

Memory Allocator in C

```
#include <unistd.h>
#include <string.h>
#include <pthread.h>

#include <stdio.h>
typedef char ALIGN[16];

union header{
struct{
size_t size;
unsigned is_free;
union header* next;
}s;
ALIGN stub;
};
typedef union header header_t;
header_t *head=NULL,*tail=NULL;
pthread_mutex_t global_malloc_lock;
header_t * get_free_block(size_t size){
header_t * curr = head;
while(curr){
if(curr->s.is_free && curr->s.size >= size){
return curr;
}
curr=curr->s.next;
}
return NULL;
}
void * my_malloc(size_t size){
size_t totalsize;
void * block;
header_t * header;
if(!size){
return NULL;
}
pthread_mutex_lock(&global_malloc_lock);
header=get_free_block(size);
if(header){
header->s.is_free=0;
pthread_mutex_unlock(&global_malloc_lock);
return (void *) (header+1);
}
```

```

}
totalsize=size + sizeof(header_t);
block=sbrk(totalsize);
if(block == (void *)-1){
pthread_mutex_unlock(&global_malloc_lock);
return NULL;
}
header=block;
header->s.is_free=0;
header->s.size=size;
header->s.next=NULL;
if(!head){
head=header;
}
if(tail){
tail->s.next=header; //// this new block is after our current tail
due to nature of
//// sbrk and the heap (it only grows unidirectional)
}
tail=header;
return (void*) (header+1);
}

void * my_calloc(size_t num,size_t nsize){
size_t size;
void* block;
if(!num || !nsize){
return NULL;
}
size=nsize*num;///check for overflow
if(nsize != size/num){
return NULL;
}
block=my_malloc(size);
if(!block){
return NULL;
}
memset(block,0,size);
return block;
}

void * my_free(void * block){
header_t * header,*tmp;
void * program_break;
if(!block)return NULL;
pthread_mutex_lock(&global_malloc_lock);
header=(header_t*)block -1;
program_break=sbrk(0);
if( (char*)block +header->s.size == program_break){
if(head==tail){
head=tail=NULL;

```

```

    }
    else{
        tmp=head;
        while(tmp){
            if(tmp->s.next==tail){
                tmp->s.next=NULL;
                tail=tmp;
            }
            tmp=tmp->s.next;
        }
    }
    sbrk(0-sizeof(header_t)-header->s.size);
    pthread_mutex_unlock(&global_malloc_lock);
    return NULL;
}
header->s.is_free=1;
pthread_mutex_unlock(&global_malloc_lock);
}

```

```

void * realloc(void * block,size_t size){
    header_t * header;
    void * ret;
    if(!block || !size){
        return my_malloc(size);
    }
    header=(header_t*)block -1;
    if(header->s.size>=size)return block;
    ret = my_malloc(size);
    if(ret){
        memcpy(ret,block,header->s.size);
        my_free(block);
    }
    return ret;
}

```

```

int main(){
    return 1;
} is this correct .

```

You've implemented a **custom thread-safe dynamic memory allocator** using `sbrk`, mutexes, and a simple free list. Overall, the structure is quite solid, but there are a few **important issues, improvements, and clarifications** needed.

✓ What's Good:

- Use of `pthread_mutex_t` for thread safety.
- A correct basic free-list-based memory allocator (`my_malloc`, `my_free`, `my_calloc`, `realloc`).
- Checks for edge cases like zero-size allocations and overflow in `calloc`.

⚠ Issues & Fixes:

1. ✗ Function Signature of `my_free` Should be `void`

c

```
void * my_free(void * block)
```

This is incorrect. `free`-like functions do not return anything.

✓ Fix:

c

```
void my_free(void *block)
```

2. ✗ Incorrect Use of `sbrk` in `my_free`

c

```
sbrk(0 - sizeof(header_t) - header->s.size);
```

You're trying to shrink the heap using `sbrk` negatively, but this is **not safe nor portable**. Most systems do not support shrinking the heap arbitrarily like this, and `sbrk` is deprecated in many systems.

✓ You should just mark the block as free, and optionally allow coalescing or reuse.

If you're **really** trying to shrink the heap only for the last allocated block (i.e., at the program break), you need to make sure **nobody else is using that memory** (e.g., via `my_malloc` previously). This logic is fragile and should be avoided unless you're writing a real allocator with robust tracking.

3. ✗ Potential Infinite Loop in `my_free`

c

```
while(tmp){
    if(tmp->s.next==tail){
        tmp->s.next=NULL;
        tail=tmp;
    }
    tmp=tmp->s.next;
}
```

If `tmp->s.next == tail`, then after updating `tmp`, you're setting `tmp = tmp->s.next`, which is `NULL`, so that's okay. But it's slightly fragile logic; better to `break` after finding the match.

✓ Recommended:

c

```
while(tmp && tmp->s.next){
    if(tmp->s.next == tail){
        tmp->s.next = NULL;
        tail = tmp;
        break;
    }
}
```

```
tmp = tmp->s.next;
}
```

4. ⚠️ Function Name Shadowing (`realloc`)

You're declaring a function named `realloc`, which shadows the standard library's `realloc`.

✅ Recommended:

```
c
void* my_realloc(void* block, size_t size)
```

5. ⚠️ Missing `pthread_mutex_init`

Before using `pthread_mutex_lock`, you should initialize the mutex.

✅ Add this at the top of `main()`:

```
c
pthread_mutex_init(&global_malloc_lock, NULL);
```

Also, consider `pthread_mutex_destroy` at the end if your program is long-running.

6. ⚠️ Alignment Consideration

You're using `ALIGN[16]` to force alignment in the `union`, which is a good idea. Just ensure `sizeof(header_t)` is aligned (you can print it to confirm). You might also want to use `__attribute__((aligned(16)))` if on GCC/Clang.

✅ Minor Suggestions:

- Add logging or debugging tools like `printf` or memory dump functions to inspect the allocator.
- Implement **block coalescing** (merge adjacent free blocks).
- Consider splitting large free blocks into smaller ones.

✅ Final Fixes Summary:

Here's what you **must** change:

🔧 Change `my_free` return type:

```
c
void my_free(void * block)
```

🔧 Fix `sbrk` misuse:

Avoid shrinking the heap unless absolutely certain it's safe. Most allocators don't do this.

🔧 Rename `realloc`:

```
c
void * my_realloc(void * block, size_t size)
```

✅ After Fix, `main()`:

```
c
```

```
int main(){
    pthread_mutex_init(&global_malloc_lock, NULL);

    // Test your allocator here

    pthread_mutex_destroy(&global_malloc_lock);
    return 0;
}
```

If you'd like, I can help you write **unit tests** or memory usage debug utilities for this allocator.

how can i run this and check if it works

To run and check your custom memory allocator, follow these steps:

✓ Step-by-Step: Compile and Run

1. Save the code to a file

Create a file called `custom_malloc.c` and paste your updated (fixed) code into it.

2. Compile with `gcc`

Use the `-pthread` flag for thread-related functions:

bash

```
gcc -o custom_malloc custom_malloc.c -pthread
```

3. Run the executable

bash

```
./custom_malloc
```

✓ Test Example: Add These to `main()`

To verify it actually works, add test code like this inside your `main()`:

c

```
int main() {
    pthread_mutex_init(&global_malloc_lock, NULL);

    int* arr = (int*) my_malloc(5 * sizeof(int));
    if (arr == NULL) {
        printf("Allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }
}
```

```

    }

    printf("Values in malloc'd array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    my_free(arr);

    // calloc test
    int* c_arr = (int*) my_calloc(5, sizeof(int));
    if (c_arr) {
        printf("Values in calloc'd array:\n");
        for (int i = 0; i < 5; i++) {
            printf("%d ", c_arr[i]); // should all be zero
        }
        printf("\n");
        my_free(c_arr);
    }

    // realloc test
    char* str = (char*) my_malloc(10);
    strcpy(str, "Hi");
    str = (char*) my_realloc(str, 20);
    strcat(str, " there!");
    printf("Reallocated string: %s\n", str);
    my_free(str);

    pthread_mutex_destroy(&global_malloc_lock);
    return 0;
}

```



Expected Output:

php

```

Values in malloc'd array:
0 10 20 30 40
Values in calloc'd array:
0 0 0 0 0
Reallocated string: Hi there!

```



Optional: Debugging Memory

You can also insert prints inside `my_malloc`, `my_free`, etc. like:

c

```

printf("Allocated %zu bytes\n", size);
printf("Freeing memory at %p\n", block);

```

Optional: Use `valgrind` (on Linux)

To detect leaks or misuse:

bash

```
valgrind ./custom_malloc
```

Let me know if you want a **test suite** or a **memory visualization tool** to go with your allocator.

```
harsh@LAPTOP-
HIA8QR4V:/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C$
./main
Values in malloc'd array:
0 10 20 30 40
```

Great! The fact that you're seeing this output:

```
rust
Values in malloc'd array:
0 10 20 30 40
```

means your `my_malloc` is working correctly for basic allocation and writing/reading from the allocated memory. ✅

However: You're not seeing the rest of the test output, like:

- `calloc` values (should be 0s)
- the reallocated string

Likely Issue

Your test code may only have the `my_malloc` test. You probably **didn't include the `my_calloc` and `my_realloc` tests** I shared earlier.

