Protection and Security

- Two main features on computer processors allow operating systems to provide protection and security

- Feature 1: **multiple processor operating modes**
  - The processor physically enforces different constraints on programs operating in different modes

- Minimal requirements:
  - Kernel mode (a.k.a. protected mode, privileged mode, etc.) allows a program full access to all processor capabilities and operations
  - User mode (a.k.a. normal mode) only allows a program to use a restricted subset of processor capabilities

- The kernel of the operating system is the part of the OS that runs in kernel mode
  - The OS may have [many] other components running in user mode

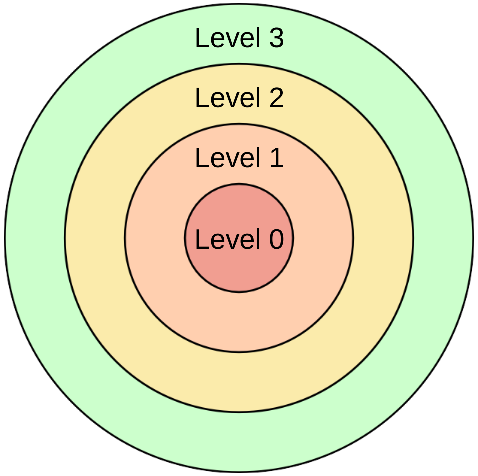
Protection and Security

- Some processors provide more than two operating modes

- These are called hierarchical protection domains or protection rings:
  - Higher-privilege rings can also access lower-privilege operations and data

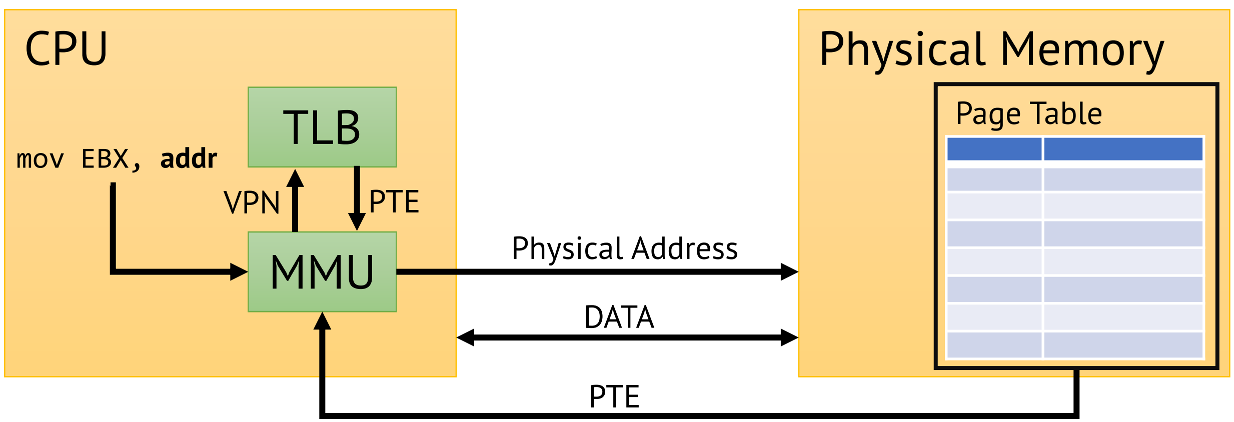
- x86 provides four operating modes
  - Level 0 = kernel mode; level 3 = user mode

- Support for multiple protection levels is ubiquitous, even in mobile devices
  - e.g. ARMv7 processors in modern smartphones have 8 different protection levels for different scenarios



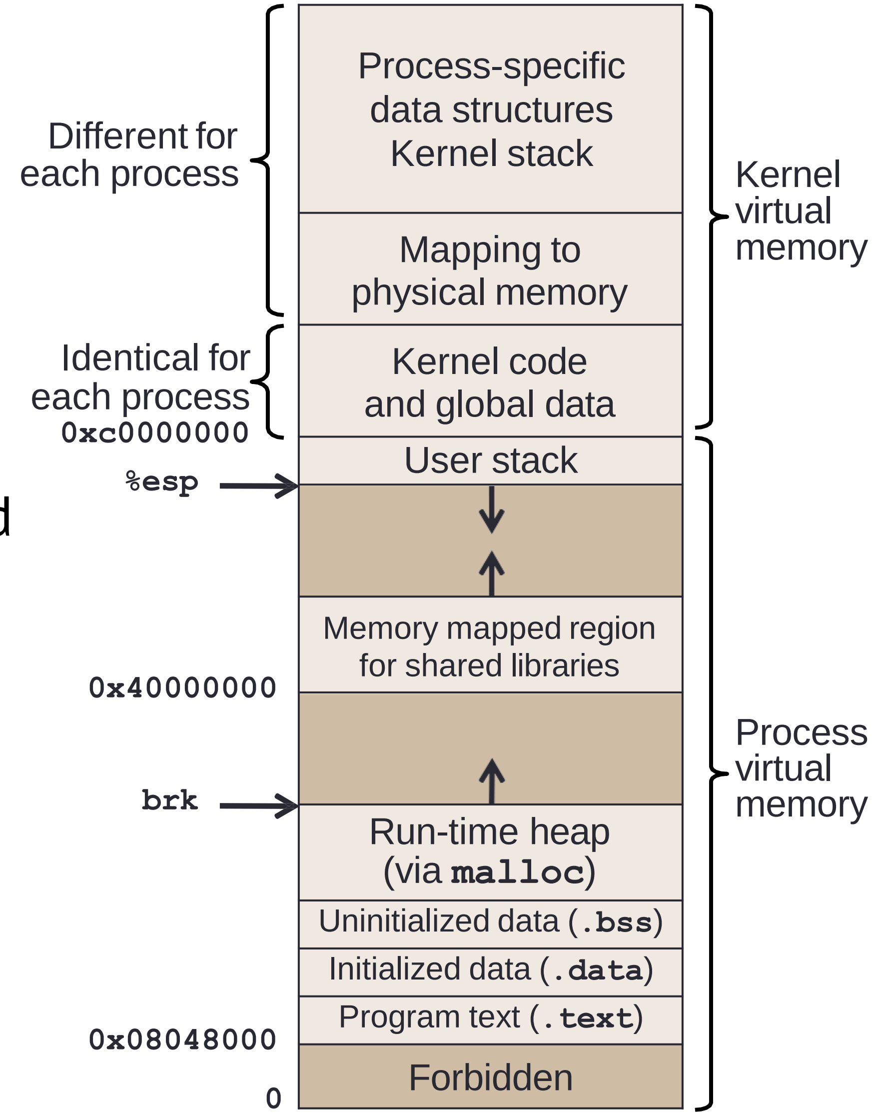
Protection and Security

- Feature 2: **virtual memory**
  - The processor maps virtual addresses to physical addresses using a page table
  - The Memory Management Unit (MMU) performs this translation
  - Translation Lookaside Buffers (TLBs) cache page table entries to avoid memory access overhead when translating addresses
  - Only the kernel can manipulate the MMU's configuration.



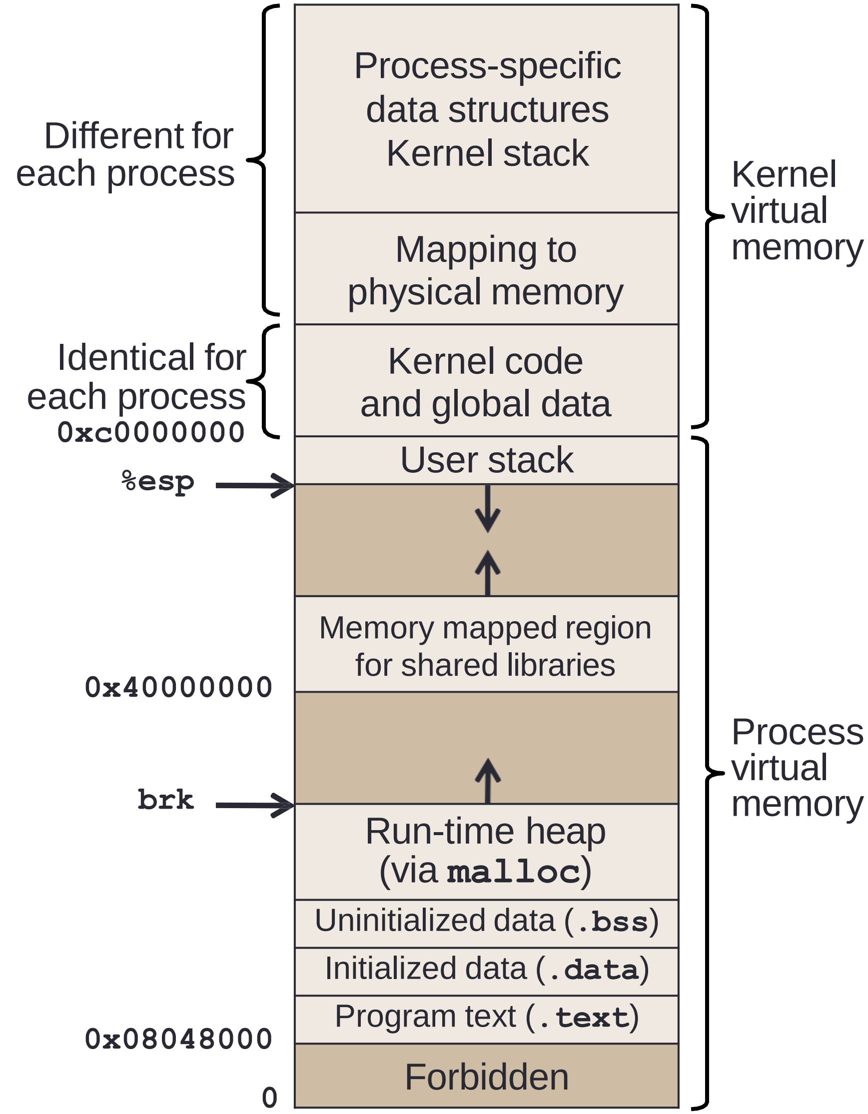
Protection and Security

- Virtual memory allows OS to give each process its own isolated address space
  - Processes have identical memory layouts, simplifying compilation, linking and loading
- Regions of memory can also be restricted to kernel-mode access only, or allow user-mode access
  - Called kernel space and user space
  - If user-mode code tries to access kernel space, processor notifies the OS
  - Only kernel can manipulate this config!



Protection and Security

- The OS must track certain details for each process
  - e.g. process' memory mapping
  - e.g. the process' scheduling configuration and behavior
- A process can't be allowed to access these details directly!
  - Just as with global kernel state, allowing direct access would open security holes
  - Processes must ask the kernel to manipulate these states on their behalf



Example: Console and File I/O

- You run a program on a Windows or UNIX system
  - The OS sets up certain basic facilities for your program to use
- Standard input/output/error streams
  - What **`printf()`** and **`scanf()`** use by default!
- Standard input/output/error streams can be from:
  - The console/terminal
  - Redirected to/from disk files
  - Your program sees the contents of a disk file on its standard input
  - What your program writes on standard output goes to a file on disk
- Redirected to/from another process!
  - Your program sees output of another process on its standard input
  - Your program’s standard output is fed to another process’ standard input

UNIX File IO

- All input/output is performed with UNIX system functions:

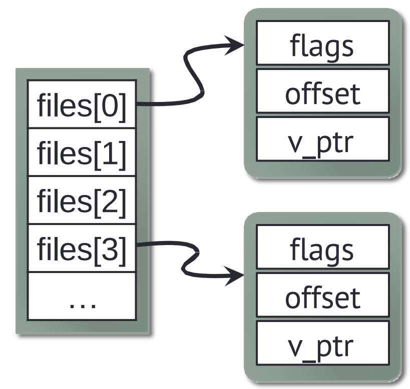
```
ssize_t read(int filedes, void *buf, size_t nbyte)
ssize_t write(int filedes, const void *buf, size_t nbyte)
```

  - Attempt to read or write nbyte bytes to file specified by **`filedes`**
  - Actual number of bytes read or written is returned by the function
  - EOF indicated by 0 return-value; errors indicated by values < 0

- The user program requests that the kernel reads or writes up to nbyte bytes, on behalf of the process:
  - **`read()`** and **`write()`** are system calls
  - Frequently takes a long time (milliseconds or microseconds; even more for user input)
  - Kernel often initiates the request, then context-switches to another process until I/O subsystem fires an interrupt to signal completion

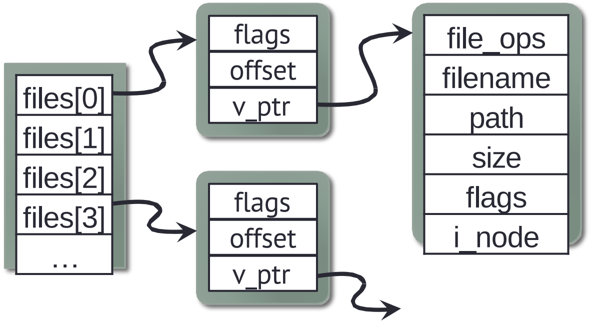
UNIX File IO

- **`filedes`** is a file descriptor
  - A non-negative integer value that represents a specific file or device
- Processes can have multiple open files
  - Each process' open files are recorded in an array of pointers
  - Array elements point to file structs describing the open file, e.g. the process' current read/write offset within the file
  - **`filedes`** is simply an index into this array
  - (Each process has a cap on total # of open files)
- Every process has this data structure, but processes are not allowed to directly manipulate it!
  - The kernel maintains this data structure on behalf of each process



UNIX File IO

- Individual **file** structs reference the actual details of how to interact with the file
  - Allows OS to support many kinds of file objects, not just disk files

