Implementation of Kernel Virtual Memory Mapping Visualizer

Ke Tu, *225040526,* Yanan Bu, *225040181*

***Abstract*—Virtual memory is a core abstraction in modern operating systems, providing processes with a contiguous logical address space while mapping to scattered physical memory. However, the dynamic mapping process is opaque to developers, making it difficult to intuitively observe memory layout changes, memory leaks, and runtime behavior. To address this problem, this paper presents a Kernel Virtual Memory Mapping Visualizer—a real-time monitoring and visualization tool for Linux process virtual address spaces. The system employs a loadable kernel module (LKM) integrated with kprobe/kretprobe to intercept memory operations such as mmap, brk, munmap, and process fork without modifying the kernel source code. A kfifo ring buffer and debugfs are used for efficient kernel-to-user-space data transmission. A Python backend processes events, maintains a VMA state machine, and broadcasts updates via WebSocket to a web-based frontend for graphical rendering. Experimental results show that the system achieves low-overhead, real-time visualization of dynamic memory mapping, effectively opening the ”black box” of process virtual memory management.**

***Index Terms*—Linux kernel, virtual memory, VMA, kprobe, visualization, memory mapping, operating systems**

I. Introduction

VIRTUAL memory is fundamental to modern operating systems, enabling processes to use a large, contiguous logical address space while the underlying physical memory is fragmented and shared among multiple processes. The kernel manages this mapping through Virtual Memory Areas (VMAs) and page tables, but the internal state and dynamic changes are not directly visible to user-space applications. Traditional tools such as `/proc/[pid]/maps` only provide static snapshots and cannot capture real-time transformations during process execution. This opacity complicates debugging, performance analysis, and understanding of memory-related issues.

To address this limitation, we developed a Kernel Virtual Memory Mapping Visualizer. This project implements a full-stack monitoring and visualization system that reveals the dynamic behavior of a process's virtual address space. The tool uses kernel instrumentation for data collection, and provides a web-based graphical interface for intuitive observation.

The main contributions of this work include a non-intrusive kernel monitoring mechanism based on LKM and kprobe for intercepting memory operations and process lifecycle events, a low-overhead and reliable kernel-to-user data transmission path using kfifo and debugfs, a user-space state machine capable of accurately reconstructing VMA layout changes including partial unmapping and VMA splitting, full support for process lifecycle events such as fork and exec with automatic child-process visualization, and a responsive web-based frontend that provides real-time rendering and dynamic updates of the process virtual memory layout.

II. Project Background and Goals

Modern applications rely on virtual memory to simplify memory management. However, the translation between virtual and physical addresses, the allocation and deallocation of VMAs, and the effects of system calls like `mmap`, `brk` are hidden from developers. Static tools cannot track dynamic changes, making runtime debugging challenging.

The project aims to:

- Reveal the structure of the process virtual address space.
- Capture memory operations in real time.
- Visualize VMA creation, resizing, splitting, and deletion.
- Support process lifecycle events including fork and exec.
- Provide an interactive graphical interface for intuitive observation.

The core requirements are divided into two parts: kernel-level monitoring and user-level visualization. The kernel module must intercept key system calls and collect VMA metadata efficiently. The user-space tool must process events, maintain consistent state, and display the memory layout graphically.

III. System Architecture

The system follows a three-tier architecture: kernel module, user-space backend, and web frontend. Shown in fig 1.