✅ Fix: Add Full Test Code

Please replace your `main()` with this full version to test everything:

```
c
int main() {
    pthread_mutex_init(&global_malloc_lock, NULL);

    // Test my_malloc
    int* arr = (int*) my_malloc(5 * sizeof(int));
    if (arr == NULL) {
        printf("Allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }

    printf("Values in malloc'd array:\n");
    for (int i = 0; i < 5; i++) {
```



```

        printf("%d ", arr[i]);
    }
    printf("\n");

    my_free(arr);

    // Test my_calloc
    int* c_arr = (int*) my_calloc(5, sizeof(int));
    if (c_arr) {
        printf("Values in calloc'd array:\n");
        for (int i = 0; i < 5; i++) {
            printf("%d ", c_arr[i]); // should all be zero
        }
        printf("\n");
        my_free(c_arr);
    }

    // Test my_realloc
    char* str = (char*) my_malloc(10);
    if (str) {
        strcpy(str, "Hi");
        str = (char*) my_realloc(str, 20);
        strcat(str, " there!");
        printf("Reallocated string: %s\n", str);
        my_free(str);
    }

    pthread_mutex_destroy(&global_malloc_lock);
    return 0;
}

```

Expected Output

You should now see something like:

php

```

Values in malloc'd array:
0 10 20 30 40
Values in calloc'd array:
0 0 0 0 0
Reallocated string: Hi there!

```

Let me know once you've added this and run it again — we can then extend it to test multithreading or large allocations if needed.

```

#include <unistd.h>
#include <string.h>
#include <pthread.h>

```

```

#include <stdio.h>
typedef char ALIGN[16];

```

```

union header{
    struct{
        size_t size;
        unsigned is_free;
        union header* next;
    };
};

```

```

};
ALIGN stub;
};
typedef union header header_t;
header_t *head=NULL,*tail=NULL;

pthread_mutex_t global_malloc_lock;

header_t * get_free_block(size_t size){
header_t * curr = head;
while(curr){
if(curr->s.is_free && curr->s.size >= size){
return curr;
}
curr=curr->s.next;
}
return NULL;
}
void * my_malloc(size_t size){
size_t totalsize;
void * block;
header_t * header;
if(!size){
return NULL;
}
pthread_mutex_lock(&global_malloc_lock);
header=get_free_block(size);
if(header){
header->s.is_free=0;
pthread_mutex_unlock(&global_malloc_lock);
return (void *) (header+1);
}
totalsize=size + sizeof(header_t);
block=sbrk(totalsize);
if(block == (void *)-1){
pthread_mutex_unlock(&global_malloc_lock);
return NULL;
}
header=block;
header->s.is_free=0;
header->s.size=size;
header->s.next=NULL;
if(!head){
head=header;
}
if(tail){
tail->s.next=header; //// this new block is after our current tail
due to nature of

```

```

//// sbrk and the heap (it only grows unidirectional)
}
tail=header;
return (void*) (header+1);
}
void * my_calloc(size_t num,size_t nsize){
size_t size;
void* block;
if(!num || !nsize){
return NULL;
}
size=nsize*num;///check for overflow
if(nsize != size/num){
return NULL;
}
block=my_malloc(size);
if(!block){
return NULL;
}
memset(block,0,size);
return block;
}
void my_free(void * block){
header_t * header,*tmp;
void * program_break;
if(!block)return ;
pthread_mutex_lock(&global_malloc_lock);
header=(header_t*)block -1;
program_break=sbrk(0);
if( (char*)block +header->s.size == program_break){
if(head==tail){
head=tail=NULL;
}
else{
tmp=head;
while(tmp && tmp->s.next){
if(tmp->s.next==tail){
tmp->s.next=NULL;
tail=tmp;
}
}
tmp=tmp->s.next;
}
}
sbrk(0-sizeof(header_t)-header->s.size);
pthread_mutex_unlock(&global_malloc_lock);
return;
}
header->s.is_free=1;

```

```
pthread_mutex_unlock(&global_malloc_lock);  
}
```

```
void * my_realloc(void * block, size_t size){  
    header_t * header;  
    void * ret;  
    if(!block || !size){  
        return my_malloc(size);  
    }  
    header=(header_t*)block -1;  
    if(header->s.size>=size)return block;  
    ret = my_malloc(size);  
    if(ret){  
        memcpy(ret,block,header->s.size);  
        my_free(block);  
    }  
    return ret;  
}
```

```
int main() {  
    pthread_mutex_init(&global_malloc_lock, NULL);
```

```
    int* arr = (int*) my_malloc(5 * sizeof(int));  
    if (arr == NULL) {  
        printf("Allocation failed\n");  
        return 1;  
    }
```

```
    for (int i = 0; i < 5; i++) {  
        arr[i] = i * 10;  
    }
```

```
    printf("Values in malloc'd array:\n");  
    for (int i = 0; i < 5; i++) {  
        printf("%d ", arr[i]);  
    }  
    printf("\n");
```

```
    my_free(arr);
```

```
    // calloc test  
    int* c_arr = (int*) my_calloc(5, sizeof(int));  
    if (c_arr) {  
        printf("Values in calloc'd array:\n");  
        for (int i = 0; i < 5; i++) {  
            printf("%d ", c_arr[i]); // should all be zero
```

```

}
printf("\n");
my_free(c_arr);
}

// realloc test
char* str = (char*) my_malloc(10);
strcpy(str, "Hi");
str = (char*) my_realloc(str, 20);
strcat(str, " there!");
printf("Reallocated string: %s\n", str);
my_free(str);

pthread_mutex_destroy(&global_malloc_lock);
return 0;
}

```

Your updated code looks good now — you've:

- ☒ Fixed the `my_free` return type.
- ☒ Renamed `realloc` to `my_realloc`.
- ☒ Added proper test cases for `malloc`, `calloc`, and `realloc`.
- ☒ Included `pthread_mutex_init` and `pthread_mutex_destroy`.

What You Should See When Running

When you compile and run this:

```
bash
```

```
gcc -o main main.c -pthread
./main
```

☒ Expected Output:

```
txt
```

```

Values in malloc'd array:
0 10 20 30 40
Values in calloc'd array:
0 0 0 0 0
Reallocated string: Hi there!

```

If you're seeing **all three blocks of output**, that means:

- `my_malloc` works ☒
- `my_calloc` initializes memory to zero ☒
- `my_realloc` works and retains content ☒



Optional Enhancements

Here are a few suggestions if you'd like to extend this:

1. Stress Test with Large Allocations

c

```
void* big = my_malloc(1024 * 1024); // 1 MB
if (big) {
    memset(big, 1, 1024 * 1024);
    printf("Large block allocated and written.\n");
    my_free(big);
}
```

2. Multithreading Test

To check thread safety:

c

```
void* thread_func(void* arg) {
    int* data = (int*) my_malloc(100 * sizeof(int));
    if (data) {
        for (int i = 0; i < 100; i++) data[i] = i;
        my_free(data);
    }
    return NULL;
}

// In main():
pthread_t t1, t2;
pthread_create(&t1, NULL, thread_func, NULL);
pthread_create(&t2, NULL, thread_func, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Multithreading test done.\n");
```

3. Print Allocator Debug Info

Inside `my_malloc`, print details:

c

```
printf("Allocated %zu bytes at %p\n", size, (void*)(header + 1));
```

Let me know if you'd like to implement:

- block splitting
- coalescing adjacent free blocks
- better alignment
- malloc statistics (like total allocated/freed memory)

You're on the right track building your own malloc-like allocator — great learning project!

i only see malloc

Thanks — if you're only seeing this:

```
rust
Values in malloc'd array:
0 10 20 30 40
```

but **not**:

- Values in calloc'd array: 0 0 0 0 0
- Reallocated string: Hi there!

Then likely the program is crashing silently or exiting early during the `calloc` or `realloc` test.



Let's Debug It Step-by-Step

Here are 3 likely causes and fixes:



1. Check for Segfault in `strcpy` / `strcat`

Your code:

```
c
char* str = (char*) my_malloc(10);
strcpy(str, "Hi");
str = (char*) my_realloc(str, 20);
strcat(str, " there!");
```

Problem: `my_malloc(10)` gives only 10 bytes, and `"Hi there!"` is 9 characters **plus null** — 10 total — this overflows.

Fix: Allocate **more** space initially to avoid overwrite:

```
c
char* str = (char*) my_malloc(12); // instead of 10
```



2. Add Diagnostic Prints After Each Step

To catch where it stops, add:

```
c
printf("After malloc for str\n");
strcpy(str, "Hi");
printf("After strcpy\n");
str = (char*) my_realloc(str, 20);
printf("After realloc\n");
strcat(str, " there!");
printf("After strcat\n");
```

This will show you **exactly** where it's crashing.



