int while loop(int x, int y, int n)
2 {
3     while ((n > 0) & (y < n)) { /* Repare no uso do operador '&' */
4         x += n;
5         y *= n;
6         n--;
7     }
8     return x;
9 }

```

Código otimizado em *assembly*:

Registo	Variável	Atribuição inicial
	x	
	y	
	n	

Código C de um programa simples (main) que usa a função `while_loop`: