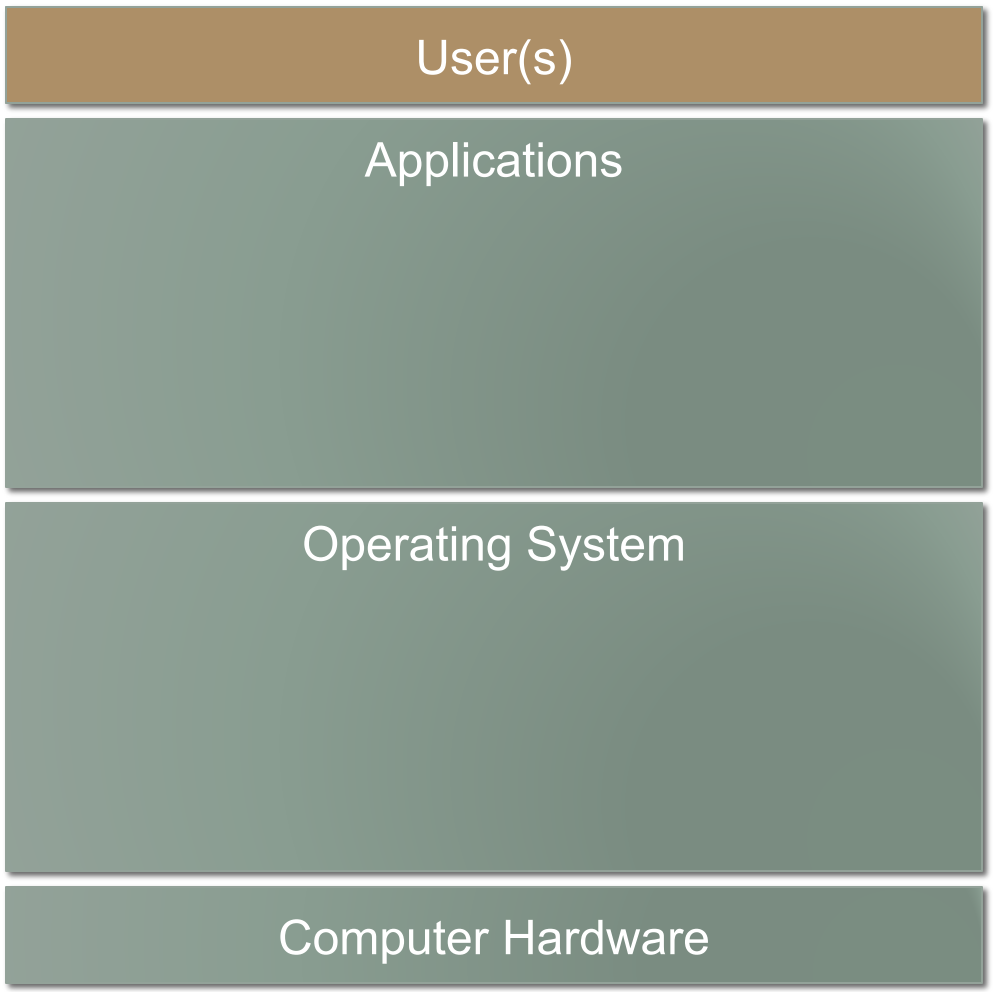
OS COMPONENTS
OVERVIEW OF UNIX FILE I/O

CS124 – Operating Systems

Spring 2024, Lecture 2

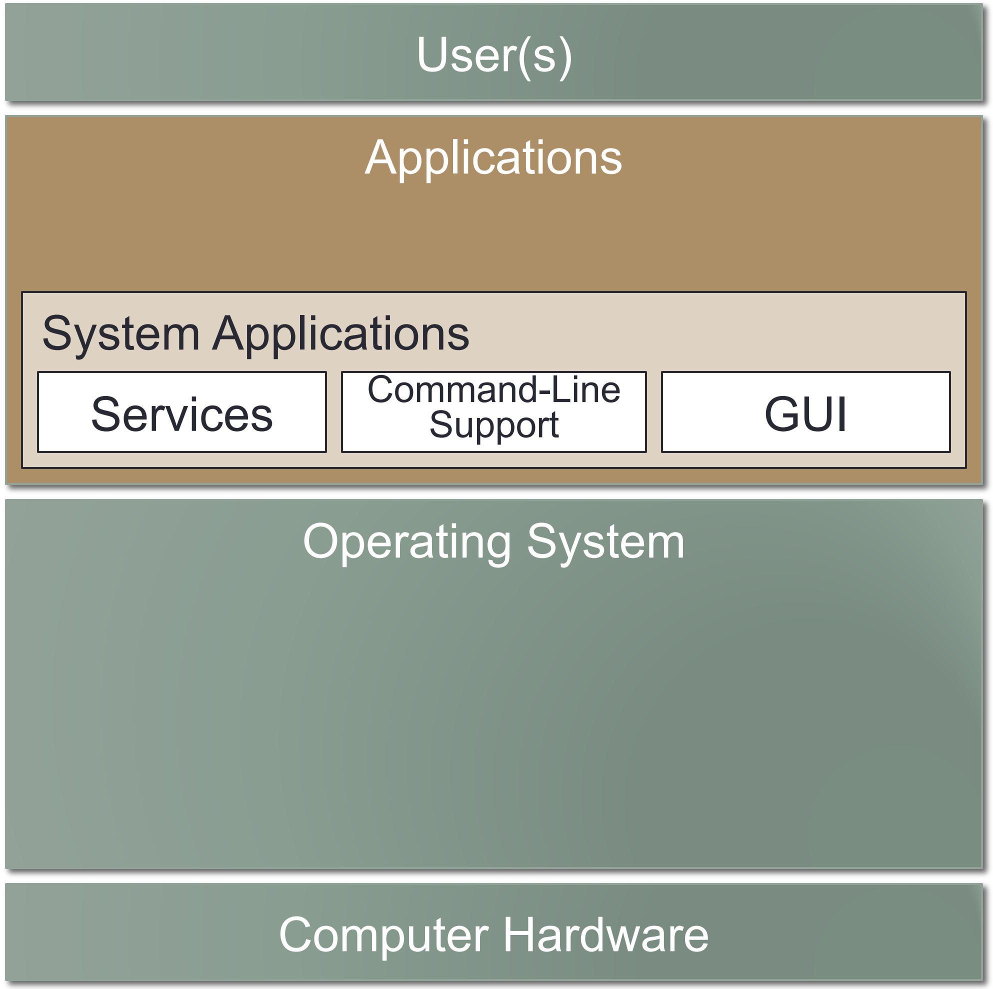
Operating System Components (1)

- Common components of operating systems:
- Users:
  - Want to solve problems by using computer hardware
  - OS may support only one user at a time, or many concurrent users, depending on system requirements
  - Some systems usually have no users, so they have an extremely minimal UI
    - e.g. automobile engine computers



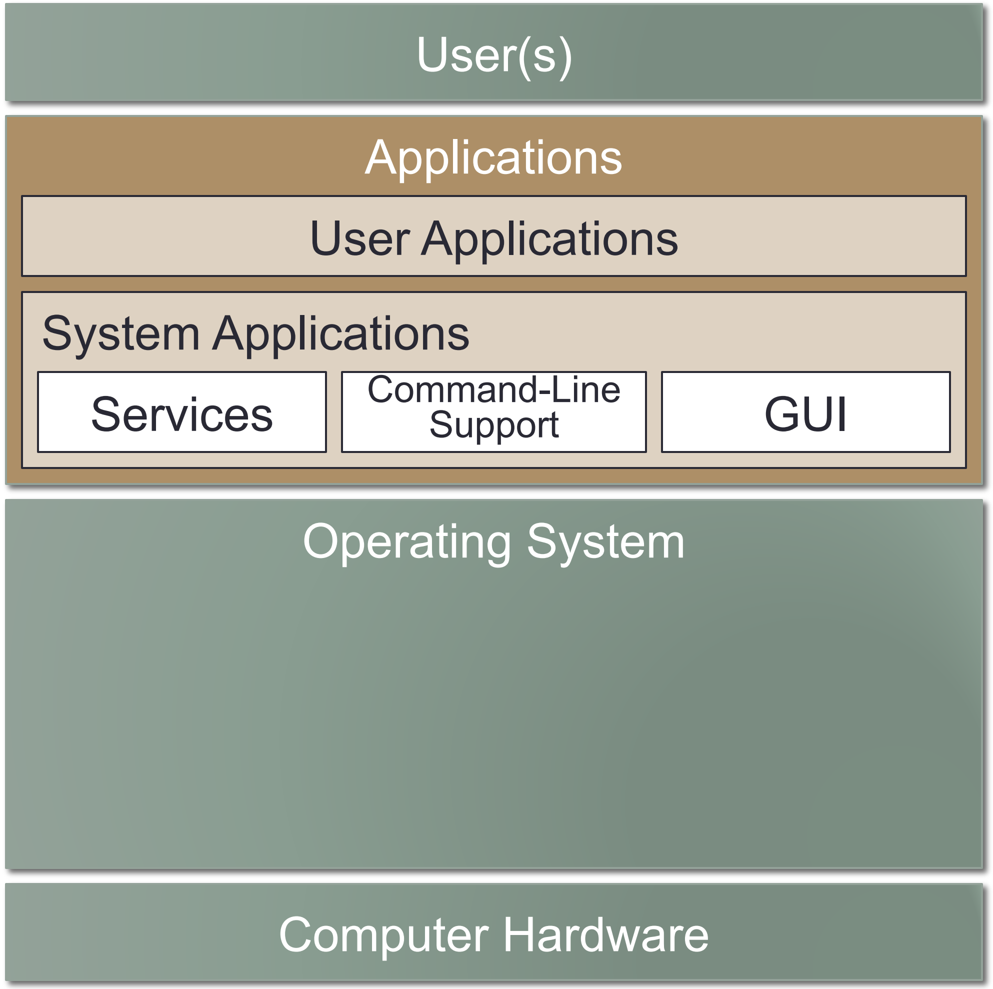
Operating System Components (2)

- Common components of operating systems:
- Applications allow users to solve problems with the computer's resources
  - Applications rely on the OS to manage those resources
- Some applications are provided by the operating system
  - Services for providing and managing system resources
  - Command shells (e.g. sh, csh, zsh, bash)
  - GUI programs (X-server, system config tools, etc.)



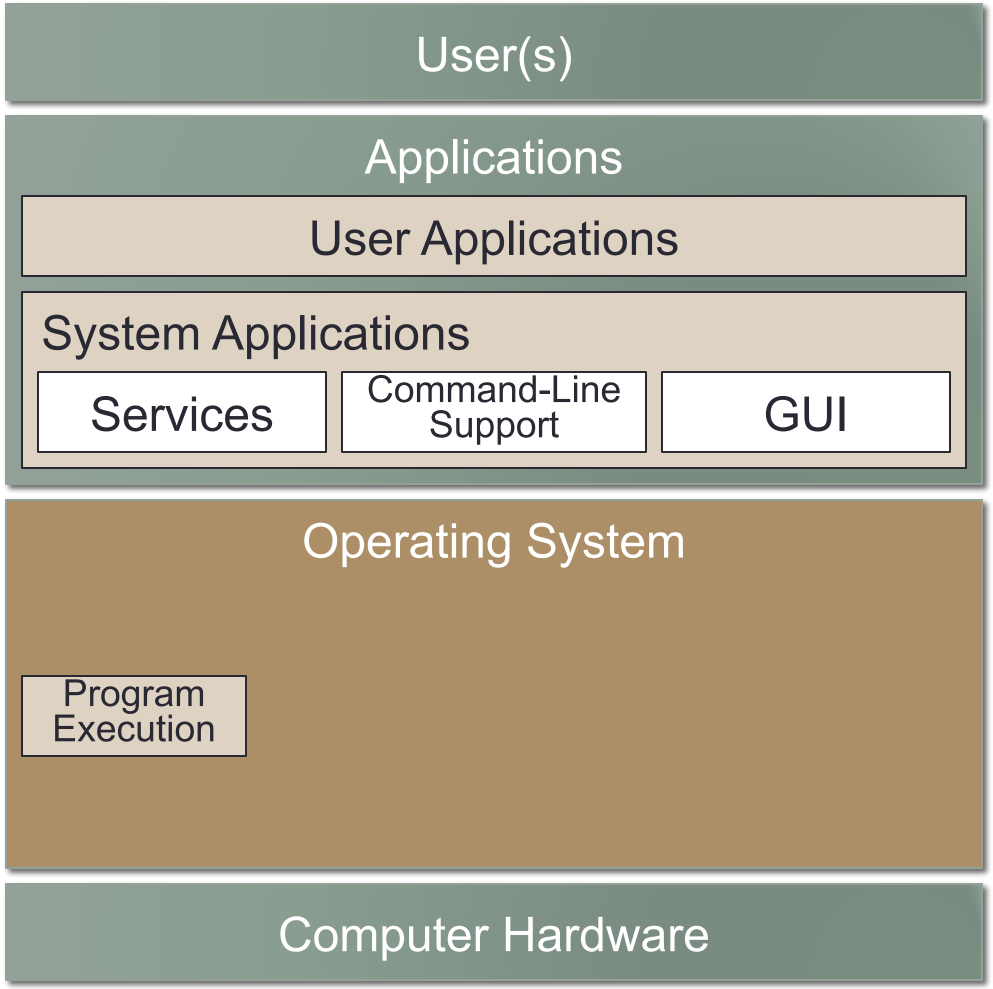
Operating System Components (3)

- Common components of operating systems:
- Applications allow users to solve problems with the computer's resources
  - Applications rely on the OS to manage those resources
- User applications are designed to solve specific problems
  - e.g. text editors, compilers, web servers
  - e.g. web browsers, word processors, spreadsheets



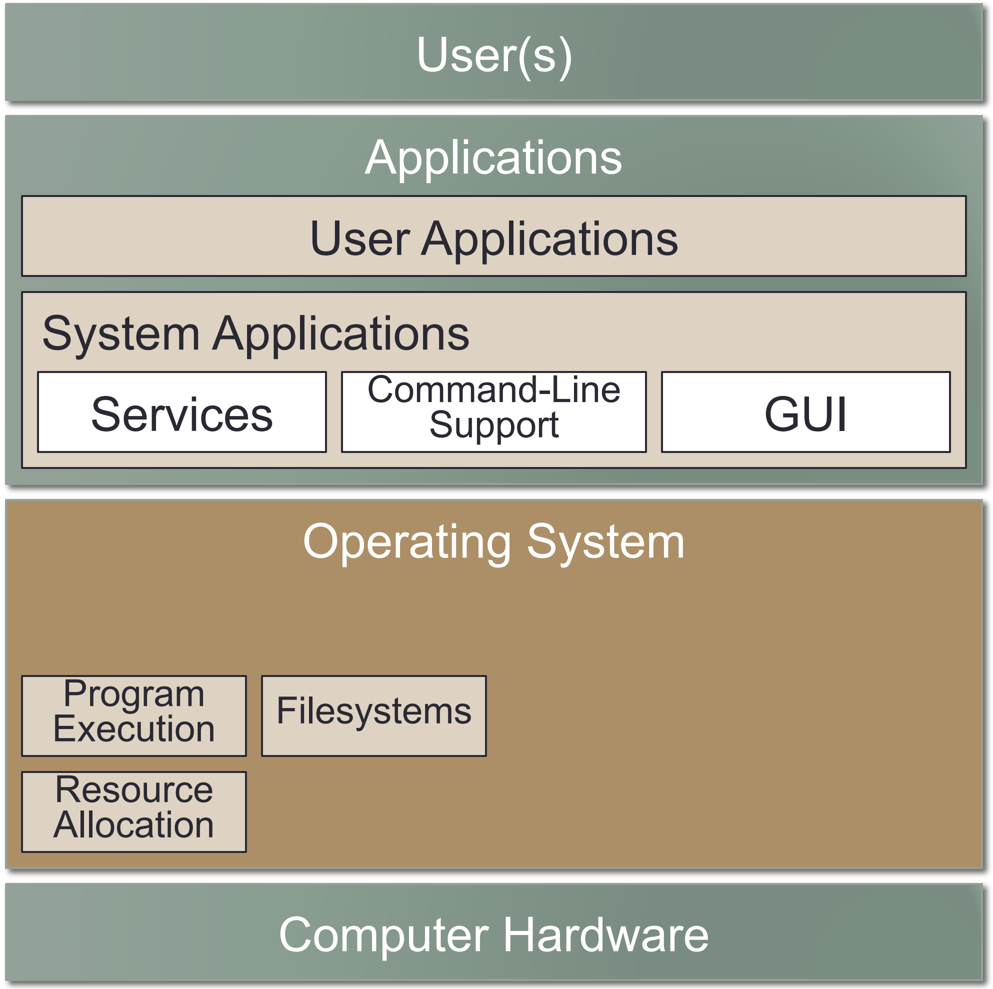
Operating System Components (4)

- Common components of operating systems:
- The OS itself can provide many different facilities
  - Not every OS provides all of these facilities…
- Most obvious facility: program execution
  - Load and run programs
  - Optionally, ability to perform runtime linking if the OS supports shared libraries
  - Handle program termination (possibly with errors!)
  - Pass along signals, etc.



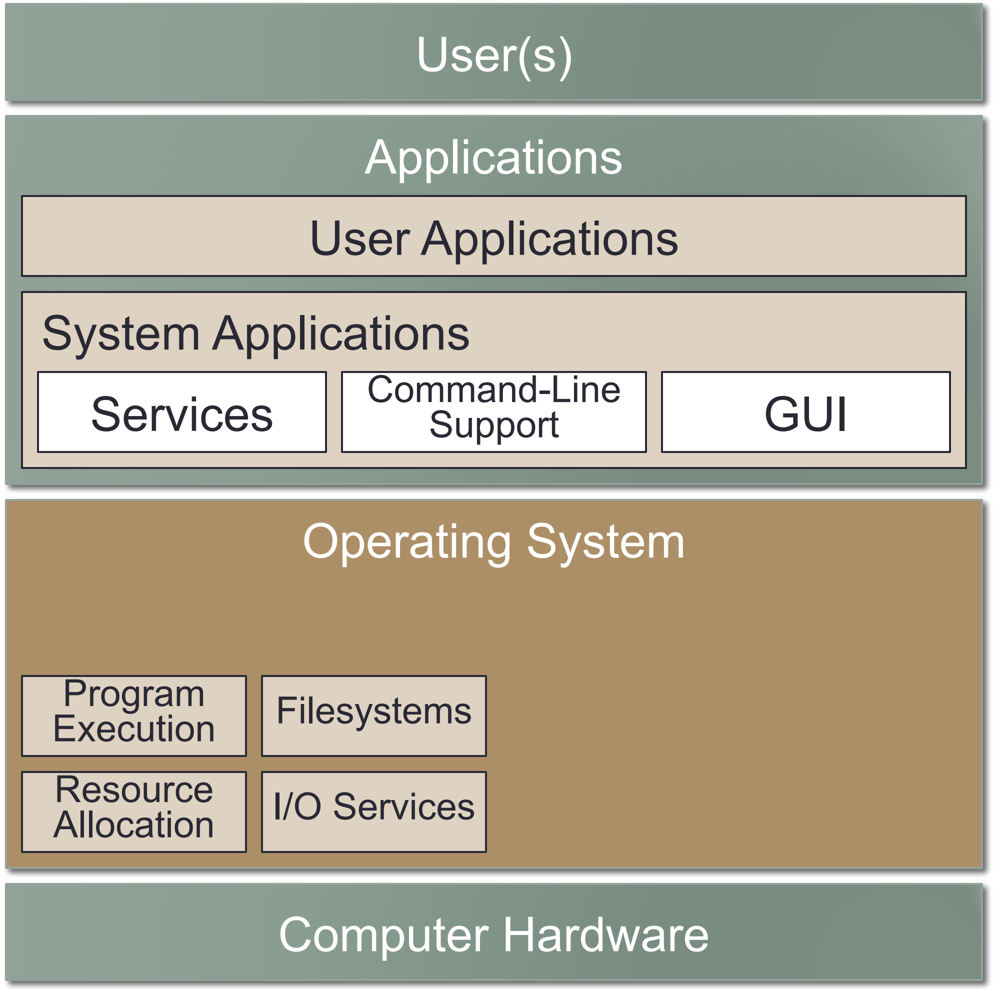
Operating System Components (5)

- Common components of operating systems:
- Another obvious facility: resource allocation
- Resources to manage:
  - Processor(s) – especially if OS supports multitasking
  - Main memory
  - Filesystem/external storage
  - Other devices/peripherals
- Filesystems:
  - OS usually supports several different filesystems
  - May also require periodic maintenance



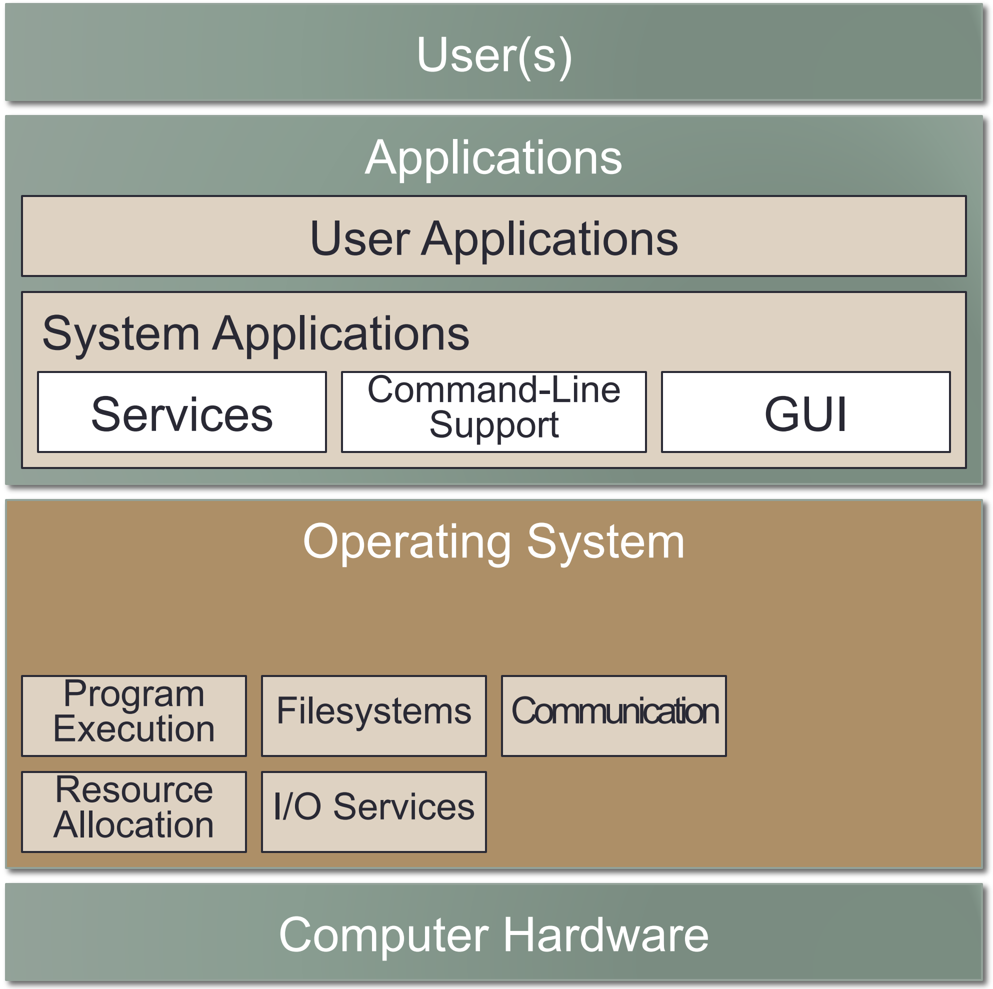
Operating System Components (6)

- Common components of operating systems:
- Disks and other peripheral devices require specific interactions to function properly
  - I/O subsystem provides facilities to control computer hardware devices
  - Often interact via I/O ports
  - Do not want apps to do this!
- Usually modularized by using a device-driver abstraction
  - Present a clean abstraction for the rest of the OS to use
  - Encapsulate gory details of talking to hardware



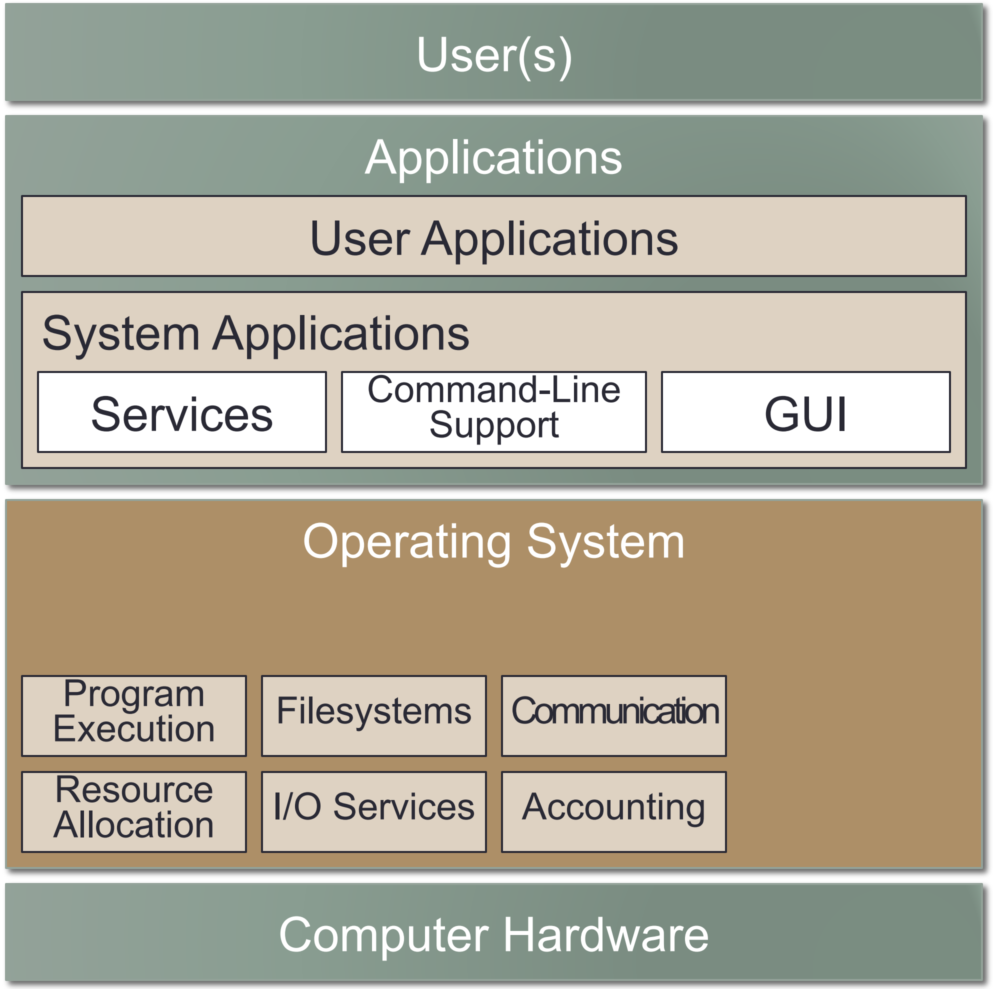
Operating System Components (7)

- Common components of operating systems:
- Many components of OS require **communication**
- Collaborating processes need to share information
  - Called **Inter-Process Communication** (IPC)
  - Many mechanisms: pipes, shared memory, message-passing, local sockets, etc.
- Some processes need to communicate with other computer systems
  - Many kinds of **networking**



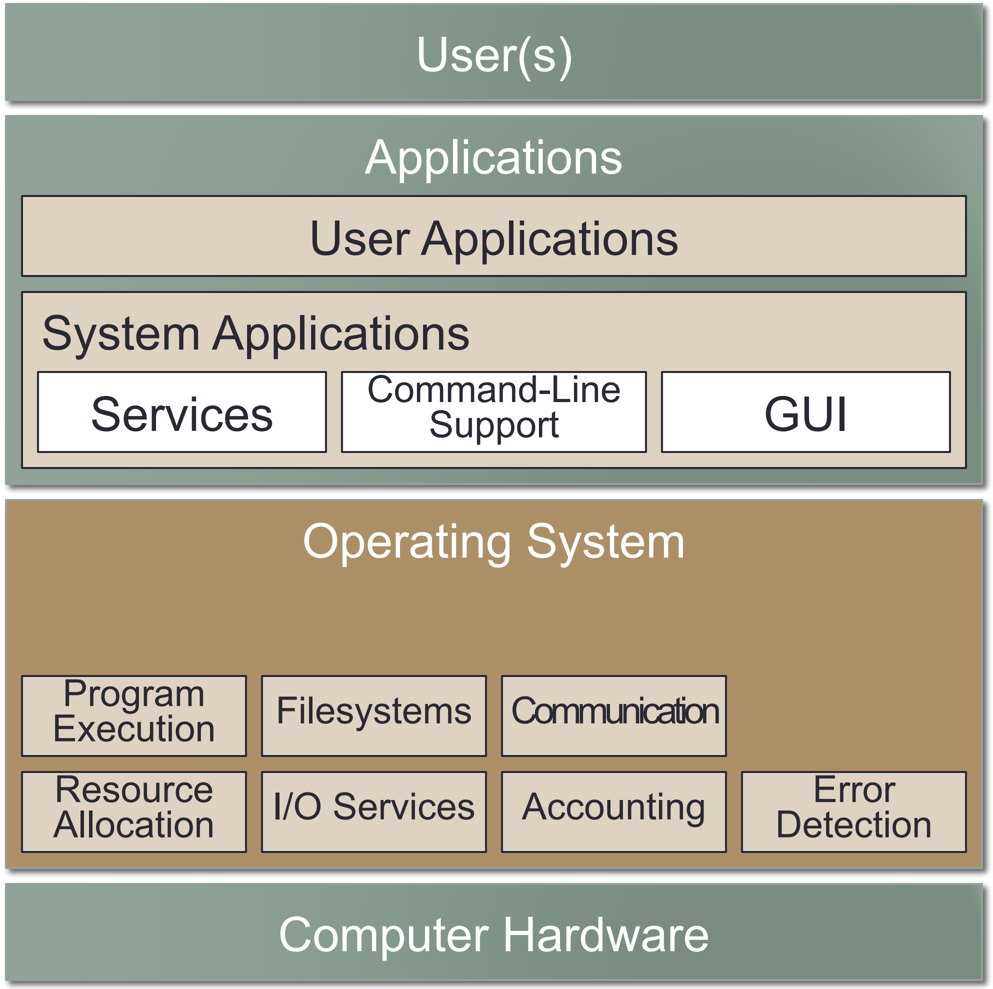
Operating System Components (8)

- Common components of operating systems:
- Some OSes record resource usage data
  - **Accounting** facility
- Purpose: systems that bill users based on CPU usage, storage, network
- Very common to bill customers for storage and network use
- Also, with hypervisors, very easy to bill per-VM for CPU use



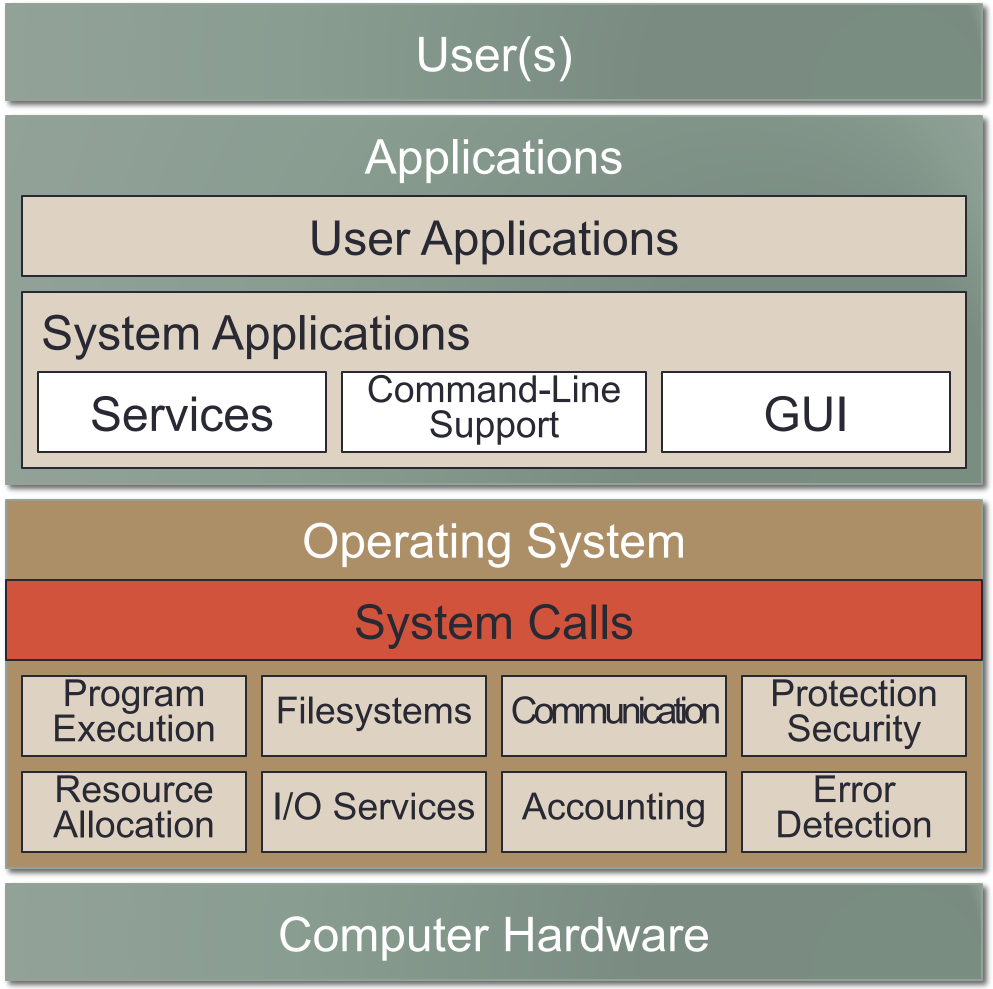
Operating System Components (9)

- Common components of operating systems:
- OSes must handle various errors that occur
  - Varies widely, depending on what the hardware can detect
- Common errors:
  - Hard disk is full, or broken
  - Filesystem is corrupt
  - Memory errors
  - A program behaves in an invalid way
  - Printer has no paper or ink
- Less common errors:
  - Processor failure, etc.



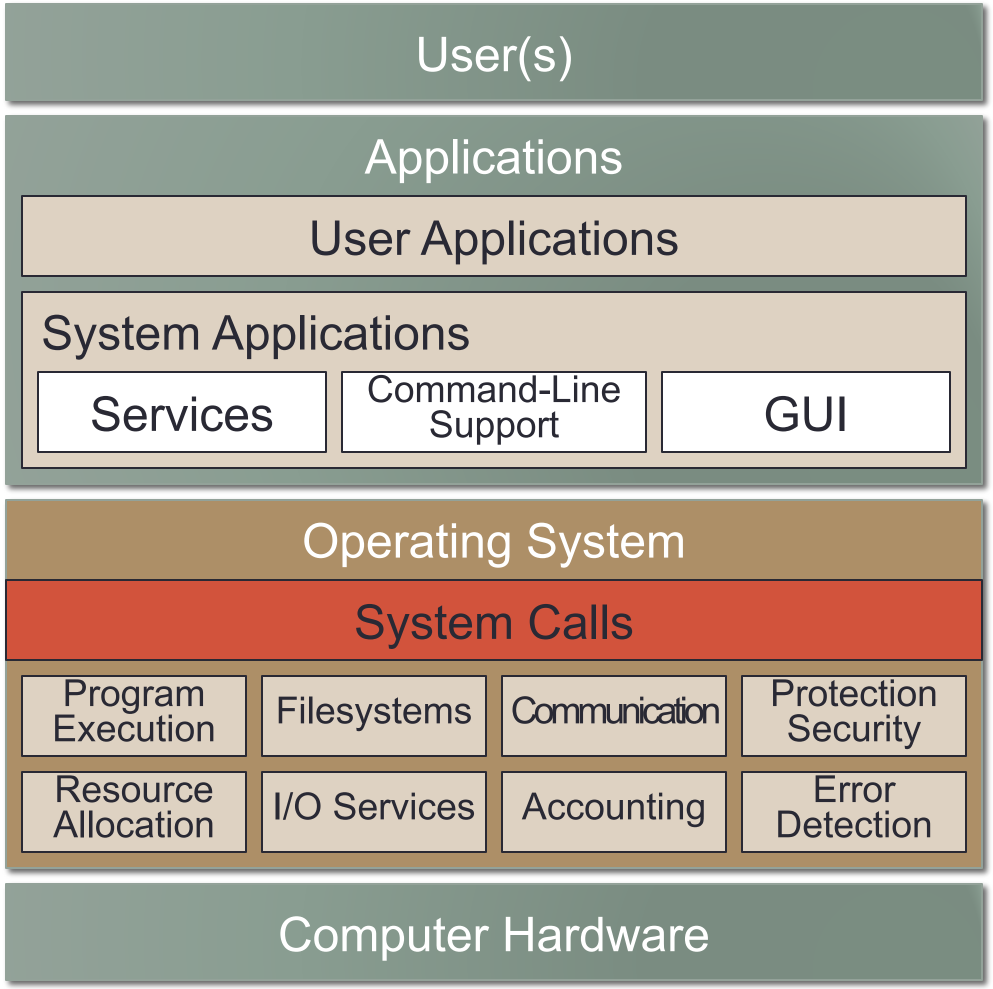
Operating System Components (10)

- Common components of operating systems:
- OSes must prevent many different kinds of abuses
- OS must be able to protect itself from malicious programs
- Applications are not allowed to directly access operating system code or data
  - (Computer hardware must provide this capability…)
- All application-interactions with OS are performed via **system calls**



Operating System Components (11)

- Common components of operating systems:
- Operating system must also protect processes from each other
  - A process should not be allowed to access another process' data, unless this is specifically allowed by the process
- Again, this requires specific support from the computer hardware

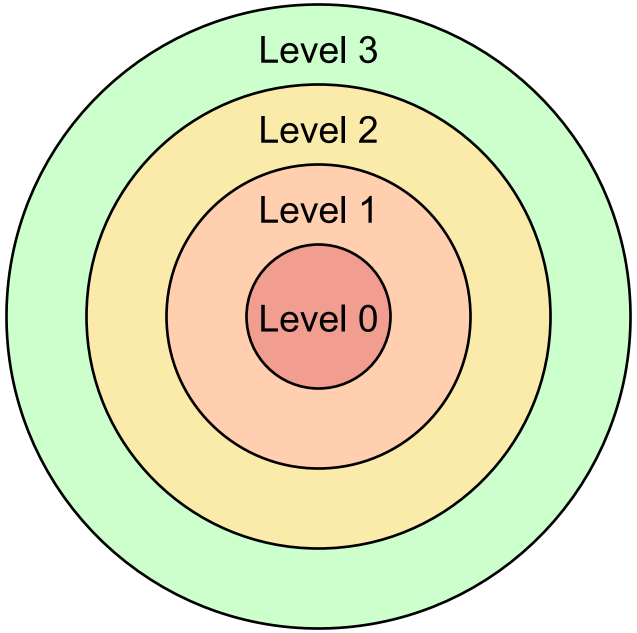


Protection and Security

- Will talk much more about computer hardware in future…
- 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 requirement:
  - **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 operating system **kernel** is the part of the OS that runs in kernel mode
  - The OS may have [many] other components running in user mode

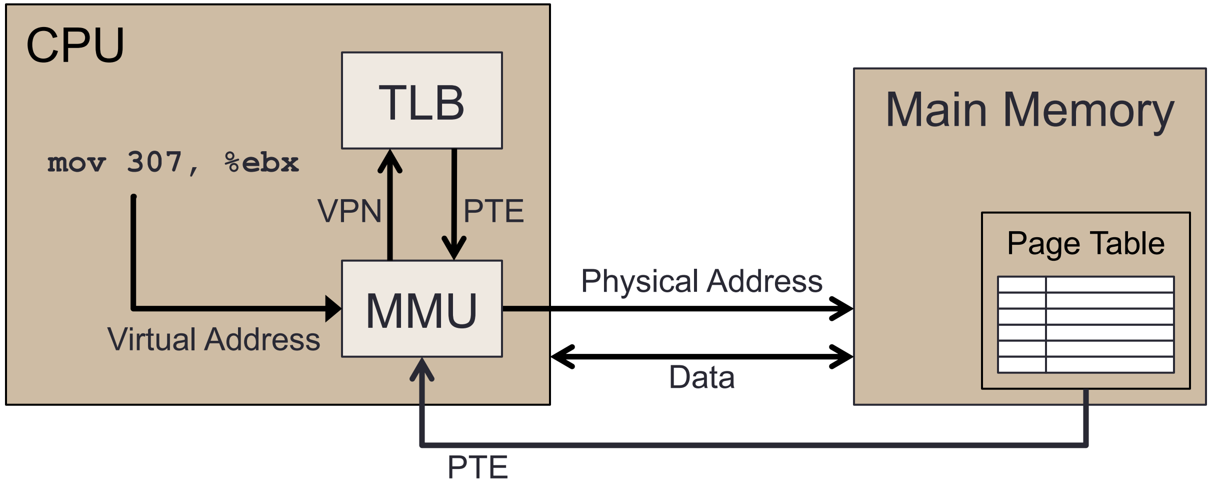
Protection and Security (2)

- Some processors provide more than two operating modes
- Called **hierarchical protection domains** or **protection rings**
  - Higher-privilege rings can also access lower-privilege operations and data
- IA32 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