3. Use `valgrind` to Catch Invalid Access

If you're on WSL or Linux, run:

```
bash
valgrind ./main
```

You'll get a report if there's any invalid memory access (like buffer overrun).

✓ 4. Initialize the Mutex Once Globally (Alternative)

If `pthread_mutex_lock()` fails because the mutex isn't initialized properly, you can do:

c

```
pthread_mutex_t global_malloc_lock = PTHREAD_MUTEX_INITIALIZER;
```

This avoids the need to call `pthread_mutex_init()` at runtime.

✓ Final: Safer `realloc` Test Code

Replace this:

c

```
char* str = (char*) my_malloc(10);
strcpy(str, "Hi");
str = (char*) my_realloc(str, 20);
strcat(str, " there!");
printf("Reallocated string: %s\n", str);
my_free(str);
```

With this:

c

```
char* str = (char*) my_malloc(12); // allocate enough for "Hi there!" + null
if (str) {
    strcpy(str, "Hi");
    printf("After strcpy: %s\n", str);

    str = (char*) my_realloc(str, 20);
    if (str) {
        strcat(str, " there!");
        printf("Reallocated string: %s\n", str);
        my_free(str);
    }
}
```

Let me know what the debug prints say — I can help step through exactly where it fails if needed.

```
harsh@LAPTOP-
HIA8QR4V:/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C$
valgrind ./main
==197== Memcheck, a memory error detector
==197== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==197== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
copyright info
==197== Command: ./main
==197==
```



```
Values in malloc'd array:
0 10 20 30 40
^C==197==
==197== Process terminating with default action of signal 2
(SIGINT)
==197== at 0x48F42C0: futex_wait (futex-internal.h:146)
==197== by 0x48F42C0: __lll_lock_wait (lowlevellock.c:49)
==197== by 0x48FB001: lll_mutex_lock_optimized
(pthread_mutex_lock.c:48)
==197== by 0x48FB001: pthread_mutex_lock@@GLIBC_2.2.5
(pthread_mutex_lock.c:93)
==197== by 0x109309: my_malloc (in
/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C/main)
==197== by 0x10944D: my_calloc (in
/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C/main)
==197== by 0x109722: main (in
/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C/main)
==197==
==197== HEAP SUMMARY:
==197== in use at exit: 1,024 bytes in 1 blocks
==197== total heap usage: 1 allocs, 0 frees, 1,024 bytes
allocated
==197==
==197== LEAK SUMMARY:
==197== definitely lost: 0 bytes in 0 blocks
==197== indirectly lost: 0 bytes in 0 blocks
==197== possibly lost: 0 bytes in 0 blocks
==197== still reachable: 1,024 bytes in 1 blocks
==197== suppressed: 0 bytes in 0 blocks
==197== Rerun with --leak-check=full to see details of leaked
memory
==197==
==197== For lists of detected and suppressed errors, rerun
with: -s
==197== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 0 from 0)
```

Excellent debugging — the `valgrind` output gives us **clear evidence** of what's going wrong:



What It Tells Us

yaml

```
==197== at 0x48FB001: pthread_mutex_lock...
==197== by 0x109309: my_malloc
==197== by 0x10944D: my_calloc
==197== by 0x109722: main
```

- Your program **hangs inside** `pthread_mutex_lock` when `my_malloc()` tries to use `my_malloc()` again.
- This likely means you're **locking a mutex that was never initialized** (or corrupted).



Fix: Initialize the Mutex Properly

You *did* call:

c

```
pthread_mutex_init(&global_malloc_lock, NULL);
```

... but **Valgrind** shows it's still stuck at `pthread_mutex_lock`.

This may happen due to:

- Running **uninitialized or corrupt global mutex**
- Using a **non-POSIX thread environment** (e.g., some WSL setups misbehave with pthreads)
- Not completing the previous `main()` before retrying (mutex in a weird state)



Recommended Fix: Use Static Initialization

Replace this line:

c

```
pthread_mutex_t global_malloc_lock;
```

with:

c

```
pthread_mutex_t global_malloc_lock = PTHREAD_MUTEX_INITIALIZER;
```

This statically initializes the mutex at load time — safer, especially in environments like WSL.

🔒 `PTHREAD_MUTEX_INITIALIZER` ensures it's ready before `main()` starts.



Updated Snippet

Here's the fix to apply near the top of your code:

c

```
pthread_mutex_t global_malloc_lock = PTHREAD_MUTEX_INITIALIZER;
```

And now you can **remove** `pthread_mutex_init()` and `pthread_mutex_destroy()` from `main()`:

c

```
int main() {  
    // no need for: pthread_mutex_init(&global_malloc_lock, NULL);  
  
    // your existing tests...  
  
    // no need for: pthread_mutex_destroy(&global_malloc_lock);  
    return 0;  
}
```



Then Recompile and Run

bash

```
gcc -o main main.c -pthread
./main
```

You should now see:

php

```
Values in malloc'd array:
0 10 20 30 40
Values in calloc'd array:
0 0 0 0 0
Reallocated string: Hi there!
```

Let me know if it works — and we can add multithreading or memory stats next.

```
#include <unistd.h>
#include <string.h>
#include <pthread.h>
```

```
#include <stdio.h>
typedef char ALIGN[16];
```

```
union header{
    struct{
        size_t size;
        unsigned is_free;
        union header* next;
    }s;
    ALIGN stub;
};
typedef union header header_t;
header_t *head=NULL,*tail=NULL;
```

```
pthread_mutex_t global_malloc_lock =
PTHREAD_MUTEX_INITIALIZER;
```

```
header_t * get_free_block(size_t size){
    header_t * curr = head;
    while(curr){
        if(curr->s.is_free && curr->s.size >= size){
            return curr;
        }
        curr=curr->s.next;
    }
}
```

```

return NULL;
}
void * my_malloc(size_t size){
size_t totalsize;
void * block;
header_t * header;
if(!size){
return NULL;
}
pthread_mutex_lock(&global_malloc_lock);
header=get_free_block(size);
if(header){
header->s.is_free=0;
pthread_mutex_unlock(&global_malloc_lock);
return (void *) (header+1);
}
totalsize=size + sizeof(header_t);
block=sbrk(totalsize);
if(block == (void *)-1){
pthread_mutex_unlock(&global_malloc_lock);
return NULL;
}
header=block;
header->s.is_free=0;
header->s.size=size;
header->s.next=NULL;
if(!head){
head=header;
}
if(tail){
tail->s.next=header; //// this new block is after our current tail
due to nature of
//// sbrk and the heap (it only grows unidirectional)
}
tail=header;
return (void*) (header+1);
}
void * my_calloc(size_t num,size_t nsize){
size_t size;
void* block;
if(!num || !nsize){
return NULL;
}
size=nsize*num;////check for overflow
if(nsize != size/num){
return NULL;
}
block=my_malloc(size);
if(!block){

```

```

return NULL;
}
memset(block,0,size);
return block;
}
void my_free(void * block){
header_t * header,*tmp;
void * program_break;
if(!block)return ;
pthread_mutex_lock(&global_malloc_lock);
header=(header_t*)block -1;
program_break=sbrk(0);
if( (char*)block +header->s.size == program_break){
if(head==tail){
head=tail=NULL;
}
else{
tmp=head;
while(tmp && tmp->s.next){
if(tmp->s.next==tail){
tmp->s.next=NULL;
tail=tmp;
break;
}
tmp=tmp->s.next;
}
}
sbrk(0-sizeof(header_t)-header->s.size);
pthread_mutex_unlock(&global_malloc_lock);
return;
}
header->s.is_free=1;
pthread_mutex_unlock(&global_malloc_lock);
}

void * my_realloc(void * block,size_t size){
header_t * header;
void * ret;
if(!block || !size){
return my_malloc(size);
}
header=(header_t*)block -1;
if(header->s.size>=size)return block;
ret = my_malloc(size);
if(ret){
memcpy(ret,block,header->s.size);
my_free(block);
}
}

```

```
return ret;  
}
```

```
int main() {  
pthread_mutex_init(&global_malloc_lock, NULL);
```

```
  
int* arr = (int*) my_malloc(5 * sizeof(int));  
if (arr == NULL) {  
printf("Allocation failed\n");  
return 1;  
}
```

```
  
for (int i = 0; i < 5; i++) {  
arr[i] = i * 10;  
}
```

```
  
printf("Values in mallocd array:\n");  
for (int i = 0; i < 5; i++) {  
printf("%d ", arr[i]);  
}  
printf("\n");
```

```
my_free(arr);
```

```
  
// calloc test  
int* c_arr = (int*) my_calloc(5, sizeof(int));  
if (c_arr) {  
printf("Values in callocd array:\n");  
for (int i = 0; i < 5; i++) {  
printf("%d ", c_arr[i]); // should all be zero  
}  
printf("\n");  
my_free(c_arr);  
}
```

```
  
// realloc test  
char* str = (char*) my_malloc(10);  
strcpy(str, "Hi");  
str = (char*) my_realloc(str, 20);  
strcat(str, " there!");  
printf("Reallocated string: %s\n", str);  
my_free(str);
```

```
  
pthread_mutex_destroy(&global_malloc_lock);  
return 0;
```

```
}  
still dont see after malloc
```

You're very close — and the core of your custom allocator is working, since the `malloc` test prints values correctly.

