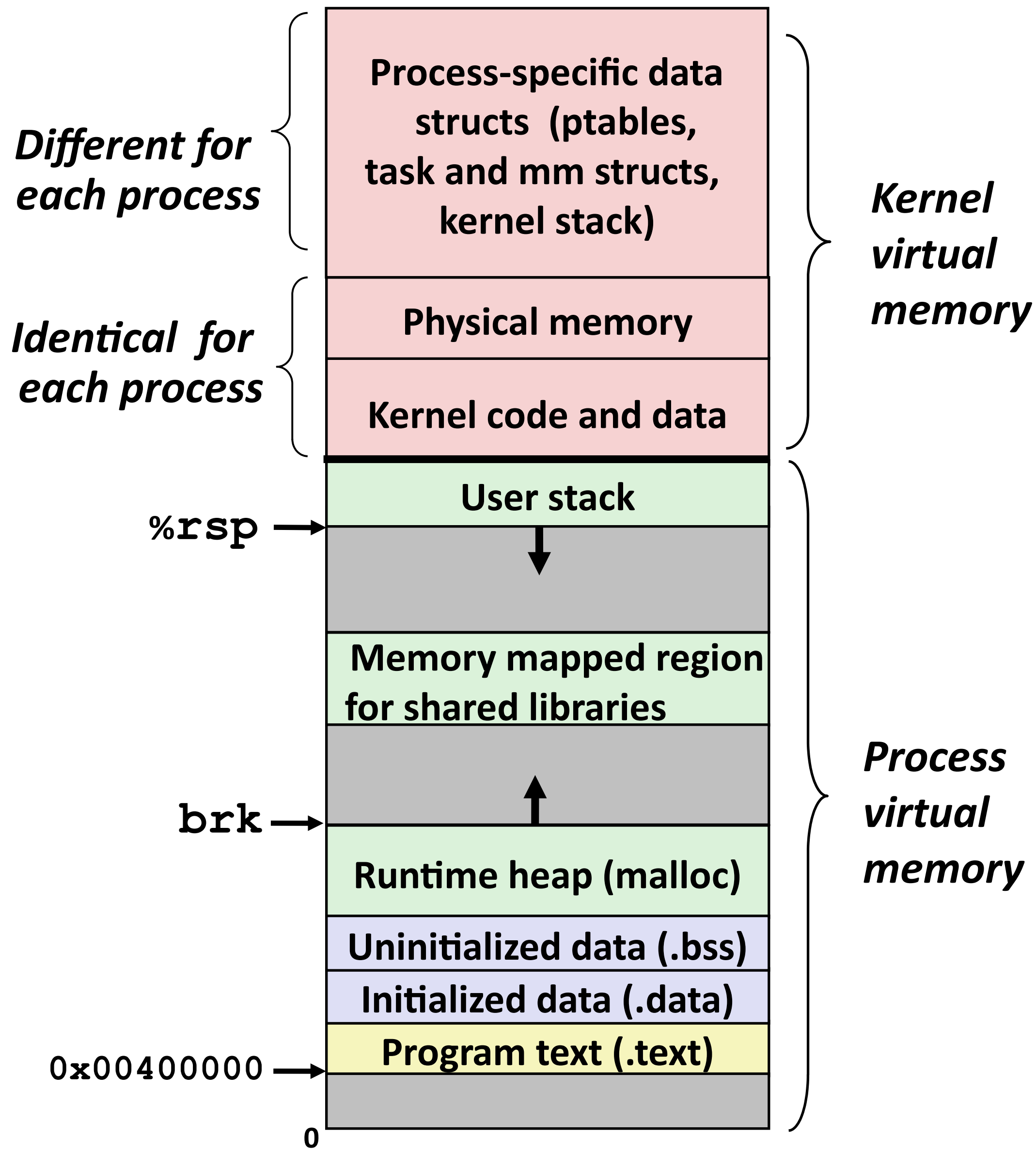
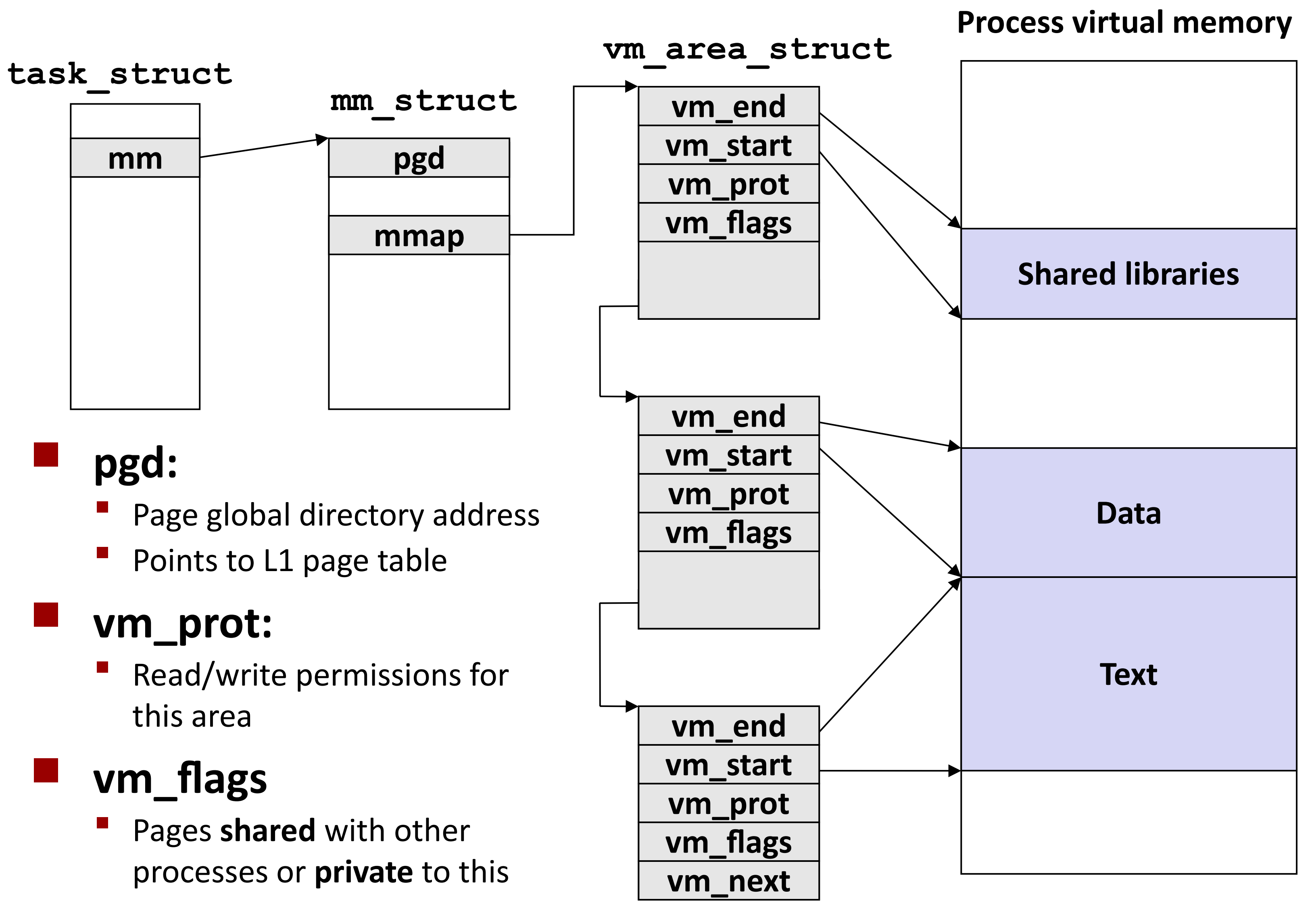
Virtual Address Space of a Linux Process



