UFSC / CTC / INE

Disciplina: Topicos Esp. em Aplic. Tecnológicas III

Curso de Ciências da Computação: INE5450

Prof. Dr. João Dovicchi*

1 Aula Prática 1 - Memória e endereços

Nesta aula vamos compreender alguns conceitos do ambiente a ser utilizado nas aulas de C para drivers de dispositivos. Em primeiro lugar, vamos compreender alguns aspectos da memória e endereçamento na arquitetura ARM e como a linguagem C lida com isto.

Roteiro 1:

Tente encontrar os tamanhos dos tipos no ARM e no x86 para podermos comparar os resultados:

A documentação do ARM, em `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/ch03s02s02.html` referencia os tipos básicos de dados na linguagem C/C++ para o ARM.

Isto pode ser comprovado pelo programa da listagem 1.

O mesmo problema pode ser endereçado de outra forma e pode ser comparado com a implementação C/C++ em outras arquiteturas (x86, por exemplo). O pode-se usar a potência de 10 ou de 2 para implementar o teste: (veja listagens 2 e 3, respectivamente).

Listagem 1: Tipos em C

```
/* sizes.c
 * Para verificar o tamanho dos tipos.
 *
 * Nota: no ARM o tipo 'long double' é do mesmo tamanho
 * que o 'double'.
 */

#include <stdio.h>
#include <math.h>
#include <limits.h>

int main(){
  printf("char:\t\t%u\n",sizeof(char));
  printf("short int:\t%u\n",sizeof(short int));
  printf("int:\t\t%u\n",sizeof(int));
  printf("long int:\t%u\n",sizeof(long int));
  printf("longlong int:\t%u\n",sizeof(long long int));
  printf("float:\t\t%u\n",sizeof(float));
  printf("double:\t\t%u\n",sizeof(double));
  printf("long double:\t%u\n",sizeof(long double));
  return 0;
}
```

Listagem 2: Limite de inteiros (unsigned long long int)

```
/* lim_int.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <limits.h>
#include <math.h>

int main (){
  unsigned long long int f;
  int i;
  f = 1;
  for(i=0;i<30;i++){
    f *= 10;
    printf("%2i %20Lu\t\t", i, f);
    i++;
    f *= 10;
    printf("%2i %20Lu\n", i, f)
  }
  exit(0);
}
```

Listagem 3: Limite de inteiros (versao com potencia de dois)

```
/* lim_int_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <limits.h>
#include <math.h>

int main (){
  unsigned long long int f;
  int i;
  double p;
  f = 0;
  for(i=0;i<65;i++){
    p = pow(2.0, (double) i);
    f = (unsigned long long int) p;
    printf("%2i %20Lu\t\t", i, f);
    i++;
    p = pow(2.0, (double) i);
    f = (unsigned long long int) p;
    printf("%2i %20Lu\n", i, f);
  }
  exit(0);
}
```

Ainda, dentro desta linha de conhecer os tamanhos de inteiros existem outros programas no arquivo de prática 1 que podem ser usados para se comparar os resultados em diversas arquiteturas.

São eles: `arm-long.c, fatorial.c, fatorial_ll.c e fatorial_ull.c`.

Roteiro 2:

Nesta parte do roteiro vamos investigar os números de ponto flutuante e as limitações do ARM. Em primeiro lugar vamos lembrar que o processador ARM não faz operações com ponto flutuante (`float` ou `double`) em uma FPU (*Floating Point Unity*) como os processadores x86. Estas operações são feitas por implementação via software (como veremos mais tarde no conjunto de instruções Assembly do ARM).

Os números de ponto flutuante são armazenados no formato IEEE 754, sendo que os números de tipo `float` são armazenados no formato de precisão

*http://www.inf.ufsc.br/~dovicchi --- dovicchi@inf.ufsc.br

Listagem 4: Teste de overflow com números de precisão simples

```
/* float_over.c */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(){
  float f;
  int i;

  f=1.0;
  for(i=0; i<40; i++){
    f *= 10.0;
    printf("%2i %e\t\t", i, f);
    i++;
    f *= 10.0;
    printf("%2i %e\n", i, f);
  }
  exit(0);
}
```

simples (32 bits) e os números do tipo `double` e `long double` são armazenados com precisão dupla (64 bits).