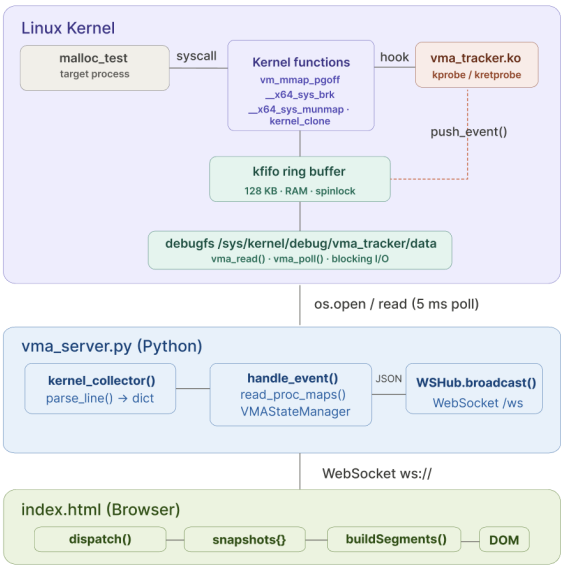

Fig. 1. System Architecture

A. Kernel Layer

The kernel layer uses a loadable kernel module (LKM) named `vma_tracker.ko`. It uses kprobe and kretprobe to hook key kernel functions involved in memory management:

- `vm_mmap_pgoff` (kretprobe): capture successful memory mappings.
- `__x64_sys_brk` (kretprobe): track heap expansion.
- `__x64_sys_munmap` (kprobe): capture unmap parameters before VMA destruction.
- `kernel_clone` (kretprobe): detect fork and retrieve child PID.

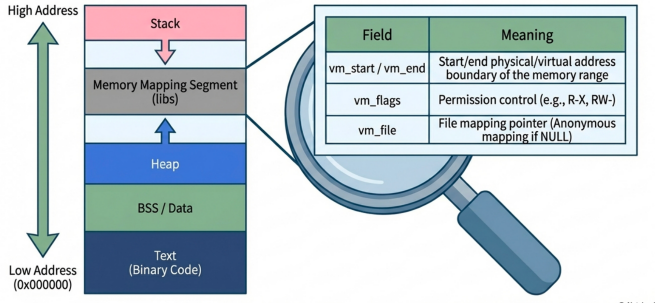

Fig. 2. Process virtual address space layout and key `vm_area_struct` fields used in the kernel module.

Events are written to a 128kb `kfifo` ring buffer and exposed to user space via `debugfs` at `/sys/kernel/debug/vma_tracker/data`.

B. User-Space Backend

The backend is implemented in Python (`vma_server.py`). It:

- Reads events from debugfs.
- Parses and validates event streams.
- Maintains a per-process VMA state machine.
- Loads initial memory layout from `/proc/[pid]/maps`.
- Broadcasts state updates to the frontend via WebSocket.

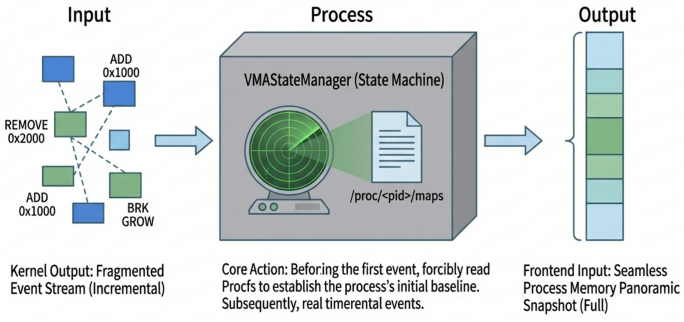

Fig. 3. VMAStateManager state machine architecture: converting fragmented kernel event streams into a consistent, full memory layout snapshot for frontend visualization.

C. Web Frontend

The frontend runs in a browser and renders the virtual address space layout in real time. It displays VMAs grouped by type (text, data, heap, stack, libraries, etc.) and updates dynamically as events arrive. Child processes are displayed adjacent to their parent to visualize fork behavior.

IV. Key Design and Implementation

A. Kprobe Instrumentation

1) Mechanism and Underlying Logic: Kprobes provide a dynamic binary instrumentation facility built into the Linux kernel. At registration time, the kernel replaces the first byte of the target instruction at the probed address with an `int3` breakpoint opcode (on x86). When the CPU executes that instruction, it raises a debug exception; the kprobe subsystem's exception handler saves the full register state into a `pt_regs` structure, invokes the user-supplied `pre_handler`, temporarily restores the original byte, single-steps the real instruction in a CPU debug mode, and then calls the optional `post_handler` before resuming normal execution. The entire mechanism operates without any source-code or kernel-image modification, making it suitable for live tracing of production kernels.

