Operating Systems: Memory Virtualization

1 Introduction

This lab investigates the mechanisms behind **memory virtualization** through hands-on experimentation with custom programs designed to reveal how operating systems abstract and manage physical memory resources. These exercises will demonstrate core concepts such as address translation, page tables, and copy-on-write semantics. All required code and resources are hosted in the **mem_virtualization** directory of the operating systems course repository, accessible here: GitHub Repository.

2 Background:

To understand how operating systems virtualize memory resources, we must examine three critical components: memory mapping primitives, low-level data transfer mechanisms, and fault handling infrastructure. This section details the essential APIs and assembly constructs used in the experimental programs.

2.1 mmap/munmap

The `mmap` system call creates virtual memory mappings that bridge user-space addresses with physical resources. Its dual functionality supports both file-backed persistence and anonymous ephemeral allocations:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Key Parameters:

- `addr`: Suggested mapping address (`NULL` lets kernel choose)
- `length`: Mapping size (page-aligned)
- `prot`: Protection flags (`PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`)
- `flags`: Mapping type (`MAP_SHARED` for IPC, `MAP_PRIVATE` for COW, and `MAP_ANONYMOUS` for non-file backed memory)
- `fd`: File descriptor for file-backed mappings

- `offset`: File offset (must be page-aligned)

Anonymous mappings (via `MAP_ANONYMOUS`) create zero-initialized memory regions not tied to files where `fd` should be -1 and `offset` should be 0.
The complementary `munmap` reverses mappings:

```
int munmap(void *addr, size_t length);
```

Mechanism:

- Unmapped pages become invalid immediately
- Kernel automatically releases physical pages
- Address range must match the original mmap call

2.2 rep movsb

The `rep movsb` assembly instruction enables optimized bulk memory transfers using x86 string operations. The wrapper function leverages this hardware capability:

```
void rep_movsb_copy(void *dst, void *src, size_t len) {
    asm volatile (
        "cld\n\t"
        "rep movsb\n\t"
        :
        : "D"(dst), "S"(src), "c"(len)
        : "memory"
    );
}
```

Key Parameters:

- `dst`: Destination pointer (`rdi`, auto-incremented)
- `src`: Source pointer (`rsi`, auto-incremented)
- `len`: Byte count (`rcx`)

Mechanism:

- `cld` clears direction flag for ascending addresses
- `rep` repeats `movsb` `rcx` times

2.3 Signal Handlers

Memory virtualization errors (invalid accesses, permission violations) trigger `SIGSEGV` signals. We intercept these to demonstrate page fault mechanics:

```
struct sigaction sa = {
.sa_flags = SA_SIGINFO,
.sa_sigaction = segv_handler,
};
sigaction(SIGSEGV, &sa, NULL);

void segv_handler(int sig, siginfo_t *info, void *ucontext);
```

Key Parameters:

- `sa_sigaction`: Three-argument handler with fault context
- `SA_SIGINFO`: Flag enabling detailed fault inspection

The handler can inspect `siginfo_t->si_addr` for faulting addresses and `ucontext` for register states, enabling sophisticated fault analysis.

3 Programs

3.1 libmap Integration

The `print_map(void *data)` function reveals virtual-to-physical address mapping by parsing `/proc/self/pagemap`. This requires `sudo` due to kernel-enforced restrictions on physical address disclosure.

3.2 anon_mmap.c

This program demonstrates anonymous memory allocation using `mmap`. It creates a 4KB private mapping not backed by any file, initializes it with a string, then unmaps the memory. The `MAP_ANONYMOUS` flag ensures zero-initialized memory without file I/O overhead, and physical pages are allocated only after the first write (via page fault). Unlike traditional heap allocation, this avoids heap fragmentation for large allocations. It illustrates efficient bulk memory management using direct kernel allocation, which is suitable for security-sensitive or large-block memory operations.

3.3 cow.c

This program analyzes copy-on-write behavior through timed write operations before/after process forking. It measures latency differences between initial writes triggering COW page duplication and subsequent writes to private pages. The first writes suffer from page fault handling, expecting a slow execution, whereas the second writes benefit from pure sequential writes, showing a faster

execution. `print_map` shows identical virtual addresses but different physical addresses post-COW.

3.3.1 Privilege Note:

This program requires `sudo` to access `/proc/self/pagemap` via `libmap`.

3.4 fork_memory.c

This program demonstrates process memory isolation through variable modification in parent/child processes. Despite sharing virtual addresses, physical page frames diverge after writes due to COW. The parent retains the original value while the child sees the local copy.

3.4.1 Privilege Note:

This program requires `sudo` to access `/proc/self/pagemap` via `libmap`.

3.5 idempotent.c

This program uses a `SIGSEGV` handler to implement fault-tolerant memory copying. When `rep_movsb_copy` encounters protected memory, it maps new pages with `mprotect` and re-executes the instruction. You can see the on-demand page allocation during copy operations.

3.6 mmap.c

This program maps file contents into memory using `MAP_SHARED`, enabling direct file manipulation through memory writes. Modifications to the mapped array automatically propagate to disk via kernel flush mechanisms.

3.7 ondemand.c

This program implements demand paging through strategic `SIGSEGV` handling. It reserves 16 pages of virtual address space with `PROT_NONE`, preventing access to that memory space. Then, it immediately unmaps the memory space. When it tries to write some values to the memory space, `SIGSEGV` occurs, and the handler maps the page with the physical address space again. Then, it retries the faulting instruction.

3.8 pthread_memory.c

This program demonstrates thread-shared memory space versus process isolation. Unlike `fork()`, pthreads share virtual address space layout and page table entries. Therefore, thread modifications are visible immediately to all threads, contrasting with process-level COW semantics.

3.8.1 Privilege Note:

This program requires `sudo` to access `/proc/self/pagemap` via `libmap`.

4 Execution:

You can compile these experimental programs by `make`, and run each program: `anon`, `cow`, `fork`, `idempotent`, `mmap`, `ondemand`, and `pthread`.