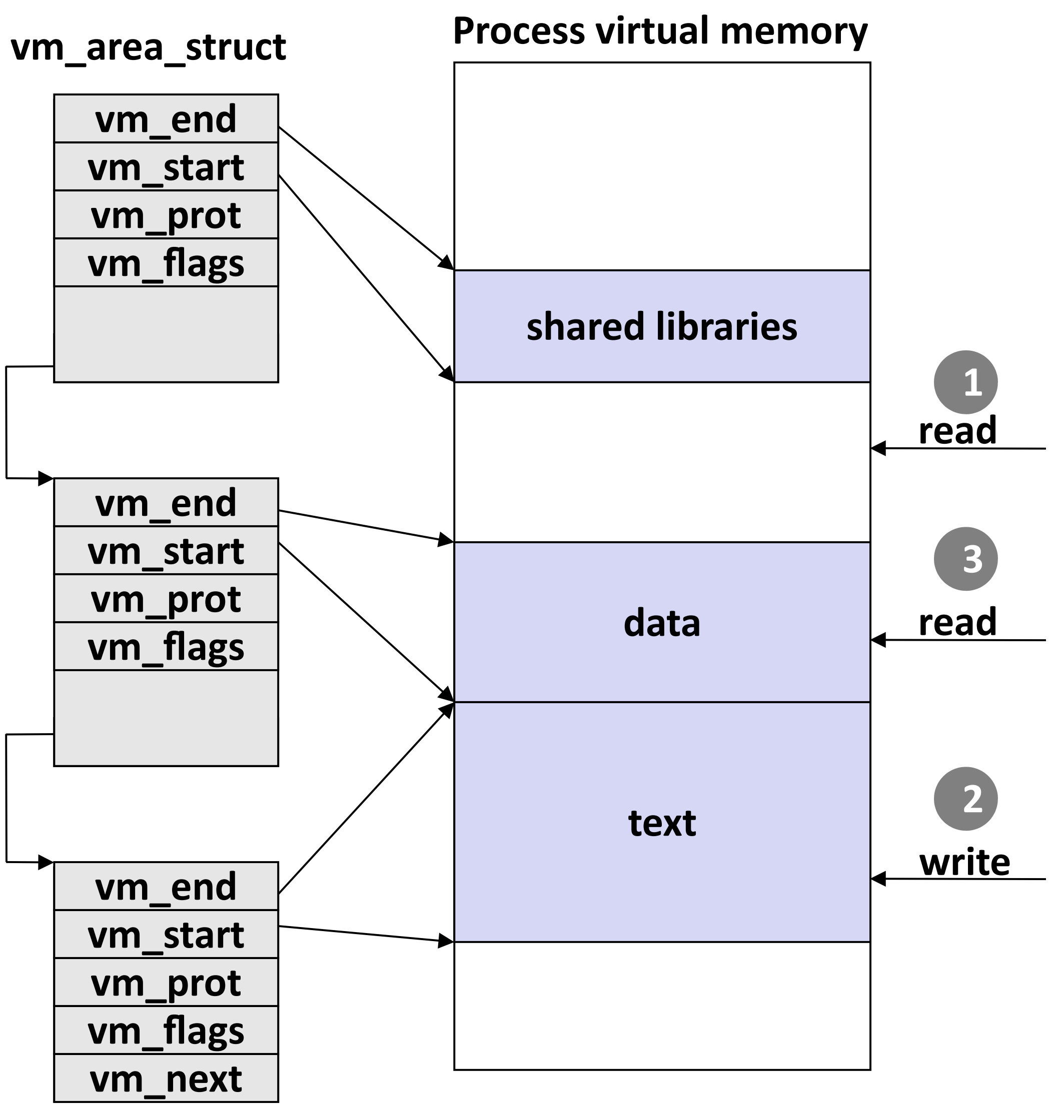
Linux Organizes VM as Collection of "Areas"



- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process

Each process has own `task_struct`, etc

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

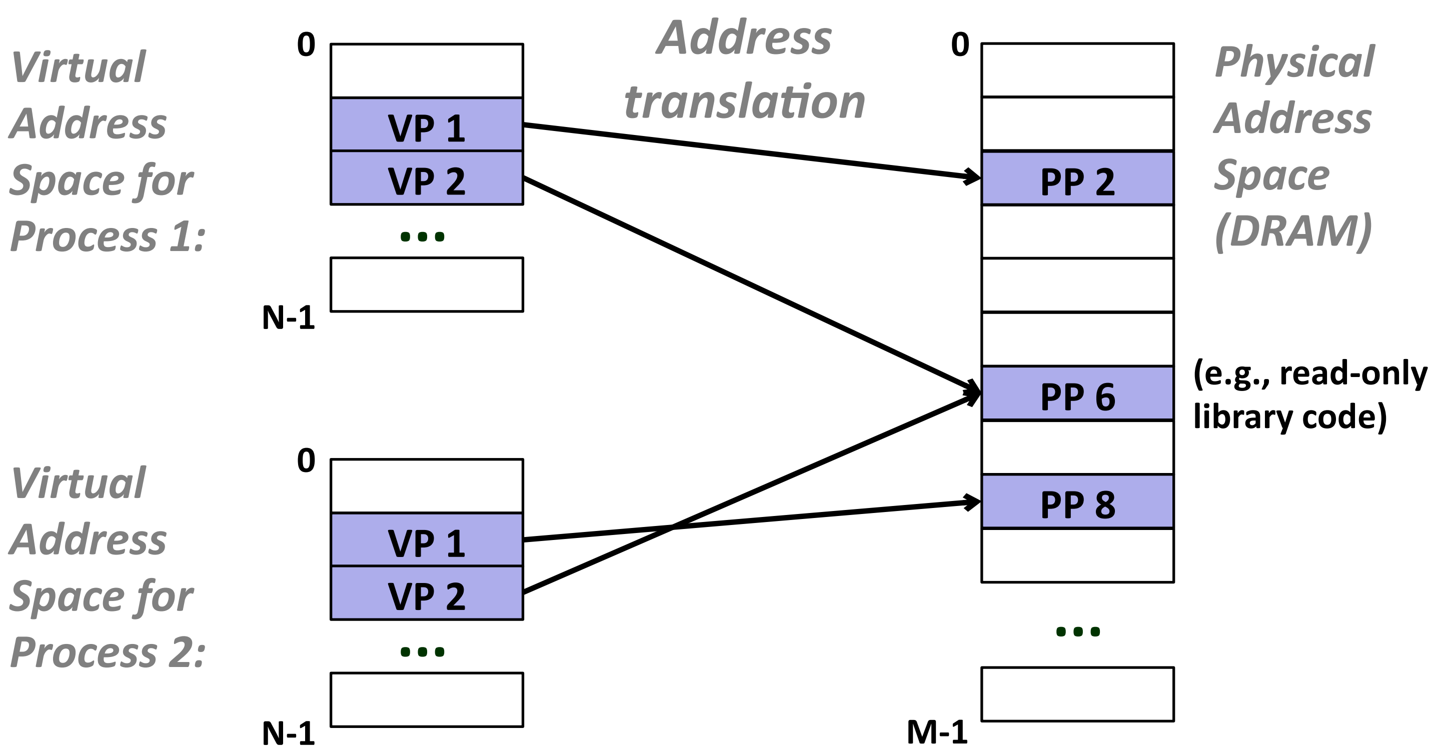
Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Memory Mapping

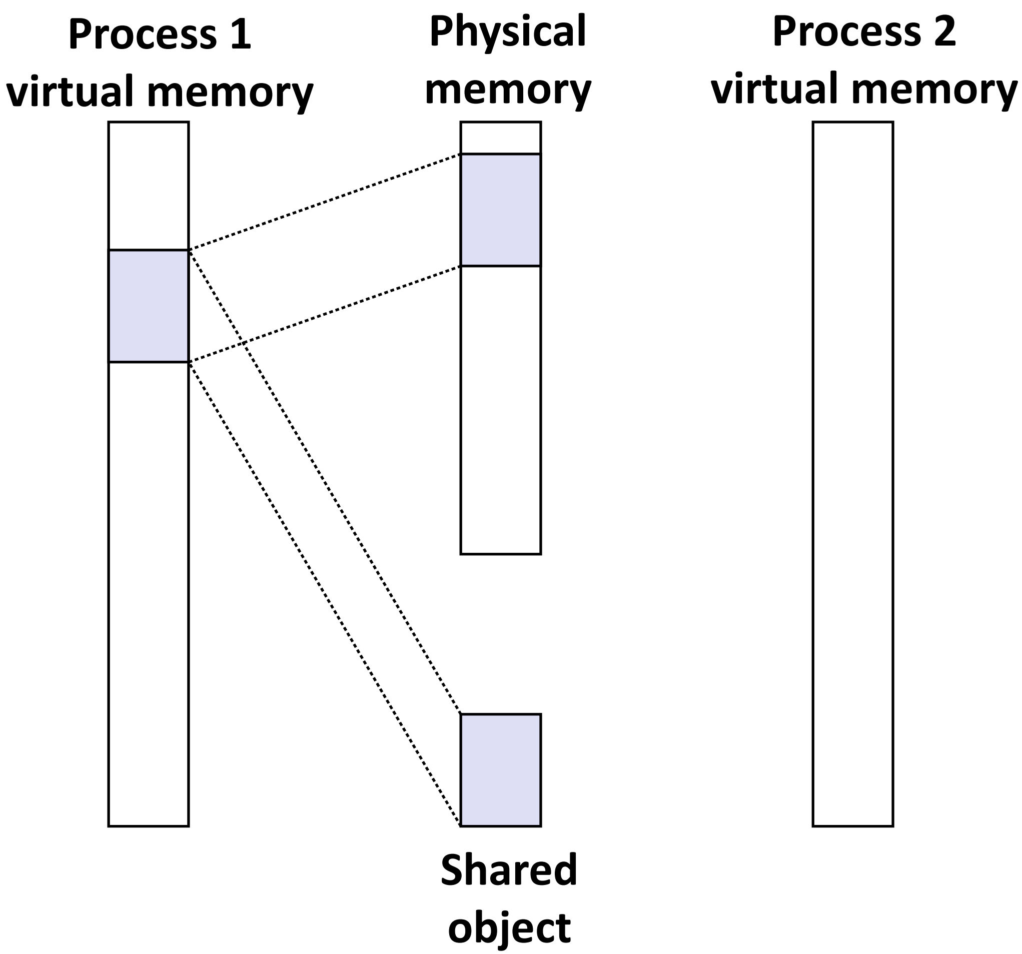
- **VM areas initialized by associating them with disk objects.**
  - Called ***memory mapping***
- **Area can be *backed by* (i.e., get its initial values from) :**
  - ***Regular file*** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - ***Anonymous file*** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (***demand-zero page***)
    - Once the page is written to (***dirtied***), it is like any other page
- **Dirty pages are copied back and forth between memory and a special *swap file*.**