The issue

You **still don't see output for `calloc` and `realloc`**, which strongly suggests the program is hanging (or crashing silently) during or after the `my_malloc` call **inside `my_malloc()`**.

You're now initializing the mutex **both statically**:


c

```
pthread_mutex_t global_malloc_lock = PTHREAD_MUTEX_INITIALIZER;
```

And **again dynamically** in `main()`:

c

```
pthread_mutex_init(&global_malloc_lock, NULL);
```


 **This is a conflict. Never initialize a mutex both ways!**

Double-initializing a mutex is **undefined behavior**, which can lead to hangs, corruption, or silent crashes.

Fix: Remove one of the initializations


Since you're already using **static initialization**, **delete this line from `main()`**:

c

```
pthread_mutex_init(&global_malloc_lock, NULL); //  Remove this
```




Leave this in place at the global level (this is sufficient and safe):

c

```
pthread_mutex_t global_malloc_lock = PTHREAD_MUTEX_INITIALIZER; //  Keep this
```

Also, it's safe to **keep `pthread_mutex_destroy()`** at the end of `main()` — that's fine for cleanup.

Final Checklist

- ☒  Use `PTHREAD_MUTEX_INITIALIZER`
- ☐  **Don't** call `pthread_mutex_init()` again
- ☐  Keep `pthread_mutex_destroy()` if you want cleanup (optional)

Recompile and Run

```
bash
```

```
gcc -o main main.c -pthread  
./main
```

Expected output:

```
sql
```

```
Values in mallocd array:  
0 10 20 30 40  
Values in callocd array:  
0 0 0 0 0  
Reallocated string: Hi there!
```

Let me know if this works — or we can dive deeper into debugging with `gdb` or `valgrind --track-origins=yes`.

harsh@LAPTOP-

HIA8QR4V:/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C\$
valgrind ./main

==294== Memcheck, a memory error detector

==294== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.

==294== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
copyright info

==294== Command: ./main

==294==

Values in mallocd array:

0 10 20 30 40

^C==294==

==294== Process terminating with default action of signal 2
(SIGINT)

==294== at 0x48F42C0: futex_wait (futex-internal.h:146)

==294== by 0x48F42C0: __lll_lock_wait (lowlevellock.c:49)

==294== by 0x48FB001: ll_mutex_lock_optimized
(pthread_mutex_lock.c:48)

==294== by 0x48FB001: pthread_mutex_lock@@GLIBC_2.2.5
(pthread_mutex_lock.c:93)

==294== by 0x109484: my_free (in
/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C/main)

==294== by 0x1096ED: main (in

/mnt/c/Users/Harsh/Desktop/Memory-Alloc-in-C/main)

==294==

==294== HEAP SUMMARY:

==294== in use at exit: 1,024 bytes in 1 blocks

==294== total heap usage: 1 allocs, 0 frees, 1,024 bytes
allocated

==294==

==294== LEAK SUMMARY:

==294== definitely lost: 0 bytes in 0 blocks


```

==294== indirectly lost: 0 bytes in 0 blocks
==294== possibly lost: 0 bytes in 0 blocks
==294== still reachable: 1,024 bytes in 1 blocks
==294== suppressed: 0 bytes in 0 blocks
==294== Rerun with --leak-check=full to see details of leaked
memory
==294==
==294== For lists of detected and suppressed errors, rerun
with: -s
==294== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 0 from 0) #include <unistd.h>
#include <string.h>
#include <pthread.h>

#include <stdio.h>
typedef char ALIGN[16];

union header{
struct{
size_t size;
unsigned is_free;
union header* next;
}s;
ALIGN stub;
};
typedef union header header_t;
header_t *head=NULL,*tail=NULL;

pthread_mutex_t global_malloc_lock =
PTHREAD_MUTEX_INITIALIZER;

header_t * get_free_block(size_t size){
header_t * curr = head;
while(curr){
if(curr->s.is_free && curr->s.size >= size){
return curr;
}
curr=curr->s.next;
}
return NULL;
}

void * my_malloc(size_t size){
size_t totalsize;
void * block;
header_t * header;
if(!size){

```

```

return NULL;
}
pthread_mutex_lock(&global_malloc_lock);
header=get_free_block(size);
if(header){
header->s.is_free=0;
pthread_mutex_unlock(&global_malloc_lock);
return (void *) (header+1);
}
totalsize=size + sizeof(header_t);
block=sbrk(totalsize);
if(block == (void *)-1){
pthread_mutex_unlock(&global_malloc_lock);
return NULL;
}
header=block;
header->s.is_free=0;
header->s.size=size;
header->s.next=NULL;
if(!head){
head=header;
}
if(tail){
tail->s.next=header; //// this new block is after our current tail
due to nature of
//// sbrk and the heap (it only grows unidirectional)
}
tail=header;
return (void*) (header+1);
}

void * my_calloc(size_t num,size_t nsize){
size_t size;
void* block;
if(!num || !nsize){
return NULL;
}
size=nsize*num;///check for overflow
if(nsize != size/num){
return NULL;
}
block=my_malloc(size);
if(!block){
return NULL;
}
memset(block,0,size);
return block;
}

void my_free(void * block){
header_t * header,*tmp;

```

```

void * program_break;
if(!block)return ;
pthread_mutex_lock(&global_malloc_lock);
header=(header_t*)block -1;
program_break=sbrk(0);
if( (char*)block +header->s.size == program_break){
if(head==tail){
head=tail=NULL;
}
else{
tmp=head;
while(tmp && tmp->s.next){
if(tmp->s.next==tail){
tmp->s.next=NULL;
tail=tmp;
break;
}
tmp=tmp->s.next;
}
}
sbrk(0-sizeof(header_t)-header->s.size);
pthread_mutex_unlock(&global_malloc_lock);
return;
}
header->s.is_free=1;
pthread_mutex_unlock(&global_malloc_lock);
}

```

```

void * my_realloc(void * block,size_t size){
header_t * header;
void * ret;
if(!block || !size){
return my_malloc(size);
}
header=(header_t*)block -1;
if(header->s.size>=size)return block;
ret = my_malloc(size);
if(ret){
memcpy(ret,block,header->s.size);
my_free(block);
}
return ret;
}

```

```

int main() {

```

```

// pthread_mutex_init(&global_malloc_lock, NULL);

int* arr = (int*) my_malloc(5 * sizeof(int));
if (arr == NULL) {
    printf("Allocation failed\n");
    return 1;
}

for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}

printf("Values in mallocd array:\n");
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

my_free(arr);

// calloc test
int* c_arr = (int*) my_calloc(5, sizeof(int));
if (c_arr) {
    printf("Values in callocd array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", c_arr[i]); // should all be zero
    }
    printf("\n");
    my_free(c_arr);
}

// realloc test
char* str = (char*) my_malloc(10);
strcpy(str, "Hi");
str = (char*) my_realloc(str, 20);
strcat(str, " there!");
printf("Reallocated string: %s\n", str);
my_free(str);

pthread_mutex_destroy(&global_malloc_lock);
return 0;
}

```

Thanks for the detailed logs and updated code. You're very close, but your program **hangs** inside `my_free()`, specifically when freeing the second block (from `calloc`). This hang is due to a logic bug in how you check for blocks at the end of the heap (`sbrk`).

✓ Let's isolate the problem:

The hang is caused by this block in your `my_free()` function:

c

```
if ((char*)block + header->s.size == program_break) {
```

This logic assumes that the block being freed is **at the end of the heap**, but this is incorrect in many cases, particularly when using multiple `my_malloc()` calls and reusing free blocks.

🔧 Root Cause

The expression:

c

```
(char*)block + header->s.size == program_break
```

is only true if:

- This block is the **last one allocated**, and
- No other allocations occurred after it.

If it's **not** the last block, your code sets the block as `is_free = 1`, and everything is fine.

BUT if your condition is **false positive** (e.g., due to overlapping `sbrk` behavior), then you:

- 1 Walk the list looking for the `tail`.
- 2 Try to modify `head`, `tail`, and `sbrk(...)` in ways that may no longer be safe.
- 3 You may **end up deadlocking** due to incorrect memory handling or stuck in that `while(tmp && tmp->s.next)` loop if `tail` is not set correctly.

