# CS 201

# Virtual Memory


**Gerson Robboy**
**Portland State University**

# Motivations for Virtual Memory

**Use Physical DRAM as a Cache for the Disk**

**Simplify Memory Management**

**Provide Protection**

# Motivation #1: DRAM a “Cache” for Disk

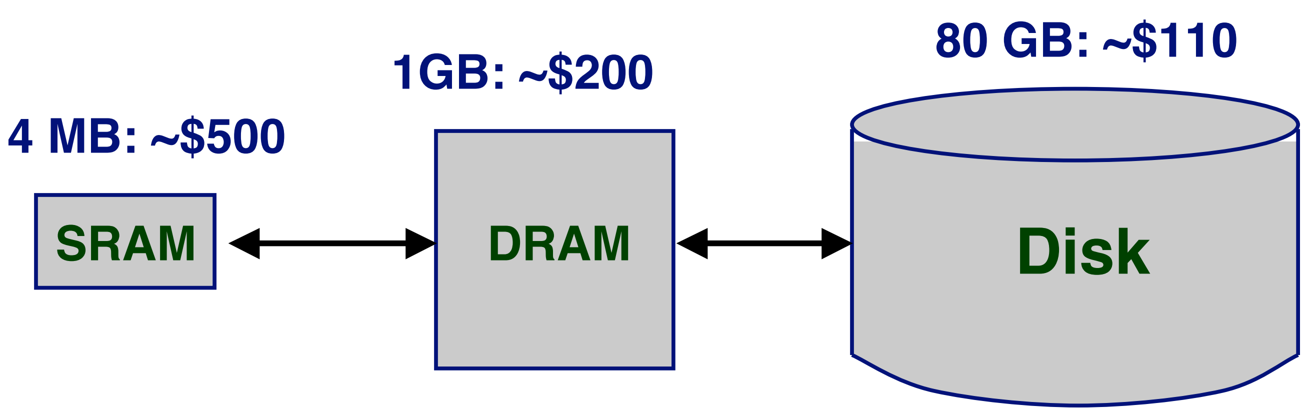
## Full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

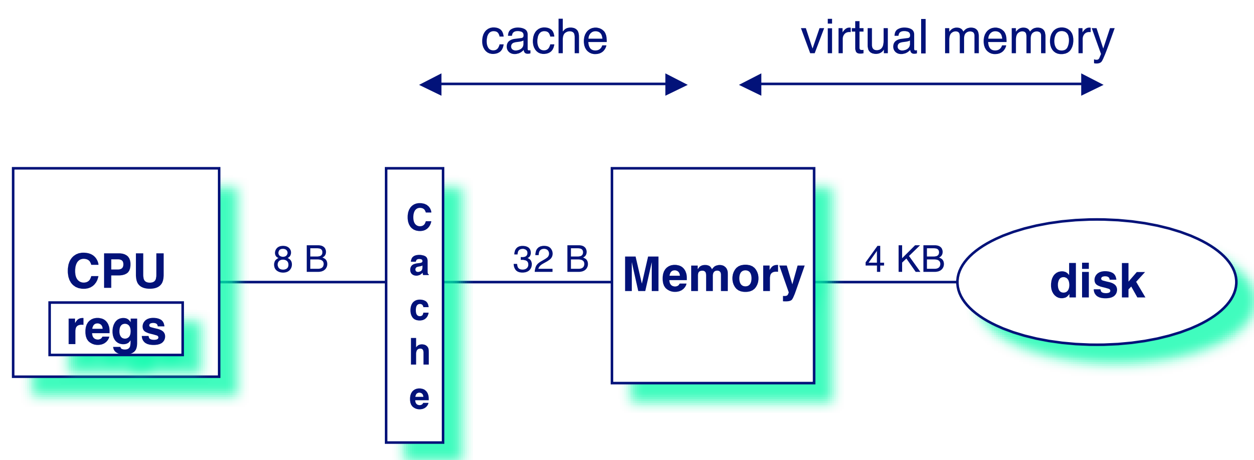
## Disk storage is ~300X cheaper than DRAM storage

- 80 GB of DRAM: ~ $33,000
- 80 GB of disk: ~ $110

## To access large amounts of data in a cost-effective manner, store the bulk of the data on disk



# Levels in Memory Hierarchy



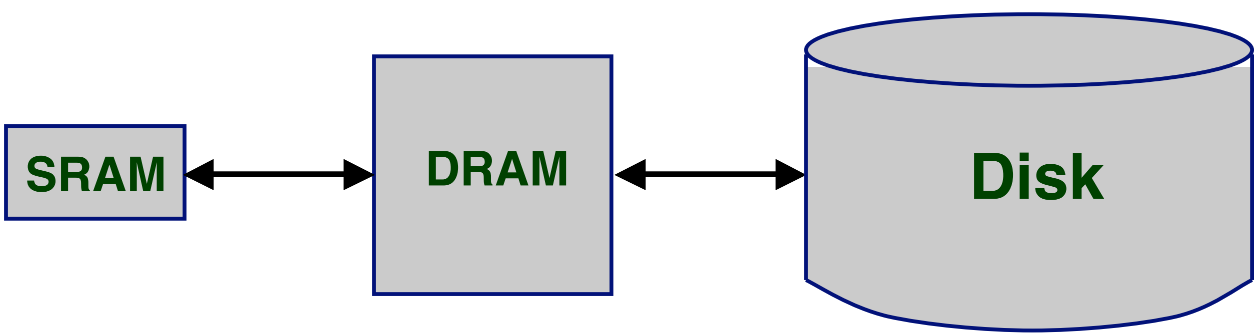

| | Register | Cache | Memory | Disk Memory |
|---|---|---|---|---|
| size: | 32 B | 32 KB-4MB | 1024 MB | 100 GB |
| speed: | 1 ns | 2 ns | 30 ns | 8 ms |
| $/Mbyte: | | $125/MB | $0.20/MB | $0.001/MB |
| line size: | 8 B | 32 B | 4 KB | |

larger, slower, cheaper

# DRAM as a "Cache"

## DRAM vs. disk is more extreme than SRAM vs. DRAM

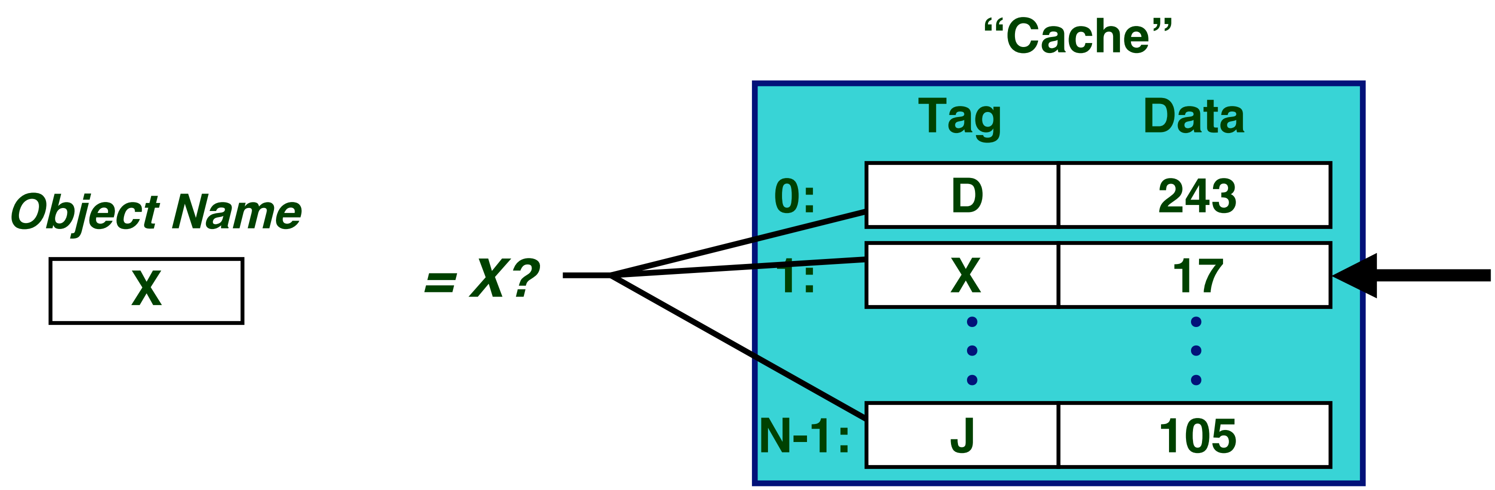
- **Access latencies:**
  - **DRAM ~10X slower than SRAM**
  - **Disk ~**100,000X **slower than DRAM**
- **Importance of exploiting spatial locality:**
  - **First byte is ~**100,000X **slower than successive bytes on disk**
- **Bottom line:**
  - **Design decisions driven by enormous cost of misses**



