memory management

TP N4

Task

Implement malloc and free

- Any Memory Allocation Policy:
  - **best_fit, first_fit, next_fit, worst_fit**.
- Describe the memory as a structure:
  - Free List Double-Linked Structure

Allocating Memory

The process of finding a block of free memory big enough to hold the requested memory block.

The free block is then split (if necessary) with the right size block marked as ‘allocated’

The remaining block (if any) is marked as ‘free’.

brk() and sbrk()

resizing the heap is actually as simple as telling the kernel to adjust its idea of where the process's program **break** is.

Initially, the program **break** lies just past the end of the uninitialized data segment.

After the program break is increased, the program may access any address in the newly allocated area, but no physical memory pages are allocated yet.

The kernel automatically allocates new physical memory pages on the first attempt by the process to access addresses in those pages.

UNIX provides 2 systems calls for manipulating the program **break:**

- **brk() and sbrk()**

brk() and sbrk()

#include <unistd.h>

int **brk**(void *end_data_segment);

Returns 0 on success, or –1 on error

void ***sbrk**(intptr_t increment);

Returns previous program break on success, or (void *) –1 on error

brk() and sbrk()

brk()
Sets the program break to the location specified by end_data_segment. Since virtual memory is allocated in units of pages, end_data_segment is effectively rounded up to the next page boundary.
The precise upper limit on where the program break can be set depends on a range of factors, including: the process resource limit for the size of the data segment and the location of memory mappings, shared memory segments, and shared libraries.

sbrk()
Adjusts the program break by adding increment to it.
sbrk(0) returns the current setting of the program break without changing it. Can be useful to track the size of the heap.

An horrible dummy malloc

```
#include <sys/types.h>
#include <unistd.h>

void *malloc(size_t size)
{
        void *p;
        p = sbrk (0);
        /* If sbrk fails , we return NULL */
        if (sbrk(size) == (void*)-1)  return NULL;
        return p;
}
```

Freeing a memory block

This is done by changing the flag on the block to ‘free’.

Then, the blocks on both sides of the newly freed block are examined. If either (or both of them are free) the free blocks must be merged together to form just one free block.

Does not lower the program break, but instead adds the block of memory to a list of free blocks that are recycled by future calls to malloc().

Read the book: Linux Programming Interface, page 141.

A memory allocated block

```
struct mab {
	int offset;
	int size;
	int allocated;
	struct mab * next;
	struct mab * prev;
};
typedef struct mab Mab;
typedef Mab * MabPtr;
```

This is the base node of your structure.
Without the reverse link, you will need to do a separate pass throuhg the memory arena after marking a block as free to merge any adjacent blocks.

Resources

Book:

The Linux Programming Interface. Chap 7. Memory Allocation

Other resources:

pdfs shown in class