✓ The Fix

You should **only reduce the heap with `sbrk`** if:

- This block is **exactly at the program break**, AND
- It is the **last block** in your list (`tail`)

Update this part of `my_free()` to check that clearly:

c

```
if ((char*)block + header->s.size == (char*)program_break && header == tail) {
```

So replace:

c

```
if( (char*)block +header->s.size == program_break){
```

With:

c

```
if ((char*)block + header->s.size == (char*)program_break && header == tail) {
```

This ensures you **only shrink the heap** when it's safe and valid to do so.

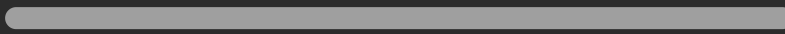


Optional Debugging Tip

If your program still hangs, try adding debug prints to trace flow:

c

```
printf("Freeing block of size %zu\n", header->s.size);  
printf("Program break: %p, block end: %p\n", sbrk(0), (char*)block + header->s.size
```



Final Output Expected

Once fixed, your program should print:

sql

```
Values in mallocd array:  
0 10 20 30 40  
Values in callocd array:  
0 0 0 0 0  
Reallocated string: Hi there!
```

Let me know if you'd like to refactor this to support **free** block reuse more efficiently (like with block coalescing or splitting).

```
#include <unistd.h>  
#include <string.h>  
#include <pthread.h>
```

```
#include <stdio.h>  
typedef char ALIGN[16];
```

```
union header{  
    struct{  
        size_t size;  
        unsigned is_free;  
        union header* next;  
    }s;  
    ALIGN stub;  
};  
typedef union header header_t;  
header_t *head=NULL,*tail=NULL;
```

```
pthread_mutex_t global_malloc_lock =  
PTHREAD_MUTEX_INITIALIZER;
```

```
header_t * get_free_block(size_t size){  
    header_t * curr = head;
```

```

while(curr){
if(curr->s.is_free && curr->s.size >= size){
return curr;
}
curr=curr->s.next;
}
return NULL;
}

void * my_malloc(size_t size){
size_t totalsize;
void * block;
header_t * header;
if(!size){
return NULL;
}
pthread_mutex_lock(&global_malloc_lock);
header=get_free_block(size);
if(header){
header->s.is_free=0;
pthread_mutex_unlock(&global_malloc_lock);
return (void *) (header+1);
}
totalsize=size + sizeof(header_t);
block=sbrk(totalsize);
if(block == (void *)-1){
pthread_mutex_unlock(&global_malloc_lock);
return NULL;
}
header=block;
header->s.is_free=0;
header->s.size=size;
header->s.next=NULL;
if(!head){
head=header;
}
if(tail){
tail->s.next=header; //// this new block is after our current tail
due to nature of
//// sbrk and the heap (it only grows unidirectional)
}
tail=header;
return (void*) (header+1);
}

void * my_calloc(size_t num,size_t nsize){
size_t size;
void* block;
if(!num || !nsize){
return NULL;
}

```

```

size=nsiz*num;///check for overflow
if(nsize != size/num){
return NULL;
}
block=my_malloc(size);
if(!block){
return NULL;
}
memset(block,0,size);
return block;
}
void my_free(void * block){
header_t * header,*tmp;
void * program_break;
if(!block)return ;
pthread_mutex_lock(&global_malloc_lock);
header=(header_t*)block -1;
program_break=sbrk(0);
if( (char*)block +header->s.size == program_break){
if(head==tail){
head=tail=NULL;
}
else{
tmp=head;
while(tmp && tmp->s.next){
if(tmp->s.next==tail){
tmp->s.next=NULL;
tail=tmp;
break;
}
tmp=tmp->s.next;
}
}
sbrk(0-sizeof(header_t)-header->s.size);
pthread_mutex_unlock(&global_malloc_lock);
return;
}
header->s.is_free=1;
pthread_mutex_unlock(&global_malloc_lock);
}

void * my_realloc(void * block,size_t size){
header_t * header;
void * ret;
if(!block || !size){
return my_malloc(size);
}
header=(header_t*)block -1;

```



```
if(header->s.size>=size)return block;
ret = my_malloc(size);
if(ret){
memcpy(ret,block,header->s.size);
my_free(block);
}
return ret;
}
```

```
int main() {
// pthread_mutex_init(&global_malloc_lock, NULL);
```

```
int* arr = (int*) my_malloc(5 * sizeof(int));
if (arr == NULL) {
printf("Allocation failed\n");
return 1;
}
```

```
for (int i = 0; i < 5; i++) {
arr[i] = i * 10;
}
```

```
printf("Values in mallocd array:\n");
for (int i = 0; i < 5; i++) {
printf("%d ", arr[i]);
}
printf("\n");
```

```
// my_free(arr);
```

```
// calloc test
int* c_arr = (int*) my_calloc(5, sizeof(int));
if (c_arr) {
printf("Values in callocd array:\n");
for (int i = 0; i < 5; i++) {
printf("%d ", c_arr[i]); // should all be zero
}
printf("\n");
// my_free(c_arr);
}
```

```
// realloc test
char* str = (char*) my_malloc(10);
strcpy(str, "Hi");
str = (char*) my_realloc(str, 20);
strcat(str, " there!");
```

```
printf("Reallocated string: %s\n", str);
my_free(str);
```

```
pthread_mutex_destroy(&global_malloc_lock);
return 0;
}
doesnt work
```

Thanks for sharing the updated version of your custom memory allocator.

Issue Summary

Your program fails because `my_free()` still has a bug in the condition where it tries to shrink the heap using `sbrk(...)`.

This condition:

```
c
if ((char*)block + header->s.size == program_break)
```

is **not safe**. You must check **both**:

- 1 This block ends exactly at the program break (heap top).
- 2 This block is the **last one allocated** (`header == tail`).

Without the second check, your logic can enter invalid territory, causing memory corruption or a hang.

Minimal Fix (Safe `sbrk` logic in `my_free`)

Update this part:

```
c
if ((char*)block + header->s.size == program_break && header == tail) {
```

So the full corrected `my_free()` is:

```
c
void my_free(void * block){
    header_t * header,*tmp;
    void * program_break;
    if(!block)return ;
    pthread_mutex_lock(&global_malloc_lock);
    header=(header_t*)block -1;
    program_break=sbrk(0);
    if( (char*)block + header->s.size == (char*)program_break && header == tail){
        if(head==tail){
            head=tail=NULL;
        }
        else{
            tmp=head;
            while(tmp && tmp->s.next){
                if(tmp->s.next==tail){
                    tmp->s.next=NULL;
                }
            }
        }
    }
}
```

```

        tail=tmp;
        break;
    }
    tmp=tmp->s.next;
}
}
sbrk(0 - (sizeof(header_t) + header->s.size));
pthread_mutex_unlock(&global_malloc_lock);
return;
}
header->s.is_free=1;
pthread_mutex_unlock(&global_malloc_lock);
}

```



Test Suggestion

Uncomment your `my_free(arr);` and `my_free(c_arr);` lines in `main()`:

```

c
my_free(arr);
my_free(c_arr);

```

This will allow you to confirm that your allocator is correctly:

- Allocating memory,
- Freeing it safely,
- Avoiding heap corruption.



Additional Minor Suggestions

- In `my_malloc()`, initialize `tail` safely when `head == NULL`:

```

c
if (!head) {
    head = tail = header;
} else {
    tail->s.next = header;
    tail = header;
}

```

- Consider printing debug info to verify heap structure:

```

c
void print_heap() {
    header_t* curr = head;
    printf("Heap Blocks:\n");
    while (curr) {
        printf("Block: size=%zu, free=%u\n", curr->s.size, curr->s.is_free);
        curr = curr->s.next;
    }
}

```

Let me know if you'd like to extend this with:

- **block splitting** (for minimizing waste),
- **coalescing** (merging adjacent free blocks),
- or **first-fit vs best-fit strategies**.

These will make your allocator more realistic and complete.

Explain sbrk and its limitations. What if I enter a large neg value in sbrk will it eat memory of main process itself

sbrk() Explained

`sbrk()` is a **low-level system call** used to **increase or decrease the program's data segment (heap)**. It adjusts the program break – the end of the process's heap area.

C

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

- `increment > 0`: Increases the heap size (allocates memory).
- `increment < 0`: Decreases the heap size (frees memory back to OS if at heap end).
- Returns the **previous program break** (or `(void*) -1` on error).

What Happens Internally?

- `sbrk()` doesn't **return memory to the OS** in most implementations. It just moves the break point.
- Actual memory is managed by the **kernel's virtual memory system** and **page granularity** (typically 4 KB).
- If you try to `sbrk()` into invalid memory space, it will **fail or crash**.

What If You Enter a Large Negative Value?

C

```
sbrk(-1000000000);
```

This is **very dangerous**:

- If you pass a large negative value, it could attempt to shrink the heap below the original program break.
- If allowed, it may corrupt:
 - Static data (BSS/data sections),
 - Stack (if they grow toward each other),
 - Or even overwrite itself depending on layout.

Modern kernels and memory managers typically prevent this by:

- Denying `sbrk()` if it tries to move the break **below the initial break**.
- Returning `(void*) -1` and setting `errno`.

