ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24
Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

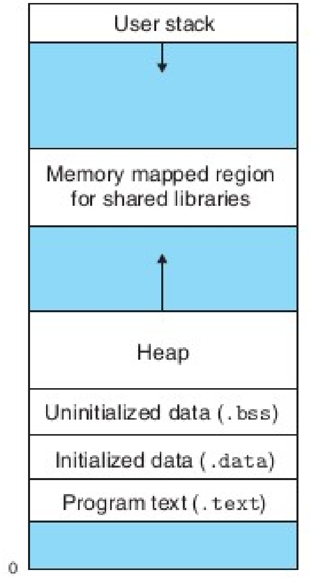
- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Memory

Program Memory Layout (address space)

- Every byte of memory in a program's memory space has an associated address.
- Everything the program needs to run is in its memory space, and different types of entities reside in different parts of a program's memory space.



Allocating Memory

```c
#define N 5

int s_array[N];
long xpto;
char a='2';
int get_max(int *v, int n){
  int i=0;
  int max = v[0];
  for(i=1;i<n;i++){
    if(v[i] > max)
    max = v[i];
  }
  return max;
}
int main(){
  int vec[N];
  int max = 0;
  ...
  max = get_max(vec,N);
  return 0;
}
```

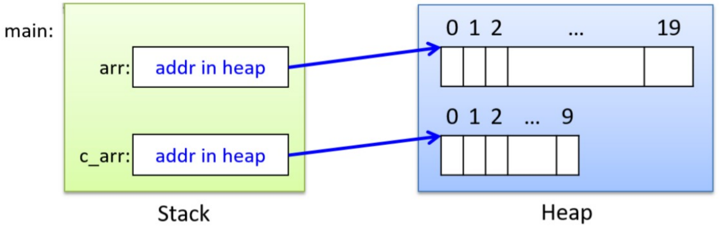
- **Static global data**
  - **Fixed size at compile-time**
  - **Entire lifetime** of the program (loaded from executable)
- Stack-allocated data
  - **Local/temporary** variables
  - **Known lifetime** (deallocated on function return)

- For instance, what if your declared array size becomes **insufficient or is more than required**?

Dynamic Memory Allocation (I)

- Memory allocated **on the fly** during run time
- Dynamically allocated memory occupies the **heap memory region of a program's address space**.
- For dynamic memory allocation, **pointers are crucial**

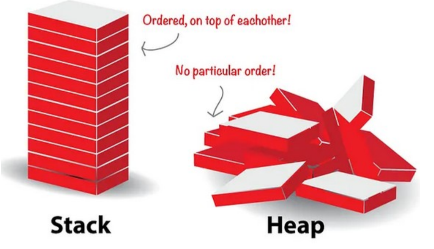
```c
int main(){
  int *arr;
  char *c_arr;
  arr = malloc(sizeof(int) * 20);// allocate an array of 20 ints on the heap:
  c_arr = malloc(sizeof(char) * 10);// allocate an array of 10 chars on the heap
  return 0;
}
```



Dynamic Memory Allocation (II)

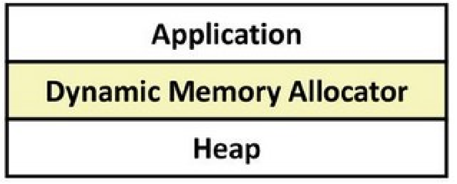
- It's important to remember that **heap memory is anonymous memory**, where "anonymous" means that addresses in the heap are not bound to variable names.
- Declaring a named program variable allocates it on the stack or in the data part of program memory.
- The allocation and deallocation of blocks from the heap follow **no set rules**, in contrast to the stack;
- Blocks may be allocated at any moment and released at any time.

- Numerous custom **heap allocators are available to tune heap performance for various usage patterns**, but this makes it much more difficult to track which parts of the heap **are allocated or free at any given time**.



