Universidade do Minho

Módulo 3

Depuração de Programas

1. Introdução

Pretende-se com esta sessão teórico-prática que os alunos acompanhem a execução de um programa instrução a instrução, visualizando as alterações ao estado da máquina (memória, registos). Para este efeito a depuração do programa, ao nível maquina, será feita usando o GDB. 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.

NOTA: A QUESTÃO ASSINALADA COM TPC DEVE SER RESOLVIDA **ANTES DA SESSÃO TP** E ENTREGUE AO DOCENTE NO INÍCIO DA MESMA. A FOLHA ANEXA A ESTE MÓDULO, ASSINALADA COMO TPC, DESTINA-SE A ESSE FIM.

---

2. Linguagem de alto nível

Escreva em C, usando o editor de texto que considerar mais adequado, o seguinte programa:

prog.c

```
int vec[5] = {-10, 20, 10, 5, -15};
int j,accum;
int main ()  {
      accum = 0;
      for (j = 0; j<5 ; j++)
        accum += vec[i];
}
```

Compile o programa `prog.c` usando o comando

`gcc –g –S prog.c`

e visualize o código *assembly* gerado. Gere o executável usando o comando

`gcc –g –o prog prog.c`

3. Depuração

Para depurar o programa vamos usar o GDB (consulte o anexo para lista de comandos):

`gdb prog`

Visualize o código *assembly* da função main() usando o comando

```
gdb: disas main
```

TPC : Questão 1 – Compare o código apresentado pelo GDB com o código *assembly* contido no ficheiro `prog.s`. Quais os endereços de memória correspondentes às variáveis `vec`, `accum` e `j`?

Questão 2 – Examine o conteúdo dessas variáveis usando o comando `examine` (e.g. `x/5dw vec`).

Questão 3 – Coloque um *breakpoint* na primeira instrução de teste da condição do ciclo.

Questão 4 – Indique ao GDB que deve apresentar no êcran os valores de `accum`, `j` e `vec[j]`. Use o comando `display`.

Questão 5 – Execute o programa durante 2 iterações, isto é, até a variável `j` ter o valor 2. Para esse efeito usando o comando `run` para iniciar a execução e o comando `continue` sempre que esta parar num *breakpoint*. Observe cuidadosamente os valores das variáveis em cada iteração.

Questão 6 – Coloque um *breakpoint* na instrução que calcula o endereço de `vec[j]`. Continue a execução até parar nesta instrução. Usando o comando `info registers` verifique o conteúdo dos registos e calcule manualmente o endereço de `vec[j]`, identificando o papel de cada campo no cálculo do endereço, isto é, a base, índice, factor de escala e deslocamento.

Questão 7 – Execute o programa instrução a instrução (`stepi`) até a operação `accum+=vec[j]` estar concluída. Descreva extensivamente o papel de cada instrução, recorrendo ao comandos `info register` e `examine` (`x`) para verificar o conteúdo dos registos e da memória.

Questão 8 – O GDB permite, além de verificar o estado do processador para cada instrução, alterar esse estado. Use o comando `set` para alterar o valor do contador `j` de forma a que esta seja a última iteração do ciclo. Corra o programa até ao fim verificando a correcção do resultado (`accum`).

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.

A tabela/figura seguinte (de CSAPP) ilustra a utilização de alguns dos comandos mais comuns para o IA32.

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

Figure 3.27 (de CSAPP e Prof. Alberto Proença): **Example GDB Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

TPC	
Número:	Nome:

TPC : Questão 1 – Comparando o código apresentado pelo GDB com o código *assembly* contido no ficheiro `prog.s`, indique quais os endereços de memória correspondentes às variáveis `vec`, `accum` e `j`?

Questão resolvida na aula	
Número:	Nome: