# CSS 430 Operating Systems
# Program 4A: Heap Space Management

## 1. Purpose

This assignment **<u>implements your own malloc( ) and free( ) using the "sbrk" system function: change data segment size.</u>** Your implementation is based on the *first-fit* and the *best-fit* strategies. Since the original malloc/free functions in Linux use "brk" (an even more legacy function than sbrk), you can compare your own and the Linux-original implementations in terms of # brk system function calls. (The few calls the better memory allocations.)

## 2. Heap Management

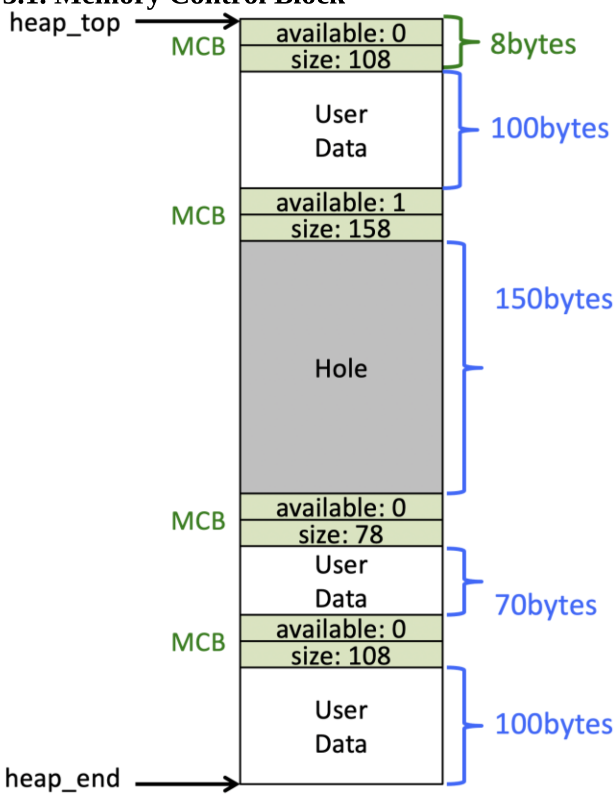
A process distinguishes three memory spaces: text (including the code and global variables), stack, and heap. Obviously, the text does not change its size. The stack grows as a new function is called to have an activation record including the return address of the caller function, parameters, and local variables. The heap grows when the program instantiates objects with "new" in C++, which is translated to malloc( ). Interestingly, malloc( ) itself is further translated to ***sbrk( a positive int )***: increases the heap size by moving down its boundary. **When a user program destroys objects through "delete" (which is automatically carried out in Java when the objects are no longer referred to by any other objects), the corresponding space is only marked as free through free( ) but not returned to the operating system immediately.** At a certain point when the heap space got too many holes (external segmentations), free( ) garbage-collects all these segments and thereafter calls sbrk( a negative int ): decreases the heap size by moving up its boundary toward the beginning of the space.

## 3. User-Level Heap Management

You will implement malloc( ) and free( ) by yourself, using sbrk( ). However, for simplicity, let's ignore garbage-collection. This, in turn, means that we will always pass 0 or a positive integer to sbrk( ): the former returns the address of the current heap boundary, and the latter moves down the boundary by this integer in bytes. Following the textbook Section 9.2.2 Memory Allocation, you will implement two versions of memory allocation:

| Function names | Strategies |
|---|---|
| malloc_f( ) and free_f( ) | **First fit:** allocate the first hole that is big enough. Searching can start from the beginning of the heap. We will stop searching as soon as we find a free hole that is large enough. |
| malloc_b( ) and free_b( ) | **Best fit:** Allocate the smallest hole that is big enough. We must search the heap entirely. This strategy produces the smallest leftover hole. |

### 3.1. Memory Control Block



# Program 4A: Heap Space Management

You will use the *variable partition* scheme for allowing a user program to receive a variety size of memory spaces from the heap. For this purpose, we define the following data structure called *memory control block* (MCB) that manages each partition. Each partition starts with its MCB followed by the actual user data. See the picture on how to manage the heap space.

```
class MCB { // memory control block
public:
  int available; // true(1): this memory partition is available, false(0): unavailable
  int size;      // MCB size + the user data size
};
```

Note that we will use the following two variables to locate the beginning and the boundary of the heap space

```
static void *heap_top; // the beginning of the heap space
static void *heap_end; // the current boundary of the heap space, obtained from sbrk( 0 )
```

### 3.2. Algorithm of malloc_f( long )

Find the template malloc.cpp file on Canvas under Files/code/prog4/. Your malloc_f function first initializes the *heap_top* and *heap_end* variables. Thereafter, increase a user-specified *size* by the size of MCB. Then, you will search the heap from heap_top to heap_end for an available partition. **This logic portion is your assignment**. If no space is available, in other words, if new_space is null, you need to move down the heap boundary with sbrk( ) and to update heap_end. **This logic portion is your assignment, too.** Once you get a new_space, you'll initialize cur_mcb->available and cur_mcb->size.

```
void *malloc_f/b( long size ) {
  struct MCB *cur_mcb;          // current MCB
  void *new_space = NULL; // this is a pointer to a new memory space allocated for a user

  if( !initialized )   {
    // find the end of heap memory, upon an initialization
    heap_end = sbrk( 0 );
    heap_top = heap_end;
    initialized = true;
  }
  // append an MCB in front of a requested memroy space
  size = size + sizeof( MCB );

  // scan from the top of the heap
  // if cur_mcb->available and cur_mcb->size fits size, new_space points to this MCB
  // Task 1: implement by yourself.  (may need 15 lines)

  // no space found yet
  if ( new_space == NULL ) {
    // move down the heap boundary, initialize new_space with heap_end, and update heap_end.
    // Task 2: implement by yourself.  (may need 5 lines)

    cur_mcb = (MCB *)new_space;
    cur_mcb->available = 0;
    cur_mcb->size = size;
  }
  // new space is after new MCB
  return (void *)( ( long long int )new_space + sizeof( MCB ) );
}
```

