

Assembly do IA-32 em ambiente Linux

TPC6 e Guião laboratorial

Baseado no guia de Alberto José Proença

Objectivo e notas

Os exercícios propostos no TPC6 introduzem o **suporte a funções em C**, no IA-32, e a utilização de um depurador (*debugger*). Estes exercícios devem ser realizados no servidor remoto que foi usado na sessão laboratorial anterior (ver TPC4).

O texto de “**Introdução ao GDB debugger**”, no fim deste guião, contém informação necessária para esta sessão, e é uma sinopse ultra-compacta do manual; uma versão integral está disponível no site da GNU.

A resolução deverá ser entregue **impreterivelmente** no início da sessão PL, com a presença do estudante durante a sessão PL para que o TPC seja contabilizado na avaliação por participação. Não serão aceites trabalhos entregues fora da PL.

Passagem de parâmetros a funções

1. a) **(TPC)** Crie um ficheiro com o nome `while_loop.c` com seguinte função e execute a sua compilação para *assembly*, usando o comando `gcc -O2 -S while_loop.c`.

```
int while_loop(int x, int y, int n)
{
    do {
        x += n;
        y *= n;
        n--;
    } while (n > 0);
    return (x+y);
}
```

- b) **(TPC)** Os argumentos `x`, `y` e `n` da função encontram-se, respetivamente, à distância 8, 12 e 16 do endereço de memória especificado em `%ebp`. Analise o *Assembly* da função e **indique na tabela os registos atribuídos a cada variável**, com base na identificação das instruções que efetuam cópias dos valores da pilha (usando `%ebp` como endereço base) para os correspondentes registos (inclua também essas as instruções na tabela).

Variável	Registo atribuído pelo compilador	Instrução <i>Assembly</i>
x		
y		
n		

- c) **(TPC)** Confirme essa atribuição dos registos identificando as três instruções no corpo do ciclo `do/while` que atualizam os valores de `x`, `y` e `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	
<i>quit</i>	Exit GDB
<i>run</i>	Run your program (give command line argum. here)
<i>kill</i>	Stop your program
Breakpoints	
<i>break sum</i>	Set breakpoint at entry to function <code>sum</code>
<i>break *0x80483c3</i>	Set breakpoint at address <code>0x80483c3</code>
<i>disable 3</i>	Disable breakpoint 3
<i>enable 2</i>	Enable breakpoint 2
<i>clear sum</i>	Clear any breakpoint at entry to function <code>sum</code>
<i>delete 1</i>	Delete breakpoint 1
<i>delete</i>	Delete all breakpoints
Execution	
<i>stepi</i>	Execute one instruction
<i>stepi 4</i>	Execute four instructions
<i>nexti</i>	Like <i>stepi</i> , but proceed through function calls
<i>continue</i>	Resume execution
<i>finish</i>	Run until current function returns
Examining code	
<i>disas</i>	Disassemble current function
<i>disas sum</i>	Disassemble function <code>sum</code>
<i>disas 0x80483b7</i>	Disassemble function around address <code>0x80483b7</code>
<i>disas 0x80483b7 0x80483c7</i>	Disassemble code within specified address range
<i>print /x \$eip</i>	Print program counter in hex
Examining data	
<i>print \$eax</i>	Print contents of <code>%eax</code> in decimal
<i>print /x \$eax</i>	Print contents of <code>%eax</code> in hex
<i>print /t \$eax</i>	Print contents of <code>%eax</code> in binary
<i>print 0x100</i>	Print decimal representation of <code>0x100</code>
<i>print /x 555</i>	Print hex representation of <code>555</code>
<i>print /x (\$ebp+8)</i>	Print contents of <code>%ebp</code> plus 8 in hex
<i>print *(int *) 0xbffff890</i>	Print integer at address <code>0xbffff890</code>
<i>print *(int *) (\$ebp+8)</i>	Print integer at address <code>%ebp + 8</code>
<i>x/2w 0xbffff890</i>	Examine 2(4-byte) words starting at addr <code>0xbffff890</code>
<i>x/20b sum</i>	Examine first 20 bytes of function <code>sum</code>
Useful information	
<i>info frame</i>	Information about current stack frame
<i>info registers</i>	Values of all the registers
<i>help</i>	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***

Transcreva aqui o código *assembly* de `while_loop.s`. e classifique cada uma das instruções de acordo com seu propósito:

- (i) copiar valores da pilha para registos
- (ii) atualizar os valores de x, y e n no corpo do ciclo `do/while`
- (iii) implementação do ciclo `do/while`
- (iv) chamada/regresso da função e gestão da pilha.
- (v) calcular o valor de retorno da função

Preencha a tabela com os registos atribuídos a cada variável

Variável	Registo
x	
y	
n	

2. Simulação da execução

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

Coloque aqui a estimativa do que irá estar nos registos após cada ponto de paragem:

Variável	Registo	Break1	Break_	Break_	Break_	Break_
x						
y						
n						