A *kretprobe* extends this to hook function *returns*. When the probed function is entered, the kretprobe subsystem replaces the on-stack return address with a *trampoline*—a small stub in kernel memory. When the function executes its `ret` instruction it jumps to the trampoline instead of the real caller; the trampoline invokes the user-supplied `handler` (with the saved `pt_regs` capturing the return value in `rax`), and then resumes at the original return address. The `maxactive` field limits how many instances of the probed function may be in flight simultaneously; excess calls are skipped rather than blocked.

2) Probe Registration: The module registers four probes in `tracker_init()` (lines 247–250).

```
if ((err = register_kretprobe(&rp_mmap))) goto fail_mmap;
if ((err = register_kretprobe(&rp_brk))) goto fail_brk;
if ((err = register_kprobe(&kp_munmap))) goto fail_munmap;
if ((err = register_kretprobe(&rp_fork))) goto fail_fork;
```

Fig. 4. Probe registration in `tracker_init()` (`vma_tracker.c`

The registrations follow a cascading cleanup pattern: each `goto` target undoes only the probes that have already been registered successfully, ensuring that a partial initialisation failure leaves the kernel in a consistent state.

- **`rp_mmap`** hooks `vm_mmap_pgoff` with a kretprobe. The handler `mmap_ret()` reads the mapped address from `regs_return_value(regs)`, calls `find_vma()` on the current process's `mm_struct` to retrieve the newly created `vm_area_struct`, extracts its flags and backing file name, and forwards the event to `push_event()`.
- **`rp_brk`** hooks `__x64_sys_brk` with a kretprobe. The handler `brk_ret()` reads the updated heap

bounds directly from `current->mm->start_brk` and `current->mm->brk`, which the kernel has already committed before the return, and emits a `BRK` record.
- **`kp_munmap`** hooks `__x64_sys_munmap` with a plain kprobe (`pre_handler`). This choice is explained in detail below.
- **`rp_fork`** hooks `kernel_clone` with a kretprobe. The handler `fork_ret()` reads the child PID from the return value and emits a `FORK` record, encoding the child PID in the `start` field of the wire format as a convention.

3) kprobe vs. kretprobe: A Timing-Critical Design Choice: The most significant instrumentation decision in the module is the deliberate asymmetry between `mmap` and `munmap`: one uses a kretprobe while the other uses a kprobe. This distinction is dictated entirely by the lifetime of the `vm_area_struct` relative to the syscall boundary.

a) Why kretprobe for mmap: `mmap` allocates a new `vm_area_struct` and inserts it into the process's VMA tree *during* the syscall. At the moment the syscall is entered, the mapping does not yet exist; a pre-handler would find nothing to inspect. Only after `vm_mmap_pgoff` returns is the VMA fully initialised and searchable via `find_vma()`. The kretprobe handler `mmap_ret()` therefore reads the return value (the mapped address) from `rax` through `regs_return_value(regs)`, and then safely calls `find_vma()` to retrieve complete VMA metadata (lines 107–123)(Fig.5)

```c
/* ── kretprobe: mmap ──────────────────────────────────────── */
static int mmap_ret(struct kretprobe_instance *ri, struct pt_regs *regs)
{
    unsigned long addr = regs_return_value(regs);
    struct vm_area_struct *vma;
    char perms[4];
    const char *path = "anon";
    const char *type;

    if (!should_track_current()) return 0;
    if (!current->mm || addr >= TASK_SIZE) return 0;

    vma = find_vma(current->mm, addr);
    if (!vma) return 0;

    perms_str(vma->vm_flags, perms);
    if (vma->vm_file) path = vma->vm_file->f_path.dentry->d_name.name;

    type = classify_vma(vma, current->mm->start_brk, current->mm->brk);
    push_event("MMAP", vma->vm_start, vma->vm_end, perms, path, type);
    return 0;
}

static struct kretprobe rp_mmap = {
    .handler = mmap_ret,
    .kp.symbol_name = "vm_mmap_pgoff",
    .maxactive = 20,
};
```

