Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 15: User vs. Kernel Space

Prof. Ryan Huang

Project 3

Process view:

- Every process has an address space starting from VM_ARENA_BASEADDR of size VM_ARENA_SIZE
- When a process starts, entire address space is invalid
- Process calls vm_map to make pages valid
- Process may load or store any valid page

Pager view:

- One process runs at a time
- Pager maintains page table, which MMU uses for translation
- MMU and pager handle vm_create, vm_map, and vm_fault to provide processes with the illusion that all pages are always readable/writable

Project 3

Swap-backed pages:

- Private to a process
- Data lifetime = process lifetime

File-backed pages:

- Shared among processes
- Data lifetime = ∞

Project 3 advice

Design all data structures before writing any code

- Can it support sharing of physical resources?

Draw state diagrams before writing any code

- Separate (but similar) diagrams for swap-backed, file-backed pages
- Don't translate the state machine directly into code (will lead to lots of redundant code)

Write and test code incrementally

- Start with swap-backed pages
- Add file-backed pages
- Add fork

Project 3 advice

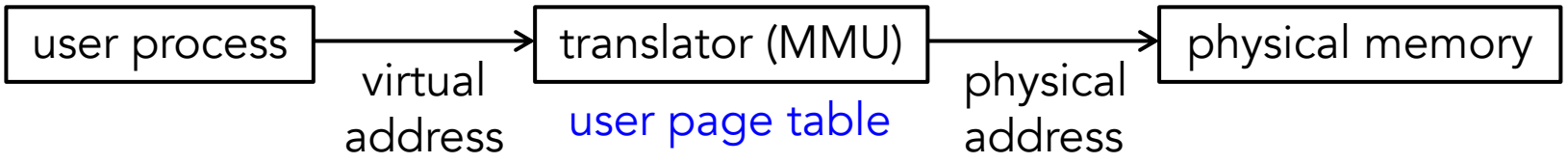
Test cases

- Enact a path through your state machine
- Some edges require multiple processes (e.g., one process evicts another process's page)
- All groups should test fork of non-empty arena

Assertions

- Check that all PTEs are consistent with state of all virtual pages (esp. virtual pages that share a physical resource)
- Check that swap blocks are accounted for correctly

User and kernel address spaces



Each process has its own address space

- Kernel sets up page table per process
- MMU looks up current page table (pointed to by PTBR) to translate any virtual address to a physical memory address (or trap to OS)

What about kernel's address space? Are kernel loads/stores translated?

- Kernel is like (but not exactly) a privileged process

Interlude: where to store page tables?

In physical or virtual memory?

- I.e., does PTBR, etc. contain a physical or virtual address?

Option 1: store page tables in physical memory

- Most common choice
  - x86: CR3 stores the physical address of the page directory

Option 2: store page tables in virtual memory

- Benefits?
- Problems?
- Solution?
- Project 3: page tables for user processes are stored in kernel's address space. Kernel's address space is managed by infrastructure.

Kernel vs. user address spaces

Kernel address space is mostly the same as a user address space, with a few key differences

1. Can you evict the kernel's virtual pages?
   - Some pages are "pinned" to physical memory

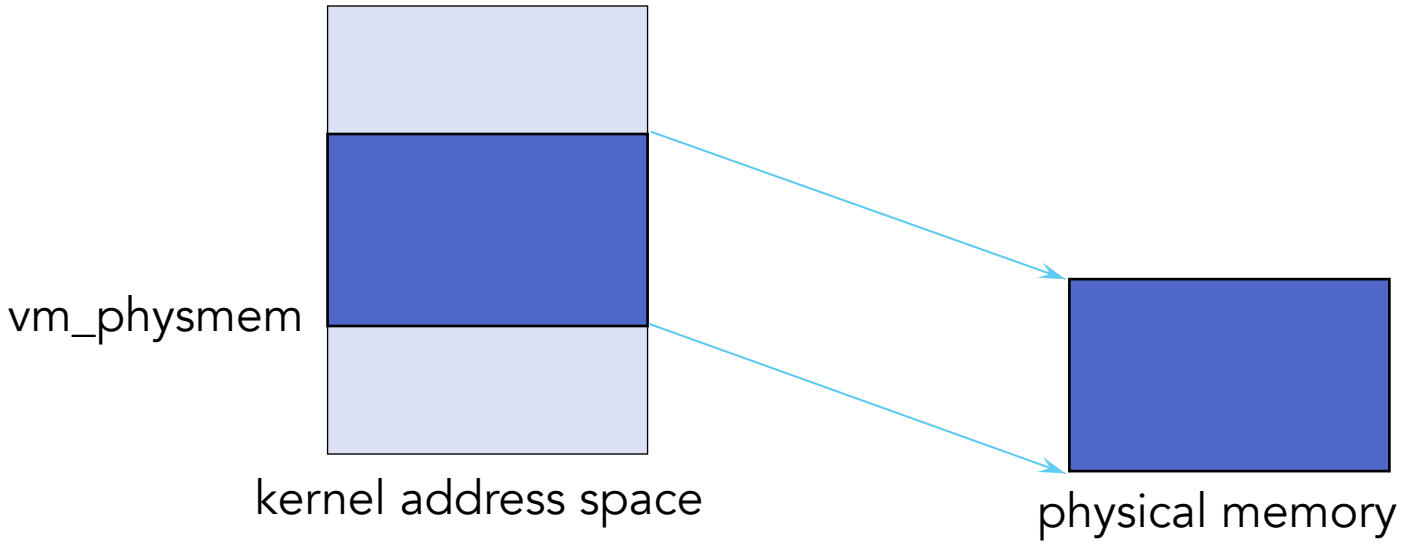
2. Sometimes, the kernel must access specific physical memory addresses
   - When?

