CS 446/646 – Principles of Operating Systems

Homework 4

Due date: Tuesday, 11/19/2024, 11:59 pm

Objectives: You will implement your own Dynamic Memory Allocator. You will be able to describe how Heap Allocation / Deallocation in a program's Virtual Address Space takes place, and how certain options may control it when using the built-in **`glibc`** Dynamic Memory Management implementations.

General Instructions & Hints:

- All work should be your own.
- The code for this assignment must be in C. Global variables are not allowed except any explicitly specified ones, and you must use function declarations (prototypes). Name files exactly as described in the documentation below. All functions should match the assignment descriptions. If instructions about parameters or return values are not given, you can use whatever parameters or return value you would like.
- All work must compile on **Ubuntu 18.04**. Use a **`Makefile`** to control the compilation of your code. The **`Makefile`** should have at least a default target that builds your application.
- To turn in, create a folder named **`PA4_Lastname_Firstname`** and store your **`Makefile`**, and your **`.c`** source code files in it. Do not include any executables. Then compress the folder into a **`.zip`** or **`.tar.gz`** file with the same filename as the folder, and submit that on WebCampus.

Background:

Heap Memory Allocation

The widely Dynamic Memory Management functions you have always been using, i.e. [**`malloc`**](https://linux.die.net/man/3/malloc) and [**`free`**](https://linux.die.net/man/3/free) are (User-Space) C-library functions, not System Calls. The reason is that they manage (i.e. do bookkeeping of) a Process' Virtual Address Space, and internally they use System Calls to the kernel, that allow to make requests to the system that will eventually allocate / release actual Physical Memory. Two such System Calls are:

a) [**`brk`** / **`sbrk`**](https://linux.die.net/man/2/brk / https://linux.die.net/man/2/sbrk): These two System Calls effectively serve the same purpose (thus they are mentioned together in the manpages), which is to change the Virtual Address location of a program's **`break`**. A program's **`break`** corresponds to the Virtual Address of the end (top) of its Heap section. At the start of a program, **`break`** is initialized at the first location after the end of the Uninitialized Data Segment, and modifying (moving) it via the **`brk`** / **`sbrk`** System Calls has the effect of Allocating -or- Deallocating more Virtual Memory to the Process.

Moving **`break`** at a higher Virtual Memory Address (using **`brk`** and its **`addr`** parameter) or shifting it by a positive increment (using **`sbrk`** and its **`increment`** parameter) has the notion of Allocating more Virtual Memory in the Process' Virtual Address Space. The opposite has the notion of Deallocating Virtual Memory in the Process' Virtual Address Space.

***CAUTION - Lazy (De-)Allocation*:** The fact that an allocating **`brk`**'s / **`sbrk`**'s return results may indicate success, does not mean that the OS has actually reserved Physical Pages for the corresponding Virtual Address Pages that were Allocated. The kernel creates a mapping in the

Process' User Page Table marked as Invalid, such that when an actual attempt to access it is performed, it will trigger a Page Fault, and a Page Fault Handler will then take care of reserving and mapping Physical Memory to the Virtual Address Page. Similarly, a deallocating **`brk`** / **`sbrk`** which decrements the program **`break`** Virtual Address location does not lead to an immediate release of Physical Pages that were so far mapped to the Virtual Pages that just got Deallocated; that would create unnecessary overhead in most situations. The kernel is free to manage Physical Memory as it deems more appropriate (e.g. once there are conditions of Memory Pressure, either do the standard Physical Page Swap-Out to disk –necessarily for Physical Pages that correspond to Virtual Pages that are presently still allocated–, or perform bookkeeping on Processes to see if it can reclaim Physical Memory that is mapped to Virtual Pages that correspond to higher addresses than the program **`break`**, or whatever other strategy is implemented by the specific OS).