Fig. 5. `mmap_ret()` kretprobe handler for capturing new VMA allocations.

b) Why kprobe (pre-handler) for munmap.: `munmap` *destroys* the target VMA during the syscall: the kernelcalls `do_munmap()` which unlinks and frees the `vm_area_struct` before returning to user space. If a kretprobe were used, the handler would execute after the VMA had already been freed; calling `find_vma()` at that point would be a use-after-free bug. Instead, the module registers a kprobe with a `pre_handler` that fires *before* `__x64_sys_munmap` executes—while the VMA is still fully valid. The syscall arguments (address and length) are read directly from the x86-64 calling-convention registers `rdi` and `rsi` via `regs->di` and `regs->si` (lines 164–165), and the VMA is looked up while it still exists in the process's address space (lines 168–173)(Fig.6)

```c
static int munmap_pre(struct kprobe *p, struct pt_regs *regs)
{
    unsigned long addr, len;
    struct vm_area_struct *vma;
    char perms[4] = "---";
    const char *path = "none";
    const char *type = "anon";

    if (!should_track_current()) return 0;
    if (!current->mm) return 0;

    addr = regs->di;
    len = regs->si;
    if (!len) return 0;

    vma = find_vma(current->mm, addr);
    if (vma && vma->vm_start <= addr) {
        perms_str(vma->vm_flags, perms);
        if (vma->vm_file) path = vma->vm_file->f_path.dentry->d_name.name;
        type = classify_vma(vma, current->mm->start_brk, current->mm->brk);
    }

    push_event("MUNMAP", addr, addr + len, perms, path, type);
    return 0;
}

static struct kprobe kp_munmap = {
    .pre_handler = munmap_pre,
    .symbol_name = "__x64_sys_munmap",
};
```

Fig. 6. Code Implementation of munmap Instruction

In summary, the choice of probe type is not arbitrary—it is governed by *when* the kernel data structure being observed is valid. Hooking after the return is correct for operations that *create* state; hooking before the call is mandatory for operations that *destroy* state. This principle applies generally to any kernel instrumentation that accesses object metadata through kprobes.

B. Data Transmission: kfifo and debugfs

Once a VMA event is captured by a kprobe or kretprobe handler, it must travel from kernel space to the user-space Python backend and ultimately be broadcast to connected WebSocket clients. This pipeline consists of three distinct stages. First, the handler serialises the event into a fixed-size text record and pushes it into a kernel ring buffer (`kfifo`). Second, a `debugfs` file node exposes the ring buffer to user space through the standard POSIX `read()` interface, with the kernel's Virtual File System (VFS) layer acting as the crossing point between the two privilege domains. Third, the Python `kernel_collector()` coroutine continuously drains the file using non-blocking I/O, reassembles complete lines, parses them into structured event dictionaries, and forwards them to the `VMAStateManager` and WebSocket hub. The overall flow is illustrated in Figure 7.