# Locating an Object in a "Cache"

## SRAM Cache

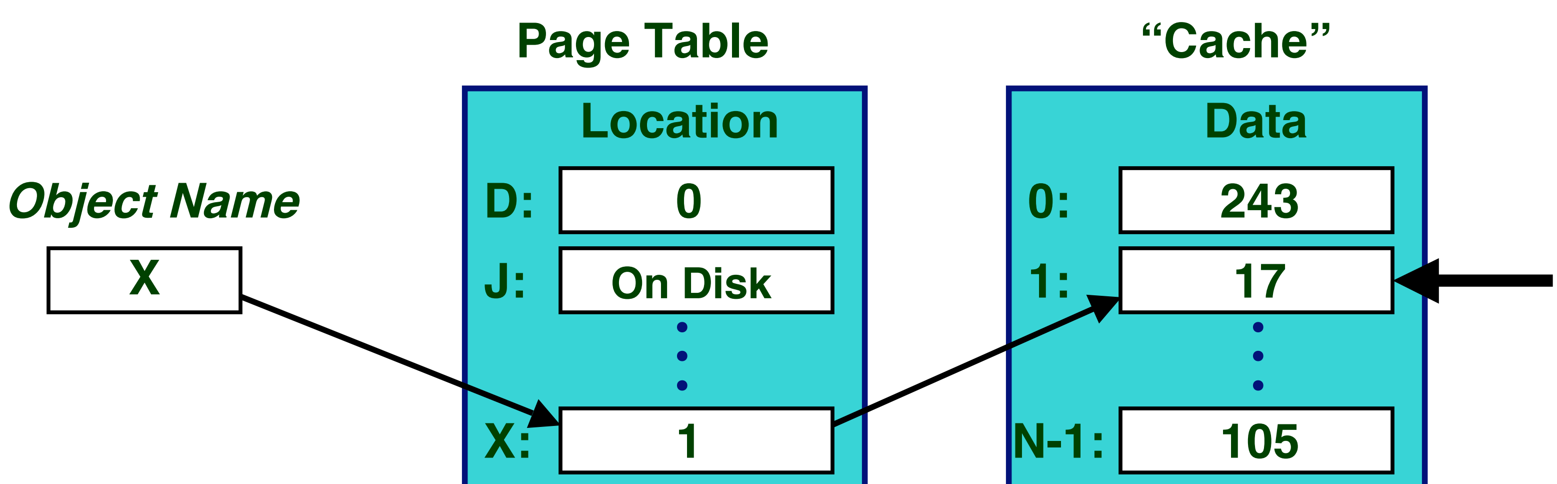
- Tag stored with cache line
  - Maps from cache block to memory address
- Hardware retrieves information
  - Cache hit:  gets it quickly from the cache
  - Cache miss:  more slowly from memory



# Locating an Object in "Cache" (cont.)

## DRAM Cache

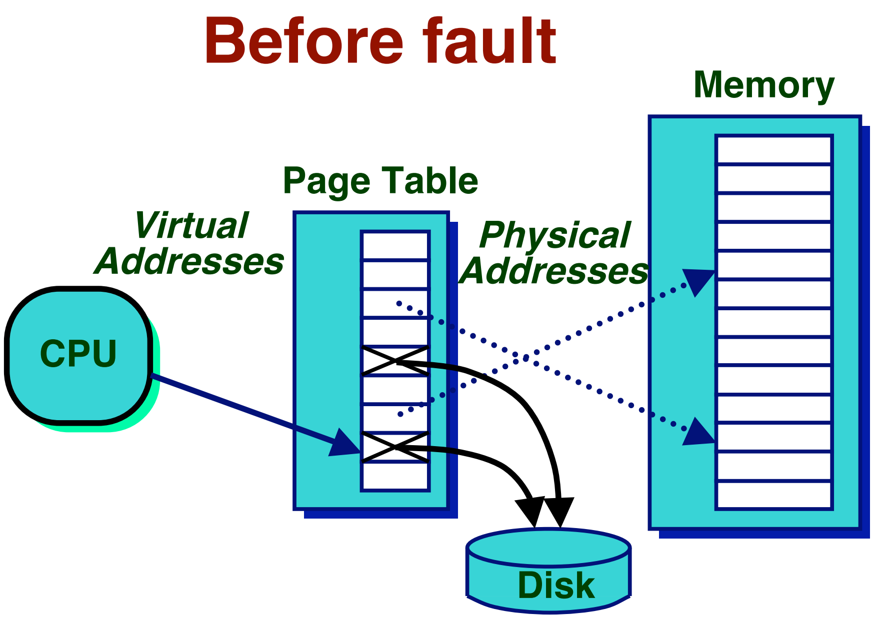
- Each allocated page of virtual memory has entry in *page table*
- Mapping from virtual pages to physical pages
  - From uncached form to cached form
- If the page is not in memory
  - "Present" bit is not set
  - Page table entry gives disk address
- OS retrieves information

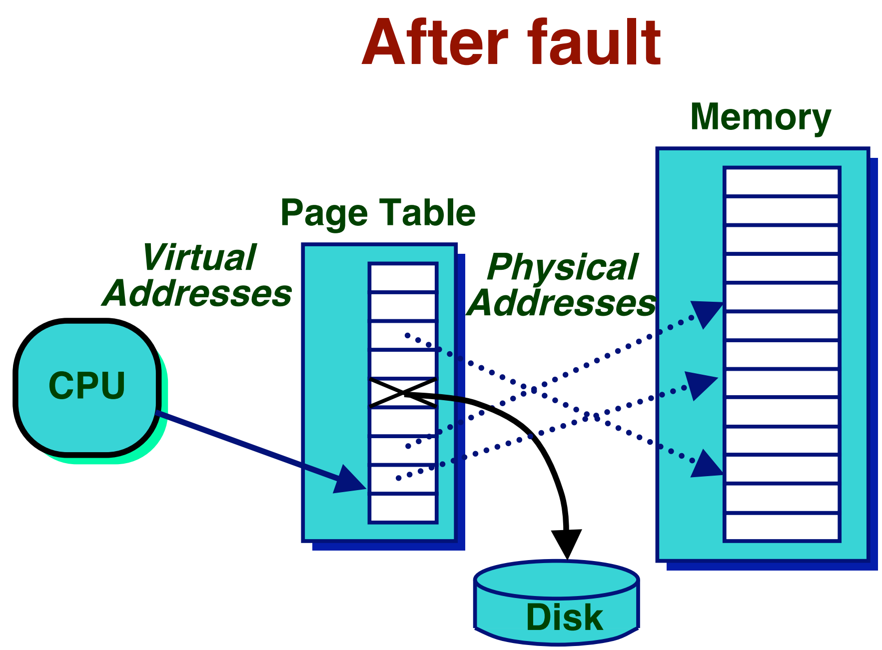


# Page Faults (like "Cache Misses")

## What if an object is not in memory?

- Page table entry indicates virtual address not present
- Page fault
- OS exception handler imoves data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.