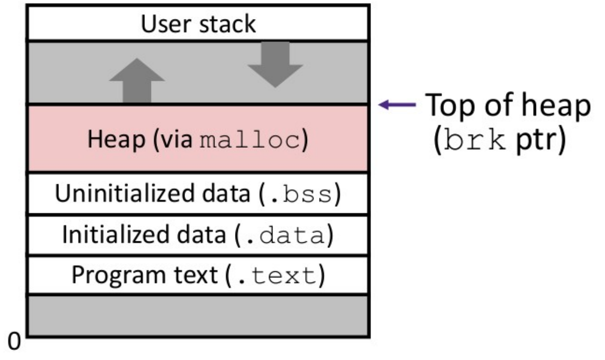
Dynamic Memory Allocation (III)

- Programmers use **dynamic memory allocators to User stack acquire virtual memory at run time**
  - For data structures whose size (or lifetime) **is known only at runtime**
  - **Manage the heap** of a program's virtual memory



- Types of allocators
  - **Explicit allocator**: programmer allocates and frees space
    - Example: `malloc` and `free` in C.
  - **Implicit allocator**: programmer only allocates space (no free)
    - Example: garbage collection in Java, Caml, and Lisp

- Allocator organizes heap **as a collection of variable- sized blocks**, which are either **allocated or free**
  - Allocator requests pages in the heap region;
  - Virtual memory hardware and OS kernel allocate these pages to the program
  - Application objects (arrays, variables and so on) are typically smaller than pages, so the allocator manages blocks within pages



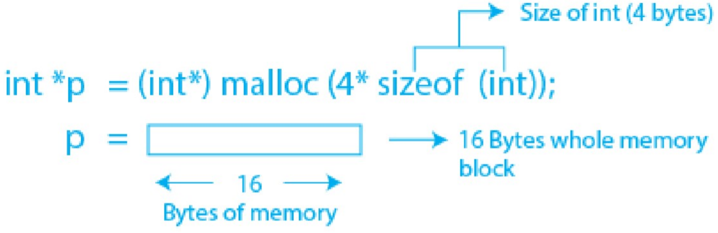
Dynamic Memory Allocation in C

Library

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - **Allocates the requested memory and returns a pointer to it.**
- `void *calloc(size_t nitems, size_t size)`
  - **Allocates the requested memory and returns a pointer to it.**
- `void *realloc(void *ptr, size_t size)`
  - Attempts to **resize the memory block pointed to by `ptr` that was previously allocated with a call to `calloc` or `malloc`.**
- `void free(void *ptr)`
  - **Deallocates the memory** previously allocated by a call to `calloc`, `malloc`, or `realloc`.
- Data types
  - `void *` (**void pointer**)
    - It is a pointer that **has no associated data type** with it.
    - It can hold an address of any type and **can be type casted to any type.**
  - `size_t`
    - It is an unsigned integer type

```
ptr = (castType*) malloc(size)
```

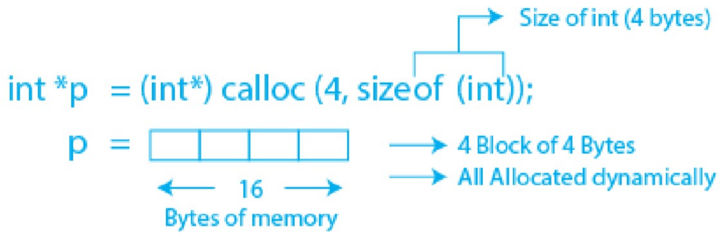
- It takes a single argument, which is the **number of bytes to allocate**, and returns **a pointer to the beginning of the allocated memory**.
- The memory allocated by `malloc` **is not initialized**, so it contains garbage values.
- It returns a pointer to the first byte of the allocated memory, or `NULL` if the request for memory could not be fulfilled.



calloc

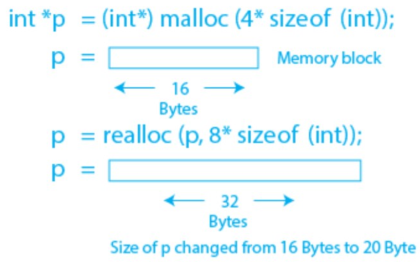
```
ptr = (castType*)calloc(n, size);
```

- It is used to allocate memory for an array of elements of a certain data type and **initializes all bits to zero**.
- The function takes two arguments: the number of elements in the array and the size of each element.
- It returns a pointer to the first byte of the allocated memory, or `NULL` if the request for memory could not be fulfilled.



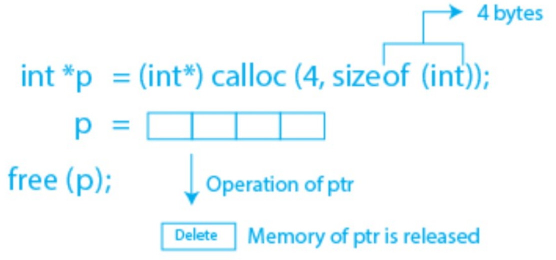
```
ptr = realloc(ptr, newSize);
```

- Itis used **to change the size of a previously allocated memory block**.
- It takes two arguments: a pointer to the memory block that you want to resize and the new size of the memory block.
- If the **new size is larger than the original size**, the function **may move the memory block to a new location in order to accommodate the increased size**.
- If the new size **is smaller than the original size**, the function **will truncate the memory block to the new size**.



```
free(ptr);
```

- It is used to **deallocate memory that has already been allocated** using the `malloc`, `calloc`, or `realloc` functions.
- Releasing the memory block indicated to by the address provided to it, it makes more allocations possible.
- To stop memory leaks in your program, you must utilize `free` on dynamically allocated memory.



