Memory Management in RISC-V

In this chapter, we implement and analyze Novix OS's foundational memory-management routines. These routines establish and verify each major memory region:

- **BSS** – Zero-initialized global data
- **Stack** – Per-core or per-task runtime stack
- **RAM** – Physical memory boundaries and page-count tracking
- **Heap** – Dynamic memory allocator

Together they bring the kernel's memory map to life.

Function: bss_init

```
void bss_init(void);
```

Purpose

Clears the BSS segment, ensuring all uninitialized global variables start at 0 before kernel code executes.

Parameters

- None

Return Value

- None

Detailed Description

1. The linker defines `__bss` and `__bss_end` symbols marking the BSS range.

2. `memset()` fills the region with zeros:

   ```
   memset(__bss, 0, (size_t)__bss_end - (size_t)__bss);
   ```

   This guarantees predictable startup behavior for all global/static variables.

Function: stack_init

```
void stack_init(void);
```

Purpose

Calculates stack boundaries and logs their addresses via UART.

Detailed Description

1. Obtains linker symbols `__stack_bottom` and `__stack_top`.

2. Computes total stack size and prints diagnostics:

   ```
   uart_printf("[stack_init] STACK initialized, size = %d KB, start = 0x%x, end = 0x%x\n",
               stack_size/1024, __stack_bottom, __stack_top);
   ```

   This confirms correct alignment and reservation of kernel stack memory.

Function: ram_init

```
void ram_init(void);
```

Purpose

Discovers the free-RAM region and calculates the number of physical pages available.

Detailed Description

1. Reads linker symbols `__free_ram` and `__free_ram_end`.
2. Computes total RAM size and divides by `PAGE_SIZE` to obtain `g_total_pages`.
3. Logs the result through UART:
   ```
   uart_printf("[ram_init] RAM initialized, size=%d MB, start=0x%x, end=0x%x total pages=%d\n", …);
   ```
   This prepares the kernel's physical memory allocator.

Function: page_allocator_total_pages

```
uint64_t page_allocator_total_pages(void);
```

Purpose

Returns the total number of physical pages available as computed by `ram_init()`.

Return Value

- `uint64_t` — Total page count.

Heap Management

Structure: Block

```
typedef struct Block {
    size_t size;
    struct Block *next;
    int free;
} Block;
```

Each allocated or free chunk begins with this header.

- `size` — Data payload length.
- `next` — Linked-list pointer to the next block.
- `free` — Flag (1 = available, 0 = in use).

Function: heap_init

```
void heap_init(void);
```

Purpose

Initializes a simple **first-fit heap allocator** over the linker-defined heap region.

Detailed Description

1. Places the initial `Block` header at `__heap_start`.
2. Sets its `size` to the total heap length minus header space.
3. Marks the block free and prints a summary:
   ```
   uart_printf("[heap_init] HEAP initialized, size=%d MB, start=0x%x, end=0x%x\n", …);
   ```
   From here, `malloc()` and `free()` manage this region.

Function: malloc

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

Purpose

Allocates a contiguous block of memory from the kernel heap.

Detailed Description

1. Rounds `size` up to 8-byte alignment.
2. Traverses the `free_list` to find a suitable block.
3. If large enough, splits the block and updates links.
4. Marks the chosen block used and returns its payload address.
5. Prints diagnostics, e.g. `[malloc] size=64 → 0x80205000`.

Return Value

- Pointer to allocated memory on success.
- `NULL` if no suitable block found.

Function: free

```
void free(void *ptr);
```

Purpose

Releases a previously allocated heap block and optionally merges it with adjacent free blocks.

Detailed Description

1. Validates pointer and computes its header.
2. Marks block as free and logs the event.
3. If next block is also free, coalesces them to reduce fragmentation.

Function: kernel_main

```
void kernel_main(void);
```

Purpose

Coordinates all memory-initialization routines in sequence.

Detailed Steps

1. **Clear BSS**
   ```
   bss_init();
   ```
2. **Display welcome screen**
   ```
   uart_cls();
   uart_printf("Novix OS (64-bits), (c) Novix, Version 0.0.1\n\n");
   ```
3. **Initialize trap vector**
   ```
   trap_init();
   ```
4. **Initialize stack, RAM & heap**
   ```
   stack_init();
   ram_init();
   heap_init();
   ```
5. **Run heap diagnostic**
   ```
   heap_test();
   ```
6. Enter infinite loop to keep kernel running.

Summary

This chapter implemented the low-level building blocks that enable every other Novix OS subsystem.
You now have:

- A clean BSS segment
- Verified stack boundaries
- Physical RAM enumeration
- A working heap allocator

These form the core of **runtime memory management** in a real 64-bit RISC-V operating system.