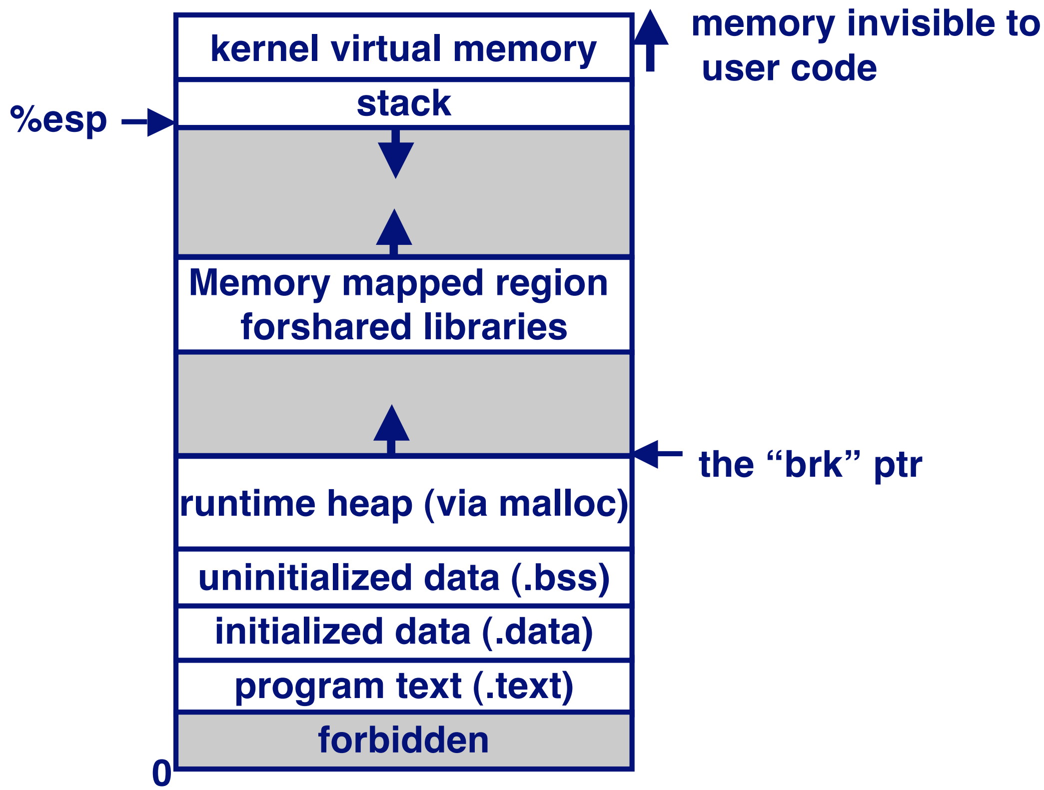
# Motivation #2: Memory Management

## Multiple processes can reside in physical memory.

## How do we resolve address conflicts?

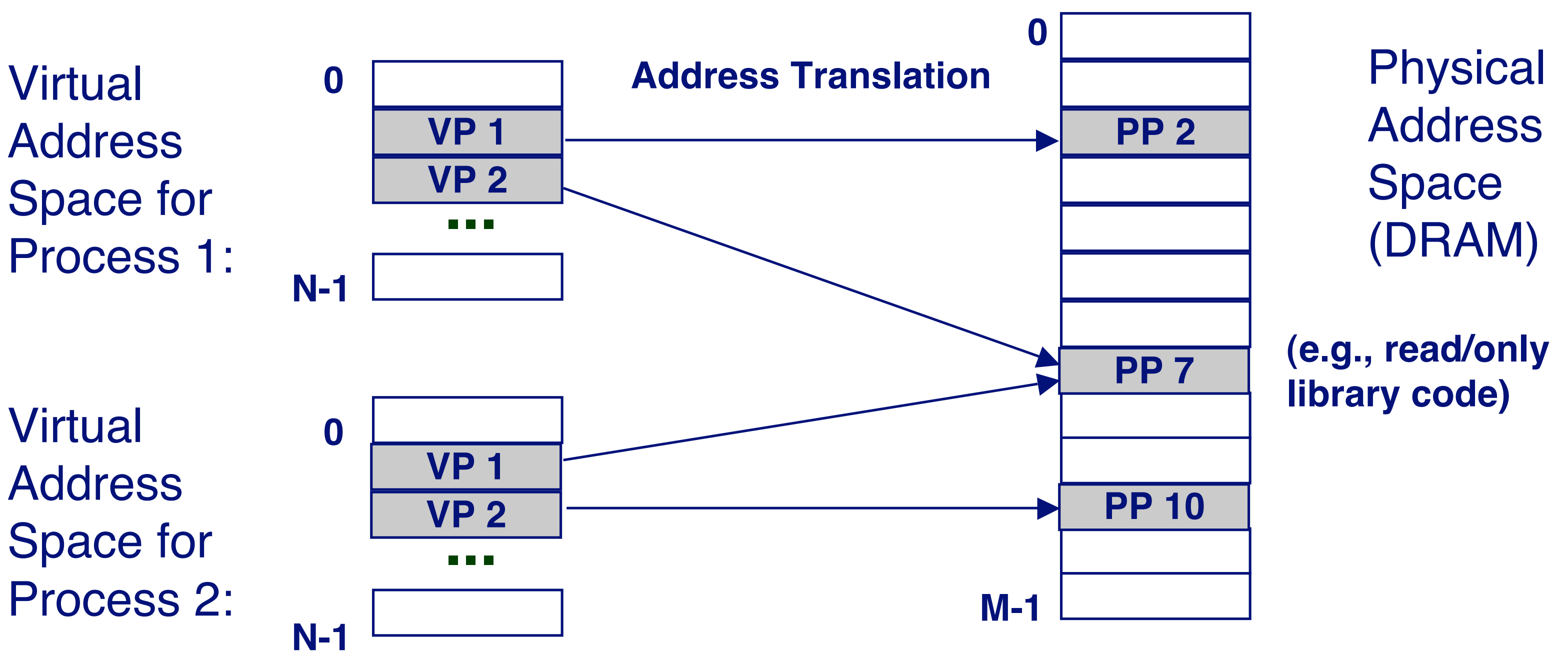
- what if two processes access something at the same address?

**Linux/x86 process memory image**



# Solution: Separate Virt. Addr. Spaces

- **Virtual and physical address spaces divided into equal-sized blocks**
  - **blocks are called “pages” (both virtual and physical)**
- **Each process has its own virtual address space**
  - **operating system controls how virtual pages as assigned to physical memory**

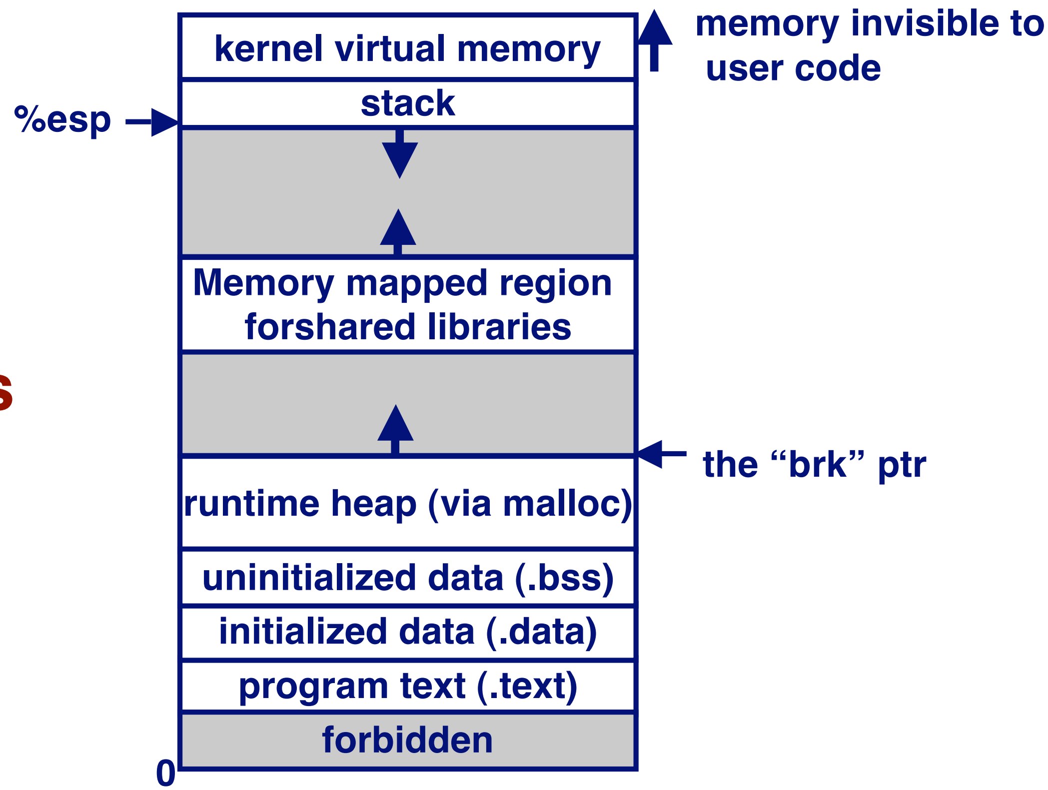


# What this means for linking/loading

## The linker binds programs to absolute addresses.

- Nothing is left relocatable.
- No relocation at load time.
- No allocation of memory segments at load time.

**All processes look just like this**



# Questions

The O. S. allocates pages for the stack on demand.

What does the hardware do when a stack overflows the allocated page?

