3. Implement system call

unmap() - Unmaps a memory region

Conceptual Overview

The munmap() system call releases a region of memory previously mapped into a process's address space using mmap(). The kernel must remove the virtual-to-physical mappings for the specified memory region, update the process's memory management data structures, and potentially handle file synchronization if the region was mapped to a file.

1. User-Space Library Function

```
// User-space library function (e.g., munmap wrapper)

int munmap(void *addr, size_t length) {

  int result;


  // Validate arguments (basic sanity checks)

  if (addr == NULL || length == 0) {

    errno = EINVAL;

    return -1;

  }


  // System call number (defined in kernel header)

  int syscall_number = SYS_MUNMAP;  // Hypothetical system call number


  // Trigger system call (architecture-dependent instruction)

  asm volatile (

      "syscall"  // x86-64 instruction

      : "=a" (result)

      : "a" (syscall_number), "D" (addr), "S" (length)
    : "memory", "rcx", "r11"
  );

  if (result < 0) {
    errno = -result; // Set errno from the (negative) return value
    return -1;
  }

  return result;
}
```

2. Kernel-Space System Call Implementation

```
// Kernel-space system call implementation
asmlinkage long sys_munmap(void *addr, size_t length) {
  struct vm_area_struct *vma;
  struct mm_struct *mm;
  unsigned long start = (unsigned long)addr;
  unsigned long end = start + length;

  // 1. Argument Validation
  if (!PAGE_ALIGNED(start) || !PAGE_ALIGNED(end)) {
    return -EINVAL; // Not page-aligned
  }
  if (length == 0) {
    return 0; // Nothing to unmap
  }
```

```
 if (end < start) {

   return -EINVAL; // Invalid address range

 }


 // 2. Get Process's Memory Management Structure

 mm = current->mm;  // current is a pointer to the current process's task_struct


 // 3. Acquire Memory Management Lock

 down_write(&mm->mmap_sem); // Protects against concurrent modifications


 // 4. Find the Virtual Memory Area (VMA)

 vma = find_vma(mm, start);

 if (!vma || vma->vm_start > start) {

   up_write(&mm->mmap_sem);

   return -EINVAL; // Invalid address

 }


 // 5. Check Permissions (Very Important for Security)

 if (!may_munmap(vma, start, length)) { //Function that checks permissions, not given as an example,
permissions depend on application

       up_write(&mm->mmap_sem);

       return -EACCES;

 }


 // 6. Handle Partial Unmapping (Splitting VMAs if Necessary)

 if (start > vma->vm_start) {
    // Split the VMA
    if (split_vma(mm, vma, start) < 0) { //Function that handles splitting the VMA, not given as an example
      up_write(&mm->mmap_sem);
      return -ENOMEM; // Out of memory
    }
    vma = find_vma(mm, start); // Get the right vma now.
  }

  // 7. Unmap the Memory Region (Core Logic)
  do {
    struct vm_area_struct *next_vma;
    unsigned long unmap_end = min(end, vma->vm_end);

    // a. Remove Page Table Entries (PTEs)
    unmap_page_range(mm, vma->vm_start, unmap_end, NULL); //Function that handles PTEs, not given
as an example

    // b. File Synchronization (If Mapped to a File)
    if (vma->vm_file) {
      // Synchronize changes to the file
      filemap_invalidate_lock(vma->vm_file->f_mapping); //Function that handles file sync, not given as
an example
      // Check if there are outstanding dirty pages in the file mapping
        if (mapping_dirty_pages(vma->vm_file->f_mapping)) {
          // If there are, flush those pages to storage
            filemap_flush(vma->vm_file->f_mapping); //Function that handles flushing pages, not given as an
example
```

```
  }
}

// c. Remove the VMA or Adjust its Size

if (unmap_end == vma->vm_end) {
  // Entire VMA is being unmapped
  next_vma = vma->vm_next;
  remove_vma(vma); //Function that removes the VMA, not given as an example
} else {
  // Adjust the VMA's end
  vma->vm_end = unmap_end;
  next_vma = NULL;
}

vma = next_vma;

if(vma){
 if(end > vma->vm_start){
  if(vma->vm_start < end){
   goto loop;
  }
 }
}
```

```
  }while(end > start);


  // 8. Update Memory Statistics (e.g., amount of free memory)


  // 9. Release Memory Management Lock
  up_write(&mm->mmap_sem);


  // 10. Return Success
  return 0;
}
```

Explanation:

1. User Space Wrapper: The munmap() function in user space prepares arguments and calls the kernel through a system call.