How does kernel access physical mem?

Option 1: Issue an untranslated address (i.e., bypass the MMU)

Option 2: Create a mapping from virtual address range to physical memory range

- Project 3: vm_physmem[n] maps to byte n of physical memory
- Problems?



Kernel vs. user address spaces

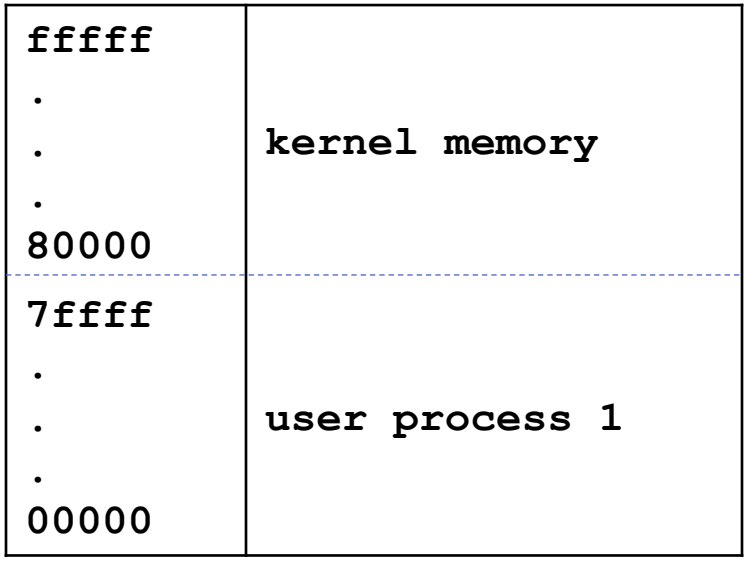
3. How does kernel access another address space?

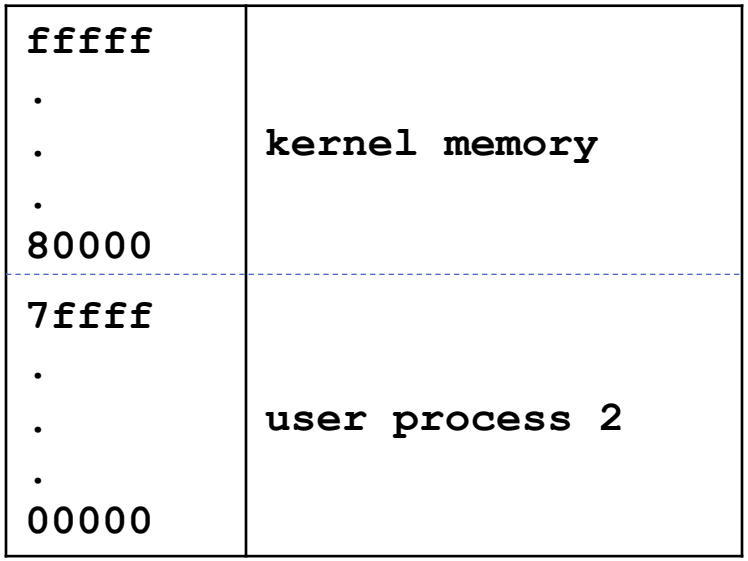
Method A: find the data in physical memory/disk

Kernel vs. user address spaces

3. How does kernel access another address space?

Method B: Map kernel address space into every process's address space, then issue virtual address





i.e., kernel does not have a dedicated, separate address space

Protection: kernel/user mode

How are we protecting a process's address space from other processes?

Who is allowed to modify the translation data?

How does CPU know kernel is running?

- Hardware support: mode bit

Who is allowed to change the mode bit?

- Kernel?
- User?
- Problem?

How can user process safely switch to kernel mode?

Faults and interrupts

- Why are these safe to transfer control to kernel?

System calls

- Process management: fork()/exec()
- Memory management: mmap
- I/O: open(), close(), read(), write()
- System management: reboot()
- …

System calls

When you call `cin >>` in your C++ program:

- `cin` calls **`read()`**, which executes assembly-language instruction **`syscall`**
- **`syscall`** traps to kernel at pre-specified location
- kernel's syscall handler calls kernel's `read()`

To handle trap to kernel, hardware atomically

- Sets mode bit to kernel
- Saves registers, PC, SP
- Changes SP to kernel stack
- Changes to kernel's address space*
- Jumps to exception handler
- Sound familiar?

*: not needed for designs where the kernel address space is mapped in every process' address space

Passing arguments to system calls

Which address space (user process's or kernel's) should be used to store the arguments?

Kernel must carefully check validity of arguments

- E.g., `read(int fd, void *buf, size_t size)`
  - Is fd a valid descriptor for an open file?
  - Are all addresses in [buf,buf+size) valid?
  - Are all addresses in [buf,buf+size) writable?

Protection summary

Safe to switch from user to kernel mode because control only transferred to certain locations

- Where are these locations stored?
- Who can modify interrupt vector table?

Aside: do administrative processes run in kernel mode or user mode?