Memory Tracking in Novix OS

Memory management is at the heart of every operating system.
When an OS allocates and frees thousands of pages dynamically, tracking who owns what becomes critical, especially for debugging **memory leaks** and verifying that processes clean up properly.

In this chapter, we'll implement a **Memory Allocation Tracker** in Novix OS that records every page allocation, associates it with a process, and automatically frees leftover memory when a process exits.

Learning Goals

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

- How Novix OS tracks page allocations system-wide
- How the kernel prevents memory leaks by linking allocations to PIDs
- How the tracker integrates with `alloc_pages()` and `free_page()`
- How to debug and inspect memory usage and leaks

Overview: Why Track Allocations?

When a process allocates memory (e.g. for a heap, stack, or program image), it may not always free it before termination.
Without tracking, those allocations would be permanently lost — causing **memory leaks**.

Our tracker solves this by maintaining a **global table of active allocations**, recording:

Field	Description
`used`	Whether the entry is active
`pa`	Physical address of the first page
`npages`	Number of pages allocated
`owner`	Process ID (0 for kernel)
`tag`	Descriptive string tag (e.g. `"alloc_pages"`)

When a process exits, the tracker automatically scans for all entries belonging to that PID and releases them.

1. Defining the Allocation Tracker

The tracker's structure and interface are defined in **`include/alloc-tracker.h`**.

`include/alloc-tracker.h`

```c
#pragma once
#include <stdint.h>
#include <stddef.h>
#include "types.h"

#define MAX_TRACKED_ALLOCS 4096*2

typedef struct {
    int     used;       // 1 = entry in use
    paddr_t pa;         // physical start address
    size_t  npages;     // number of pages
    pid_t   owner;      // process ID (0 = kernel)
    char *tag;          // descriptive tag
} alloc_track_t;

void track_alloc(paddr_t pa, size_t npages, pid_t owner, char *tag);
void track_free(paddr_t pa, size_t npages);
void dump_allocs(void);

void dump_allocs_for_pid(int pid, int debug_flag);
void free_all_tracked_for_pid(pid_t pid, int debug_flag);
```

2. Implementing the Tracker

Let's implement the logic that records, frees, and dumps allocations.

kernel/alloc-tracker.c

```c
#include "alloc-tracker.h"
#include "uart.h"
#include "riscv.h"
#include "page.h"

alloc_track_t alloc_table[MAX_TRACKED_ALLOCS];
size_t alloc_table_count = 0;

void track_alloc(paddr_t pa, size_t npages, pid_t owner, char *tag) {
    for (int i = 0; i < MAX_TRACKED_ALLOCS; i++) {
        if (!alloc_table[i].used) {
            alloc_table[i].used = 1;
            alloc_table[i].pa = pa;
            alloc_table[i].npages = npages;
            alloc_table[i].owner = owner;
            alloc_table[i].tag = tag;
            alloc_table_count++;
            return;
        }
    }
}

void track_free(paddr_t pa, size_t npages) {
    for (int i = 0; i < MAX_TRACKED_ALLOCS; i++) {
        if (alloc_table[i].used && alloc_table[i].pa == pa && alloc_table[i].npages == npages) {
            alloc_table[i].used = 0;
            alloc_table_count--;
            return;
        }
    }
}
```

3. Debugging Tools

To help debug memory usage, we provide dump functions that list active allocations.

