LAB 2: Interação com Memória

OBJETIVO: COM ALOCAR E GERIR MEMÓRIA

Nos programas UNIX/C, compreender como alocar e gerir memória é crítica para a a construção de um software robusto e confiável. Que interfaces se utilizam habitualmente? Que erros devem ser evitados?

CRUX: HOW TO ALLOCATE AND MANAGE MEMORY

In UNIX/C programs, understanding how to allocate and manage memory is critical in building robust and reliable software. What interfaces are commonly used? What mistakes should be avoided?

1. Tipos de memória

Num programa C existem dois tipos de memória que pode ser alocada. A primeira é a memória em pilha (*stack memory*) e o processo de alocação e libertação é gerido de forma implícita pelo compilador. Por essa razão é. habitualmente, designada por memória automática.
A declaração de memória na pilha em C é simples. Vamos assumir que precisamos de alguma espaço de memória na função func() para um inteiro x;

```
void func() {
      int x; // declares an integer on the stack
      ...
}
```

Fig.1: *Uma variável na pilha declarada como variável a guardar no stack.*

O compilador guarda espaço na pilha quando é chamada a função. Quando se retorna da função, o compilador liberta a memória.
A memória que se mantém durante mais tempo é habitualmente designada por memória de acervo (*heap memory*). Neste caso, as alocações e a libertação são geridas pelo programador. Esta é uma grande responsabilidade e causadora de muitos erros.

```
void func() {
      int *x= (int *) malloc(sizeof(int));
      ...
}
```

Fig.2: *Uma variável na pilha guardada no heap.*

O compilador cria espaço para um apontador para um inteiro quando vê a declaração *int *x*; em seguida, quando o programa chama *malloc()*, vai requer espaço para um inteiro no espaço de memória no *heap*; a função *malloc()* retorna o endereço do inteiro, endereço esse que é guardado no stack para ser utilizado no programa.

O processo filho criado não é exatamente uma cópia. Embora ele tenha agora o seu espaço de endereços (isto é, o seu espaço de memória privado), os seus registos, o seu PC, o valor que retorna de *fork()* é diferente. Enquanto o pai recebe o PID do processo filho criado, o filho recebe o valor zero. Esta diferença é útil porque, desta forma, é fácil escrever o código que corre em cada um dos casos.

2. Chamada sistema *malloc()*

A chamada malloc() é muito simples: é passado um parâmetro com o tamanho da memoria a guardar no heap e se houver sucesso retorna um apontador para o sítio onde fica guardado o espaço na memória ou falha retornando o valor NULL (experimente o comando man malloc).:

```
#include <stdlib.h>
      ...
      void *malloc(size_t size);
      ...
}
```

Fig.2: A assinatura da função malloc().

A informação que devemos tomar em conta é o facto de termos de incluir o ficheiro stdlib.h. Esta biblioteca é, por vezes, incluída automaticamente. O único parâmetro da funçaõ é do tipo size-t que simplesmente indica com quantos bytes são necessários. A maior parte dos programadores não usam um valor direto mas utilizam uma forma que permite indicar a dimensão pretendida.

```
      ...
      double *d=(double*) malloc(sizeof(double));
      ...
}
```

Fig.3: Alocar variável com a dimensão de *double*.

Podemos também criar espaço multiplicado por um inteiro. No exemplo da figura 4, aloca-se espaço para 10 inteiros. A segunda instrução imprime não o espaço ocupado no *heap* para a variável mas a dimensão do apontador! Para imprimir o espaço ocupado temos de recorrer ao conceito de vetor como mostra a figura 5. O resultado mostra que cada inteiro ocupa 4 bytes.

```
      ...
      int *x = malloc(10 * sizeof(int));
      printf("%d\n", sizeof(x));
      ...
}
```

Fig.4: Alocar variável com a dimensão de *int*. O que é impresso é a dimensão do apontador e não da dimensão da memória reservada no heap.

```
      ...
      int x[10];
      printf("%d\n", sizeof(x));
      ...
}
```

Fig.5: Alocar variável com a dimensão de *int*. O que é impresso é a dimensão do apontador e não da dimensão da memória reservada no heap.

Como nota final, duas chamadas de atenção:

- Para criar espaço para strings, deve-se utilizar a expressão `malloc(strlen(s)+1).` Dessa forma obtêm-se a comprimento da string e adiciona-se 1 de forma a garantir espaço para o carácter no final da *string*.

- A função *malloc()* retorna um apontador do tipo *void*. Desta forma o C permite retornar um endereço e permite ao programador fazer o que quer com ele. Habitualmente utiliza-se a função **cast** garantindo desta forma um tipo associado ao valor retornado por *malloc()*.

3. Chamada sistema free()

Como é de esperar pelo nome, esta função liberta espaço de memória heap. Para libertar esta memória, o programador apenas precisa de executar a função free() como mostra a figura 6.

```
        ...
        int x= malloc(10*sizeof(int))
        ...
        free(x)
}
```

Fig.6: *Libertar espaço de memória do inteiro x.*

4. Erros comuns

Há um número de erros comuns que surgem quando se usa a função *malloc()*.

4.1 Esquecer alocar memória
4.2 Esquecer libertar memória
4.3 Libertar memória antes do tempo
4.4 Libertar memória de forma repetida
4.5 Chamar free() de forma incorreta

5. QUESTÕES

5.1 Escreve um programa simples designado por null.c que cria um apontador para um inteiro, e que atribui o valor NULL. Depois atribui ao local onde o apontador aponta (ou seja para nada) o valor 2.
a) Corra o programa. O que acontece?

b) A seguir compile o programa com a flag -g. Corra o programa utilizando o depurador escrevendo *gdb null* e depois escreva *run* já dentro do programa. O que mostra o *gdb*?

c) Finalmente vamos utilizar o programa valgrind (instale-o executando o comando *apt-get install valgrind*). Vamos utilizar o memchek que é parte do valgrind para analisar o que acontece. Corra-o executando o seguinte comando *valgrind –leak-check=yes ./null*. Consegue interpretar o resultado que aparece no ecrã?

5.2 Escreva um programa simples que aloca memória utilizando o *malloc()* mas que se esquece de a libertar antes de terminar. O que acontece quando este programa corre? Utilize o *gdb* novamente para verificar o que acontece. Testa também utilizando o *valgrind* com a mesma flag *–leak-check* ativa.

5.3 Escreva um programa que cria um vetor de inteiros usando *malloc*. Atribui o valor *data[100]=0*. O que acontece quando o programa é executado? O que acontece quando se usa a ferramenta *valgrind*?
5.4 Cria um programa que aloca um vetor de inteiros (como em cima), liberta-os e depois tenta imprimir o valor de um dos elementos do vetor. Será que o programa corre? Experimenta utilizar a ferramenta *valgrind*.
5.5 Agora utiliza o comando free() para libertar o apontador mas quando este aponta para o meio do vetor. O que acontece? Testa a execução do programa com o valgrind.