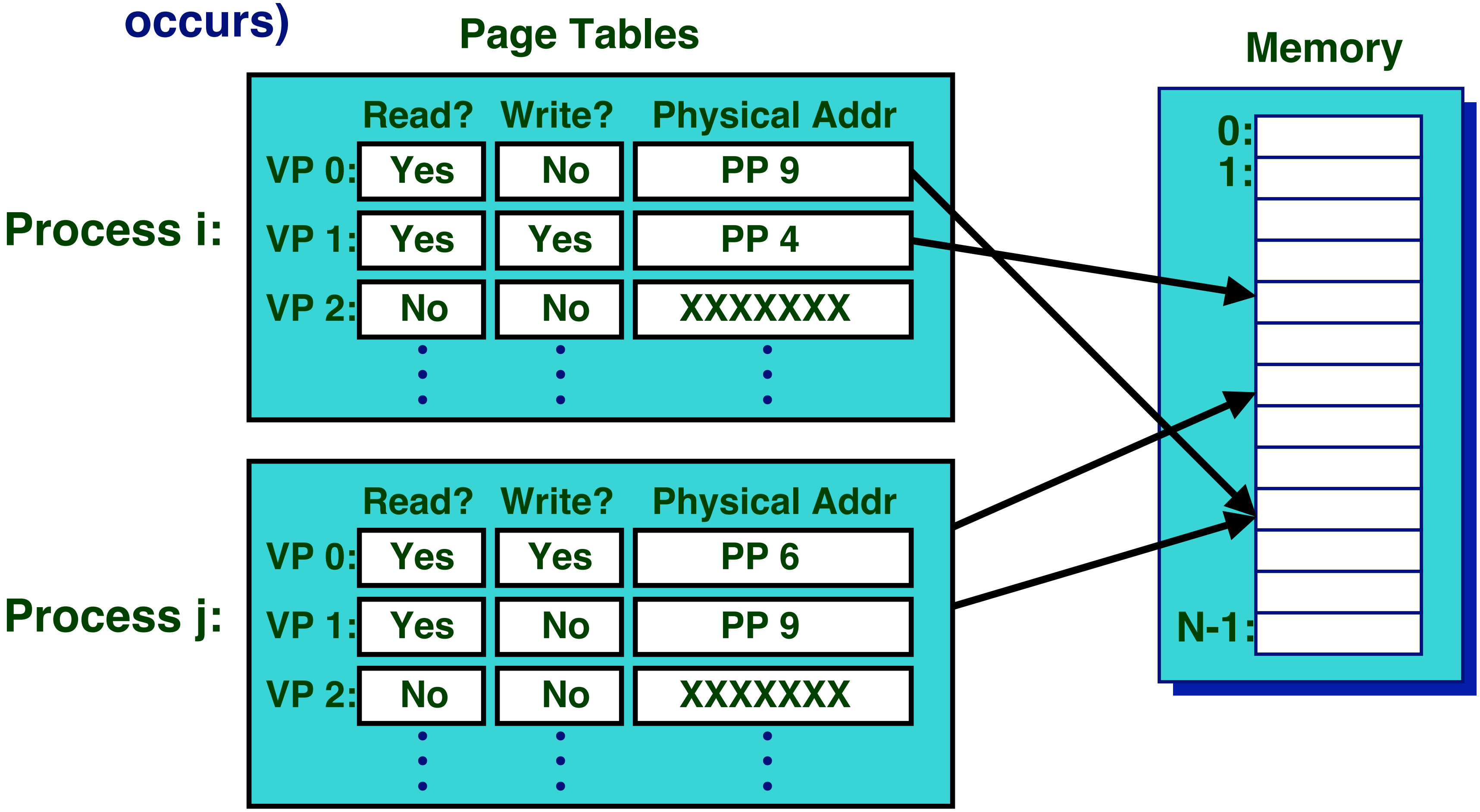
What does the O. S. do in response?

Is it possible for the program to keep running?

# Motivation #3: Protection

## Page table entry contains access rights information

- hardware enforces this protection (trap into OS if violation occurs)



# Protection

The O. S. kernel gives each process a virtual memory space

- Each process has its own set of page tables
- Page tables for other processes are not visible
- Process B's memory is not merely protected from Process A by permissions
  - It doesn't even exist in Process A's memory space

Physical memory belonging to another process is completely outside the memory space.

- Note:  sharing is also possible
- In Linux, threads are a kind of process with shared memory.

# Virtual Memory = Swapping

Pieces of processes are swapped in and out.

The granularity is the page, not the whole process

Pages are not in memory until needed

- “Demand Paging”
- Pull pages in on demand; i. e., when accessed

# What happens when a new process starts running?

No pages are in memory.

The first access to an instruction causes a page fault.

Pages are pulled in as needed ("demand paged")

# What happens when you say malloc(32000000) ?

What exactly does the O. S. allocate at that time?

Is it necessary to allocate 32 MB of physical memory?

# Question

Suppose a process has a page fault, the kernel must allocate a physical page of memory, and all physical memory is in use by this and other processes.

What does the O. S. do?

# Page Replacement algorithms

## Analogous to cache line replacement.

## A complex topic.

- Beyond the scope of this class
- A popular topic with computer scientists because it lends itself to research.

# What about code?

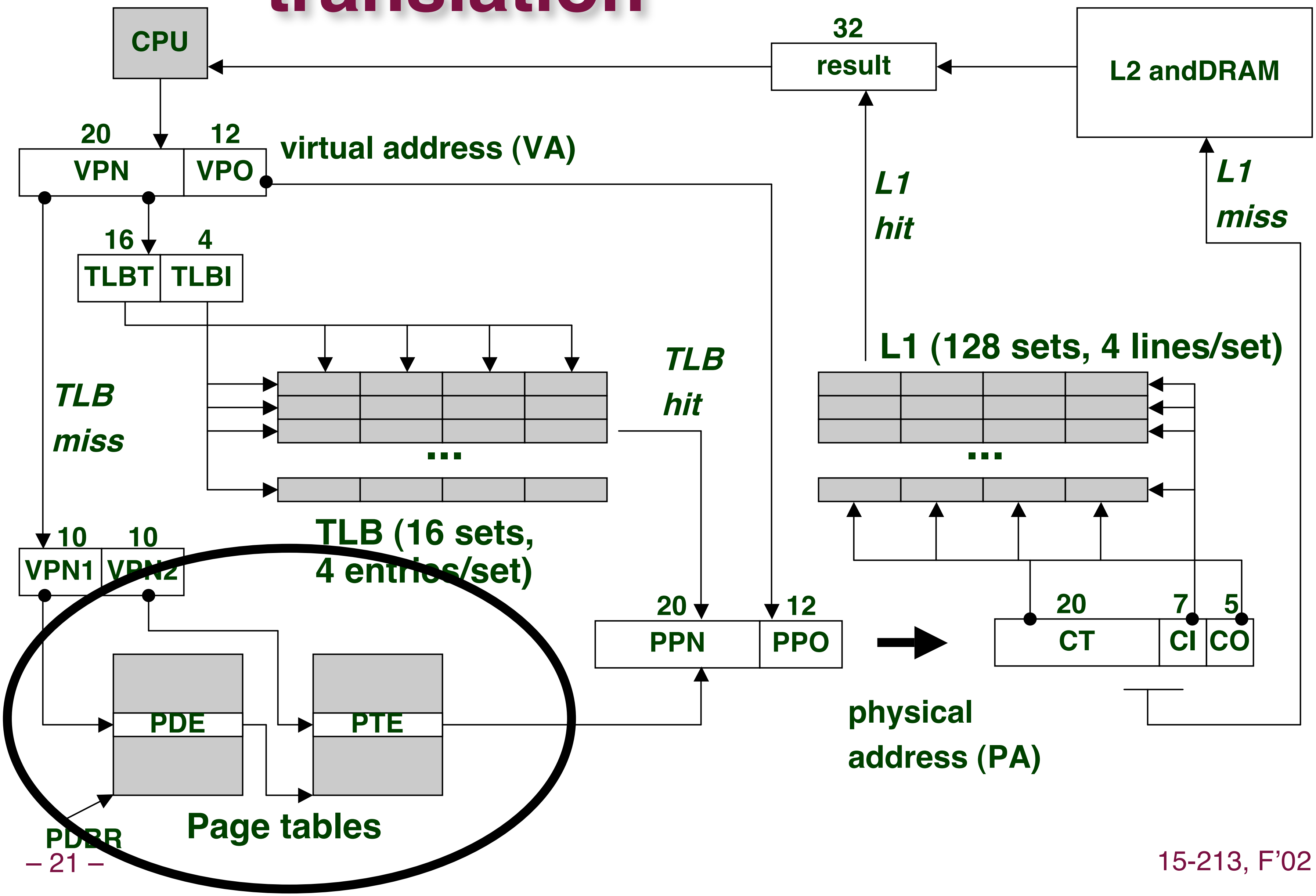
Code is read-only.

We execute it but we don’t write it.

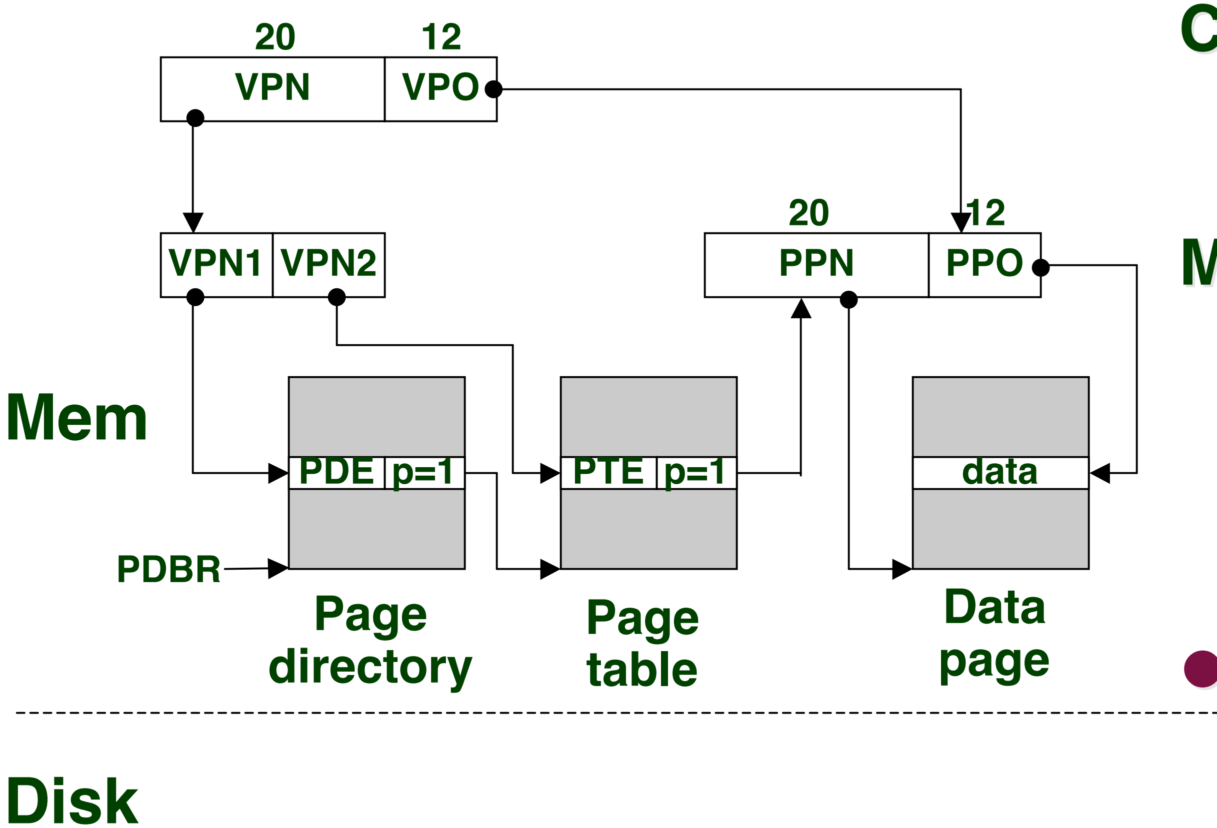
What happens when the O. S. must evict a page of code?