dump_allocs() and dump_allocs_for_pid()

```c
void dump_allocs(void) {
    uart_printf("=== OUTSTANDING ALLOCS ===\n");
    for (int i = 0; i < MAX_TRACKED_ALLOCS; i++) {
        if (alloc_table[i].used) {
            uart_printf("PA=0x%x npages=%d owner=%d tag=%s\n",
                alloc_table[i].pa,
                (int)alloc_table[i].npages,
                (int)alloc_table[i].owner,
                alloc_table[i].tag ? alloc_table[i].tag : "?");
        }
    }
}
```

```c
void dump_allocs_for_pid(int pid, int debug_flag) {
    if (debug_flag) uart_printf("=== ALLOCS for pid=%d ===\n", pid);
    for (int i = 0; i < MAX_TRACKED_ALLOCS; i++) {
        if (alloc_table[i].used && alloc_table[i].owner == pid) {
            if (debug_flag)
                uart_printf("  leak: pa=0x%x npages=%d tag=%s\n",
                    alloc_table[i].pa,
                    (int)alloc_table[i].npages,
                    alloc_table[i].tag ? alloc_table[i].tag : "?");
        }
    }
    if (debug_flag) uart_printf("=== END of list - ALLOCS for pid=%d ===\n", pid);
}
```

4. Automatic Cleanup on Process Exit

When a process terminates, the kernel must ensure that all pages owned by that process are released. The cleanup occurs in **`free_all_tracked_for_pid()`**.

`kernel/alloc-tracker.c`

```c
void free_all_tracked_for_pid(pid_t pid, int debug_flag) {
    disable_interrupts();
    if (debug_flag) {
        uart_printf("[free_proc] Allocations still owned by pid %d:\n", pid);
        uart_printf("[free_all_tracked] Force freeing leftover allocations for pid=%d\n", pid);
    }

    for (int i = 0; i < MAX_TRACKED_ALLOCS; i++) {
        alloc_track_t *e = &alloc_table[i];
        if (!e->used || e->owner != pid) continue;

        if (debug_flag)
            uart_printf("  freeing leak: pa=0x%x npages=%d tag=%s",
                e->pa, (int)e->npages, e->tag ? e->tag : "?");

        for (size_t p = 0; p < e->npages; p++) {
            paddr_t pa_page = e->pa + p * PAGE_SIZE;
            free_page(pa_page);
        }

        if (debug_flag) uart_puts(" = destroyed\n");

        e->used = 0;
        e->pa = 0;
        e->npages = 0;
        e->owner = 0;
        e->tag = NULL;
        alloc_table_count--;
    }

    enable_interrupts();
}
```

5. Integration with Process Cleanup

The process cleanup routine in **`kernel/process.c`** calls the tracker to destroy leftover allocations.

```
void free_proc(proc_t *proc) {
    ...
    free_all_tracked_for_pid(proc->pid, 0);
    dump_allocs_for_pid(proc->pid, 0);
    ...
}
```

By setting the second argument to `1`, you can enable detailed debug output.

To Enable Debug Output

```
free_all_tracked_for_pid(proc->pid, 1);
dump_allocs_for_pid(proc->pid, 1);
```

To Disable Debug Output

```
free_all_tracked_for_pid(proc->pid, 0);
dump_allocs_for_pid(proc->pid, 0);
```

Example Output (with debug enabled)

```
Novix RISC-V 64 OS, (c) NovixManiac, Shell version : 0.0.1
$ ls
Files in tarfs:
idle.elf
shell.elf
hello.elf
ps.elf
ls.elf
mem.elf
date.elf
echo.elf
[free_proc] Allocations still owned by pid 3:
[free_all_tracked] Force freeing leftover allocations for pid=3
  freeing leak: pa=0x80537000 npages=1 tag=alloc_pages = destroyed
  freeing leak: pa=0x80536000 npages=1 tag=alloc_pages = destroyed
  ...
=== ALLOCS for pid=3 ===
=== END of list - ALLOCS for pid=3 ===
```

Summary

In this chapter, we've achieved:

- Implemented a **global memory tracker** for all allocations
- Automatically freed leaked pages when a process terminates
- Added debugging utilities for leak detection
- Integrated the tracker into the page allocator
- Improved system reliability and memory hygiene

This new **Memory Tracking System** makes Novix OS far more robust — ensuring that no process can leak pages permanently.
It's a key step toward a stable, production-quality memory management subsystem.

```
*A smile on my face - because Novix OS ensuring that no process can leak memory!*
```