Review: Memory Management & Protection

- **Code and data can be isolated or shared among processes**

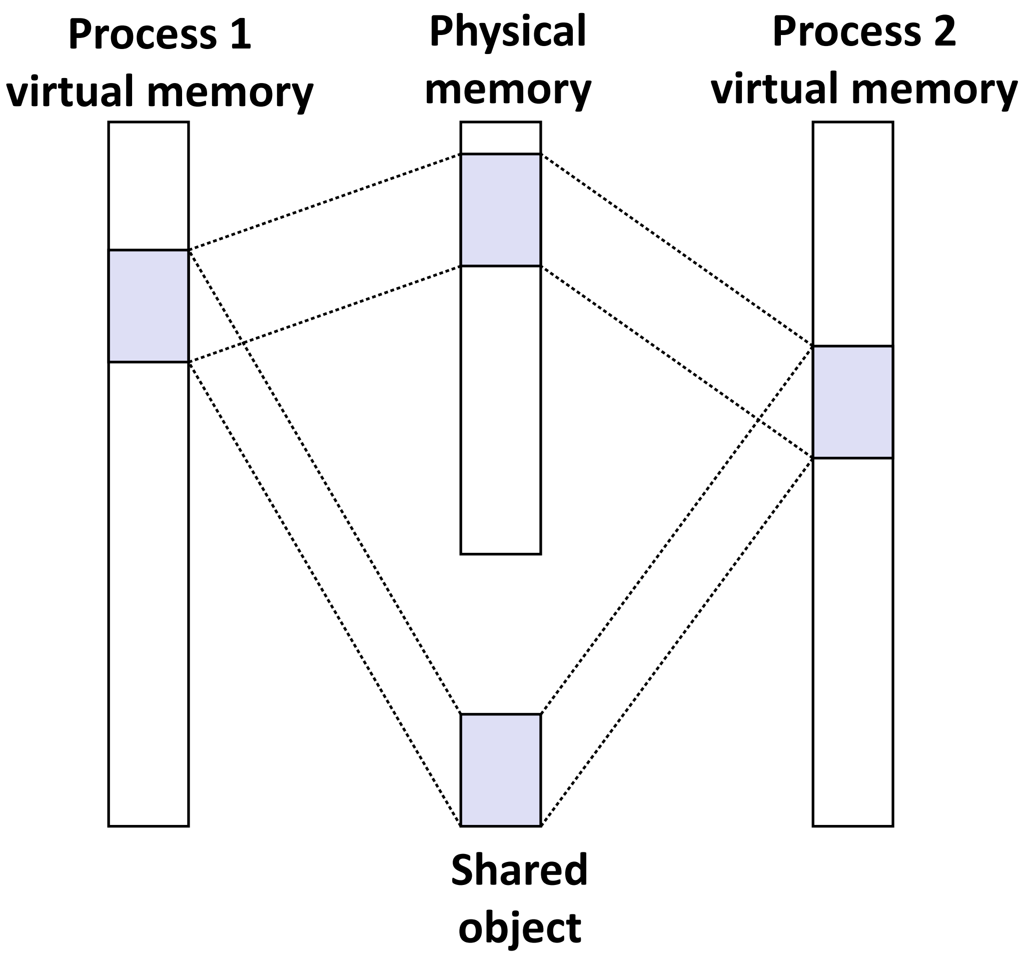


Sharing Revisited: Shared Objects



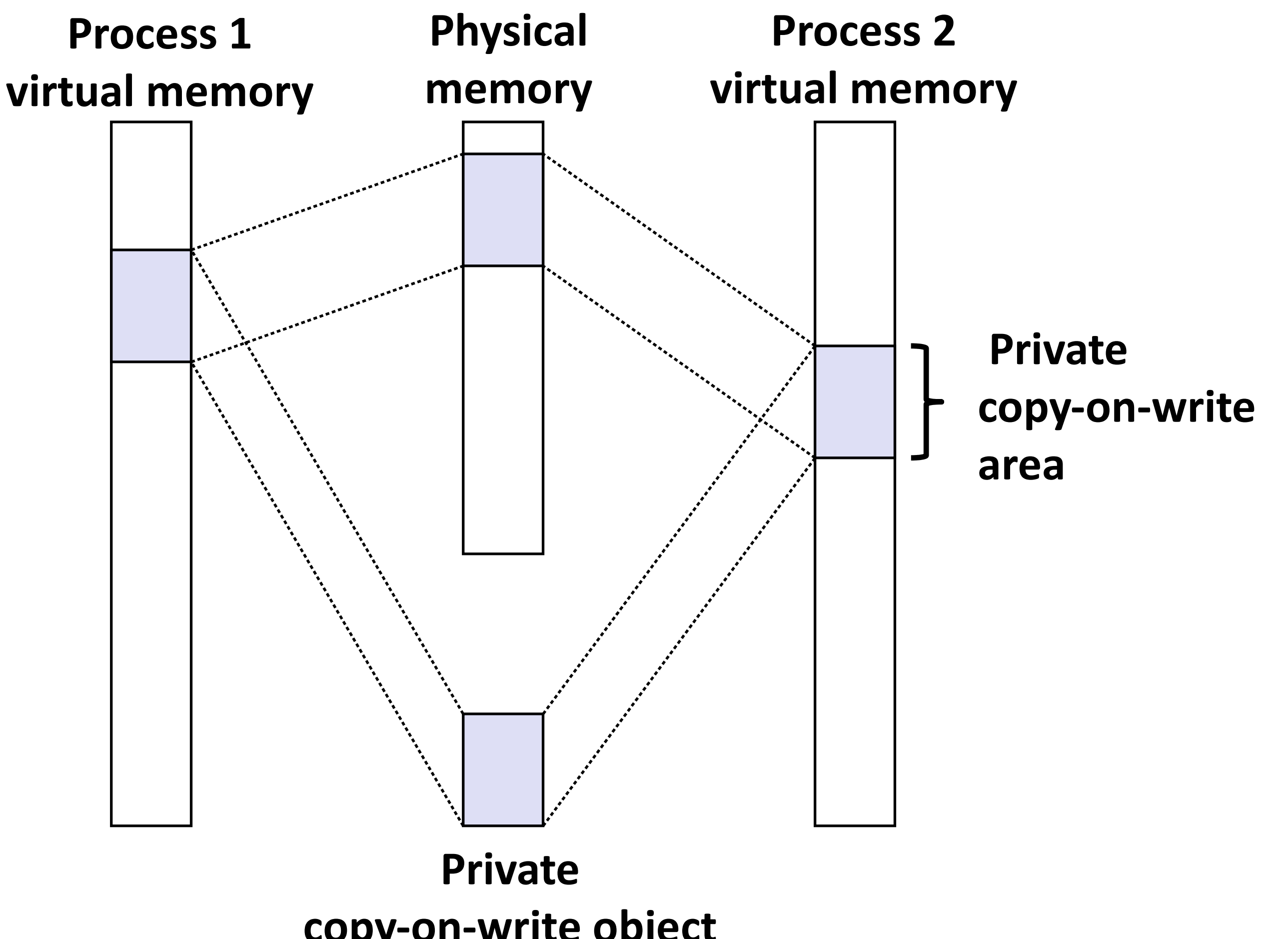
- Process 1 maps the shared object (on disk).

