



CS3281 / CS5281

# Advanced Virtual Memory

CS3281 / CS5281

Spring 2026

*\*Some lecture slides borrowed and adapted from CMU's  
"Computer Systems: A Programmer's Perspective"  
and MIT's 6.S081 Course*



Tel (615) 343-7472 | Fax (615) 343-7440  
1025 16th Avenue South Nashville, TN 37212  
[www.isis.vanderbilt.edu](http://www.isis.vanderbilt.edu)



# Today

- Simple memory system example
- Memory mapping

# Review of Symbols

- Basic Parameters
  - **N** =  $2^n$  : Number of addresses in virtual address space
  - **M** =  $2^m$  : Number of addresses in physical address space
  - **P** =  $2^p$  : Page size (bytes)
- Components of the virtual address (VA)
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN**: Physical page number

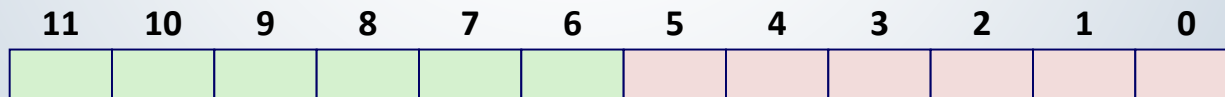
# Simple Memory System Example

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



Virtual Page Number

Virtual Page Offset



Physical Page Number

Physical Page Offset

# Simply Memory System Page Table

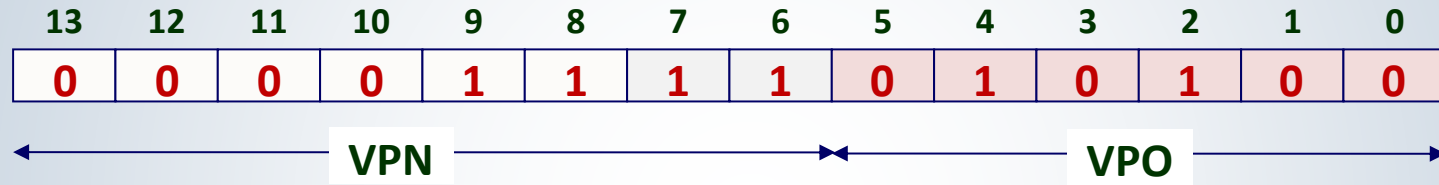
- Only show first 16 entries (out of )

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1
06	—	0
07	—	0

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	—	0
0C	—	0
0D	2D	1
0E	11	1
0F	0D	1

# Address Translation Example #1

Virtual Address: 0x03D4



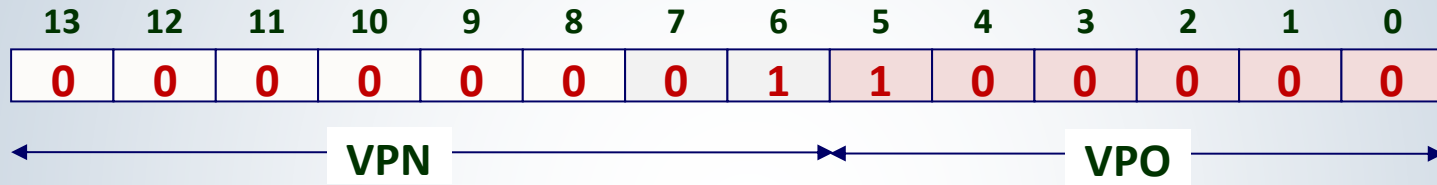
VPN:

Page Fault:

PPN:

# Address Translation Example #2

Virtual Address: 0x0060



VPN: 0x01

Page Fault: Y

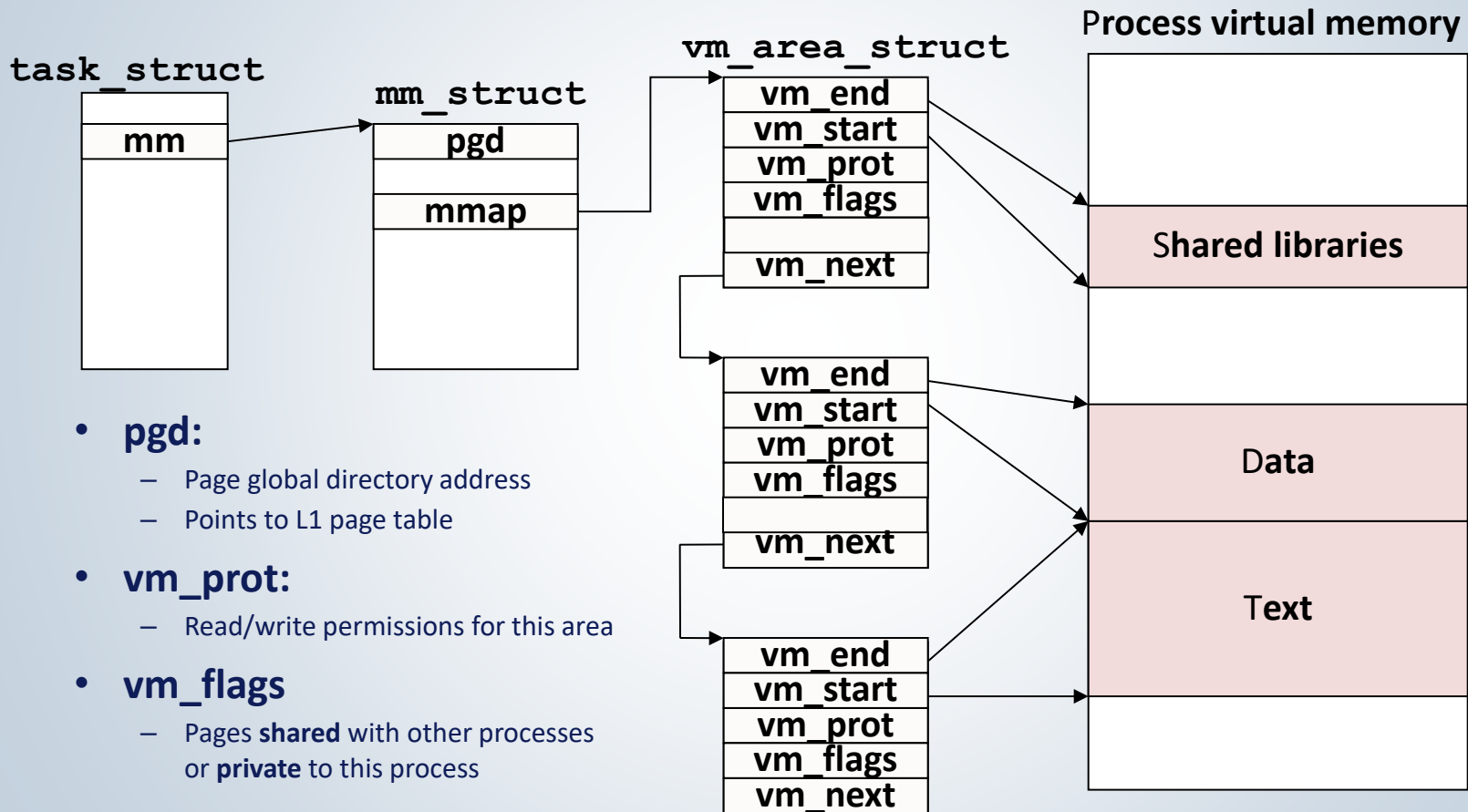
PPN:

# Segment

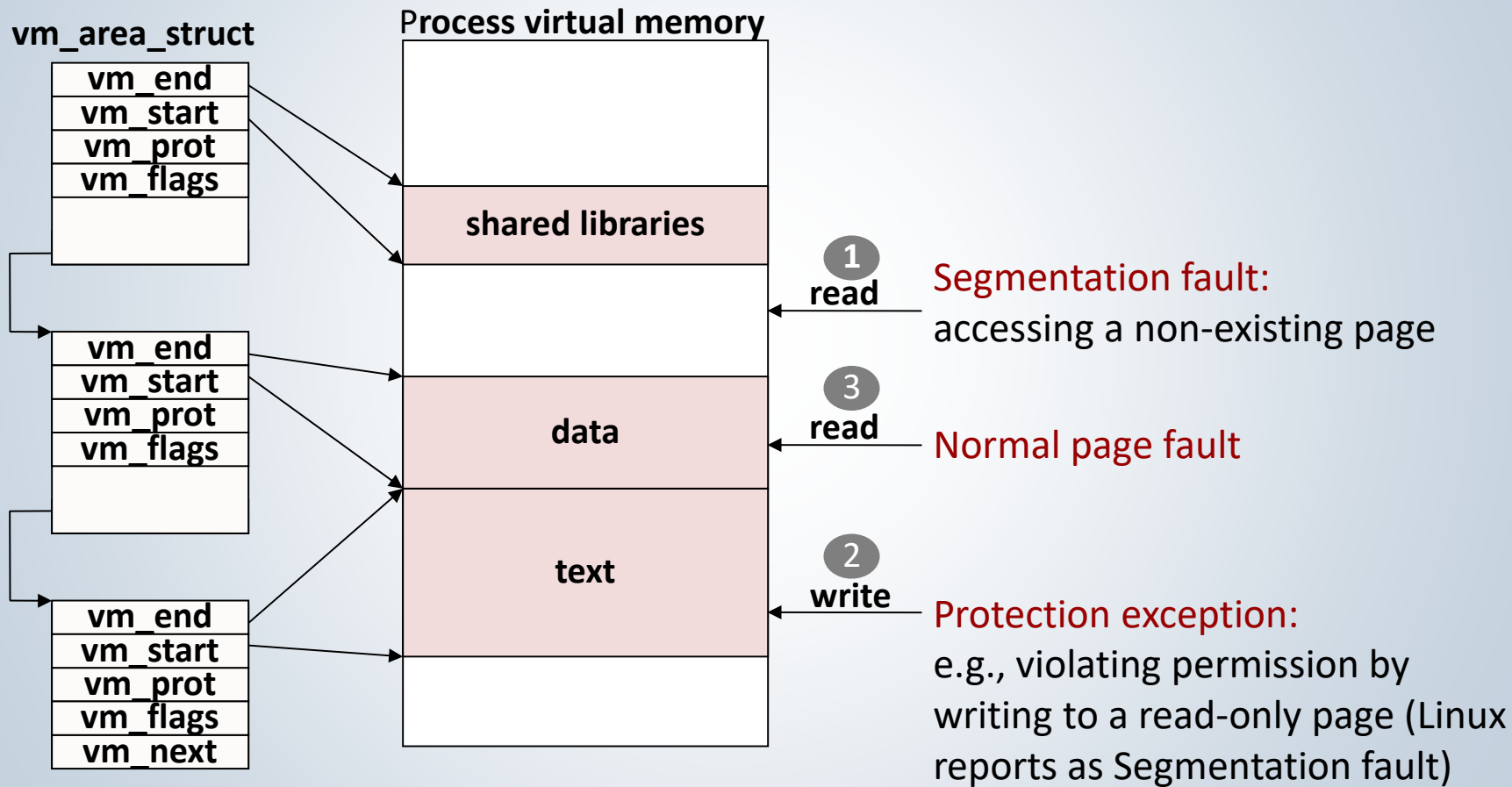
- Segment: Contiguous portion of address space
- Segfault arises from a memory access on a segmented machine to an illegal address
- In OS, segments can grow, or shrink
  - malloc(): Find free space for the object
    - Put it on the existing heap, or the heap has to grow