Note: The strategy of using **`brk`** / **`sbrk`** to Allocate Heap Memory has its caveats, especially when dealing with the allocation of large objects. Imagine allocating dynamically (variably) a number of huge objects (e.g. each object can be a **`double[1000000]`** array), such that a sequence of: **allocate array_1**, **allocate array_2**, …, **allocate array_10** is performed using the **`brk`** /**`sbrk`** method. At this point the program **`break`** is at the next location after the **array_10** object. If we now wish to Deallocate for example **array_3** - **array_7**, we cannot move back the program **`break`** at the next location after the **array_2**, because this would mean Deallocating **array_8** - **array_10** as well. This strategy will eventually create Memory Pressure.

Generally, the whole incremental notion of this strategy, implies it is suitable to Deallocate some of Virtual Memory chunk that was Allocated just before; so this is how it's used. It Allocates **a large enough Virtual Memory chunk** to fit a number of objects, and then uses it to fit whatever Dynamic Object Allocation request are made within it. If a new Dynamic Object Allocation request cannot fit, it then attempts to grow the allocated Virtual Address Space by using **`brk`** / **`sbrk`** incrementally again. If as a result of some Dynamic Object Dellocation requests it deems that lived at the higher Virtual Memory Addresses, it deems that it can reduce the Virtual Address Space, it can do so by using **`brk`** / **`sbrk`** decrementally; this is however inefficient and not done in practice, once Virtual Address Space has been Allocated, the Process might as well not utilize it (or re-use it in the future if multiple Dynamic Objects need to be Allocated again).

I.e. the strategy of using **`brk`** / **`sbrk`** is very fast, but we need to perform bookkeeping at the User-Space level (inside the implementation of **`malloc`**) by keeping track of which Virtual Addresses between the bottom of the Heap and the current location of the **`break`** are Free (with a so-called "Freelist" data structure), in order to fit Dynamic Object Allocation requests within one of the available chunks.

b) [**`mmap`**](https://linux.die.net/man/2/mmap): This creates a Virtual Address mapping in the Process' Virtual Address Space at the location indicated by the **`addr`** parameter, and of a size indicated by the **`length`** parameter. This creates a so-called "Anonymous Memory region" in the program's Virtual Address Space (i.e. non-"File-backed Memory" – see Lectures). We can use this functionality offered by this System Call in order to **treat the `mmap`-ed Anonymous Memory Regions as Heap memory**.

The benefit this offers is evident once large objects need to be allocated. In the case of a fragmented Virtual Adress Space (due to many prior Allocations/Deallocations), using the "Freelist" to find a chunk large enough to fit the large object could easily take too long and even fail most of the time (necessitating an incremental **`brk`** / **`sbrk`** to grow the Heap Virtual Address Space). Now imagine multiple large objects needing to be allocated back-to-back; we

would have to go though the **brk** / **sbrk** interface every time, defeat the design purpose of a pre-Allocated Heap Virtual Address Space which is used to quickly find slots to fit new objects, and also create non-homogeneous fragmentation (all ranges of large-to-small objects mixed up).

So instead, for objects over a certain (configurable) size, we elect to **place them in Virtual Address Space regions somewhere between the downwards growing Stack, and the upwards growing Heap** (with large enough region boundaries to ensure there will be no overlap). This direct Allocation of a Process' Virtual Address Space locations is accomplished with **mmap** (through its **addr** parameter). Quite conveniently, Deallocation of Virtual Address Space Regions corresponding to previously allocated objects can be accomplished via the System Call [**munmap**](https://linux.die.net/man/2/munmap).

Note: When using the system-provided **malloc** implementation, the size limit over which the Dynamic Memory Allocation behavior reverts to the **mmap** strategy is configurable via the [**mmallopt**](https://linux.die.net/man/3/mallopt) C-library function and the **M_MMAP_THRESHOLD** parameter.

You are required to implement your own Dynamic Memory Allocator **mymalloc**, which should have the same interface as the built-in **glibc** version of **malloc**:

```
void * mymalloc(size_t size);
```

size: Requested allocation size

return value: Pointer to Virtual Address location which was allocated and can be safely used (up-to **size** Bytes in length). **NULL** if allocation failed.