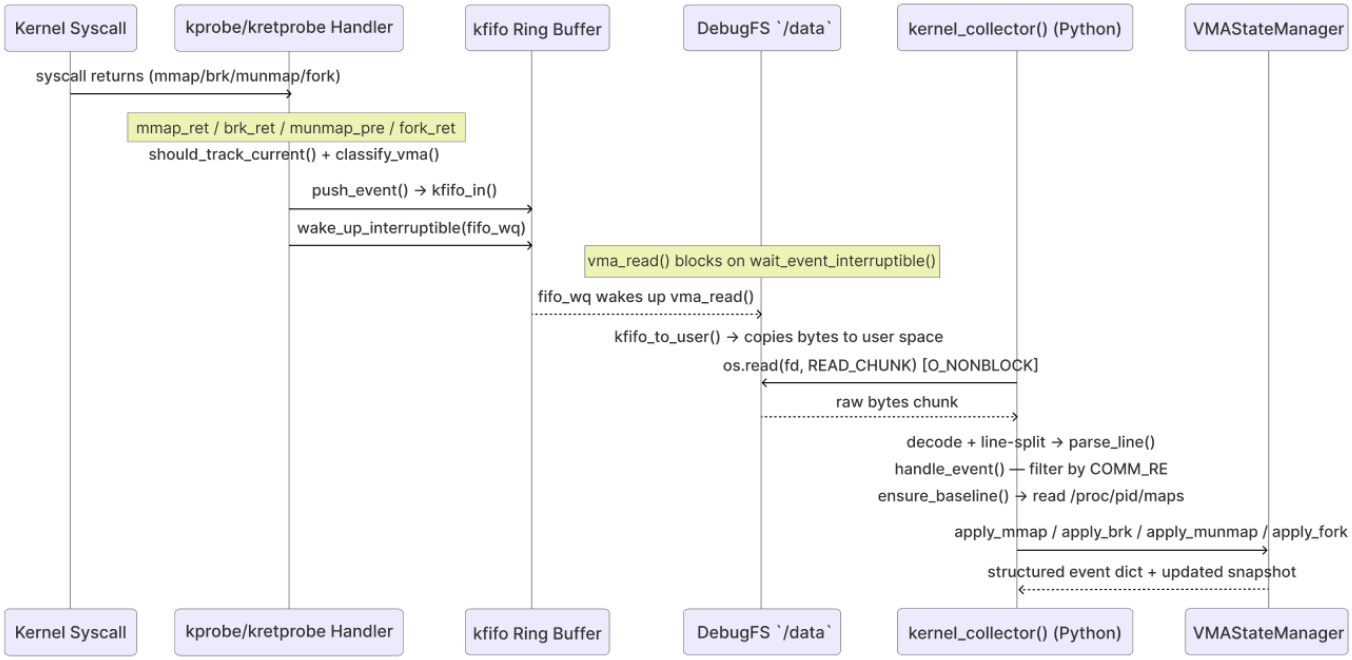

Fig. 7. Swimlane Diagram of Data Transmission Lifecycle

1) Kernel-Side: Ring Buffer and Synchronisation:

a) The `kfifo` Ring Buffer.: All event data is buffered in a statically allocated, 128 KB kernel ring buffer declared as Figure 8 shows.

```
/* ── kfifo ring buffer ──────────────────────────────────────── */
#define FIFO_SIZE (1 << 17)   /* 128 KB ring buffer */
static DECLARE_KFIFO(vma_fifo, char, FIFO_SIZE);
static DEFINE_SPINLOCK(fifo_lock);
static DECLARE_WAIT_QUEUE_HEAD(fifo_wq);
```

Fig. 8. Ring buffer and synchronisation primitives (`vma_tracker.c`, lines 48–51).

`DECLARE_KFIFO` allocates the buffer at compile time, avoiding any `kmalloc` call at runtime, which is critical because probe handlers may execute in atomic context where dynamic allocation is forbidden. The buffer size is a power of two so that the wrap-around index arithmetic reduces to a single bitwise AND, making every push and pop an $O(1)$ operation.

b) Serialisation Format and the `push_event()` Function: `push_event()` is the sole write path into the ring buffer. It formats the event as a pipe-delimited ASCII record using `snprintf`, then atomically enqueues it as Figure 9 shows. Each record includes a monotonic timestamp (in nanoseconds since boot), the process PID and command name, the operation type (e.g., `MMAP`, `MUNMAP`), the start and end virtual addresses, memory permissions, the mapped file path, and the VMA type classification. This fixed-width, line-based format is designed to be both compact and human-readable, making it easy to parse in the user-space backend.

```
static void push_event(const char *op, unsigned long start, unsigned long end,
                       const char *perms, const char *path, const char *type)
{
    char buf[320];
    int len;
    u64 ts = ktime_get_ns();           // monotonic nanoseconds since boot

    len = snprintf(buf, sizeof(buf),
                   "%llu|%d|%s|%s|0x%lx|0x%lx|%s|%s|%s\n",
                   ts, current->pid, current->comm,
                   op, start, end, perms, path, type);

    spin_lock(&fifo_lock);
    if (kfifo_avail(&vma_fifo) >= (unsigned int)len)
        kfifo_in(&vma_fifo, buf, len);   // atomic byte-wise push
    spin_unlock(&fifo_lock);

    wake_up_interruptible(&fifo_wq);    // wake any sleeping reader
}
```