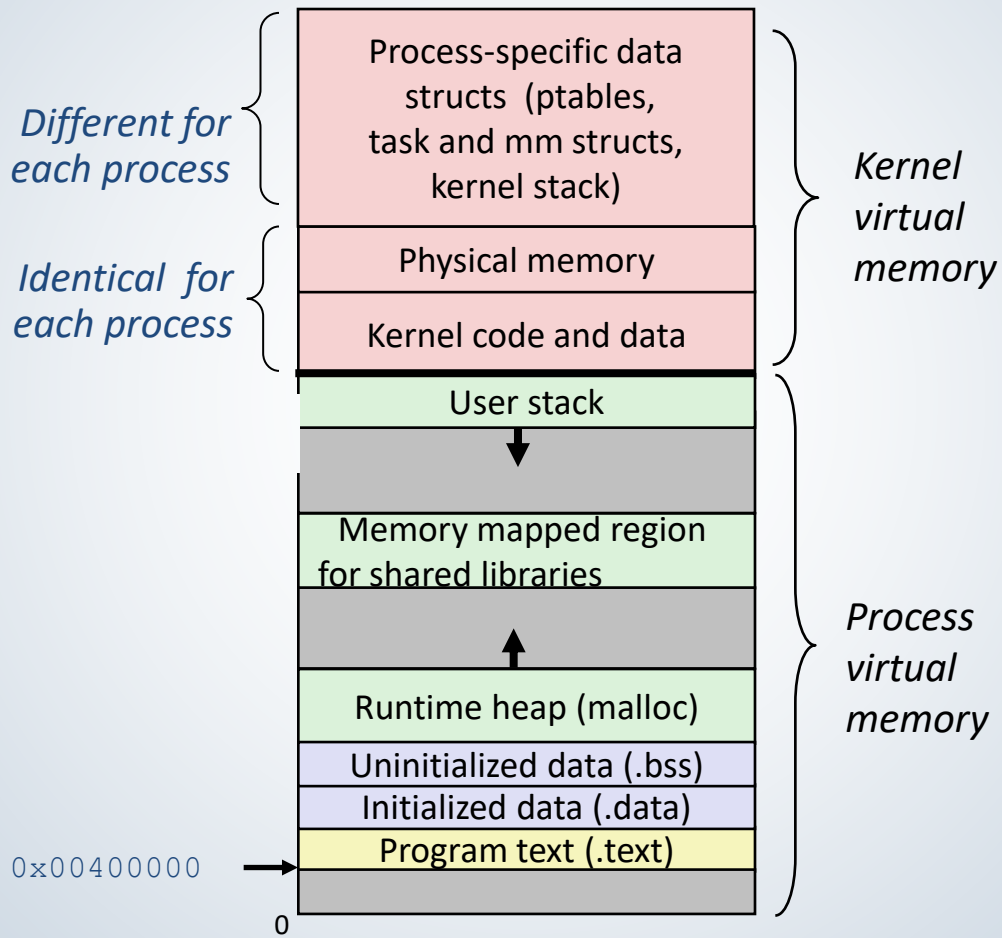
# Linux Organizes VM as Collection of “Areas”



# Linux Page Fault Handling



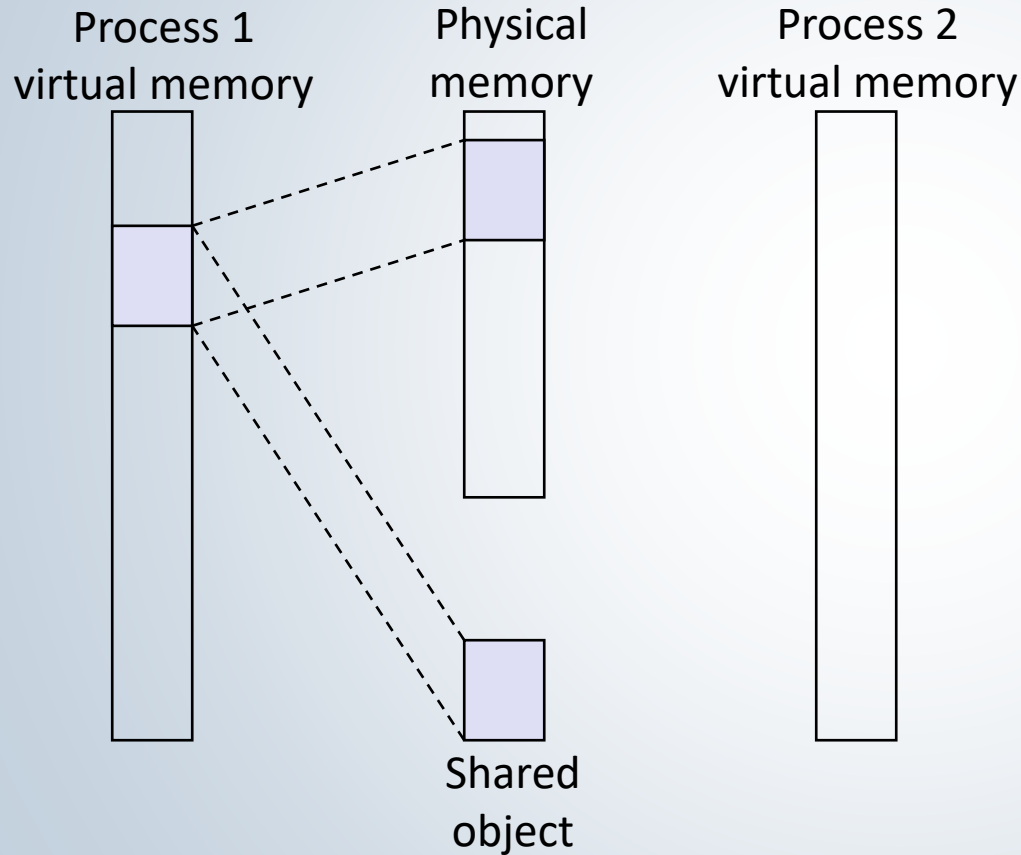
# Virtual Address Space of a Linux Process