You will not be required to implement both functionalities in your custom **mymalloc** implementation, only strategy a) which uses **sbrk**. In order to do so, you are required to implement a Memorylist data structure that will facilitate the bookkeeping required.

IMPORTANT Note: The Memorylist will be a list of Virtual Memory Blocks (in effect the nodes of a Doubly-Linked-List). Each Memory Block (node) is a **struct mblock_t** which contains the information about the **next** and the **prev**ious Memory Block (**mblock_t*** both), the **size** of the *useful* memory in the Memory Block (**size_t**), its Allocation **status** (**int**, **1**: Allocated, **0**: Free), and the **payload** buffer (**void***) whose *field address* (**&(mblock_t.payload)**) is the location in memory which can be used for a Dynamically Allocated object's actual content. Therefore you should not confuse the size of the *entire* Memory Block with the **size** of the data it is able to hold

```
typedef struct _mblock_t {
    struct _mblock_t * prev;
    struct _mblock_t * next;
    size_t size;
    int status;
    void * payload;
} mblock_t;
```

The **offsetof** pointer arithmetic can be leveraged to get the offset (in Bytes) between the beginning of an **mblock_t** struct and the starting location of its **payload** field (**&(mblock_t.payload)** - location at which where any newly Allocated data should be placed):

```
#define MBLOCK_HEADER_SZ offsetof(mblock_t, payload)
```

This also allows to easily infer the size of the *entire* Free Block (summing **mblock_t.size** and **MBLOCK_HEADER_SZ**).

Your implementation for Dynamic Object Allocation should follow the "First-Fit" strategy, i.e.:

i) Traverse the Memorylist to find the first available Free Block which can fit the requested object allocation size.

   a. Once found, split up the Memory Block into the chunk that will be *reserved* to place the object, and the *remaining* free Memory Block chunk. Adjust the Memorylist in order to indicate that the reserved chunk has been taken away, and only the remaining free chunk is part of the Memorylist.

      To do so, you need to find the address where the *reserved* Memory Block will end (somewhere inside the *Address Space* starting at **`&current_block`** and ending at **`&current_block + MBLOCK_HEADER_SZ + current_block.size`**). Then reinterpret (Pointer-cast) that address as a **`mblock_t*`** to indicate the beginning of a new Memory Block **`remaining_block`** (splitting). Then initialize the **`prev`**, **`next`**, **`status`**, **`size`** fields of the **`remaining_block`** to attach it to the Memorylist and mark it as Free. Finally, adjust the **`current_block->next`** MemoryBlock's field **`prev`** to attach it to the **`remaining_block`**, and also adjust the **`current_block`**'s fields **`next`**, **`size`** to mark it as Allocated and attach it to the newly created **`remaining_block`** itself.

      Note: Think how to deal with cases such as a) fitting an object that is exactly the same size as the one available in the Free Block, or b) fitting an object that is smaller, but not as small as to allow enough remaining free space post-split for a new Free Block that will be able to fit at least **`sizeof(mblock_t)`**+at least 1 Byte of an actual object (in practice, would use a more conservative threshold rather than just 1 Byte).

   b. Return from **`mymalloc`** the *field address* of **`payload`** (**`&(mblock_t.payload)`**, which is the newly reserved Memory Block's Virtual Address location, which can now be used for whatever Dynamic object we wanted to Allocate when we called it.

ii) If there is no available Free Block large enough in the Memorylist, then grow the Heap.

   c. Use **`sbrk`** to move the Heap **`break`** by the larger value between i) the requested allocation size (+whatever offset might be necessary), ii) a fixed amount of large-enough memory (e.g. 1KB).
   Note: You need to verify the return value of **`sbrk`** to ensure that the Heap Virtual Address Space growth has succeeded.

   d. Adjust the Memorylist to attach the newly allocated Virtual Address Space at the end of it, by creating a new Memory Block. Effectively this means:
      i. Reinterpret (i.e. Pointer-cast) the starting location of the Virtual Memory chunk you just obtained by **`sbrk`** as a **`mblock_t*`**.
      ii. Initialize its **`prev`**, **`next`**, **`status`**, **`size`** fields appropriately.
      iii. Attach it to the end of the Memorylist by pointing its last Memory Block's **`next`** Pointer to point to its **`mblock_t*`** address.

   e. Now, allocate the requested memory size (step a) and return the appropriate address (step b).