Allocating

- To allocate heap memory, call `malloc` (or `calloc`), passing in the total number of bytes of contiguous heap memory to allocate.

```c
#include <stdlib.h>

int main() {
  int *p;
  p = (int *) malloc(sizeof(int)); // allocate heap memory for storing an int
  if (p != NULL) {
    *p = 6; // the heap memory p points to gets the value 6
  }
  ...
  return 0;
}
```

  - Use the `sizeof` operator to compute the number of bytes to request.

Deallocating

- When a program no longer needs the heap memory it dynamically allocated with `malloc`, it should explicitly deallocate the memory by calling the `free` function.

```
#include <stdlib.h>

int main() {
  int *p;

  ...
  free(p);
  p = NULL;
  return 0;
}
```

  - It is also a good idea to set the pointer's value to `NULL` after calling `free`,

Dynamically Allocated Arrays

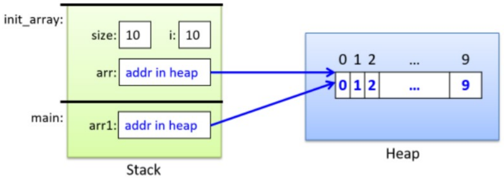
```
int i;
int s_array[20];
int *d_array;
d_array = malloc(sizeof(int) * 20);
if (d_array == NULL) {
  printf("Error: malloc failed\n");
  exit(1);
}
for (i=0; i < 20; i++) {
  s_array[i] = i;
  d_array[i] = i;
}
free(d_array);
d_array = NULL;
```

- After dynamically allocating heap space for an array, a program can access the array through the pointer variable.
  - Because the pointer variable's value represents the base address of the array in the heap, we can **use the same syntax to access elements in dynamically allocated arrays as we use to access elements in statically declared arrays**.
- When a program is finished using a dynamically allocated array, it should call free to deallocate the heap space.

Pointers to Heap Memory and Functions

- When passing a dynamically allocated array to a function, the pointer variable argument's value is passed to the function (i.e., the **base address of the array in the heap is passed to the function**).
  - Thus, when passing either statically declared or dynamically allocated arrays to functions, the parameter gets exactly the same value –the **base address of the array in memory**.

```
int main(void) {
  int *arr1;
  arr1 = malloc(sizeof(int) * 10);
  ...
  init_array(arr1, 10);
  ...
}
void init_array(int *arr, int size) {
  int i;
  for (i = 0; i < size; i++) {
    arr[i] = i;
  }
}
```



Allocating Memory for an Array in a Function

```
int main(void) {
  int *arr1 = NULL;
  int r = arr_alloc(&arr1, 10);
  ...
  free (arr1);
}
```

```
int main(void) {
  int *arr1 = NULL;
  int arr1 = arr_alloc(10);
  ...
  free (arr1);
}
```

```
int arr_alloc(int **p, int n) {
  int res = 1;
  *p = malloc(sizeof(int) * n);
  if (*p == NULL) {
    printf("malloc error\n");
    res = 0;
  }
  return res;
}
```

```
int* arr_alloc(int n) {
  int *p = malloc(sizeof(int) * n);
  return p;
}
```

Reallocating Memory for an Array

```
int i, *p;
/* allocate block of n ints */
p = (int*) malloc(n*sizeof(int));
/* check for allocation error */
if (p == NULL) {
  perror("malloc");
  exit(0);
}
/* initialize int array */
for (i=0; i<n; i++)
  p[i] = i;
 /* add space for m ints to end of p block */
p = (int*) realloc(p,(n+m)*sizeof(int));
/* check for allocation error */
if (p == NULL) {
  perror("realloc");
  exit(0);
}
/* initialize new spaces */
for (i=n; i < n+m; i++)
  p[i] = i;
/* print new array */
for (i=0; i<n+m; i++)
  printf("%d\n", p[i]);

free(p);
```