# Today

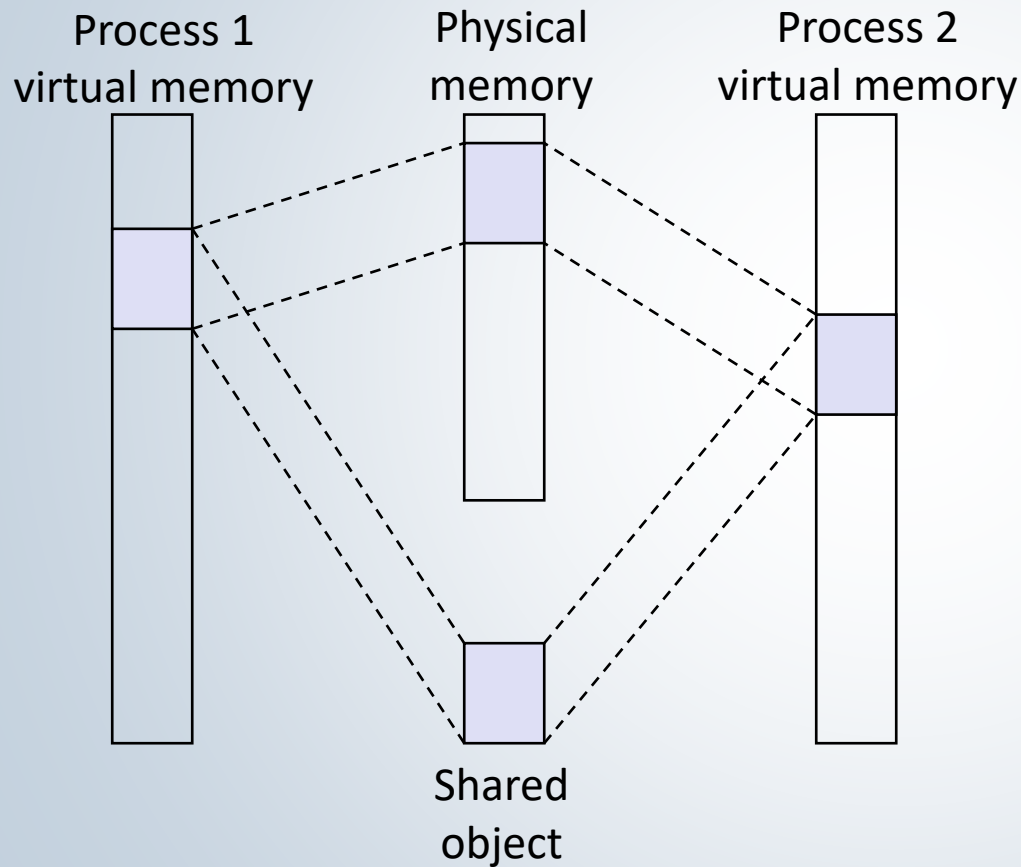
- Simple memory system example
- Shared Memory and Copy-on-Write
- Memory mapping

