User Memory Management in Novix OS

In the previous section, we gave user programs a way to communicate with the kernel using **system calls**. Now, we're taking another giant step forward: letting user programs **dynamically allocate memory**.

In this chapter, you'll learn how **Novix implements its own user-space heap**, managed by `malloc()` and `free()`,backed by the kernel's syscall `SYS_SBRK`.

Learning Goals

By the end of this lesson, you'll understand:

- How userland requests heap memory via `sbrk()`
- How the kernel extends the heap on demand
- How `malloc()` and `free()` manage user memory
- How dynamic allocation works safely in user mode

The Concept of the Heap

Every user process in Novix OS has its own memory layout, defined by:

Segment	Purpose
Text	Program instructions (ELF-loaded code)
Data	Global/static variables
Heap	Dynamic memory (`malloc`, `calloc`, `realloc`)
Stack	Local variables and function calls

When a program calls `malloc()`, it requests more memory from the heap.
If the heap needs to grow, it calls `sbrk()` — a system call that asks the kernel to map more pages.

The `sbrk()` System Call

Let's start in **userland**, with the function `sbrk()` inside `user/lib/sbrk.c`:

```c
#include "../include/syscall.h"

static uintptr_t heap_end = 0;

void *sbrk(long increment) {
    if (heap_end == 0)
        heap_end = (uintptr_t)&__user_heap_start;

    uintptr_t prev = heap_end;
    uintptr_t new_end = heap_end + increment;

    int64_t res = syscall(SYS_SBRK, increment, 0, 0);

    if (res < 0)
        return (void *)-1;

    heap_end = new_end;
    return (void *)prev;
}
```

How it works:

1. The first time it's called, it initializes `heap_end` to the start of the user heap.
2. It then requests `increment` bytes from the kernel via syscall `SYS_SBRK`.
3. If the kernel approves, `heap_end` moves forward, effectively extending the heap.
4. The function returns the previous heap pointer, which becomes usable memory.

Kernel Implementation: `sys_sbrk()`

In the kernel (`kernel/syscall.c`), the corresponding handler looks like this:

```
long sys_sbrk(long increment) {
    proc_t *p = current_proc;
    uintptr_t old_heap = p->heap_end;
    uintptr_t new_heap = old_heap + increment;

    if (increment > 0) {
        for (uintptr_t addr = PGROUNDUP(old_heap); addr < new_heap; addr += PAGE_SIZE) {
            paddr_t pa = alloc_pages(1);
            if (!pa) return -1;
            map_page(p->page_table, addr, pa, PTE_U | PTE_R | PTE_W | PTE_V, 0);
        }
    }

    p->heap_end = new_heap;
    return old_heap;
}
```

Step-by-step:

- The kernel retrieves the current process (`current_proc`).
- It checks the process's current heap end and adds the requested increment.
- If more pages are needed, it allocates them and maps them into the process's page table.
- Finally, it updates `heap_end` and returns the old heap address.

  Just like Linux, this allows user processes to grow their heap dynamically and safely.

The Building Blocks of Dynamic Memory

Now that `sbrk()` can grow memory, we can implement our own `malloc()` and `free()` — the heart of user memory management.

The Memory Block Structure

In `user/include/malloc.h`, we define a small structure to describe each heap block:

```
typedef struct block {
    size_t size;
    struct block *next;
} block_t;

#define BLOCK_SIZE sizeof(block_t)
```

Each block contains: - The **size** of the allocated memory - A pointer to the **next** free block

This forms a simple **linked list** called the *free list*, which tracks available memory chunks.

malloc() Implementation

Here's the simplified version from `user/lib/malloc.c`:

```
void *malloc(size_t size) {
    block_t *prev = NULL;
    block_t *curr = free_list;

    size = (size + 7) & ~7;  // Align size to 8 bytes

    while (curr) {
        if (curr->size >= size) {
            // Reuse or split existing block
            if (curr->size >= size + BLOCK_SIZE + 8) {
                block_t *new_block = (block_t*)((char*)curr + BLOCK_SIZE + size);
                new_block->size = curr->size - size - BLOCK_SIZE;
                new_block->next = curr->next;
                curr->size = size;
                if (prev)
                    prev->next = new_block;
                else
                    free_list = new_block;
            } else {
                if (prev)
                    prev->next = curr->next;
                else
                    free_list = curr->next;
            }
            return (char*)curr + BLOCK_SIZE;
        }
        prev = curr;
        curr = curr->next;
    }

    // No free block found - grow the heap
    void *mem = sbrk(size + sizeof(block_t));
    if (mem == (void*)-1)
        return NULL;

    block_t *new_blk = (block_t*)mem;
    new_blk->size = size;
    return (char*)new_blk + BLOCK_SIZE;
}
```

How it works:

1. It first looks through the free list for a suitable block.
2. If one is found, it either reuses or splits it.
3. If not, it grows the heap using `sbrk()`.
4. It then returns a pointer to usable memory just after the metadata block.

free() Implementation

The `free()` function simply adds the released block back to the free list:

```
void free(void *ptr) {
    if (!ptr) return;

    block_t *blk = (block_t*)((char*)ptr - BLOCK_SIZE);
    blk->next = free_list;
    free_list = blk;
}
```

This is a simple, first-fit allocator — effective for educational systems and small embedded OSes.

Testing malloc/free

Let's test this in `user/shell.c`:

```
#include "include/stdio.h"
#include "include/syscall.h"
#include "include/string.h"
#include "include/malloc.h"

void main(void) {
    cls();
    puts("\nHello world from userland! (shell.elf)\n\n");

    puts("malloc/free test start...\n\n");
    void *a = malloc(100);
    void *b = malloc(200);
    void *c = malloc(300);

    printf("a = %p\n", a);
    printf("b = %p\n", b);
    printf("c = %p\n", c);

    free(b);
    void *d = malloc(180);  // could reuse b
    printf("d = %p\n", d);

    puts("\nmalloc/free test done.\n\n");

    char* str = malloc(64);
    strcpy(str, "malloc works correctly!\n");
    puts(str);

    for (;;);
}
```

When you run this program, you'll see memory addresses being printed, confirming successful allocations and reuse of freed blocks.

Summary

In this lesson, you've built a complete **user-space heap manager** in Novix OS:

- Implemented `sbrk()` to request heap space from the kernel

- Added kernel-side `sys_sbrk()` to map new pages dynamically
- Built `malloc()` and `free()` to manage memory inside user space
- Verified it all with a working shell test

You now have a functioning **user-mode memory allocator**, the foundation for all higher-level features like strings, buffers, and IPC message storage.

```
*A smile on my face - because Novix OS can now dynamically manage memory in userland!*
```