Your Memorylist data structure effectively only needs to be:

```
typedef struct _mlist_t {
    mblock_t * head;
} mlist_t;
```

and in your program you will need 1 Global Variable **`mlist_t mlist;`** which will be the global (for the entire Process) Heap Memorylist.

Note: Remember to properly initialize **`head`** to **`NULL`**.

You are also required to implement your own Dynamic Memory Deallocator **`myfree`**, which should have the same interface as the built-in **`glibc`** version of **`free`**:

`void myfree(void * ptr);`

`ptr:` Pointer to Virtual Address location which was previously allocated and should now be Deallocated.

Your implementation for Dynamic Object Deallocation should:

iii) Assume the **`ptr`** Virtual Memory Address is part of a **Memory Block which was previously a *free* Memory Block, and then it was Allocated** (and therefore **`ptr`** is the address of a **`payload`** field (**`&(mblock_t.payload)`**).

iv) Thus, **`myfree`** should first backtrack enough to go to the beginning of the Memory Block Header (use **`MBLOCK_HEADER_SZ`**).

v) From that point, it can reinterpret the address as a **`mblock_t*`**, which will now allow it to easily manipulate the **`prev`** and **`next`** Pointers to the corresponding Memorylist Blocks, as well as observe the **`size`** of the *useful* data this Memory Block can store.

vi) **Coalesce** this Memory Block which was just Freed, with other adjacent Memory Blocks that are marked in their **`status`** fields as Free as well.

To do this you need the reverse process of i)-a. Which means to fuse two sequential Memory Blocks (if both are Free), by absorbing the second one into the first one. Effectively this can be accomplished by adjusting the **`second->next`** Memory Block's **`prev`** to point to the **`first`**, and adjusting the **`first->next`** to point to the **`second->next`** (simple Doubly-Linked List node deletion). Finally you have to adjust the fused Memory Block's **`size`** field appropriately.

Note: This is called "*Immediate Coallescing*". There is also the option to perform *"Deferred Coallescing"*, which is effectively the execution of the operation while the Memorylist is being traversed to find Free Memory Blocks.

General Directions:

Name your program **`mymalloc.c`** . You will turn in C code for this. **You are not required to make your implementation Thread-Safe.**

You may only use the following libraries, and not all of them are necessary:

```
<stdio.h>
<string.h>
<stdlib.h>
<stddef.h>
<sys/wait.h>
<sys/types.h>
<unistd.h>
<fcntl.h>
<errno.h>
<sys/stat.h>
```

You will need to write a minimum of the following functions (you may implement additional functions as you see fit):

- **main**
  Input Params: **int argc, char* argv[]**
  Output: **int**
  Functionality: It is not required to do any parsing of command line arguments, it is only required to call **mymalloc** a number of times to test it. Use the following test sequence (a different sequence may be used for grading purposes):

```
void * p1 = mymalloc(10);
void * p2 = mymalloc(100);
void * p3 = mymalloc(200);
void * p4 = mymalloc(500);
myfree(p3); p3 = NULL;
myfree(p2); p2 = NULL;
void * p5 = mymalloc(150);
void * p6 = mymalloc(500);
myfree(p4); p4 = NULL;
myfree(p5); p5 = NULL;
myfree(p6); p6 = NULL;
myfree(p1); p1 = NULL;
```

  You can use the provided function with this assignment (look in Files folder of Webcampus):

```
void printMemList(const mblock_t* headptr);
```

  to debug the contents of your Memorylist.

