

Assembly do IA-32 em ambiente Linux

TPC6 e Guião laboratorial

Alberto José Proença

Objectivo

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*). Os exercícios para serem resolvidos e entregues antes da aula TP estão assinalados com uma caixa cinza, e repetem-se na última folha.

O texto anexo “**Introdução ao GDB debugger**”, contém informação pertinente ao funcionamento da sessão laboratorial, e é uma sinopse ultra-compacta do manual; a versão integral está disponível no site da GNU, e o acesso electrónico a este documento (em servidor local) e a outros complementares, está referido nas notas de apoio da disciplina (na Web).

Ciclo *While*

1. 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) ^(A) Considerando que os argumentos x , y , e n , passados para a função, se encontram respectivamente 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 | |
| | n | |
| | y | |
| | | |
| | | |
| | | |

b) ^(R) Faça nova compilação, usando um comando semelhante mas sem qualquer optimização e criando um ficheiro em assembly com uma designação diferente. **Analise os 2 códigos em assembly e identifique e caracterize** as principais diferenças que encontrar.

c) **Confirme esta utilização dos registos**, directamente no computador:

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, respectivamente). *(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 usando o comando `gcc -Wall -O2 -g .`

iii. ^(A) 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; escreva aqui essas instruções em *assembly* e respectiva localização:

iv. ^(A) Invocando o *debugger* (com `gdb <nome_fich_executável>`), **insira pontos de paragem** (*breakpoints*) imediatamente antes da execução dessas instruções; explicita aqui os comandos usados (e registre o nº de *breakpoint* atribuído a cada endereço):

v. ^(A) Valide o conjunto de atribuições aos registos, **preenchendo esta tabela sem executar qualquer código** (apenas com base na análise do código). 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`; de notar que o `gdb` apenas aceita especificação de registos de 32 bits).

| Registo | Variável | Break1 | Break2 | Break3 | Break4 | Break5 |
|---------|----------|--------|--------|--------|--------|--------|
| | x | | | | | |
| | n | | | | | |
| | y | | | | | |
| | | | | | | |
| | | | | | | |

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 IA32.

| 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: |
|-----------|--------------|

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 | |
| | n | |
| | y | |
| | | |
| | | |
| | | |

Principais diferenças entre as versões do *assembly* (otimizada e não-otimizada):

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