- Does the O. S. write the page out to disk?

# P6 page table translation



# Translating with the P6 Page Tables (case 1/1)



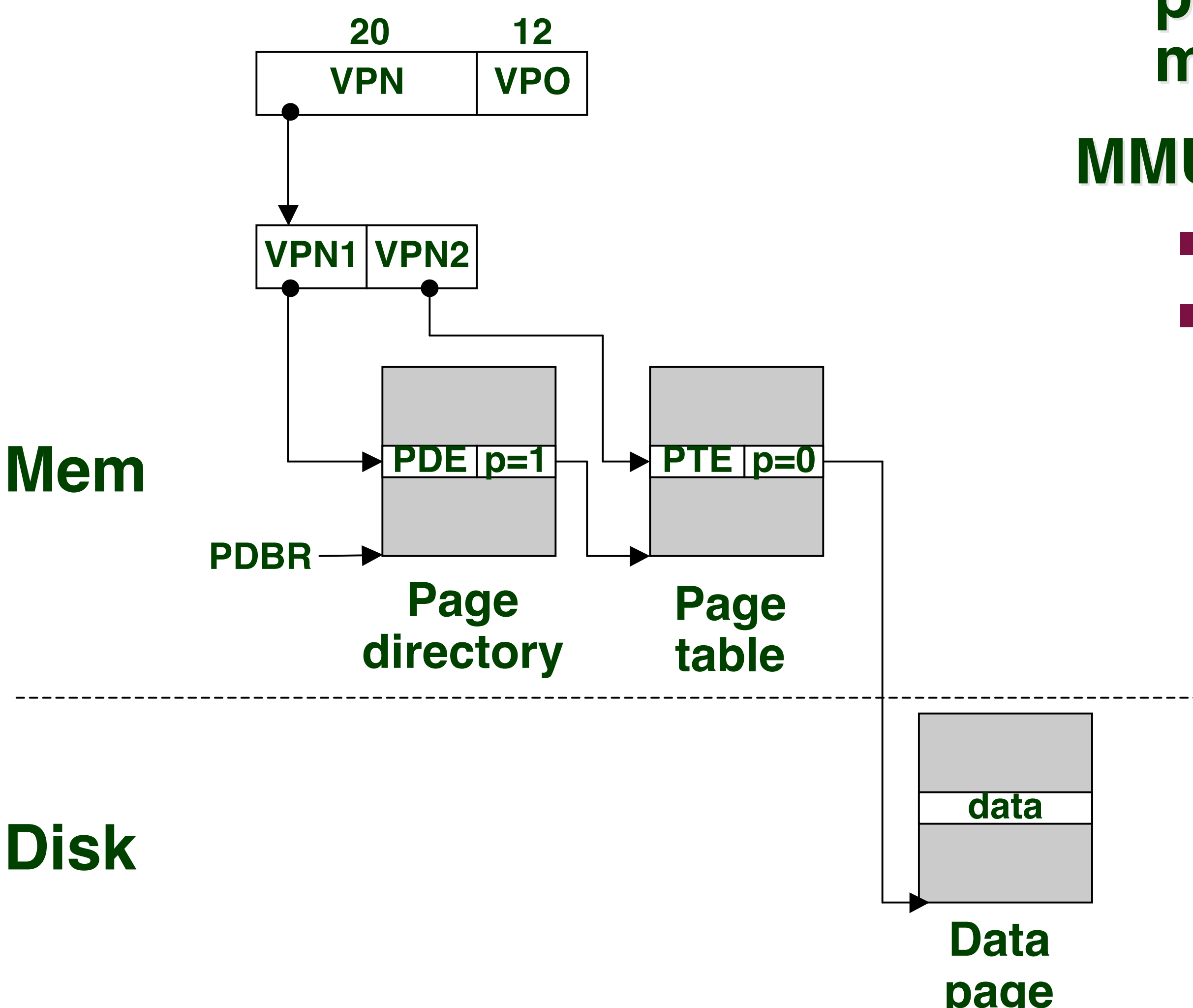

Case 1/1: page table and page present.

MMU Action:

- MMU builds physical address and fetches data word.

- OS action
  - none

# Translating with the P6 Page Tables (case 1/0)

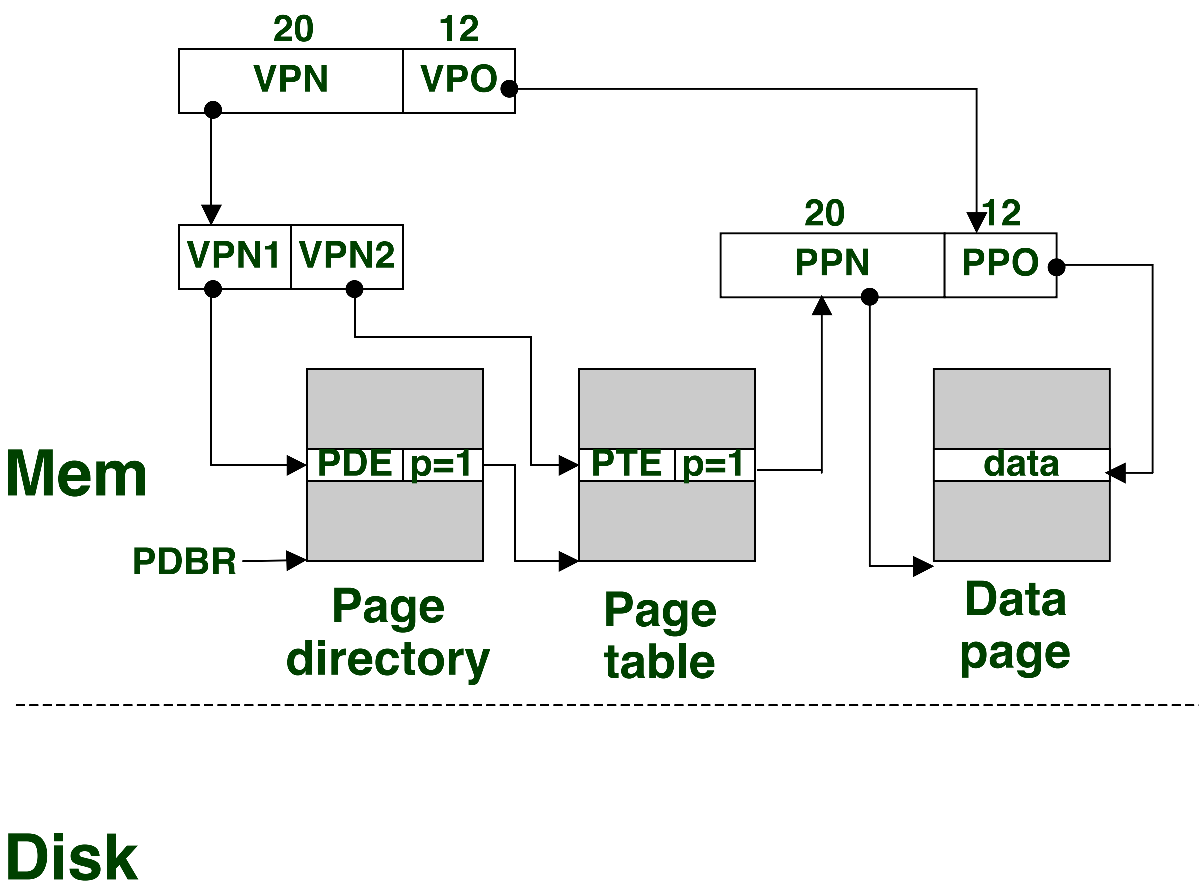



**Case 1/0: page table present but page missing.**

**MMU Action:**

- page fault exception
- handler receives the following args:
  - VA that caused fault
  - fault caused by non-present page or page-level protection violation
  - read/write
  - user/supervisor