# Sharing Revisited: Shared Objects



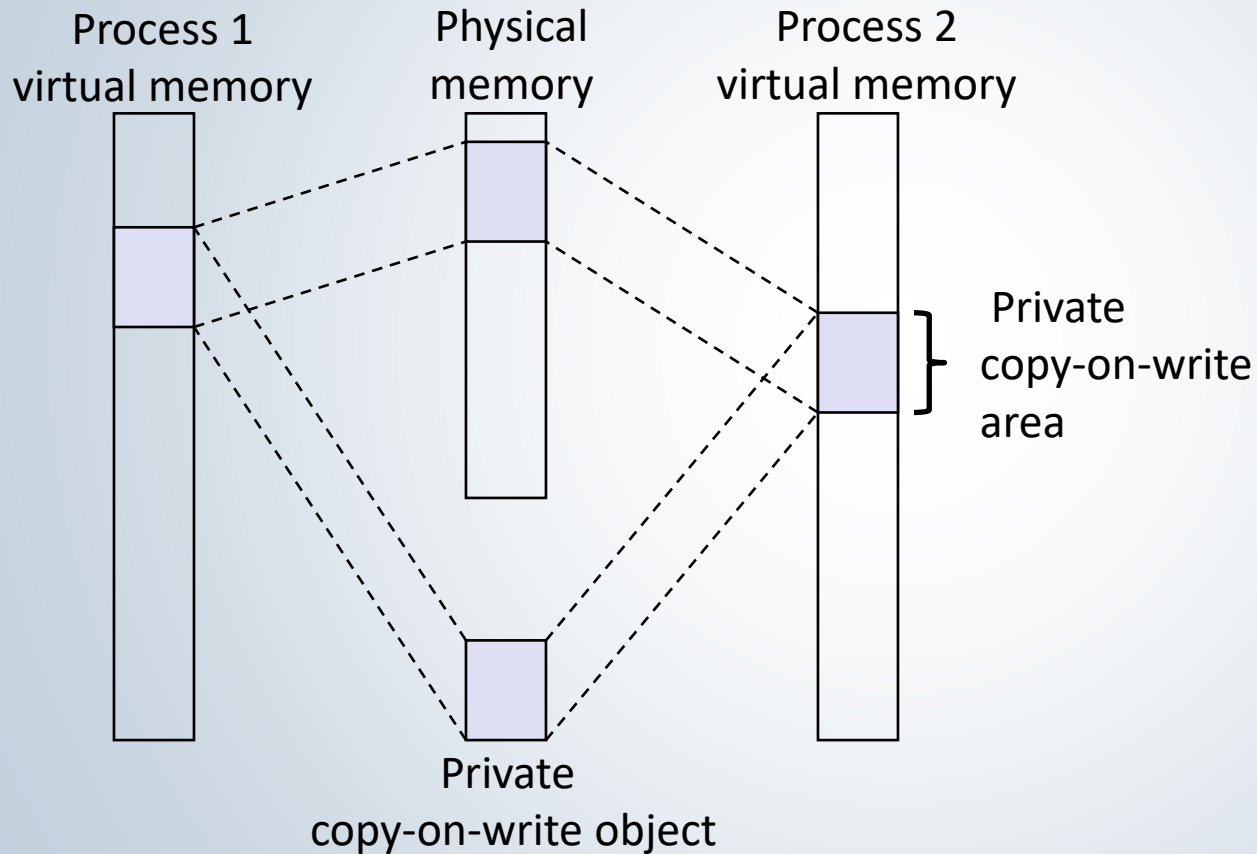
- **Process 2 maps the shared object (on disk).**