- **mymalloc**
  Input Params: **size_t size**
  Output: **void ***
  Operates as described above.

- **myfree**
  Input Params: **void * ptr**
  Output: **void**
  Operates as described above.
  Should also perform basic sanity checking with respect to Virtual Memory bounds by checking whether the address of the Block to be freed is greater than the **head** of the Memorylist, and lesser than the current program **break** (Remember: can be obtained by **sbrk(0)**).

- You are **strongly encouraged** to organize your code by splitting tasks and responsibilities into separate functions, e.g: (indicative prototypes – can adjust in own implementation as required):
  - **mblock_t * findLastMemlistBlock()**
    Traverse Memorylist to find the last Memory Block.
  - **mblock_t * findFreeBlockOfSize(size_t size)**
    Traverse Memorylist to find the first Free Memory Block that can fit a requested payload data size.
  - **void splitBlockAtSize(mblock_t * block, size_t newSize)**
    Split a MemoryBlock such that 2 Memory Blocks are created, the first one being able to accommodate a specific payload new size requested, and the second one getting whatever remains. Makes sure the Memorylist structure is kept up-to-date.
  - **void coallesceBlockPrev(mblock_t * freedBlock)**
    Coalesce a Memory Block with the previous one in the Memorylist (assuming it is Free). Makes sure the Memorylist structure is kept up-to-date.

- **`void coallesceBlockNext(mblock_t * freedBlock)`**
  Coalesce a Memory Block with the next one in the Memorylist (assuming it is Free). Makes sure the Memorylist structure is kept up-to-date.
- **`mblock_t * growHeapBySize(size_t size)`**
  Increase the Heap allocation and create a new Memory Block in the Virtual Address Space which was reserved. Attach it to the Memorylist.

CAUTION – Read the MANual: Make sure you read carefully how the API of any System Calls / C-Library functions you use works. For example, **`sbrk`** which can be used to increment/decrement the program **`break`** returns the location of the **previous program `break`** location (before the change).
CAUTION – Pointer Arithmetic: Make sure you are extremely careful when using Pointer arithmetic. E.g. you have to be able to understand what the difference between:
```
int * a = 0xffff;  int * b = a + 100;
```
and
```
int * a = 0xffff;  int * b = (char *)a + 100;
```
is.
You also have to understand what the problem with:
```
int * a = 0xffff;  int * b = (void *)a + 100;
```
is, i.e. try compiling the last expression with the **`-pedantic`** flag, and understand what the displayed warning is about.

Submission Directions:
When you are done, create a directory named **`PA4_Lastname_Firstname`**, place your **`Makefile`** (which has to have at least one default target that builds your **`mymalloc`** application executable), and your **`mymalloc.c`** source code file into it, and then compress it into a **`.zip`** or **`.tar.gz`** with the same filename as the folder.
Upload the archive (compressed) file on Webcampus.

Early/Late Submission: You can submit as many times as you would like between now and the due date.
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects that are up-to 24 hours late will receive a 15% penalty, ones that are up-to 48 hours late will receive a 35% penalty, ones that are up-to 72 hours late will receive a 60% penalty, and anything turned in 72 hrs after the due date will receive a 0.

Verify your Work:
You will be using **`gcc`** to compile your code. Use the **`-Wall`** switch to ensure that all warnings are displayed. Use the **`-pedantic`** switch to ensure Pointer your arithmetic does not rely on GCC-specific extensions. Strive to minimize / completely remove any compiler warnings; even if you don't warnings will be a valuable indicator to start looking for sources of observed (or hidden/possible) runtime errors, which might also occur during grading.

After you upload your archive file, re-download it from WebCampus. Extract it, build it (e.g. run **`make`**) and verify that it compiles and runs on the ECC / WPEB systems.

- Code that does not compile will receive an automatic 0.