Operating Systems Project: Topic 8

Kernel Virtual Memory Mapping Visualizer

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.22

Outline

1. Project Goals & Requirements
2. Theory: The Virtual Address Space
3. Kernel Mechanics: mmap Internals
4. Implementation Guide
5. Next Steps

Outline

1. Project Goals & Requirements
2. Theory: The Virtual Address Space
3. Kernel Mechanics: mmap Internals
4. Implementation Guide
5. Next Steps

The Mission: Topic 8 Requirements

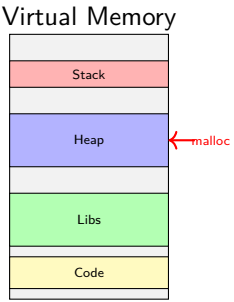
Objective: Open the "Black Box" of process address spaces. Visualize memory layout changes in real-time.

Requirement 1: Kernel Instrumentation

- **Target:** Intercept `mmap()`, `munmap()`, and `brk()` operations.
- **Data:** Capture start address, length, permissions (R/W/X), and backing file.
- **Output:** Stream this data to user space (e.g., via `/proc` or `debugfs`).

Requirement 2: Visualization Tool

- **User Space:** Develop a GUI (Python/Web).
- **Function:** Read the live stream and render the process's Virtual Memory map dynamically.



Advanced Options

Option A: Lifecycle Tracking (Fork/Exec)

- **Challenge:** `fork()` duplicates the entire memory map.
- **Task:** Visualize this "Cloning" event. Show how Parent and Child memory maps diverge over time (COW splits).

Option B: Page Table Walk Visualization

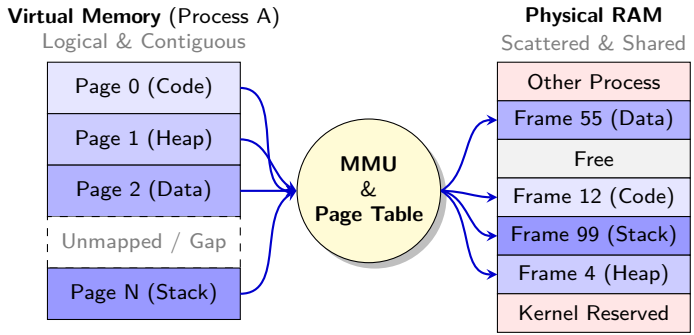
- **Challenge:** VMA is logical; Page Table is physical.
- **Task:** When a Page Fault occurs, visualize the hardware "Walk" from PGD $\rightarrow$ PUD $\rightarrow$ PMD $\rightarrow$ PTE to show physical allocation.

Outline

1. Project Goals & Requirements
2. Theory: The Virtual Address Space
3. Kernel Mechanics: mmap Internals
4. Implementation Guide
5. Next Steps

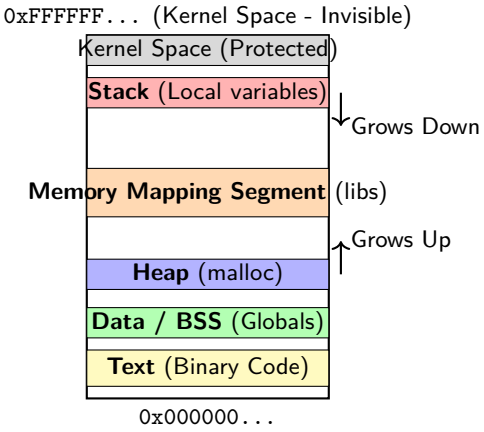
Theory 1: The Grand Illusion (Visualized)

The Illusion: The process sees a neat, contiguous block of memory. **The Reality:** Data is scattered, fragmented, and mixed with other processes.



Theory 2: Standard Layout of a Linux Process

When you run a program, the OS builds a standard environment. Your visualizer should identify these zones.