Sharing Revisited: Shared Objects



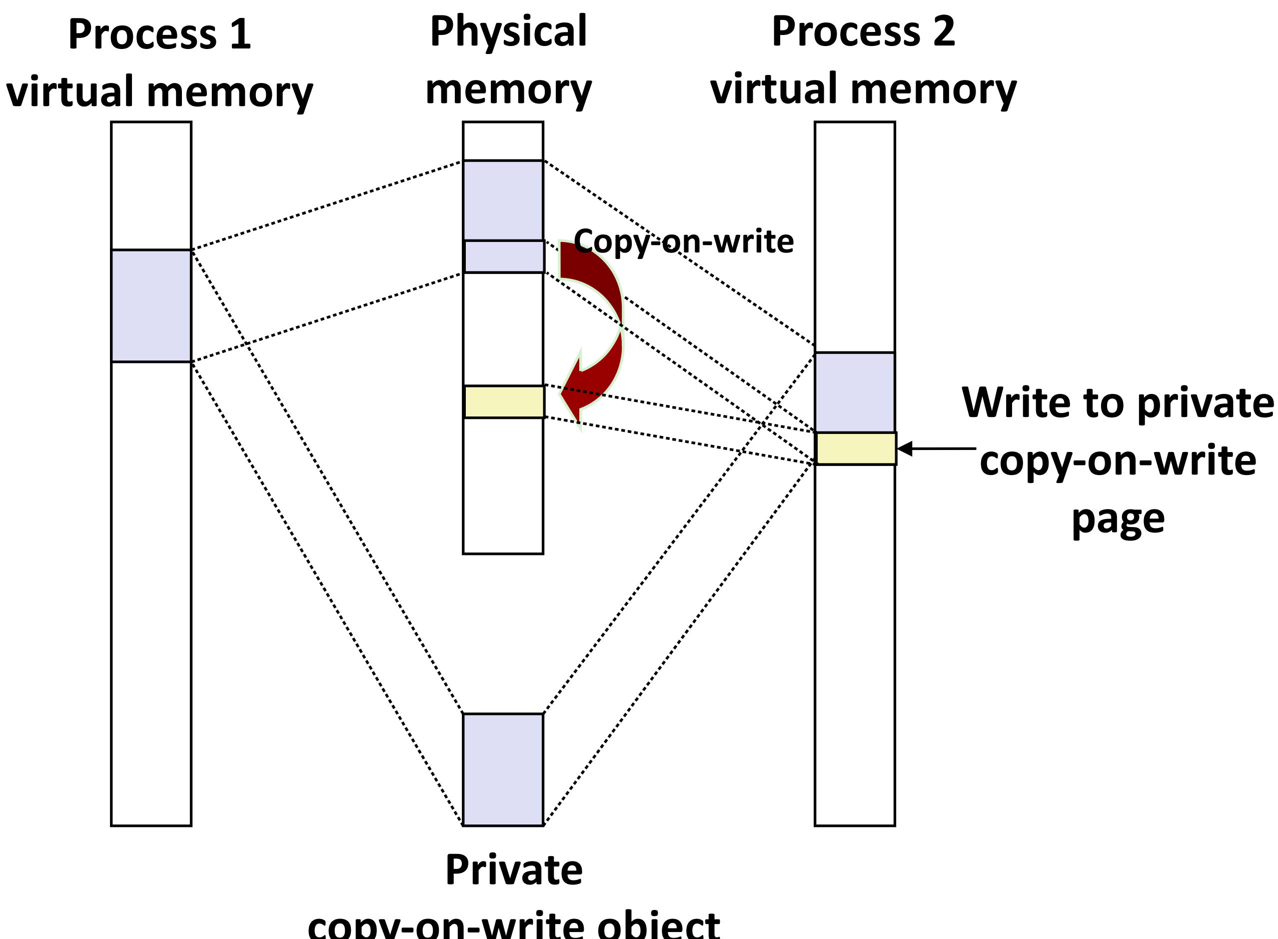
- Process 2 maps the same shared object.
- Notice how the virtual addresses can be different.
- But, difference must be multiple of page size

Sharing Revisited:
Private Copy-on-write (COW) Objects



- Two processes mapping a ***private copy-on-write (COW)*** object
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