Veja *Operations on floating-point types* em `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/ch03s02s02.html`.

Neste experimento vamos testar a capacidade de operações com números de ponto flutuante do ARM e comparar com o x86. Para testar o *overflow* vamos usar o código da listagem 4 e para testar o *underflow* usaremos o código da listagem 5

Nota: Faça o teste com o formato `double` também (aumente o número de iterações, claro!). Experimente, também o programa `ser1.c` que aproxima o número 1 por uma série de Taylor.

Roteiro 3:

Nesta parte, vamos estudar o endereçamento da memória na arquitetura ARM e compará-lo com a arquitetura x86. Para isso, o programa da listagem 6 declara algumas variáveis e identifica o endereço delas na memória.

Listagem 5: Teste de underflow com números de precisão simples

```
/* float_under.c */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(){
  float f;
  int i;

  f=1.0;
  for(i=0; i<50; i++){
    f /= 10.0;
    printf("%2i %e\t\t", i, f);
    i++;
    f /= 10.0;
    printf("%2i %e\n", i, f);
  }
  exit(0);
}
```

Listagem 6: Programa para evidenciar os endereços de variáveis

```
/* addr.c
 *
 * A alocacao de variaveis pode ser contigua ou separada em blocos. Este
 * programa imprime os enderecos das variaveis na memoria.
 *
 */

#include <stdio.h>

int main(){
  char c='a',d='b';
  int i=0,j=0;
  float k=0.0,l=0.0;
  double m=0.0,n=0.0;

  printf("variavel\ttipo  \tendereco\n\n");
  printf("   c    \tchar  \t%x\n",(unsigned int) &c);
  printf("   d    \tchar  \t%x\n",(unsigned int) &d);
  printf("   i    \tint   \t%x\n",(unsigned int) &i);
  printf("   j    \tint   \t%x\n",(unsigned int) &j);
  printf("   k    \tfloat \t%x\n",(unsigned int) &k);
  printf("   l    \tfloat \t%x\n",(unsigned int) &l);
  printf("   m    \tdouble\t%x\n",(unsigned int) &m);
  printf("   n    \tdouble\t%x\n",(unsigned int) &n);
  return 0;
}
```

O resultdado é algo como o mapa abaixo. Note que os primeiros 6 bytes de endereço variam a cada carga do programa.

```
0xbeb0f298 +-----+
           |  m  | } 8 bytes (double)
0xbeb0f2a0 +-----+
           |  n  | } 8 bytes (double)
0xbeb5f2a8 +-----+
           |void | } 4 bytes
0xbeb5f2ac +-----+
           |  i  | } 4 bytes (int)
0xbeb5f2b0 +-----+
           |  j  | } 4 bytes (int)
0xbeb5f2b4 +-----+
           |  k  | } 4 bytes (float)
0xbeb5f2b8 +-----+
           |  l  | } 4 bytes (float)
0xbeb5f2bc +-----+
           |void | } 2 bytes
0xbeb5f2be +-----+
           |  c  | } 1 byte (char)
0xbeb5f2bf +-----+
           |  d  | } 1 byte (char)
           +-----+
```

A reserva de memória para o armazenamento das varáveis é separado em blocos dos tamanhos primeiro os blocos de 8 bytes (doubles), depois os blocos de 4 bytes (int e floats) e finalmente os blocos de 1 byte (1 char).

Rode o mesmo programa em um Linux numa arquitetura x86. deve resultar em algo semelhante a:

```
0xbeb0f21e +-----+
           |  c  | } 1 byte  (char)
0xbeb0f21f +-----+
           |  d  | } 1 byte  (char)
0xbeb5f220 +-----+
           |  i  | } 4 bytes (int)
0xbeb5f224 +-----+
           |  j  | } 4 bytes (int)
0xbeb5f228 +-----+
           |  k  | } 4 bytes (float)
0xbeb5f22c +-----+
           |  l  | } 4 bytes (float)
0xbeb5f230 +-----+
           |  m  | } 8 bytes (double)
0xbeb5f2bc +-----+
           |  n  | } 8 bytes (double)
0xbeb5f2be +-----+
```