# Sharing Revisited: Shared Objects



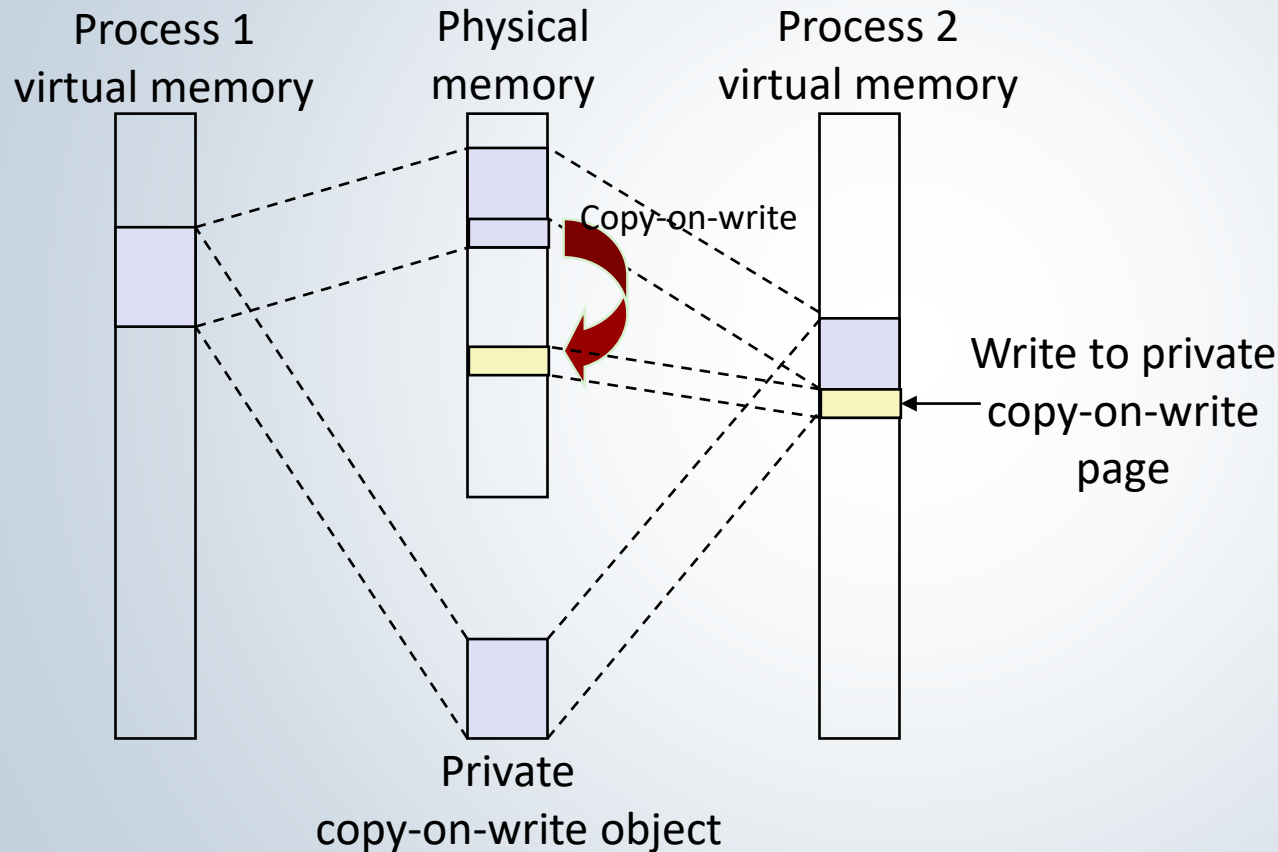
- **Process 2 maps the shared object (on disk).**
- Notice how the virtual addresses can be different.

# Sharing Revisited: Copy-On-Write (COW) Objects



- Two processes mapping a *private copy-on-write (COW)* object
- Initially, this object is like a shared object
- The access right for the pages of this object is read-only in PTEs

# Sharing Revisited: Copy-On-Write (COW) Objects



- Instruction writing to private page triggers protection fault
- Handler creates new R/W page for the offending process (Process 2)
- Instruction restarts upon handler return
- Copying deferred as long as possible!



# The fork() Function Revisited

- Can use COW memory mapping in `fork()` to provides private address space for each process without duplicating physical memory unnecessarily
- To create virtual address for new process
  - Create exact copies of current page tables
  - Flag each page in PETs of both processes as (clear `write` flag)
  - Linux flags the VM area structs as copy-on-write
- On return, each process has identical view of memory but only one copy of physical memory exists
- Subsequent writes, e.g., with `exec()`, trigger COW mechanism and force pages to be duplicated when needed

# Memory Mapping

- VM areas initialized by associating them with disk objects.
  - Process is known as **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**.