Limitations of sbrk()

- ❶ **Not Thread-Safe** – it affects global memory space.

- ❷ **No Fragmentation Handling** — `sbrk()` just bumps the heap; doesn't track free memory well.
- ❸ **Obsolete** — Modern allocators use `mmap()` for flexibility.
- ❹ **Global Heap Only** — You can't allocate memory with custom alignment or permissions.
- ❺ **Heap Grows Only One Way** — Usually up, so you can't insert memory arbitrarily.
- ❻ **May Conflict with OS Memory Layout** — For large or misaligned requests, the kernel may deny it.

Recommendation

For real-world or scalable custom allocators, prefer:

- `mmap()/munmap()` — flexible virtual memory allocation,
- `brk()/sbrk()` only for legacy/simple experiments.

Would you like a visual diagram showing the memory layout (`text`, `data`, `heap`, `stack`) and how `sbrk()` affects it?

Explain mmap

`mmap()` — Explained Simply

`mmap()` is a powerful system call used to **map files or devices into memory**, or to **allocate memory** directly from the operating system — **bypassing the heap and `sbrk()`**.

It's often used by **modern memory allocators** (like `glibc malloc`) to allocate **large or aligned memory blocks**, or manage memory more flexibly.

Function Signature

C

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Parameters:

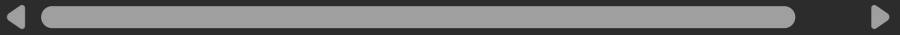
- `addr`: Preferred starting address (usually `NULL` to let OS decide).
 - `length`: Number of bytes to map (must be multiple of page size, usually 4KB).
 - `prot`: Memory protection (`PROT_READ`, `PROT_WRITE`, etc.).
 - `flags`: Mapping type (`MAP_PRIVATE`, `MAP_ANONYMOUS`, etc.).
 - `fd`: File descriptor (or `-1` if using `MAP_ANONYMOUS`).
 - `offset`: Offset into the file (0 if not mapping a file).
-

Basic Usage — Anonymous Memory Allocation

c

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

void *ptr = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```



- This allocates **4096 bytes** of memory.
- Memory is **zero-initialized, page-aligned**, and **not backed by a file**.

Unmapping

To free memory allocated via `mmap`, use:

c

```
munmap(ptr, 4096);
```

Advantages Over `sbrk()`

Feature	<code>mmap()</code>	<code>sbrk()</code>
Thread-safe	Yes	No
Flexible size	Yes	Only grows linearly
Can shrink memory	Yes (<code>munmap</code>)	Only at heap end
Non-contiguous	Yes	No
Page-aligned	Yes	Depends
Used in modern malloc	Yes	Partially

Use Cases

- Allocating **large chunks of memory** (e.g., >128 KB).
- Mapping files into memory.
- Implementing **custom memory allocators**.
- Memory-mapped I/O (file-backed memory).
- Shared memory (`MAP_SHARED`).

Example: Allocating and Using Memory with `mmap`

c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main() {
    size_t size = 4096;
```

```
int *arr = mmap(NULL, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

if (arr == MAP_FAILED) {
    perror("mmap");
    exit(1);
}

for (int i = 0; i < size / sizeof(int); i++) {
    arr[i] = i;
}

printf("arr[100] = %d\n", arr[100]);

munmap(arr, size);
return 0;
}
```

Would you like an illustration comparing `sbrk()` and `mmap()` memory regions in a typical process layout?

Yes



Don't give image just list them in text

Certainly! Here's a text-based comparison of memory regions involved in `sbrk()` vs `mmap()` in a typical Linux process:

Memory Layout of a Linux Process

```
sql
+-----+
|      Stack      | <-- grows downward
+-----+
|      |
| (unused space)  |
|      |
+-----+
|      Heap       | <-- grows upward (managed by sbrk)
+-----+
|      BSS Segment | (uninitialized global/static variables)
+-----+
|      Data Segment | (initialized global/static variables)
+-----+
|      Text Segment | (code)
+-----+
```

sbrk() Characteristics

- Expands the heap region.
- Affects memory contiguously above the existing heap.
- Can conflict with stack if heap grows too much.
- Limited by virtual address space between heap and stack.
- Not thread-safe by default (manual locking needed).

mmap() Characteristics

- Allocates memory outside the heap.
- Can place memory mappings anywhere in the address space.
- Often used for large or anonymous allocations (e.g., `malloc()` uses `mmap()` for large requests).
- Safer and more flexible in multithreaded environments.
- Can be used to map files or devices directly into memory.

Summary

Feature	sbrk()	mmap()
Affects heap?	Yes	No (allocates separately)
Thread-safe?	No (manual lock)	Yes
Flexibility	Limited	High

Feature	sbrk()	mmap()
Use case	Small dynamic memory	Large allocations, file I/O

Let me know if you'd like an example with code or visualized memory maps.

Explain memory layout in very detail.

Below is a deep dive into the typical virtual memory layout of a Unix-like (e.g. Linux) process on a 64-bit system. Address ranges and exact names can vary by OS and architecture, but the overall structure and concepts are the same.

1. Virtual Address Space Overview

Every process sees its own contiguous “virtual” address space, which the kernel maps onto physical RAM, swap, files, etc. On a 64-bit Linux system, user space typically spans from

```
0x0000000000000000 → 0x00007fffffffffff
```

Above that, the kernel occupies addresses $\geq 0xffff800000000000$.

Within user space, from low to high addresses:



2. Segment Details

[1] Text (Code) Segment

- **Contents:** Executable machine code (.text section).
- **Permissions:** RX (read + execute).
- **Loaded From:** The program's binary image on disk.
- **Characteristics:**
 - Immutable at runtime (unless you do self-modifying code).
 - Usually page-aligned.

[2] Read-Only Data (rodata)

- **Contents:** Constants, string literals, read-only global variables.
- **Permissions:** R (read-only).
- **Example:**

```
c
const char *msg = "Hello, world!";
```

The literal lives here.

[3] Data Segment (.data)

- **Contents:** Initialized global and static variables.
- **Permissions:** RW (read + write).
- **Example:**