Fig. 9. `push_event()` serialisation and enqueue (`vma_tracker.c`, lines 89–101).

c) Wait Queue: `DECLARE_WAIT_QUEUE_HEAD` initialises the wait queue used to synchronise the producer (probe handlers) with the consumer (`vma_read`).

When a probe handler enqueues a record it calls wake_up_interruptible, which transitions waiting tasks from `TASK_INTERRUPTIBLE` back to `TASK_RUNNING` and schedules them on the next available CPU tick. A `spinlock` (rather than a `mutex`) must be used here because the Linux scheduler forbids sleeping inside a spinlock critical section, and probe handlers—which may run in softirq or NMI context—must never block.

2) The Kernel/User Space Boundary: During module initialisation, `tracker_init()` creates a `debugfs` directory and file node. `debugfs` is a RAM-backed pseudo-filesystem mounted under `/sys/kernel/debug`, intended exclusively for development and diagnostic purposes.

When user space calls `read()` on /sys/ker-

nel/debug/vma_tracker/data, the VFS layer looks up `vma_fops.read` and dispatches to `vma_read()`, transparently bridging a POSIX file-I/O call into the module's ring buffer consumer.

3) User-Side I/O Strategy: The Python backend's `kernel_collector()` coroutine (lines 355–404 of `vma_server.py`) runs as a background `asyncio` task. It opens the `debugfs` file with `O_RDONLY | O_NONBLOCK` on each loop iteration and attempts to read up to `READ_CHUNK` (4 096) bytes.

A single `os.read()` call may return a partial record if the kernel wrote fewer bytes than requested or the chunk boundary falls mid-line. The collector handles this with a classical streaming reassembly pattern: bytes are appended to a string accumulator `buf`, and complete records are extracted by splitting on `'\n'` in a loop, leaving any incomplete trailing fragment in `buf` for the next iteration.

Complete lines are forwarded to `parse_line()` (lines 184–205 of `vma_server.py`), which splits on `'|'` into exactly nine fields and validates the hex address fields with a pre-compiled regular expression before constructing a typed Python dictionary, the function `parse_line` is shown in Fig10.

```python
def parse_line(line: str):
    # Wire format:  ts|pid|comm|op|0xSTART|0xEND|perms|path|type
    parts = line.strip().split("|", 8)
    if len(parts) != 9:
        return None
    ts, pid, comm, op, start, end, perms, path, vtype = parts
    if not (_HEX_RE.match(start) and _HEX_RE.match(end)):
        return None
    try:
        return {
            # Resulting dict:
            "ts": int(ts),
            "pid": int(pid),
            "comm": comm,
            "op": op,
            "start": int(start, 16),
            "end": int(end, 16),
            "perms": perms,
            "path": path if path != "-" else "",
            "type": vtype,
        }
    except ValueError:
        return None
```

Fig. 10. Wire format parsed by `parse_line()`

Any line that does not match the expected format—for example a partial write caused by a race between `kfifo_in` and `kfifo_to_user`—is discarded and counted in the `parse_errors` diagnostic counter, ensuring that malformed data never reaches the state manager.

C. VMA State Machine

Initially, we employed a direct visualization of the raw event stream based on arrival order; however, this approach yielded fragmented and incoherent results, failing to capture the dynamic evolution of memory regions. To address this, we developed a robust state machine that categorizes kernel-side semantics and organizes data at the backend. By maintaining a per-process ordered list of Virtual Memory Areas (VMAs), this system enables precise state management at the Python layer. It efficiently handles incremental updates, including interval fragmentation during `munmap` operations, directional logic for `brk` heap adjustments (expansion or contraction), and the replication of memory mappings during fork events.