2. Argument Validation: The kernel validates the arguments passed from user space, including:

- addr: Must be a valid pointer within the process's address space. It must also be page-aligned.
- length: Must be a positive integer representing the size of the region to unmap.

3. Memory Management Structure: The kernel obtains a pointer to the process's memory management structure (mm_struct). This structure contains information about the process's address space, including a list of virtual memory areas (VMAs).

4. Acquire Memory Management Lock: The kernel acquires a lock on the memory management structure to prevent concurrent modifications from other threads or processes. This is essential for data integrity.

5. Find VMA: The kernel searches for the VMA that contains the specified address range. A VMA represents a contiguous region of virtual memory with specific properties (e.g., read/write permissions, mapping to a file).

6. Check Permissions: The kernel checks that the process has the necessary permissions to unmap the specified memory region. This is a crucial security check.

7. Handle Partial Unmapping:

- If the unmap operation only covers part of a VMA, the kernel may need to split the VMA into smaller regions.

8. Unmap Memory Region (Core Logic): This is the heart of the system call and involves the following:

   - Remove Page Table Entries (PTEs): The kernel iterates through the page tables for the specified memory region and removes the PTEs that map virtual addresses to physical addresses. This effectively removes the mappings. The TLB may also need to be flushed.

   - File Synchronization: If the VMA is mapped to a file (e.g., a shared memory segment or a file opened with mmap), the kernel must synchronize any changes made to the memory region back to the file. This might involve writing dirty pages to disk.

   - Remove VMA or Adjust its Size: Depending on whether the entire VMA is being unmapped or just a portion, the kernel either removes the VMA from the process's memory map or adjusts the VMA's size to reflect the unmapped region.

9. Update Memory Statistics: The kernel updates memory statistics (e.g., amount of free memory, amount of memory used by the process).

10. Release Lock: The kernel releases the lock on the memory management structure.

11. Return Success: The system call returns a success code to user space.

Key Challenges and Security Considerations:

- Memory Management Complexity: Managing memory in an operating system is one of the most complex tasks.

- Page Table Manipulation: Directly manipulating page tables is low-level and requires a deep understanding of the hardware architecture.

- TLB Invalidation: The Translation Lookaside Buffer (TLB) caches recent virtual-to-physical address translations. When PTEs are removed, the TLB must be invalidated to ensure that the CPU doesn't use stale mappings.

- Concurrency: Multiple threads or processes might be trying to access or modify the same memory region concurrently. Proper synchronization is essential to prevent race conditions and data corruption.

- Security Vulnerabilities: Incorrectly implemented munmap() can lead to security vulnerabilities, such as:

  - Use-After-Free: If a process continues to use a memory region after it has been unmapped, it can lead to unpredictable behavior or security exploits.

  - Double-Free: If a memory region is unmapped twice, it can corrupt the kernel's memory management data structures.

- Privilege Escalation: If a process can unmap memory regions that it doesn't have permission to access, it can potentially gain unauthorized access to sensitive data or resources.
- Error Handling: All operations in the kernel must be carefully checked for errors. If an error occurs, the system call must return an appropriate error code to user space.
- Kernel Modules: System calls should be used with extreme caution. Improper manipulation of system memory can result in system corruption.

How to (Attempt) Implementing (Simplified for Linux):

Adding a new system call to Linux involves the following steps. Always back up your data.

1. Get the Kernel Source: Ensure you have the Linux kernel source code matching your running kernel version.
2. Define the System Call Number: In arch/x86/include/asm/unistd_64.h, add your SYS_MUNMAP to the list.
3. Implement the System Call: Create a source file (kernel/sys_munmap.c) with the function implementation outlined above.
4. Update the System Call Table: Add an entry for your function in arch/x86/entry/syscalls/syscall_64.tbl.
5. Update the Makefile: Update the Makefile in the kernel/ directory to include your new source file.
6. Recompile the Kernel: Recompile and install the new kernel.
7. Write a User-Space Test Program: Create a user-space program to call your munmap() system call.
8. Test and Debug: Thoroughly test your system call.