Sharing Revisited: Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

The `fork` Function Revisited

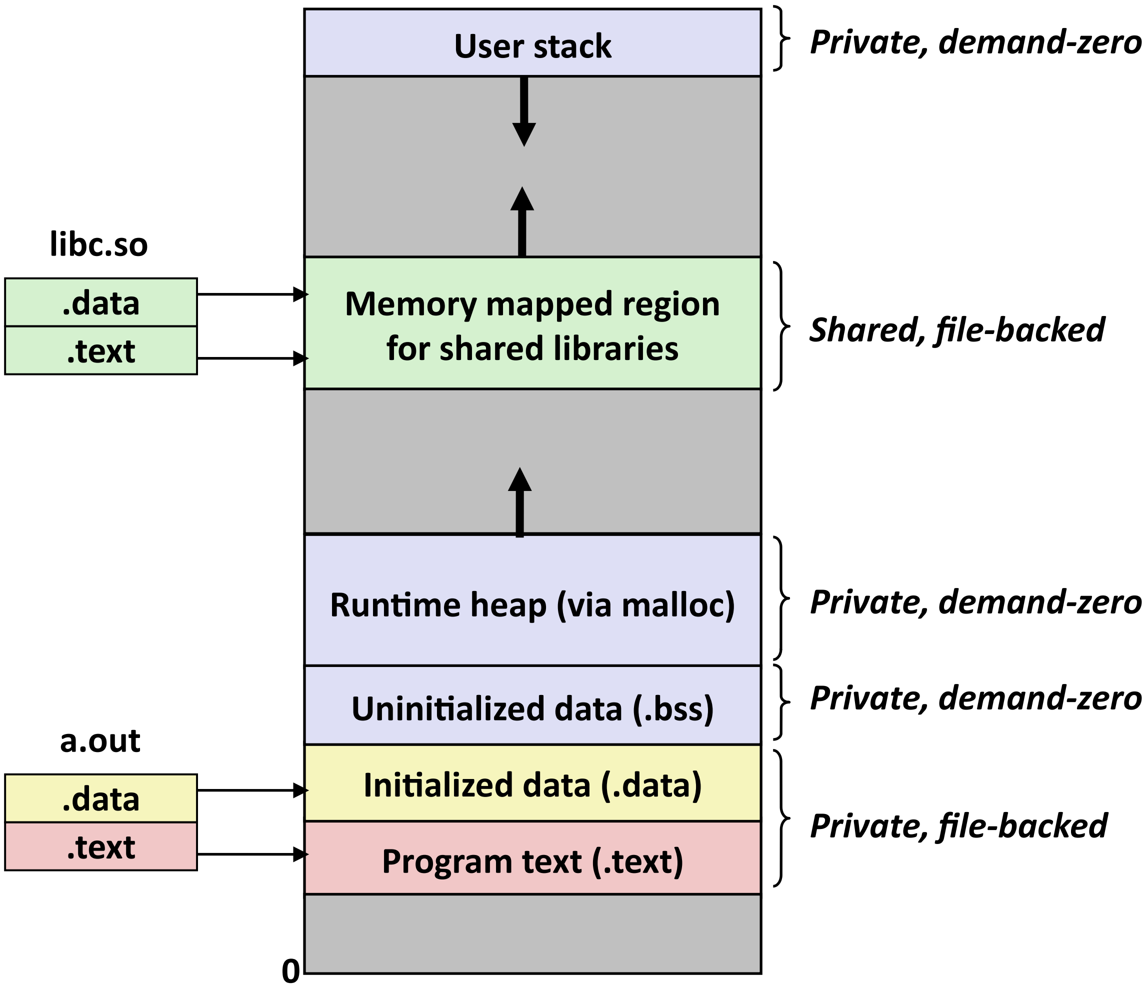
- **VM and memory mapping explain how `fork` provides private address space for each process.**

- **To create virtual address for new process:**
  - Create exact copies of current **`mm_struct`**, **`vm_area_struct`**, and page tables.
  - Flag each page in both processes as read-only
  - Flag each **`vm_area_struct`** in both processes as private COW

- **On return, each process has exact copy of virtual memory.**

- **Subsequent writes create new pages using COW mechanism.**

The execve Function Revisited



- **To load and run a new program `a.out` in the current process using `execve`:**

- **Free `vm_area_struct`'s and page tables for old areas**

- **Create `vm_area_struct`'s and page tables for new areas**
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files.

- **Set PC to entry point in `.text`**
  - Linux will fault in code and data pages as needed.