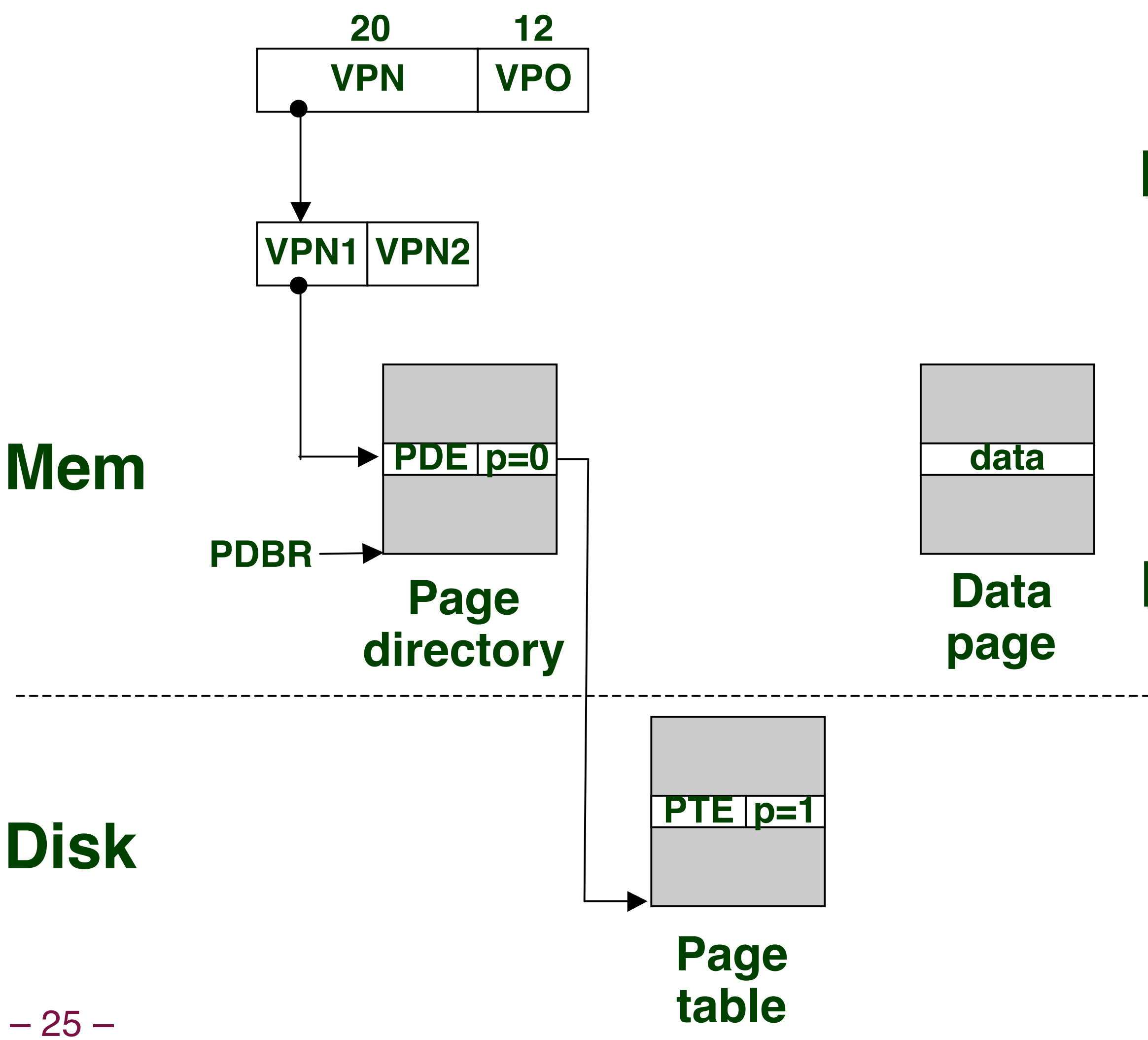
# Translating with the P6 Page Tables (case 1/0, cont)




## OS Action:

- Check for a legal virtual address.
- Read PTE through PDE.
- Find free physical page (swapping out current page if necessary)
- Read virtual page from disk and copy to virtual page
- Restart faulting instruction by returning from exception handler.

# Translating with the P6 Page Tables (case 0/1)




Case 0/1: page table missing but page present.

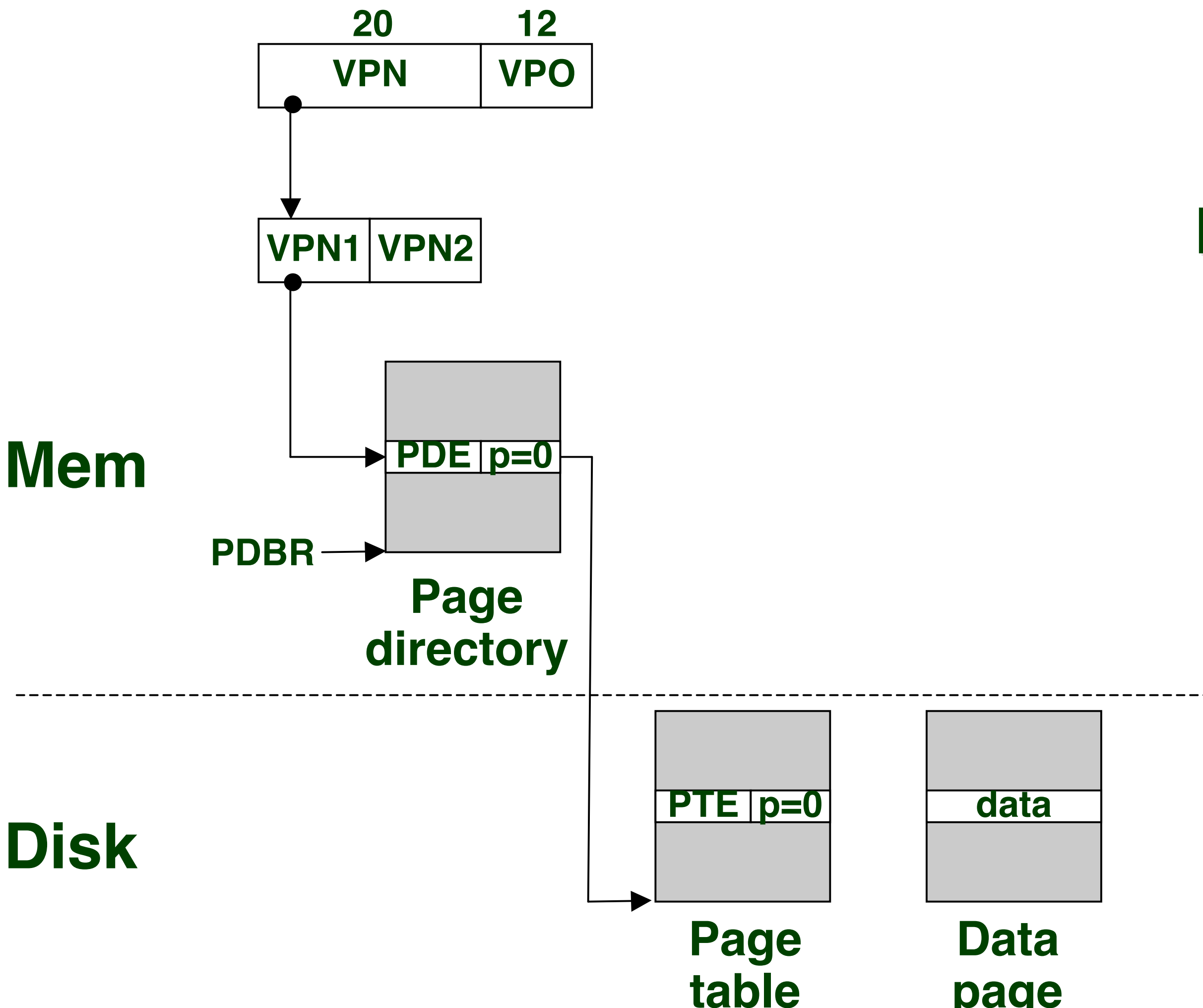
Introduces consistency issue.

- potentially every page out requires update of disk page table.

Linux disallows this

- if a page table is swapped out, then swap out its data pages too.