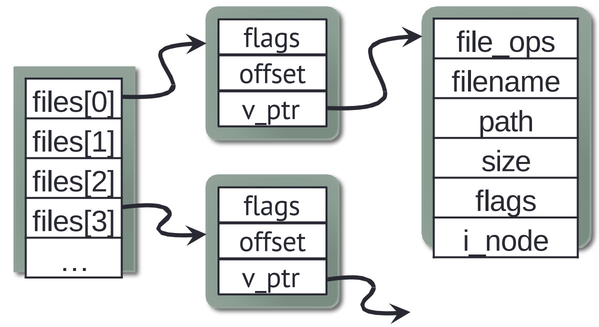


`file_ops` is a struct containing function-pointers for common operations supported by all file types, e.g.

```
struct file_operations {
  ssize_t (*read)(file *f, void *buf, size_t nb);
  ssize_t (*write)(file *f, void *buf, size_t nb);
  ...
};
```

UNIX File/Console IO

- Individual file structs reference the actual details of how to interact with the file
  - Allows OS to support many kinds of file objects, not just disk files

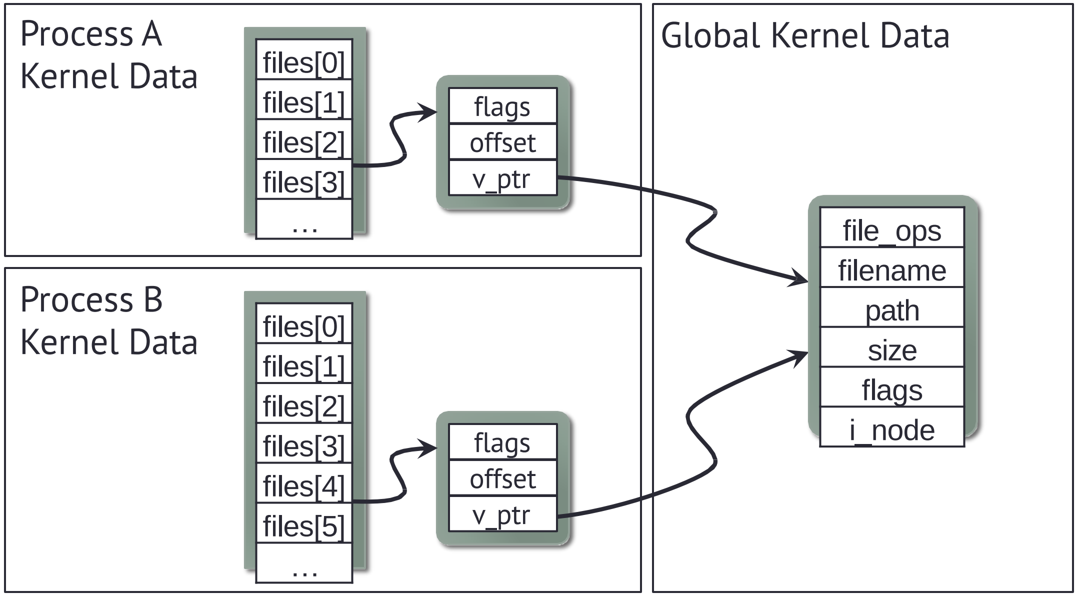


- Kernel can easily read and write completely different file types using indirection

```
// Kernel code for read(filedes,buf,nbyte)
file *f = files[filedes];
f->v_ptr->file_ops->read(file, buf, nbyte);
```

UNIX File IO

- Levels of indirection also allow multiple processes to have the same file open
  - Each process has its own read/write offset for the file
  - Operations are performed against the same underlying disk file



UNIX Standard I/O

- When a UNIX process is initialized by the OS, standard input/output/error streams are set up automatically

- Almost always:

  File descriptor 0 = standard input

  File descriptor 1 = standard output

  File descriptor 2 = standard error

- For sake of compatibility, always use constants defined in **unistd.h**

- standard header file

  STDIN_FILENO = file descriptor of standard input
  STDOUT_FILENO = file descriptor of standard output
  STDERR_FILENO = file descriptor of standard error

UNIX Standard I/O and Command Shells

- Most programs don't really care about where **stdin** and **stdout** go, if they work
  - Command shells care very much!

- **`grep Allow < logfile.txt > output.txt`**
  - Shell sets grep's **stdin** to read from logfile.txt
  - Shell sets grep's **stdout** to write to the file output.txt
  - (If output.txt exists, it is truncated)

- Once **stdin** and **stdout** are properly set, grep is invoked:
  argc = 2, argv = {"grep", "Allow", NULL}