More specifically, the backend maintains an in-memory representation of each process's VMAs. The state machine handles:

- MMAP: add new VMA regions.
- BRK: update heap boundaries.
- MUNMAP: remove or split VMAs.
- FORK: clone parent VMA state to child.

This class is described in fig 11

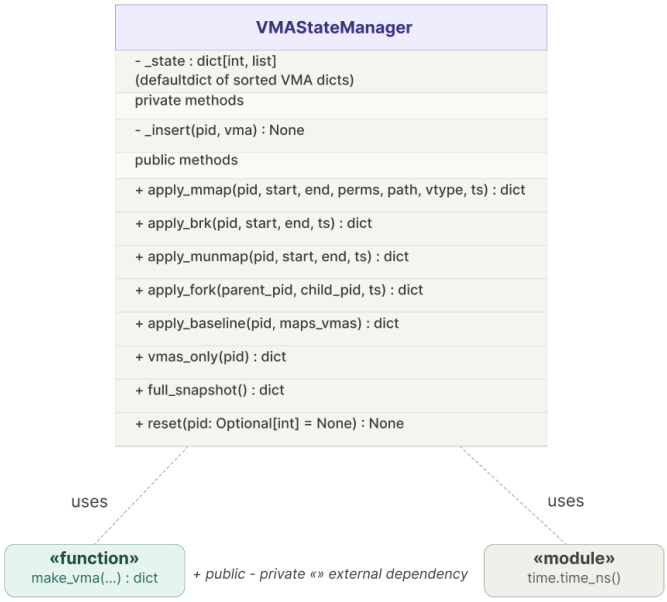

Fig. 11. VMA State Machine Class

The core data structure is encapsulated within the `_state` attribute, which utilizes a `defaultdict(list)` to organize sorted lists of Virtual Memory Area dictionaries, indexed by PID. Internally, the system relies on the private `_insert()` utility method to maintain chronological consistency, ensuring that new VMAs are inserted into their respective PID-indexed lists according to their starting addresses.

The public interface of the handler is categorized into two functional domains:

- **State Modification Operations:** A suite of methods, including `apply_mmap`, `apply_brk`, `apply_munmap`, `apply_fork`, `apply_baseline`, and `reset`—governs the dynamic evolution of the memory state by processing allocation and deallocation events.
- **Read-Only Query Operations:** Diagnostic and retrieval functions such as `vmas_only` (providing a single-process snapshot) and `full_snapshot` (providing a comprehensive system-wide snapshot) facilitate state inspection without side effects.

Furthermore, the implementation leverages critical external dependencies to ensure data integrity and temporal precision. The `make_vma()` factory function is systematically invoked across various `apply` methods to standardize VMA dictionary construction. For temporal logging, `time.time_ns()` is integrated into `vmas_only`, `full_snapshot`, and `apply_baseline` to provide high-resolution timestamps for each captured state.

A critical challenge is partial unmapping, which splits a VMA into up to two fragments. The state machine uses interval overlap checking to clip regions accurately. The code implementation of `apply_munmap` is shown in fig 12.

For every MUNMAP event, `apply_munmap()` walks the current VMA list and checks each entry for overlap with the unmapped range. If there is no overlap, the VMA is kept unchanged. If there is a full overlap, the VMA is simply removed. If there is a partial overlap on the left side — meaning the existing VMA starts before the unmap range — we create a clipped copy ending at the unmap start. If there is a partial overlap on the right side, we create a clipped copy starting at the unmap end. This interval clipping algorithm ensures the user-space view stays byte-accurate with the actual kernel memory layout at all times.

```python
def apply_munmap(self, pid, start, end, ts):
    lst = self._state[pid]
    new_lst, removed = [], []
    for v in lst:
        vs, ve = v["start"], v["end"]
        if ve <= start or vs >= end:
            new_lst.append(v)
            continue
        removed.append(v)
        if vs < start:
            new_lst.append(make_vma(
                                    vs, start, v["perms"], v["path"], v["type"],
                                    pid, ts, "MUNMAP_CLIP"
                                    ))
        if ve > end:
            new_lst.append(make_vma(
                                    end, ve, v["perms"], v["path"], v["type"],
                                    pid, ts, "MUNMAP_CLIP"
                                    ))
    self._state[pid] = sorted(new_lst, key=lambda x: x["start"])
    return {"event": "MUNMAP", "pid": pid, "ts": ts,
            "start": start, "end": end, "removed": removed}
```