# Translating with the P6 Page Tables (case 0/0)

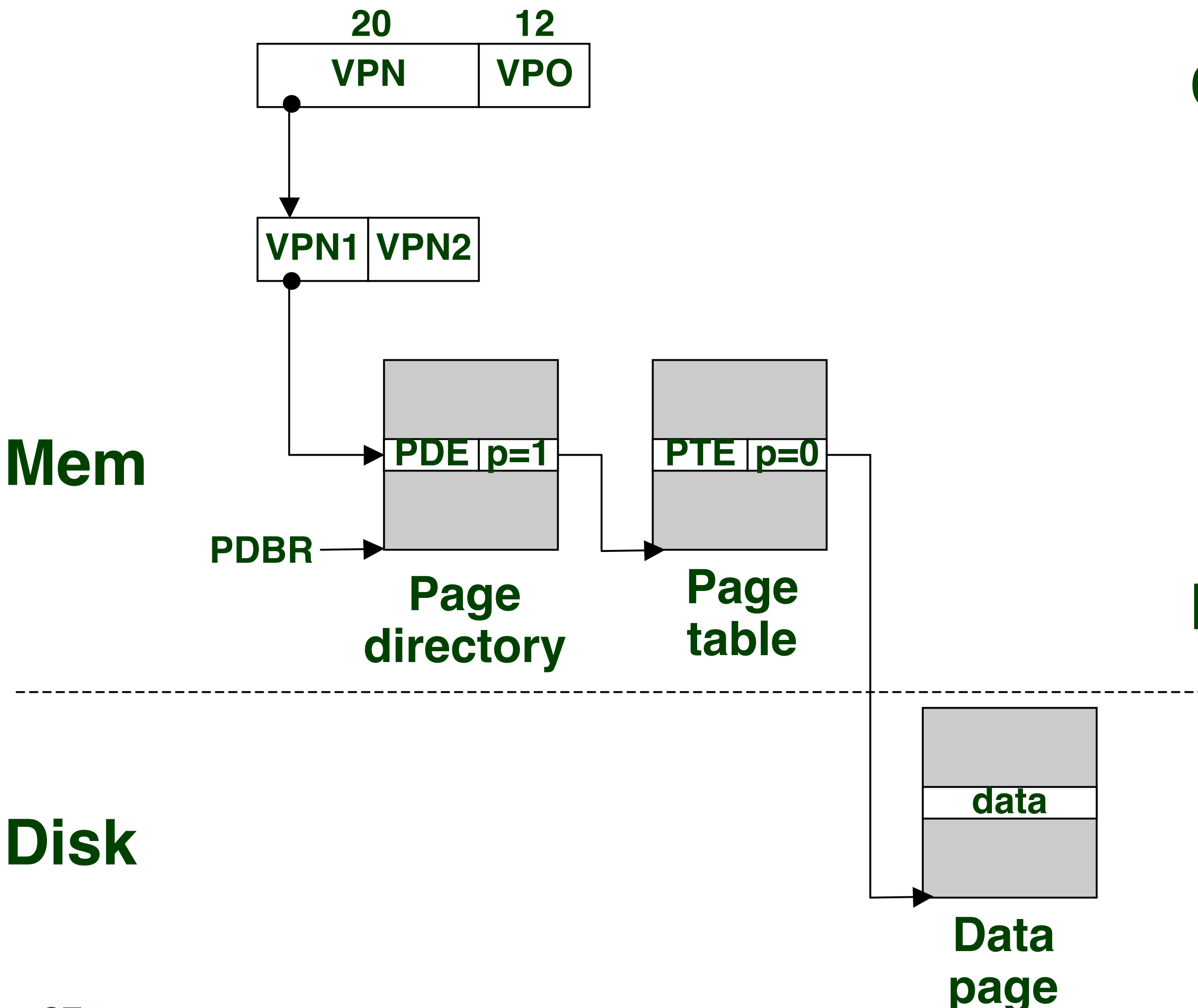



**Case 0/0: page table and page missing.**

**MMU Action:**

- page fault exception

# Translating with the P6 Page Tables (case 0/0, cont)




OS action:

- swap in page table.
- restart faulting instruction by returning from handler.

Like case 0/1 from here on.

# What happens after a page fault is handled?

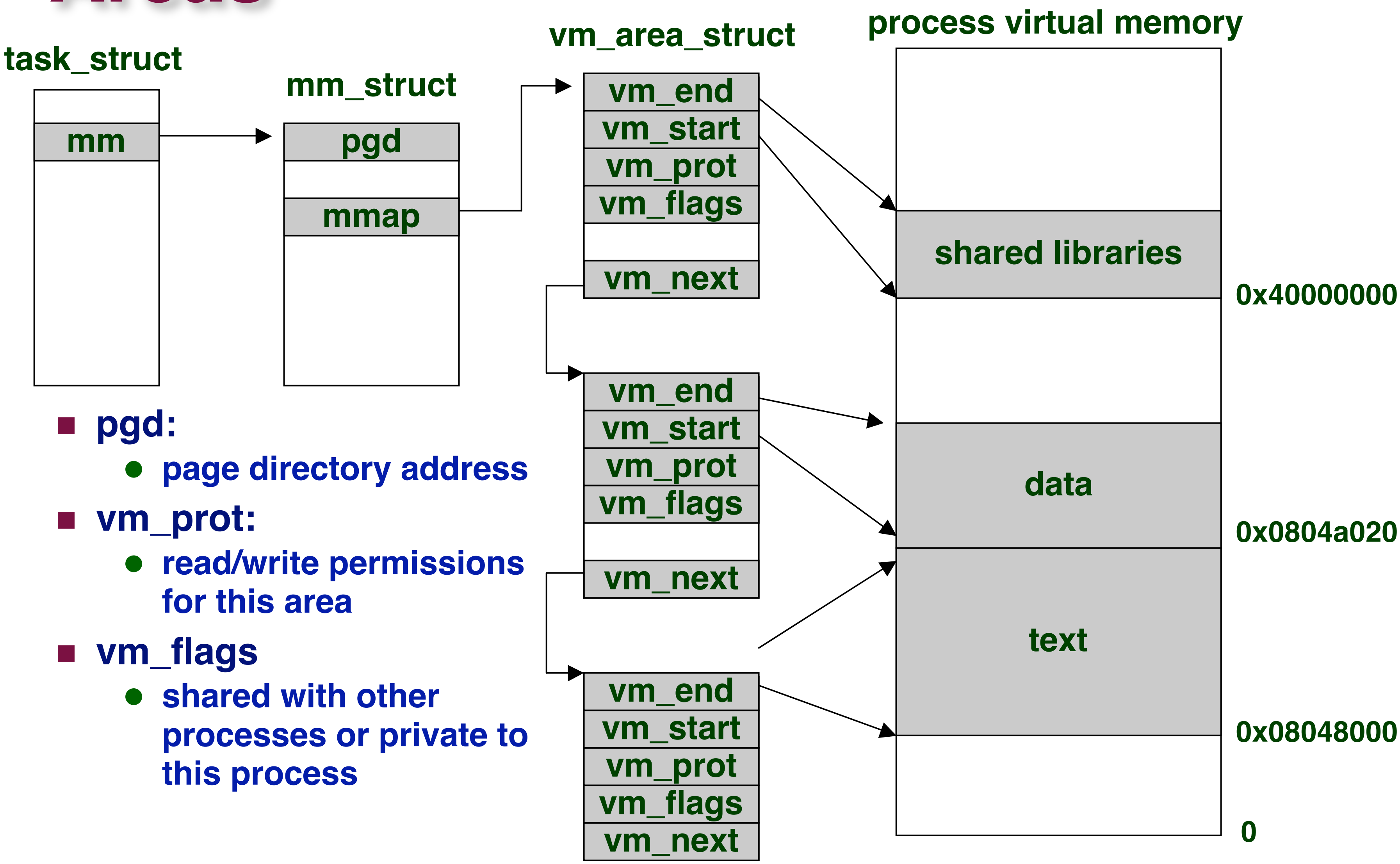
A process touches an unmapped memory address.

The hardware generates a fault.

The O. S. kernel gets the required page and maps it.