Reallocating Memory for an Array in a Function

```
int main(void) {
  int *arr1 = NULL;
  int r = arr_alloc(&arr1, 10);
  ...
  r = arr_realloc(&arr1, 10);
  ...
  free (arr1);
}
```

```
int arr_realloc(int **p, int n) {
  int res = 1;
  *p =(int*) realloc(*p,(n+m)*sizeof(int));
  if (*p == NULL) {
    printf("malloc error\n");
    res = 0;
  }
  return res;
}
```

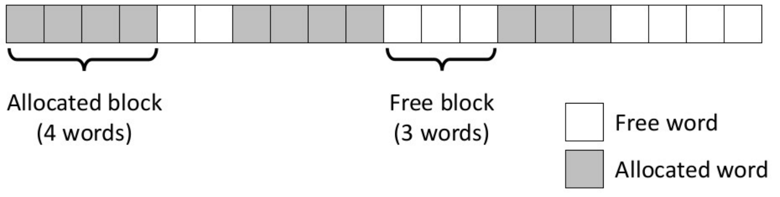
```
int main(void) {
  int *arr1 = NULL;
  ...
  int arr1 = arr_alloc(10);
  ...
  arr1 = arr_realloc(10);
  ...
  free (arr1);
}
```

```
int* arr_alloc(int *p, int n) {
  int *rp = (int*) realloc(p,(n+m)*sizeof(
        int));
  return rp;
}
```

Dynamic Memory Allocators

Assumption & Notation

- Let us assume that memory is divided into 32 bits (4 bytes) word.
  - Memory is word addressed (words are int-sized)
    - E.g. `malloc(4)` will be equivalent to `malloc(4*sizeof(int))`
- Allocations will be in sizes that are a multiple of boxes (i.e., multiples of 4 bytes).



Allocating and freeing blocks

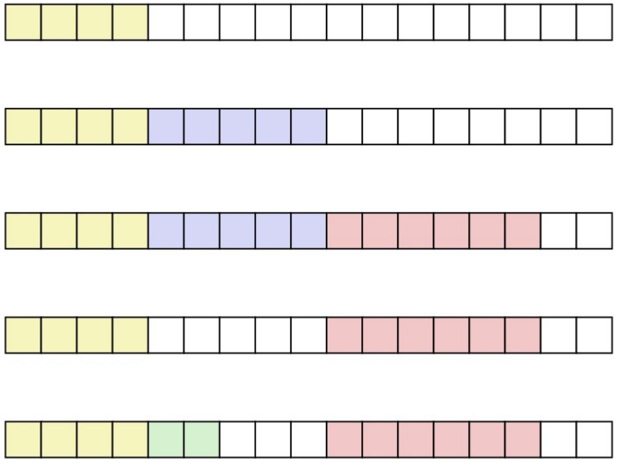
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`



Performance

- **Goals**:
  - Given some sequence of `malloc` and `free` requests $R_0, R_1, ..., R_k, R_{k+1}$, maximize **throughput** and **peak memory utilization**
    - These goals are often conflicting
- **Aggregate payload** P_k:
  - `malloc(p)` results in a block with a payload of `p` bytes
  - After request R_k has completed, the aggregate payload P_k is the **sum of currently allocated payloads**
- **Current heap size** H_k
  - Assume H_k is **monotonically non-decreasing**
    - Allocator can increase size of heap using `sbrk`

Performance

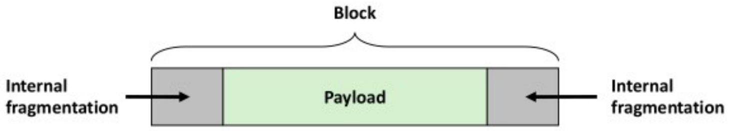
- **Throughput**
  - Number of completed requests per unit time
  - Example:
    - If 5000 `malloc` calls and 5000 `free` calls completed in 10 seconds, then throughput is 1000 operations/second
- **Peak Memory Utilization**
  - Defined as $U_k = (\max_{i \leq k} P_i)/H_k$ after $k + 1$ requests
  - Goal: maximize utilization for a sequence of requests

Fragmentation

- Poor memory utilization is caused by **fragmentation**
- Sections of memory are not used to store anything useful, but cannot satisfy allocation requests
- Two types: **internal** and **external**