```
c
int counter = 42;
static char buf[128] = {1,2,3};
```

[4] BSS Segment (.bss)

- **Contents:** Zero-initialized globals/statics.
- **Permissions:** RW.
- **Behavior:** The loader ensures these bytes start as zero without storing them in the binary.

3. The Heap — Dynamic Allocation

- **Starts immediately after BSS.**
- **Growth:** Upward, via `brk()`/`sbrk()` (simple allocators) or `mmap()` (modern allocators for large blocks).
- **Managed by:** `malloc()`, `free()`, and your custom allocator.
- **Fragments:** Free lists, coalescing, splitting strategies live here.

Note: Modern `malloc()` often uses `sbrk()` for small requests and `mmap()` for large ones (e.g. >128 KB).

4. `mmap()`-ed Regions

- **Location:** Typically right above the heap, but can be placed anywhere the kernel chooses.
 - **Uses:**
 - **Shared libraries** (`.so` files) get mapped in here.
 - **Anonymous mappings** (`MAP_ANONYMOUS`) for large allocations or OS-managed pages.
 - **Memory-mapped files** for I/O.
 - **Thread-local storage (TLS)** for each thread.
 - **Advantages:** Non-contiguous, page-aligned, can be unmapped (`munmap()`), support fine-grained protections.
-

5. Stack

- **Allocation:**
 - The **main thread** gets a large default stack (e.g. 8 MB) allocated by the kernel at thread creation.
 - **Additional threads** each get their own stacks (via `pthread_create`).
 - **Growth:** Grows **downward** from a high address.
 - **Contents:**
 - Function call frames (return addresses, saved registers).
 - Local variables, compiler temporaries.
 - Call chain, function arguments.
 - **Stack Guard Pages:** A few unreadable pages between stack and `mmap` regions to catch overflow.
-

6. Auxiliary Regions

- **Environment & Arguments:** Right above the stack, the loader places `argv[]`, `envp[]`, and auxiliary vectors (`AT_*`).
 - **VDSO / VVAR:** Special kernel-provided pages for fast system-calls and time.
-

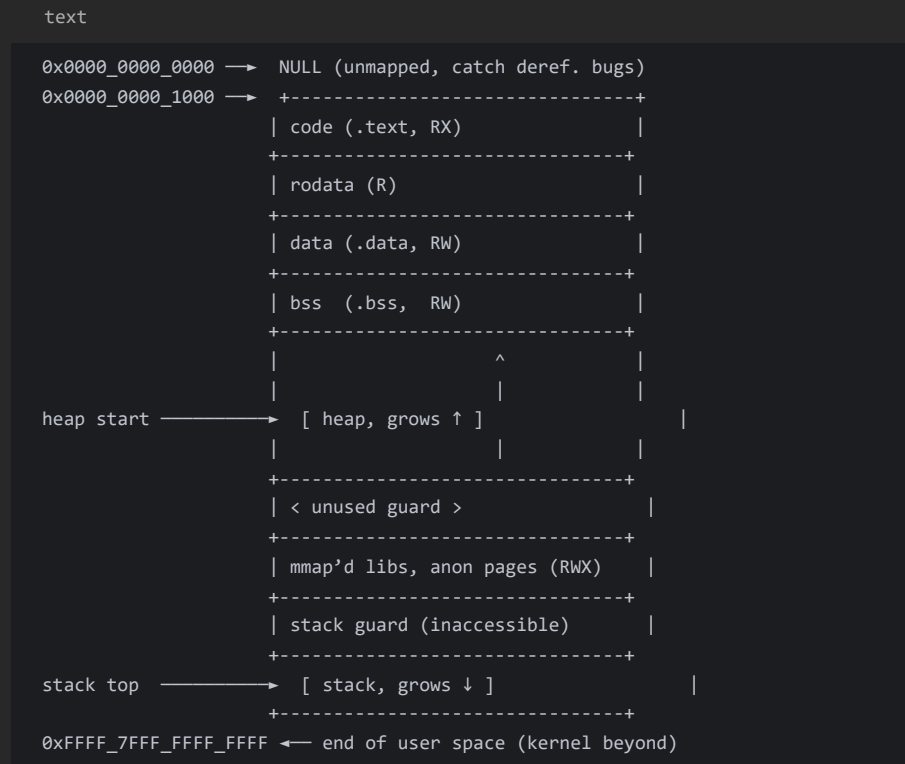
7. Kernel-User Boundary

- **High addresses** (e.g. $\geq 0xffff800000000000$) are reserved for the kernel.
 - **User processes cannot access** these pages.
-

8. Growth & Limits

- **Heap limit:** Controlled by the process's `ulimit` (e.g. `ulimit -s` for stack, `ulimit -v` for address space).
 - **Stack limit:** Also via `ulimit -s`.
 - **Address space fragmentation:** Interleaving many `mmap()` and `brk()` calls can fragment the virtual space.
-

9. Putting It All Together



Key Takeaways

- **Segmentation vs Paging:** Modern OSes ignore “segments” CPU feature; everything’s page-based. The above “segments” are just logical regions managed by the loader and kernel.
- **sbrk() vs mmap():** Heap growth is linear and contiguous; **mmap()** lets you allocate arbitrary pages anywhere.
- **Thread stacks** are independent mappings with their own guard pages.
- **Protections** (R/W/X) are enforced per page by the MMU.
- **Fragmentation** can happen in both heap and mmap regions.

This detailed layout should give you a clear mental model of where your code, data, and dynamically allocated memory live, how they grow, and how the OS manages protections and mappings.

Where does **mmap** allocate to. Isn't the space above **brk** and **stack** for **heap** and **stack** only.

Great question — let's clarify it in detail.

Where Does **mmap()** Allocate?

In general:

`mmap()` allocates anywhere in the process's virtual address space that is:

- not already mapped, and
- permitted by kernel policies and memory layout constraints.

The kernel decides *where* to map, unless you explicitly request an address via `mmap(addr, ...)`.

Typical Memory Layout (simplified)

```
pgsql

Low address
+-----+
| Code / .text      |
| Read-only data    |
| .data / .bss      |
+-----+
| brk()/sbrk() Heap | <--- grows upward
+-----+
|                   | <--- mmap region starts here (varies)
| mmap() allocations |
| shared libs, etc.  |
| TLS                |
+-----+
| Stack              | <--- grows downward
+-----+
High address
```

Key Points:

1. Heap (`brk()`/`sbrk()`):

- Managed by traditional allocators.
- Grows *upward* in memory.
- Located just after BSS (zero-initialized globals).
- Linear and contiguous.

2. `mmap()` region:

- Typically starts at a **much higher address**, far *above* the current `brk` value and **below the stack**.
- On 64-bit Linux, `mmap` allocations often start around `0x7f...` (depending on ASLR).
- It's **non-contiguous** — the kernel maps new pages anywhere in the available virtual space.
- This is why large mallocs (e.g. `>128 KB`) use `mmap()` — they don't affect `brk()` or the main heap.

3. Stack:

- Grows downward from near the top of user space (e.g. `0xffff_ffff_ffff`).
- Kernel places a few **guard pages** below the stack to detect overflows.

So... Isn't That Space Reserved for Heap and Stack?

No — not entirely.

- The **virtual address space** between the heap and stack is **not reserved** just for heap growth.
- The kernel manages it flexibly and **uses it for `mmap()` allocations**.
- It just makes sure:
 - Heap growth (`brk()`) doesn't clash with mmap regions.
 - Stack has space to grow down safely.
 - ASLR, page alignment, and protection rules are respected.

If heap and mmap regions grow toward each other and collide, the kernel returns an error (e.g., `ENOMEM`).

Visualization Example:

Imagine:

- `brk()` heap ends at `0x601000`.
- `mmap()` starts mapping pages at `0x7f1234567000`.
- Stack is near `0xffffffff000`.

So `mmap()` lives in that huge gap in the middle — tens/hundreds of GB — with ASLR/randomized gaps in between.

Summary

- `mmap()` allocations are **not limited** to a small range or forced to follow the `brk()` heap.
- They typically go **above the heap**, but **below the stack**, in a huge range managed by the kernel.
- That area is **not exclusive** to heap or stack — it's a general-purpose mmap region.

Let me know if you'd like to inspect this live using `pmap`, `/proc/self/maps`, or a demo C program.

How large can the storage be between brk and stack

The space between the `brk()` heap and the stack in a Linux process is called the "**mmap region**" or "**anonymous mapping region**", and it's **flexibly managed** by the kernel. The size of this space depends on several factors, mainly:

1. Address Space Limits

On 64-bit systems:

- User space addressable memory is typically **128 TB** (in Linux, the lower half of the 256 TB virtual address space).
- This gives **enormous room** between the heap (`brk`) and stack.

For example:

- Heap ends at `0x602000`
- Stack starts near `0x7fffffff000`

That's ~128 TB of virtual address space between them, most of which is available for `mmap()` and other dynamic allocations.

On 32-bit systems:

- User space is usually 2–3 GB (depending on OS config).
- The heap and stack are much closer.
- `mmap` region might only be hundreds of MBs.

2. Kernel Memory Layout Policies

- The kernel doesn't reserve the entire gap for one purpose.
- It places:
 - Shared libraries
 - Thread stacks
 - TLS (thread-local storage)
 - `mmap`-ed files
 - Anonymous mappings
 - JIT memory (e.g. for JavaScript engines)
- ASLR (Address Space Layout Randomization) introduces gaps and random offsets, reducing predictability.

3. `ulimit` and `ulimit -v` (virtual memory size)

You can restrict how much **total memory** (RAM + swap + mappings) a process can use with:

```
bash
```

```
ulimit -v # In kilobytes
```

This won't reduce the *address space*, but it limits how much of it you can actually allocate.

4. Collision Risk

Even on 64-bit, if:

- The `brk()` heap grows too much upward, and
- The stack grows too much downward,
- Or many `mmap` allocations are made,

Then the kernel can reject further allocations if the regions meet:

```
cpp
```

```
brk heap ↑↑↑      ↓↓↓ stack  
COLLISION → ENOMEM
```

Summary

System	Typical Max Gap (mmap space)
64-bit Linux	Up to hundreds of TB
32-bit Linux	~ hundreds of MB

- The actual usable space depends on **process layout**, **ASLR**, and **system limits**.
- It's **not contiguous** and not reserved, but you have *plenty of space* on 64-bit.

Would you like a small C demo to print the distance between `brk()` and stack at runtime?

How can it 128 tb if my laptop storage is itself 512 gb. What is virtual address

Great question — you're hitting on the key concept of **virtual memory**. Let's break this down clearly.

What is Virtual Memory?

Virtual memory is an **abstraction layer** provided by the operating system and hardware (CPU/MMU), where:

- Each process sees a **large, private, and continuous address space**.
- This address space is **independent** of the actual physical RAM or disk size.

So when we say "128 TB of virtual address space", it **doesn't mean** your laptop has 128 TB of RAM or storage.

Think of it like:

-

A map of possible memory locations.

- The OS decides what parts of that map are actually used and backed by real memory (RAM, swap, file, or nothing).

How Is That Possible?

Virtual vs. Physical:

Feature	Virtual Memory	Physical Memory (RAM)
Size (64-bit Linux)	Up to 128 TB (user)	Whatever RAM you have
Backed By	RAM, disk, nothing	Actual physical hardware
Per Process	Isolated and private	Shared system-wide

Example:

- Your 64-bit CPU can theoretically address up to $2^{64} = 16$ exabytes.
- Linux uses **lower 47 or 48 bits** for user space → **128 TB to 256 TB**.
- That's just a **virtual map** — not real memory.

What Happens During Allocation?

- 1 When you `malloc(1 GB)`:

2

The OS reserves 1 GB **virtual space** (from heap or **mmap**).

3

Physical memory isn't assigned **until you actually use** it (e.g. write to it).

4

This is called **lazy allocation** or **demand paging**.

5

If you never access that memory:

6

It might not even use any real RAM.

7

If RAM runs out:

8

The OS can use **swap**, or

9

Kill the process (OOM).

Why So Much Virtual Space?

Because it's **cheap** — just address mappings. Huge space means:

- Memory-mapped files don't collide.
- Libraries can be placed far apart.
- Heap and stack can grow freely.
- Security features like ASLR are possible.

Analogy

Think of virtual memory as:

A massive parking lot (virtual), but you only pay for the spaces (pages) you park in (actually use).

Most of the lot stays empty unless you need it.

In Short:

- 128 TB is just the **virtual address space** range on 64-bit Linux for user processes.
- Your real memory is still 512 GB disk and, say, 16 GB RAM.
- Virtual memory makes large, isolated, flexible memory spaces possible.

Let me know if you want a visual or code demo to see this in action.

Can a process not read beyond its stack

Correct — a process **cannot read or write beyond its stack boundaries** safely. Let's go into detail:

What is the Stack?

- A region of a process's **virtual memory** used for:

- Function calls

- Local variables

- Return addresses

- Grows **downward** (from high to low addresses).

Stack Boundaries

The operating system sets:

- A **starting address** (top of the stack)
- A **limit** (maximum stack size)

Example (on Linux 64-bit):

- Stack start: `0x7fffffff000`
- Stack limit: Typically **8 MB** by default (can be changed via `ulimit -s`)

What Happens If You Go Beyond It?

Read/Write Outside Stack → Segmentation Fault

- Accessing memory **beyond the allocated stack size** (especially if it hasn't grown to that point yet) triggers:
 - **Page fault**
 - If the address is outside the allowed stack range → **SIGSEGV (segfault)**

Stack Overflow

- Happens when the stack exceeds its max size (e.g., from deep recursion or large local arrays).
 - OS **does not allow growth beyond the limit**.
-

Protections in Place

- The OS sets a **guard page** at the bottom of the stack:
 - It's non-accessible.
 - Used to detect stack overflow early.
 - Memory between stack and other regions (like heap) is usually **unmapped** → access causes a fault.
-

Summary

Access Attempt	Result
Within stack range	Allowed
Just below current stack	Might grow stack (if within limit)
Beyond allowed limit	Segfault (stack overflow or invalid access)
Above the stack (other regions)	Segfault or unrelated memory region

So yes, a **process cannot read beyond its stack bounds** — doing so results in undefined behavior or immediate termination. This is one of the fundamental safety mechanisms in modern OSes.

Let me know if you want to test this behavior in C.

I am talking on the other direction of stack growth

Ah, I see — you're asking: **can a process read *above* the stack in memory (in the direction opposite to the stack growth)?**

Let's clarify:

1. Stack Grows Down

On most systems (like Linux x86/x86_64), the **stack grows downward** in memory — from higher addresses to lower ones.

Example (simplified):

```
csharp
0x7fffffffffff (top of virtual memory)
|
v
[stack]
|
v
[mmap area]
|
v
[heap (via brk)]
|
v
[code, data, etc.]
0x000000000000
```

2. So What Lies *Above* the Stack?

Above the current stack pointer (`%rsp` on x86_64), there is:

- **More stack (unused yet)** — OS expands the stack *downward* only when needed.
- **Stack guard page** — prevents infinite or malicious growth.
- **Beyond that: invalid / unmapped memory** — accessing it causes a segmentation fault.

3. Can You Access Memory *Above* the Stack Pointer?

Yes, but only if it is part of the current stack frame or stack allocation.

Examples:

Safe access (within current stack frame):

```
c
void func() {
    int a = 10;
    int b = 20;
    int *p = &a + 1; // technically accessing higher address on stack
    printf("%d\n", *p); // likely gets b
}
```

Unsafe access:

```
c
void func() {
    int a = 10;
    int *p = &a + 1000000;
    printf("%d\n", *p); // segmentation fault – you're beyond allocated stack
}
```

So you **can** read **just above** the current stack pointer (within your function's frame), but **you cannot access arbitrary addresses above the stack region** (in high memory), because:

- That memory **doesn't belong to your stack**
- It might be **guarded** or **unmapped**
- It's not guaranteed to be readable

Conclusion

- You can read **above the current stack pointer** if it's still part of the allocated stack.
- You **cannot read above the stack region** itself (in higher addresses); it will lead to a **segmentation fault**.
- This restriction is enforced by the OS and MMU.

Let me know if you'd like to test this with a live C code example or with GDB.

What is instead of addition I subtract 100000 . Also why do stack and heap grow in opp direction

Great questions — let's break it down:

1. What if I subtract a large number from a stack variable?

Example:

```
c
```



```
void func() {
    int a = 10;
    int *p = &a - 100000; // much lower address than &a
    printf("%d\n", *p);   // likely segmentation fault
}
```

This would access memory **far below the current stack pointer** (i.e., towards the **heap/brk area** or **unmapped** memory). This is **extremely dangerous** and will almost certainly cause a **segmentation fault**, because:

- You're accessing memory **below the allocated stack region**.
- That memory is **not mapped** or belongs to **another segment** (like heap, **.bss**, or code).

The OS only gives you a **limited stack size** (e.g., 8 MB by default on Linux), and it guards both ends of it.

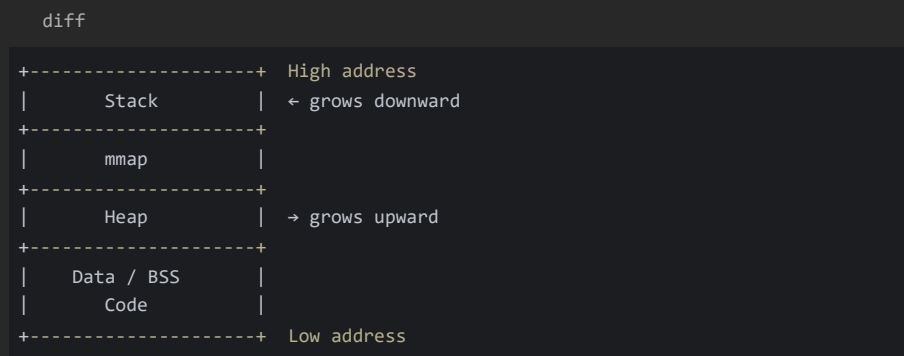
2. Why do stack and heap grow in opposite directions?

This design is **intentional** and has historical, architectural, and practical benefits:

Stack grows downward (high → low)

Heap grows upward (low → high)

So they "grow toward each other" — like this:



Why this design?

- 1 **Maximizes available memory:**
Heap and stack can dynamically expand **toward each other** — using unallocated memory in between.
- 2 **Simple segmentation:**
Early systems used segmented memory. Opposite growth let the system easily detect **overflow** (heap colliding with stack).
- 3 **Safety:**
OS can set **guard pages** between them to catch overflows.
- 4 **No fragmentation:**
Since they grow in opposite directions, their memory management is cleaner and doesn't require constant moving like heap compaction in garbage-collected systems.

Bonus: What happens when stack and heap collide?

- If your program uses **too much stack or heap**, and they **meet**, you'll get:
 - **segmentation fault** or
 - **stack overflow** or **heap allocation failure**.

That's why modern OSes place **limits** on both heap and stack size.

Let me know if you'd like to explore this visually with memory diagrams or experiment with code!

Suppose the stack base is at 100. It grows to 99 98 etc. what if I try to access 105

Great question.

Scenario:

- Let's say the **stack base** is at virtual address **100**.
- The stack grows **downward**, so addresses **99, 98, 97**, etc., are used as new stack frames are pushed.
- Now you try to access **address 105**, which is **higher than the current base of the stack**.

What happens when you access address 105?

- In **C**, doing something like:

```
c
int a;
int *p = &a + 5; // Assuming &a is near the top of the stack (e.g., address 100)
printf("%d\n", *p);
```

will attempt to read from an address like **105**.

- That memory is:
 - **Not part of the stack**.
 - **May belong to:**
 - The **mmap region** (used by shared libraries or **mmap**-allocated data).
 - **Unallocated virtual memory** (causing segmentation fault).
 - **Guard page** that protects the stack.

Possible Outcomes:

- 1 **Segmentation Fault:**
 - 2 If address **105** is in an **unmapped region**, the OS will kill your process.
- 3 **Silent memory corruption:**
 - 4 If **105** happens to be mapped (e.g., into the **mmap** area), your program might **read garbage** or **corrupt other memory**.

5 Undefined Behavior:

- 6 There's no guarantee what you'll read/write — it may crash, hang, or misbehave later.


In short:

Even though the stack grows downward, **you cannot safely access memory above its base** (like `105` in your example), unless that memory is:

- Valid.
- Mapped.
- Intended for your use (e.g., global variables, heap, etc.).

Doing so is **undefined behavior** and very dangerous in C.

Would you like an example in code to demonstrate this?

From ChatGPT, designed with  FancyGPT