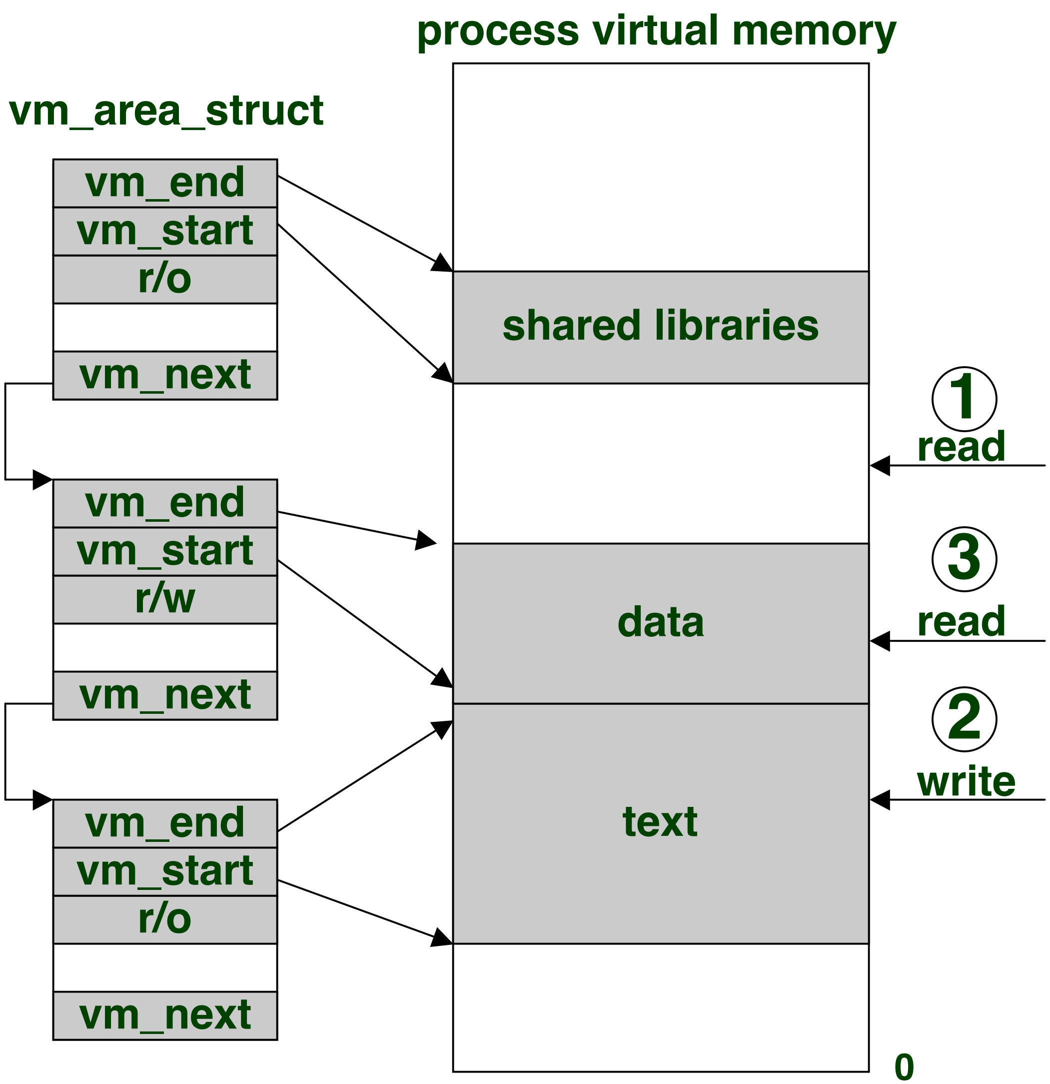
The fault handler returns control to … where?

# Linux Organizes VM as Collection of "Areas"




- **pgd:**
  - page directory address
- **vm_prot:**
  - read/write permissions for this area
- **vm_flags**
  - shared with other processes or private to this process

# Linux Page Fault Handling




## Is the VA legal?

- i.e. is it in an area defined by a vm_area_struct?
- if not then signal segmentation violation (e.g. (1))

## Is the operation legal?

- i.e., can the process read/write this area?
- if not then signal protection violation (e.g., (2))

## If OK, handle fault

- e.g., (3)

# Memory Mapping

## Creation of new VM *area* done via "memory mapping"

- create new vm_area_struct and page tables for area
- 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
  - nothing (e.g., bss)
    - » initial page bytes are zeros
- dirty pages are swapped back and forth between a special swap file.

## Key point: no virtual pages are copied into physical memory until they are referenced!

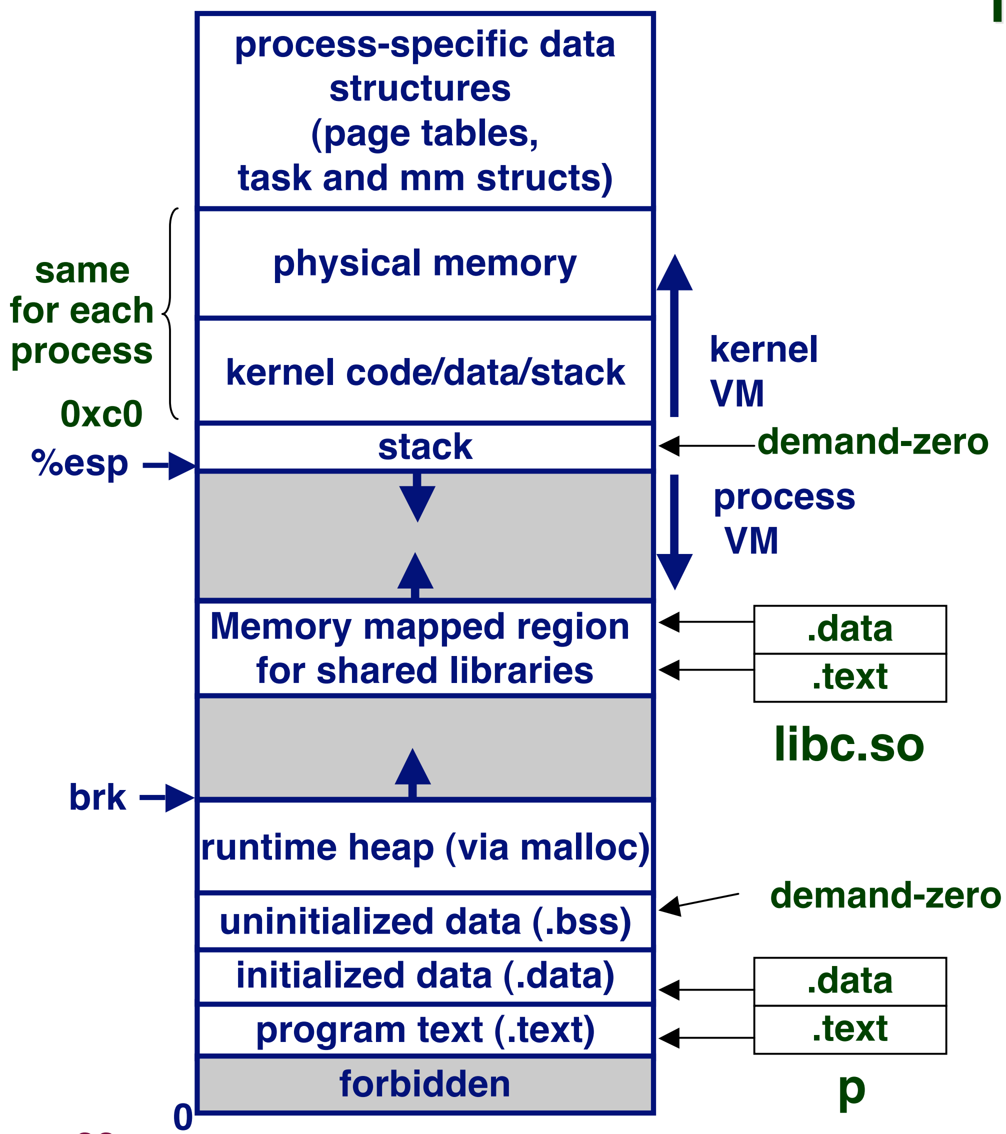
- known as "demand paging"
- crucial for time and space efficiency

# Putting it all together

Do practice problem 10.4 on page 715 of B & O.

After each of the 4 parts, let’s reconvene and review the solution to that part, then go on.

# Exec() Revisited




To run a new program p in the current process using `exec()`:

- free vm_area_struct's and page tables for old areas.
- create new vm_area_struct's and page tables for new areas.
  - stack, bss, data, text, shared libs.
  - text and data backed by ELF executable object file.
  - bss and stack initialized to zero.
- set PC to entry point in .text
  - Linux will swap in code and data pages as needed.

# Fork() Revisited

To create a new process using `fork()`:

make copies of the old process's mm_struct, vm_area_struct's, and page tables.

- at this point the two processes are sharing all of their pages.
- How to get separate spaces without copying all the virtual pages from one space to another?
  - "copy on write" technique.

# Fork() Revisited

## copy-on-write

- make pages of  writeable areas read-only
- flag vm_area_struct's for these areas as private "copy-on-write".
- writes by either process to these pages will cause page faults.
  - fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

## Net result:

- copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

# What does copy-on-write buy us?

**What do most  child processes do soon after a fork?**

# When a process calls exec…

**All its vm_area_structs and page tables are freed.**

**Is the physical memory freed?**

**How does the O. S. deal with shared pages?**

# Review Problem

When a process calls fork, what does the operating system do in terms of memory managament? Describe what physical memory is allocated or freed, for which processes.

When a process calls exec, what does the operating system do in terms of memory managament? Describe what physical memory is allocated or freed, for which processes.

# Can we write VM-friendly programs?

Yes, definitely.

Remember “cache-friendly” code?

In this case a “cache line” is analogous to a page.

General principles:

- Spatial and temporal locality
- Try to re-use recently used data
- Keep the “working set” relatively small

With VM we don’t have a fixed-size cache

- How many pages we can use depends on the system load and total size of system memory
- Factors we can’t control or even know

# Main Themes

## Programmer’s View

- Large “flat” address space
  - Can allocate large blocks of contiguous addresses
- Process “owns” machine
  - Has private address space
  - Unaffected by behavior of other processes

## System View

- User virtual address space created by mapping to set of pages
  - Need not be contiguous
  - Allocated dynamically
  - Enforce protection during address translation
- OS manages many processes simultaneously
  - Continually switching among processes
  - Especially when one must wait for resource
    - » E.g., disk I/O to handle page fault