### 3.3. Algorithm of malloc_b( long )

The malloc_b( ) function differs from malloc_f in only scanning the heap space. You have a little more complexity: scan all MCBs to identify the smallest partition that fits a user-requested size. **This is your assignment (task3 probably can be coded in up to 20 lines)**. All the other logics including heap_top/heap_end initialization, the heap boundary move-down with sbrk( ), and a new MCB initialization are the same as malloc_f( ).

### 3.4. Algorithm of free_( void* )

Given a pointer to the space to be deallocated (i.e., dealloc_sapce below), locate this space's MCB and set its available true.

```
    void free_( void *dealloc_space ) {
      MCB *mcb;

      // locate this partition's mcb address from dealloc_space
      // Task 4: implement by yourself. (one line of code may work)
      mcb->available = true;
      return;
}
```

### 3.4. driver.cpp

The driver.cpp program is provided for the verification and measurement of your program execution. Find the driver.cpp on Canvas under Files/code/prog4/. The program receives 1 or 2 arguments:

| Arguments | Actions | Example |
|---|---|---|
| 1st argument (l, f, or b) | distinguishes the malloc logics. l uses Linux-original malloc; f calls malloc_f( ); and b calls malloc_b( ). | ./a.out l<br>./a.out f<br>./a.out b |
| 2nd argument (p, n, or nothing) | This is optional. p prints out all memory allocations/de-allocations called by the driver program. n has no printings. The default is n. | ./a.out l p<br>./a.out f p<br>./a.out b p |

The program first allocates 10 different memory chunks whose sizes are randomly chosen but less than 1024 bytes. Thereafter, it repeats 100 iterations, each randomly choosing one of these chunks to de-allocate or re-allocate if previously de-allocated. You don't have to change this driver program at all.

### 3.5. The strace command

For the measurement of your code execution, you should use ***strace*** to trace what system function a given execution (i.e., a.out) called.

```
      $ strace ./a.out l
```

Note that strace prints out all traced results into standard error (fd = 2). Furthermore, we are interested in counting only brk( ) system calls rather than tracing all. Therefore, we will execute our a.out as follows:

```
      $ strace ./a.out l 2>&1 | grep brk
```

## 4. Statement of Work

**Task 1**: Implement malloc_f( )'s heap-scanning logic. It can be done in at most 15 lines.
**Task 2**: Implement malloc_f( )'s logic to move down the heap boundary. It can be coded in at most 5 lines.
**Task 3**: Implement malloc_b( )'s heap-scanning logic. It can be done in at most 20 lines.
**Task 4**: Implement free_( )'s MCB address calculation in one line.
**Task 5**: Test your malloc.cpp with driver.cpp as follows:

```
      [css430@cssmpi1h prog4]$ strace ./a.out l 2>&1 | grep brk
      brk(NULL)                                = 0xebf000
      brk(NULL)                                = 0xebf000
      brk(0xee0000)                            = 0xee0000
      brk(NULL)                                = 0xee0000
      [css430@cssmpi1h prog4]$ strace ./a.out b 2>&1 | grep brk
      brk(NULL)                                = 0xa88000
      brk(NULL)                                = 0xa88000
      brk(NULL)                                = 0xa88000
      brk(0xa8816f)                            = 0xa8816f
      ...
      [css430@cssmpi1h prog4]$ strace ./a.out f 2>&1 | grep brk
      brk(NULL)                                = 0x1659000
      brk(NULL)                                = 0x1659000
      brk(NULL)                                = 0x1659000
      brk(0x165916f)                           = 0x165916f
      ...
      [css430@cssmpi1h prog4]$
```

# brk( ) calls must be the smallest with the Linux-original malloc/free; the 2nd smallest is your malloc_b/free; and the largest (i.e., the most inefficient) must be your malloc_f/free.

## 5. What to Turn in

| | Materials | Points | Note |
|---|---|---|---|
| 1 | malloc.cpp<br>**a.** Code organization: **+2pts**<br>• Well organized: 2pts,<br>• Poor comments or bad organization: 1pt, or<br>• No comments and horrible code: 0pts<br>b. malloc_f( ): **+6pts**<br>• Heap-scanning logic: +4pts and<br>• Logic to move down the heap boundary: +2pts<br>c. malloc_b( ): **+5pts**<br>• Heap-scanning logic: 5pts<br>**d.** free_( ): **+2pts**<br>• Correct MCB address calculation: 2pts | 15 | Submit this .cpp file individually. Please also submit driver.cpp individually, though you don't need to change this file at all. |
| 2 | an execution snapshot<br>• Correct (# brk( ) calls must be the smallest with the Linux-original malloc/free; the 2nd smallest is your malloc_b/free; and the largest (i.e., the most inefficient) must be your malloc_f/free): 5pts<br>• Wrong: 3pts, or<br>• No outputs: 0pts | 5 | Include the snapshots in a pdf file named: FirstNameLastName_prog4A.pdf. Submit this pdf file individually. Please clearly label each snapshot and add proper explanation. |