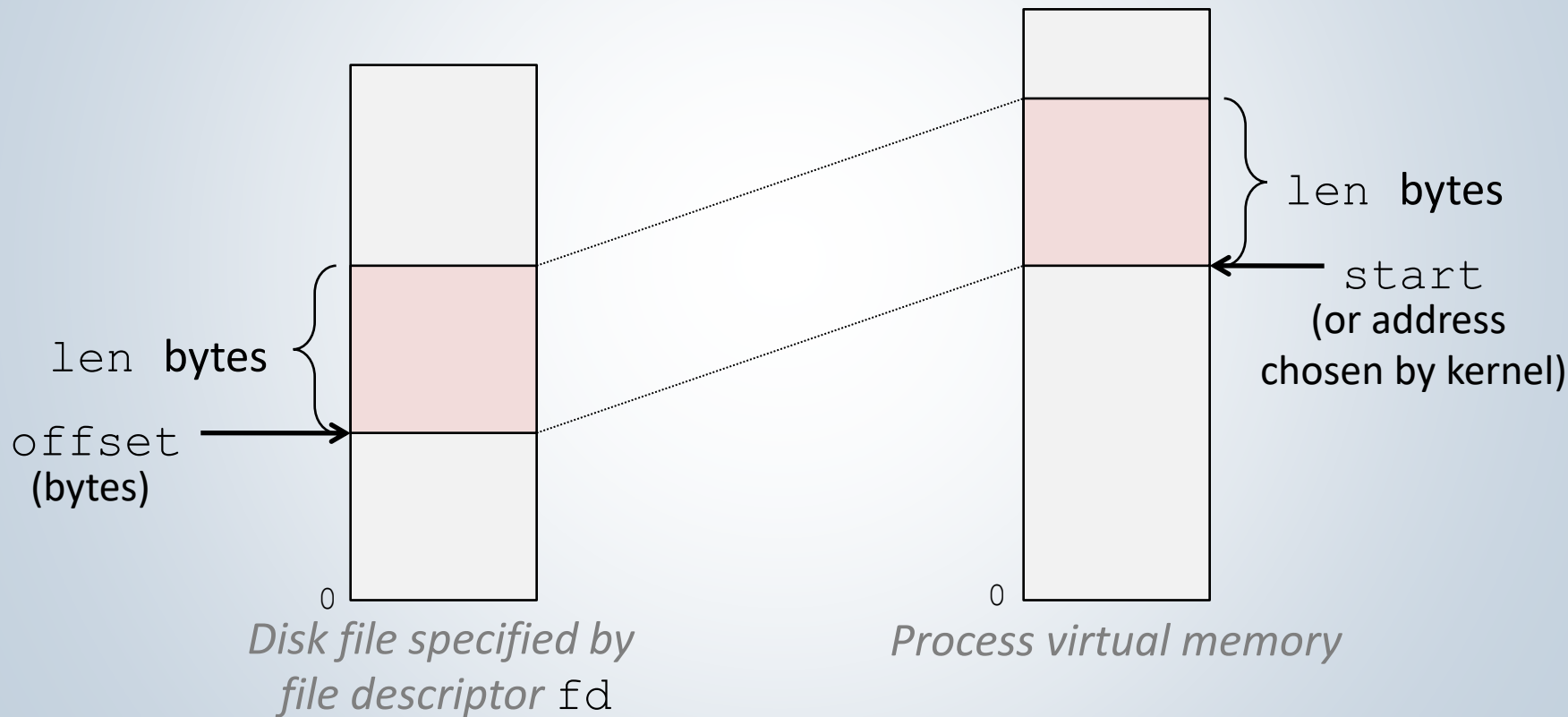
# User-Level Memory Mapping

```
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, ...
  - **flags**: MAP\_ANON, MAP\_PRIVATE, MAP\_SHARED, ...
- Return a pointer to start of mapped area (may not be **start**)
- **malloc()** calls **mmap()** to allocate new pages

# User-Level Memory Mapping

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



# Using mmap() to Copy Files (Linux)

- Copying a file to `stdout` without transferring data to user space

```
#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(1, bufp, size);
    return;
}
```

```
/* 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);
}
```