Internal Fragmentation

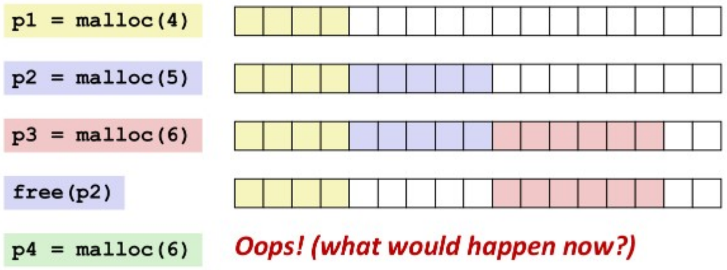
- For a given block, internal fragmentation occurs if payload is smaller than the block
  - It is the difference between de block size minus payload size



- Causes:
  - Padding for alignment purposes
  - Overhead of maintaining heap data structures (inside block, outside payload)

External Fragmentation

- For the heap, external fragmentation occurs when **allocation/free pattern leaves "holes" between blocks**
  - That is, the aggregate payload is non-continuous
  - Can cause situations where there is enough aggregate heap memory to satisfy request, but no single free block is large enough



  - Depends also on the pattern of future requests

Applications & Allocators

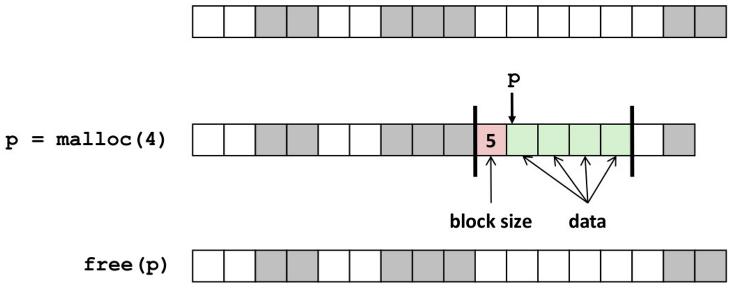
- Applications
  - Can issue **arbitrary sequence** of `malloc` and `free` requests
  - Must never access memory not currently allocated
  - Must never free memory not currently allocated
    - Also must only use `free` with previously `malloc`'ed blocks
- Allocators
  - Cannot control **number or size of allocated blocks**
  - Must respond immediately to `malloc` requests
    - i.e., cannot reorder or buffer requests
  - Must allocate blocks from free memory
    - i.e., can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
  - Can manipulate and modify only free memory
  - **Cannot move the allocated blocks**
    - i.e., defragmentation is not allowed

Implementation Issues

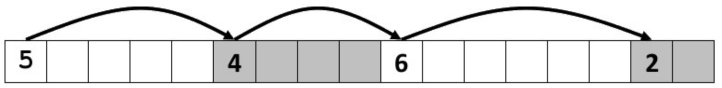
- How do we know **how much memory to free given just a pointer**?
- How do we **keep track of the free blocks**?
- How do we **pick a block to use for allocation** (when many might fit)?
- What do **we do with the extra space when allocating a structure that is smaller than the free block it is placed in**?
- How do we **reinsert a freed block into the heap**?

Knowing How Much to Free

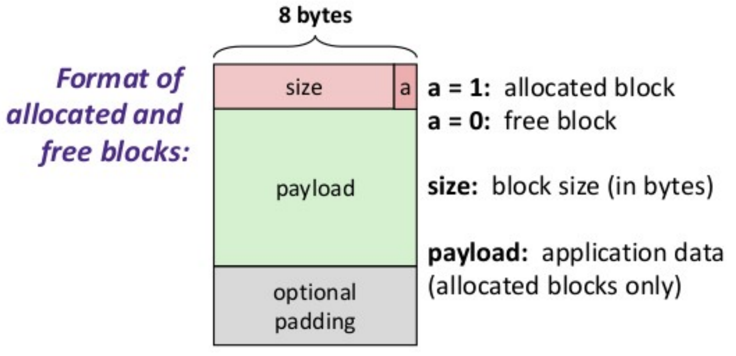
- Standard method
  - Keep the length of a block in the word preceding the data
    - This word is often called the header field or header
  - Requires an extra word for every allocated block



1. Implicit free list using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free

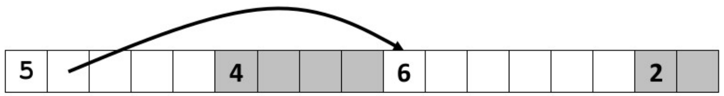


   - For each block we need both size and allocation status



Keeping Track of Free Blocks (II)

2. **Explicit free list** among only the free blocks, using pointers



3. **Segregated free list**
   - Different **free lists for different size “classes”**
4. **Blocks sorted by size**
   - Can use a **balanced binary tree** (e.g., red-black tree) with pointers within each free block, and the length used as a key