Theory 3: Segments Explained

The Static Parts (Fixed at Load Time)

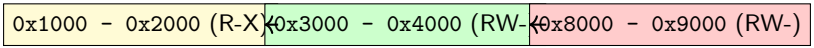
- **Text Segment:** Read-Only, Execute. Contains CPU instructions.
- **Data Segment:** Read-Write. Global initialized variables (e.g., `int x = 5;`).
- **BSS Segment:** Read-Write. Global uninitialized variables (e.g., `int y;`).

The Dynamic Parts (Change at Runtime)

- **Heap:** Managed by `brk()`. Expands upwards when you `malloc()`.
- **Stack:** Managed automatically. Expands downwards. Holds function frames and return addresses.
- **Mmap Region:** Used for `mmap()` allocations and shared libraries (`.so` files).

Theory 4: The Kernel Object - `vm_area_struct`

To the Linux Kernel, "Segments" are just **VMAs** (Virtual Memory Areas). Each VMA represents a contiguous range of addresses with the **same permissions**.



Project Task: Your job is to intercept the creation and destruction of these struct nodes.

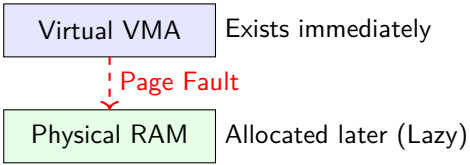
Theory 5: Demand Paging (Virtual vs Physical)

Crucial Concept: When `malloc` returns, do you have physical RAM? **NO.** You only have a VMA (a promise).

The Sequence:

1. `malloc(1MB)` calls `mmap`.
2. Kernel creates a VMA. Returns address.
3. App tries to write to that address.
4. **CPU Exception (Page Fault)!**
5. Kernel catches fault, allocates Physical Frame, updates Page Table.
6. App resumes.

This project visualizes the top box (The VMA).



Theory 6: VMA Types (Anonymous vs File-Backed)

Not all VMAs are the same. Your visualizer should color-code them.

1. **Anonymous Mappings (Anon)**
   - **Content:** Pure RAM (initialized to zero).
   - **Used For:** Heap, Stack, BSS.
   - **Backing:** Swap File (if RAM is full).
   - **Kernel Flag:** `vm_file == NULL`.
2. **File-Backed Mappings (File)**
   - **Content:** Direct view of a file on disk.
   - **Used For:** Code (.exe), Shared Libraries (.so), Data files.
   - **Backing:** The file itself.
   - **Kernel Flag:** `vm_file != NULL`.

Outline

1 Project Goals & Requirements

2 Theory: The Virtual Address Space

3 Kernel Mechanics: mmap Internals

4 Implementation Guide

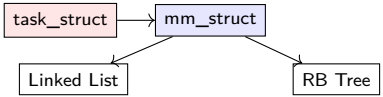
5 Next Steps

Mechanics 1: The Memory Descriptor (`mm_struct`)

Every process (`task_struct`) points to an `mm_struct`. This describes the entire address space.

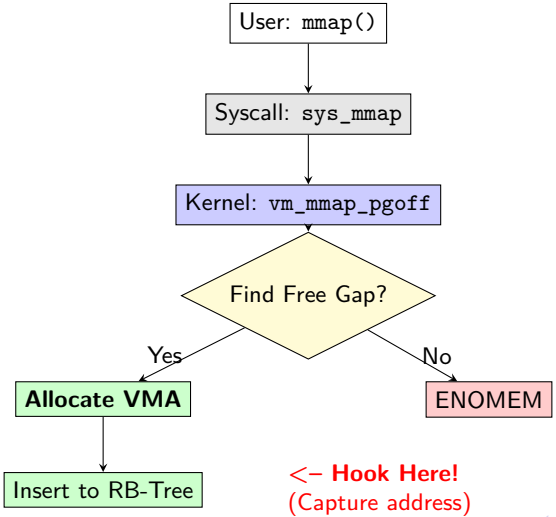
Key Fields in `mm_struct`:

- `mmap`: Head of the VMA Linked List (for sequential access, e.g., `/proc/maps`).
- `mm_rb`: Root of VMA Red-Black Tree (for fast lookup during Page Faults).
- `start_code`, `end_code`: Boundaries of the text segment.
- `start_brk`, `brk`: Boundaries of the Heap.



