Insper

04 - Representação de dados em RAM

Sistemas Hardware-Software - 2019/2

Igor Montagner

No handout passado usamos o *gdb* para listar variáveis globais, nomes de funções e examinar endereços de memória. Neste handout vamos começar a usar o *gdb* também para examinar nossos programas *durante sua execução*.

Parte 1 - parando e continuando a execução de um programa.

Exercício 0: Compile *funcao1.c* usando as flags da disciplina (`-Og -Wall -std=c99`)

Exercício 1: Abra o arquivo com o *gdb* e liste suas funções e suas variáveis globais. **Não rode o programa ainda!** Anote abaixo seus endereços e nomes.

Exercício 2: O comando `disas` é usado para mostrar as instruções de máquina de uma função. Use-o para ver o código de máquina da função `funcao1`. Liste abaixo quais registradores foram usados nesta função e qual o tamanho dos dados guardados neles.

Exercício 3: Com base nos tamanhos identificados, você consegue associar os registradores acima com as variáveis no código de `funcao1`?

Exercício 4: Use o comando `breakpoint funcao1` para parar a execução do programa quando a função `funcao1` começar a executar. Execute o programa usando `run`. O quê ocorre?

Exercício 5: Podemos usar o comando `info registers edi` para ver o conteúdo atual do registrador `%edi`. Qual é este valor? Verifique quais argumentos são passados para `funcao1` no código *C* e, com estas informações em mãos, verifique sua resposta do exercício 3 e escreva abaixo suas novas conclusões.

Exercício 6: O comando `stepi` executa exatamente uma instrução de máquina. Use-o uma vez e execute novamente `disas funcao1`. O que aconteceu? É possível saber em qual instrução o programa está parado?

Exercício 7: Cheque novamente o valor de `%edi`. Este valor condiz com a instrução executada? O que ela faz, exatamente?

Exercício 8: Use `stepi` para parar logo antes do retorno da função. Verifique o conteúdo do registrador `%eax` e compare-o com os prints feitos pelo program. Você consegue dizer seu uso?

Exercício 9: Vamos agora analisar o registrador `%rsi`. Toda vez que um registrador aparece entre `( )` estamos fazendo um acesso a memória. Ao mostrar seu conteúdo usando `info registers rsi` recebemos o endereço de memória que contém o dado que queremos acessar. Use o comando `x` para mostrar, em decimal, o `int` que está armazenado neste endereço.

Exercício 10: Execute o comando `continue` para continuar rodando o programa. Ele irá rodar até que o próximo *breakpoint* seja alcançado ou até que o programa termine.

Parte 2 - endereçamento relativo e variáveis globais

Na parte anterior analisamos o código Assembly de nossa primeira função e vimos como

- mostrar o código fonte de uma função usando `disas`
- mostrar o conteúdo de um registrador usando `info registers`
- executar exatamente uma instrução usando `stepi`

Também vimos que ao colocar um registrador entre `( )` estamos fazendo um acesso a memória. Esta operação é equivalente a desreferenciar um ponteiro usando `*p`. Neste roteiro iremos adicionar um detalhe importante: podemos fazer contas com endereços usando esta notação. Nos exemplo abaixo nos referimos a memória como um grande vetor de bytes `unsigned char M[]`. Ou seja, ao acessar `M[%rax]`, por exemplo, estamos acessando o lugar na memória cujo endereço está escrito em `%rax`.

1. `10(%rax)`: acessa a memória `M[%rax + 10]`.
2. `(%rax, %rdi, 4)`: acessa a memória `M[%rax + 4 * % rdi]`. Note que isto se parece com aritmética de ponteiros cujo tipo apontado seja inteiro, pois os endereços pulam de 4 em 4 bytes.

Exercício 0: Saia e abra o *gdb* novamente. Mostre o código de máquina da função `funcao2` e coloque um breakpoint em sua primeira instrução.

Exercício 1: Execute agora o programa. A execução deve ter parado no início de `funcao2`. Rode `disas funcao2`.

Exercício 2: Você consegue identificar acessos a memória em `funcao2`? Quais são de leitura e quais são de escrita? Qual o tamanho dos dados lidos/escritos?

Exercício 3: Qual o significado do registrador `%rip`?

O tipo de acesso a memória que estamos realizando se chama `rip relative addressing`. Este tipo de acesso é reservado para variáveis globais e dados somente leitura. Estes dados tem uma característica especial: eles são copiados para a memória seguindo o mesmo layout do arquivo executável. Ou seja, as posições *relativas* entre o código e os dados globais são fixas.

Exercício 4: Anote abaixo o endeço das funções `MOV` que utilizam este acesso. Baseado nos exemplos acima, descubra o endereço das variáveis acessadas.

Exercício 5: Confira se o valor identificado na questão anterior é o mesmo mostrado a direita das instruções `MOV` na saída do `disas`. O *gdb* já calcula este endereço para facilitar nossa vida, mas é interessante calcular isto manualmente uma vez para entender melhor o processo.

Exercício 6: Use o comando `continue` para continuar o programa. Você deve estar agora na segunda execução de `funcao2`. Use o comando `x` para mostrar o valor armazenado na memória calculada acima. Lendo o código do programa, você consegue dizer qual variável é armazenada neste endereço? O valor atual é o esperado para a segunda execução de `funcao2`?.

Além de poder mostrar valores na memória podemos **escrever** valores também. A sintaxe usada é a seguinte:

```
set *( (tipo *) 0x.....) = valor
```

onde devemos substituir `tipo` por um tipo básico de *C*, `0x...` pelo endereço desejado e `valor` pelo valor que queremos escrever. Note que o que estamos fazendo é um *cast* do endereço `0x....` para um ponteiro de `tipo` e depois estamos acessando o valor apontado usando `*`!

Exercício 7: Escreva o valor `-10` na memória da variável global usada em `funcao2`. Rode o programa até o fim. O resultado foi o esperado? Escreva abaixo os comandos utilizados.

Desafio: localize na função `main` as chamadas ao comando `printf`. Encontre então o endereço das strings de formatação e use o comando `x` para mostrá-las no gdb. Escreva os comandos usados abaixo.