Fig. 12. Implementation of `munmap` in the State Machine

D. Process Lifecycle Support

By hooking `kernel_clone`, the system detects fork events and captures the child PID. The backend clones the parent's VMA state and initializes the child process view. The frontend places the child next to the parent for clear visualization of address-space inheritance.

V. Evaluation and Results

The system was deployed and tested on an Alibaba Cloud ECS instance with 8GB RAM running Ubuntu 22.04 LTS. The evaluation focused on correctness, timeliness, and runtime overhead.

Fig. 13. Visualization of a single process's virtual address space.

Figure 13 shows the visualization of a single process's virtual address space, including its stack, heap, mapped libraries, data/bss, and text segments. The layout is consistent with the process's actual memory map as reported by `/proc/[pid]/maps`.

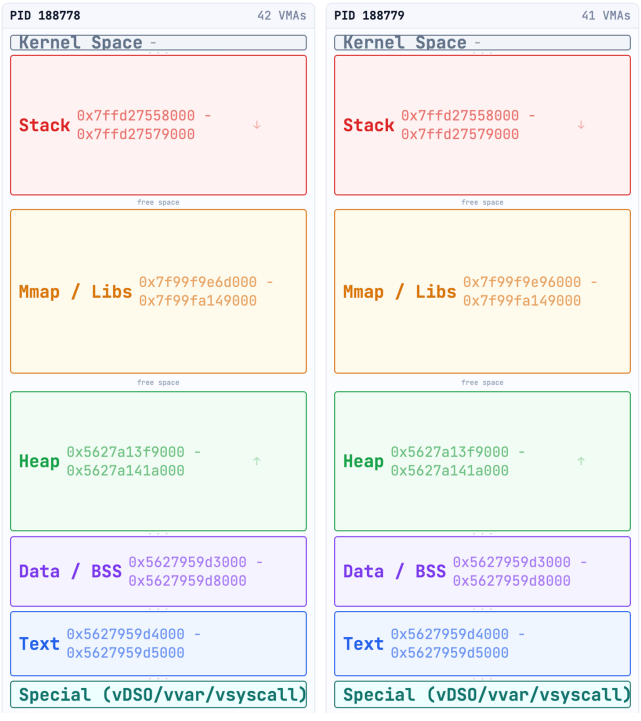

Fig. 14. Side-by-side visualization of a parent and child process after `fork()`.

To demonstrate the support for process lifecycle events, Figure 14 shows a parent process and its child immediately after a `fork()` operation. The child inherits the same memory layout as the parent, which is visualized side-by-side to highlight the address space duplication. The number of VMAs decreases from 42 in the parent to 41 in the child due to copy-on-write and minor internal adjustments, but all major segments remain consistent.

Tests with allocation-intensive programs, fork loops, and partial unmapping confirm that the tool correctly visualizes dynamic memory changes. The kernel module introduces negligible overhead, suitable for continuous monitoring.

VI. Conclusion

This project presents a complete, non-intrusive tool for real-time visualization of Linux process virtual memory mapping. By combining kernel probing, efficient data transmission, a robust state machine, and web-based rendering, the system opens the black box of virtual memory management. The tool supports full process lifecycle tracking and accurately handles complex operations such as partial munmap and VMA splitting.

Future improvements could include physical memory mapping visualization, performance statistics, and support for additional architectures.

Acknowledgment

Thanks for the instructor of CSC5031 Operating Systems Chung Yehching at The Chinese University of Hong Kong (Shenzhen), for his guidance, and thanks for our groupmates for their support and collaboration.