

Computer Architecture (Practical Class)

Dynamic Memory Allocation

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

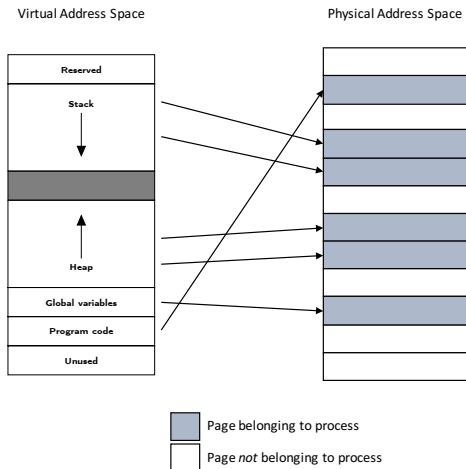
lmn@isep.ipp.pt

2022/2023

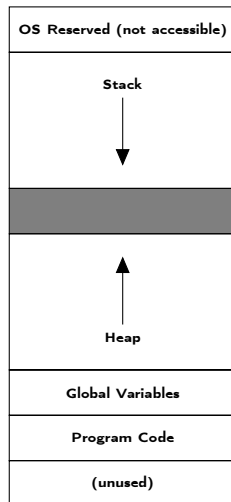
Virtual Memory (1/2)

- The operating system gives the illusion that a program¹ has exclusive access to a contiguous memory address space
- This continuous memory address space is known as the *virtual memory* address space

¹ More precisely, a *process*; you will learn about this distinction in SCOMP



- Like the stack, the heap expands and contracts dynamically at run time as a result of calls to allocate/free memory
- However, the heap grows toward higher addresses
- C programmers typically use a dynamic memory allocator when they need to acquire additional virtual memory at run time



Allows for a more flexible and efficient memory management, allocating and freeing space on the heap, according to the real needs of the program

Consider the following scenarios:

- The programmer does not know the amount of needed memory to handle all the data
 - Must be pessimistic when allocate memory statically
 - Allocate enough for worst possible case → memory is used inefficiently
- Complex data structures: lists and trees
- We often need a way to reserve a section of memory that remains available throughout our entire program (not only until the end of a function), or until we want to destroy it (give it back to the operating system)

The C standard library provides calls to dynamically allocate/free memory on the heap

- `void* malloc(size_t size);`
 - Allocates a continuous memory block of at least `size` bytes and returns a pointer to it
 - The allocated memory is *not initialized*
 - If `malloc()` encounters a problem (e.g., the program requests a block of memory that is larger than the available virtual memory), then it returns `NULL`
- `void free(void *ptr);`
 - Free allocated heap memory blocks previously allocated with a memory allocation call
 - The `ptr` argument must point to the beginning of an allocated block that was obtained from a memory allocation call

malloc() and free() Example

```
#include <stdlib.h> /* malloc() and free() are part of stdlib */

/* declare pointer */
int *ptr_int=NULL;
/* allocate 10 integers in the heap */
ptr_int=(int *) malloc(10 * sizeof(int)); malloc (size_t size)

/* use allocated memory */
ptr_int[0] = 10;
*(ptr_int + 1) = 20;

/* free memory */
free(ptr_int);
```

- `void* calloc(size_t n, size_t size);`

- Reserves a continuous block of memory of at least $n * \text{size}$ bytes
- The memory block is initialized to 0
- Returns a pointer to the allocated memory block, or NULL in error

`calloc()` Example

```
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointer */
int *ptr_int=NULL;
/* allocate 100 integers in the heap */
ptr_int=(int *) calloc(100, sizeof(int));
...
/* free memory */
free(ptr_int);
```

- `void *realloc(void *ptr, size_t size);`
 - Changes the size of a memory block previously allocated with `malloc()` or `calloc()`
 - Returns a pointer to the allocated memory block, or `NULL` in error
 - Data is kept if size is increased, or truncated if size is decreased
 - If size is increased, the added memory is not initialized

`realloc()` Example

```
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointer */
int *ptr_int=NULL;
/* allocate 100 integers in the heap */
ptr_int=(int *) calloc(100, sizeof(int));

/* allocate one more integer in the heap */
ptr_int=(int *) realloc(ptr_int, 101 * sizeof(int));

/* free memory */
free(ptr_int);
```


Important note about the usage of `realloc()`

- If `realloc()` fails, the pointer to the memory block is lost!
 - Use a temporary pointer and check the return of `realloc()`
-
- It may be the case that there is sufficient memory, but not available in one contiguous chunk that can satisfy the allocation request
 - This situation is called **memory fragmentation** and will be discussed in a lecture class

Check Return of realloc() Example

```
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointers */
int *ptr_int=NULL, *ptr_tmp=NULL;
/* number of ints to allocate */
int n=100;

/* allocate n integers in the heap */
ptr_int=(int *) calloc(n, sizeof(int));

/* allocate one more integer in the heap
NOTE: return is stored in temporary pointer */
ptr_tmp=(int *) realloc(ptr_int, (n+1) * sizeof(int));

/* check realloc() return */
if(ptr_tmp!=NULL){
    ptr_int=ptr_tmp;
    ptr_tmp=NULL;
}

/* free memory */
free(ptr_int);
```

- We can dynamically manage memory by creating memory blocks as needed in the heap
- In dynamic memory allocation, memory is allocated at a run time
- Dynamic memory allocation permits to manipulate data structures (e.g. strings and arrays) whose size is flexible and can be changed anytime in your program
- It is required when you have no idea how much memory a particular data structure is going to occupy
- `malloc()` and `calloc()` are used to reserve a continuous block of n bytes in the heap
- `realloc()` is used to reallocate memory according to the new specified size
- `free()` is used to clear the dynamically allocated memory

- Implement, in C, a program that requests the user for integer values and stores them
- The program should request values until the user enters a negative integer.
- The program starts by reserving space for 10 integers, and increases this space as needed.
- The value read from the user should be copied to the array (at index) by the following function, implemented in assembly:
`void write_value(int *array, int index, int value)`

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "handle_vec.h"

int main(){
    int *vec, i, n;

    vec = alloc_mem_for_vec(10);

    /* realloc can return a different pointer from what was passed in */
    vec = fill_vec(vec);

    i = 0;
    while(*(vec + i) >= 0){
        printf("vec[%d] = %d\n", i, *(vec+i));
        i++;
    }

    free(vec);

    return 0;
}
```

Practice problem

handle_vec.c

```
int* alloc_mem_for_vec(int size){
    int *vec = (int*)malloc(size * sizeof(int));
    return vec;
}

int* fill_vec(int *vec){
    int *tmp, value = 0, pos = 0;

    do{
        printf("Enter a number: ");
        scanf("%d", &value);
        if(pos >= 10){
            tmp = (int*)realloc(vec, (pos+1)*sizeof(int));
            if(tmp == NULL){
                printf("Unable to resize vector!\n");
                break;
            }
            vec = tmp;
        }
        write_value(vec, pos, value);
        pos++;
    }while(value >= 0);

    /* realloc can move the block to accomodate the new size */
    return vec;
}
```

asm.s

```
.section .text
.global write_value

# void write_value(int *vec, int pos, int value)
write_value:
    # vec in %rdi, pos in %esi, value in %edx

    movl %edx, (%rdi,%rsi,4)    # vec[pos] = value
    ret
```