Mechanics 2: The Lifecycle of an `mmap`

What actually happens inside the kernel?



Outline

1 Project Goals & Requirements

2 Theory: The Virtual Address Space

3 Kernel Mechanics: mmap Internals

4 Implementation Guide

5 Next Steps

Step 1: Where to Hook?

You need to intercept VMA creation, destruction, and resizing.

Target File: `mm/mmap.c`

This file manages the address space layout.

- **Creation:** Hook `mmap_region()` or `do_mmap()`. This is where ‘vm_start‘ is finalized.
- **Destruction:** Hook `do_munmap()` or `__vm_munmap()`.
- **Resizing:** Hook `do_brk()` (used for small Heap allocations).

Tools: You can use `tracepoints` (safe) or direct source modification (flexible).

Step 2: What to Log?

Inside the kernel, you have access to the `vm_area_struct`. Extract these fields:

```
// Pseudo-code injection in mm/mmap.c
struct vm_area_struct *vma = ...; // The created VMA

if (current->pid == TARGET_PID) {
    // 1. Basic Range
    unsigned long start = vma->vm_start;
    unsigned long end = vma->vm_end;

    // 2. Permissions (R/W/X)
    unsigned long flags = vma->vm_flags;

    // 3. Name (Backing File)
    char *name = "anon";
    if (vma->vm_file) {
        name = vma->vm_file->f_path.dentry->d_name.name;
    }

    trace_printk("VMOP: ADD %lx-%lx %s\n", start, end, name);
}
```

Step 3: Handling the Data Stream

How do you send this data to User Space efficiently?

Approach A: `printk` (Simple)

- Easy to implement.
- Read via `dmesg` or `/proc/kmsg`.
- **Con:** Slow, rate-limited, can drop messages if updates are too fast.

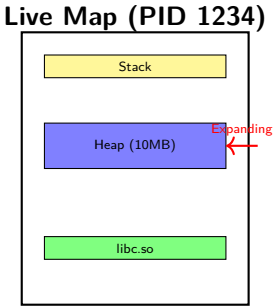
Approach B: Netlink / RelayFS (Advanced)

- Create a socket-like channel between kernel and user.
- Zero-copy or ring-buffer based.
- **Pro:** Very fast, suitable for heavy workloads (e.g., compiling code).

Step 4: The User Space Visualizer

Architecture:

1. **Target App:** A dummy C program that ‘malloc‘s randomly.
2. **Parser:** Python script reading log stream.
3. **Renderer:** Matplotlib / PyGame.



Step 5: What to look for?

Your tool should be able to spot these standard patterns:

1. **Memory Leak**
   - **Visual:** The Heap VMA (Blue) keeps growing continuously without shrinking.
   - **Cause:** `malloc` without `free`.

2. **Stack Growth**
   - **Visual:** The Stack VMA (Top) expands downwards automatically as function recursion deepens.
   - **Note:** You don't see 'mmap' calls for stack growth; the kernel handles Page Faults to expand it automatically.

Outline

1. Project Goals & Requirements
2. Theory: The Virtual Address Space
3. Kernel Mechanics: mmap Internals
4. Implementation Guide
5. Next Steps

Resources & Next Steps

Action Plan:

1. **Explore:** Run `cat /proc/self/maps`. This is the text output you want to visualize.
2. **Trace:** Use `ftrace` to find when `mmap_region` is called.
3. **Code:** Modify kernel to export this data.
4. **Build:** Write a Python script to animate the boxes.

Resources:

- *Understanding the Linux Kernel*: Chapter 20 (The Process Address Space).
- `man proc`: Read about the format of `/proc/[pid]/maps`.