Finding More Shareable Pages

- **Easy places to identify shareable pages**
  - Child create via **`fork`**
  - Processes loading the same binary file
    - E.g., bash or python interpreters, web browsers, ...
  - Processes loading the same library file
- **What about others?**
  - Kernel Same-Page Merging
  - OS scans through all of physical memory,
    looking for duplicate pages
  - When found, merge into single copy, marked as copy-on-write
  - Implemented in Linux kernel in 2009
  - Limited to pages marked as likely candidates
  - Especially useful when processor running many virtual machines

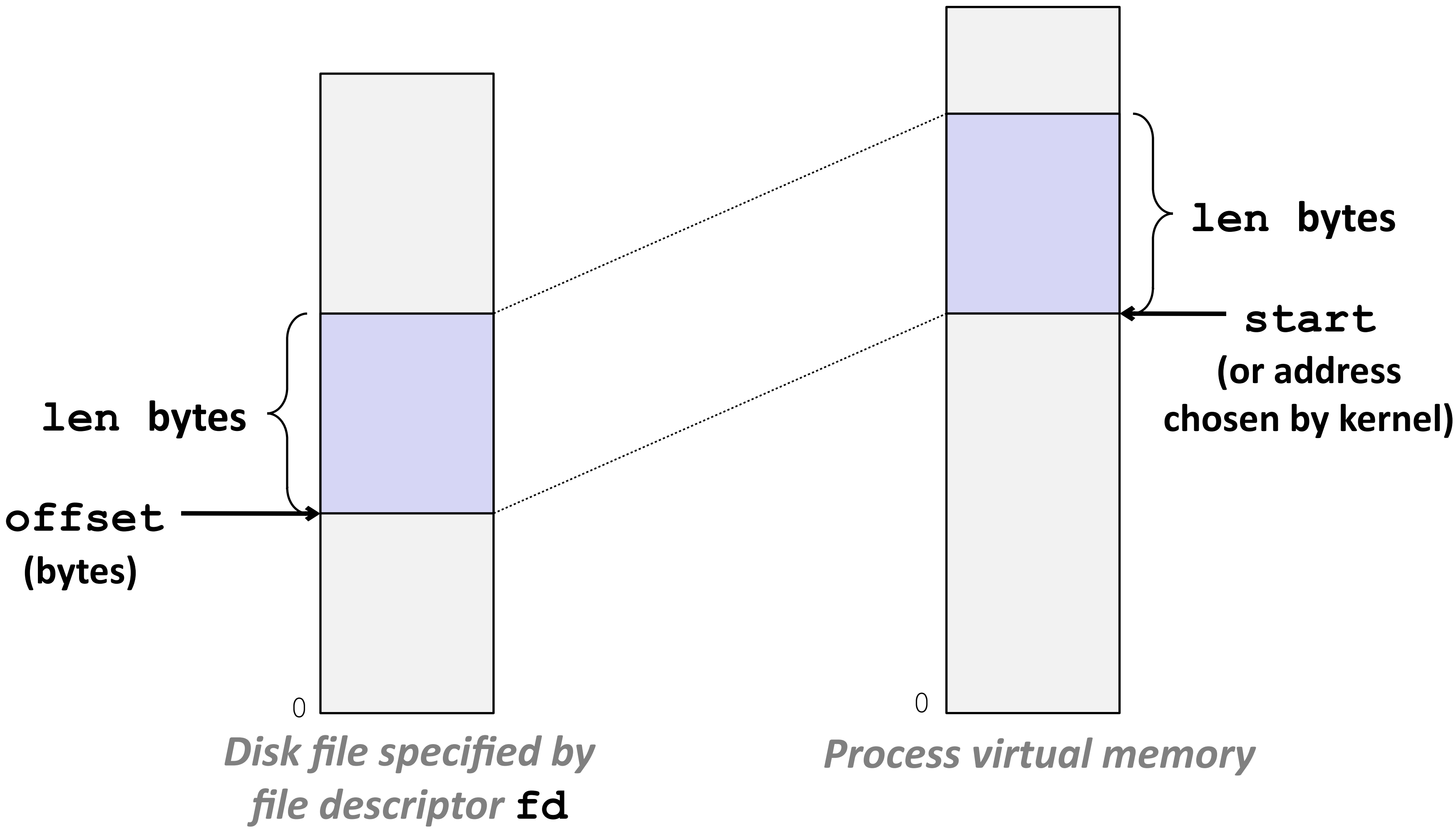
User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- **Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`**
  - `start:` may be 0 for "pick an address"
  - `prot`: PROT_READ, PROT_WRITE, PROT_EXEC, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- **Return a pointer to start of mapped area (may not be `start`)**

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Example: Using `mmap` to Copy Files

- Copying a file to `stdout` without transferring data to user space
  - This code does not meet our coding standards.

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size,
                PROT_READ,
                MAP_PRIVATE,
                fd, 0);
    write(STDOUT_FILENO,
          bufp, size);
    return;
}
```

mmapcopy.c

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
               argv[0]);
        exit(0);
    }

    /* Copy input file to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

mmapcopy.c

Some Uses of mmap

- **Reading big files**
  - Uses paging mechanism to bring files into memory
- **Shared data structures**
  - When call with **`MAP_SHARED`** flag
    - Multiple processes have access to same region of memory
    - Risky!
- **File-based data structures**
  - E.g., database
  - Give **`prot`** argument **`PROT_READ | PROT_WRITE`**
  - When unmap region, file will be updated via write-back
  - Can implement load from file / update / write back to file

Summary

- **VM requires hardware support**
  - Exception handling mechanism
  - TLB
  - Various control registers
- **VM requires OS support**
  - Managing page tables
  - Implementing page replacement policies
  - Managing file system
- **VM enables many capabilities**
  - Loading programs from memory
  - Forking processes
  - Providing memory protection