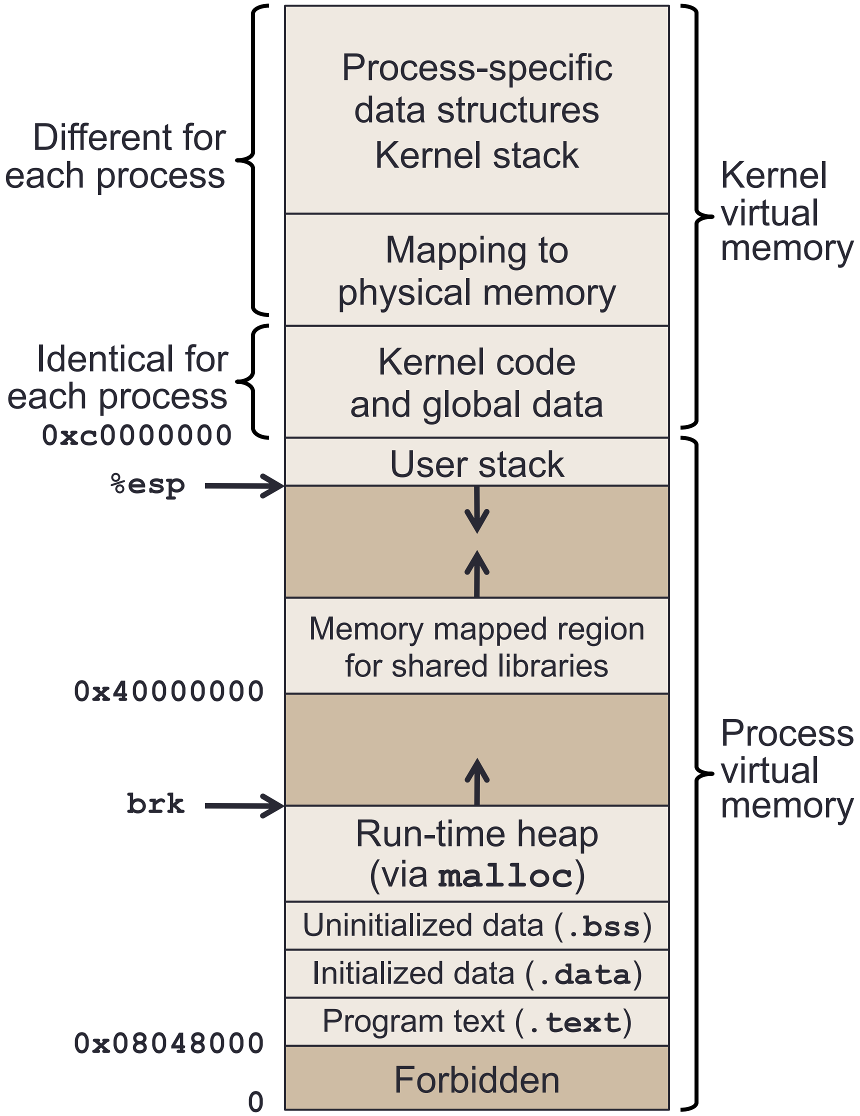
Protection and Security (3)

- 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, etc.**
- Again, will discuss virtual memory much more in the future



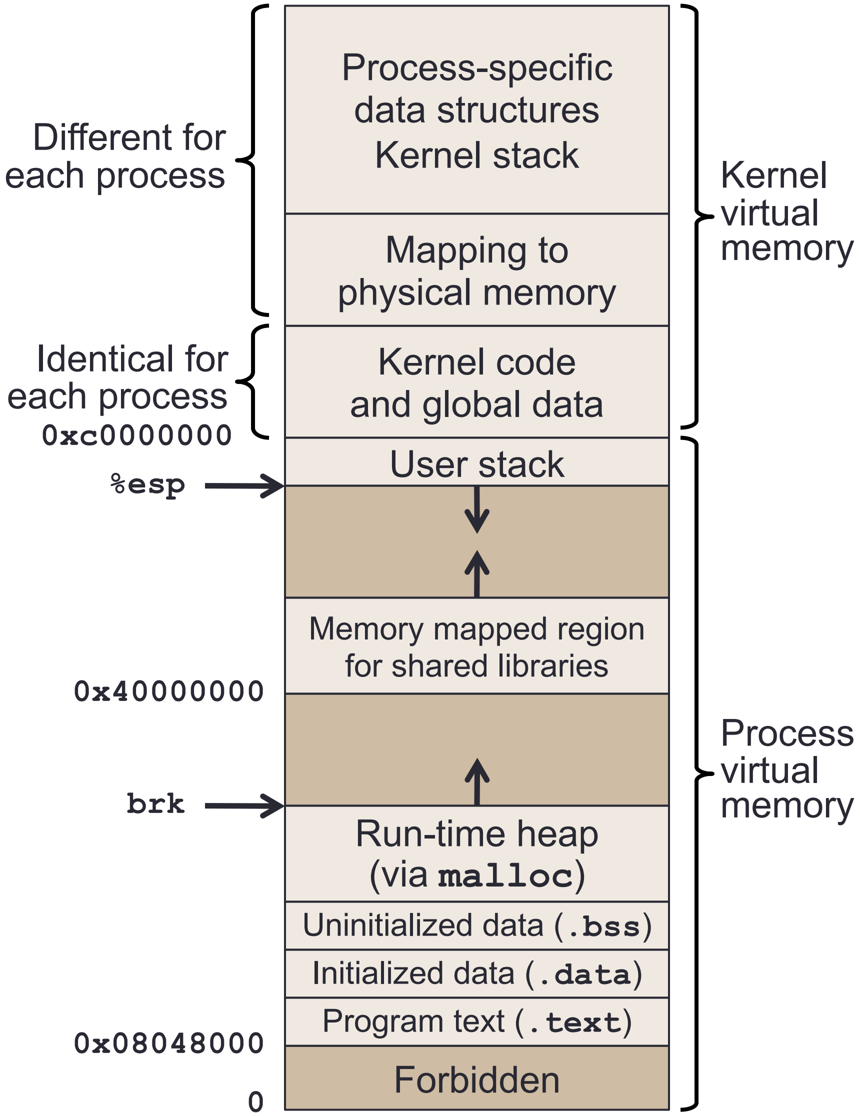
Protection and Security (4)

- 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 configuration!



Protection and Security (5)

- 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
  - Process must ask the kernel to manipulate this state on its behalf
- Example: Console and file IO



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/Console IO

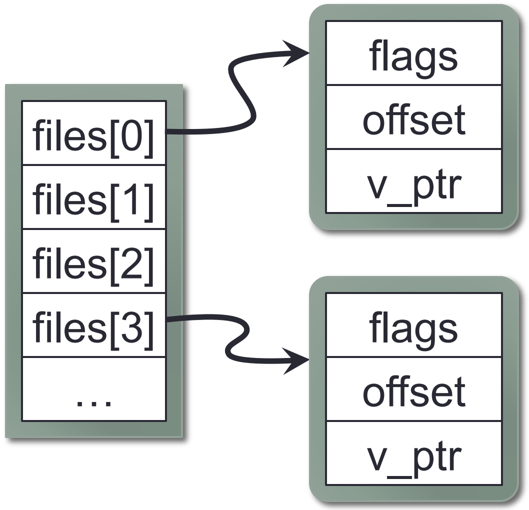
- All input/output is performed with UNIX system functions:

```
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

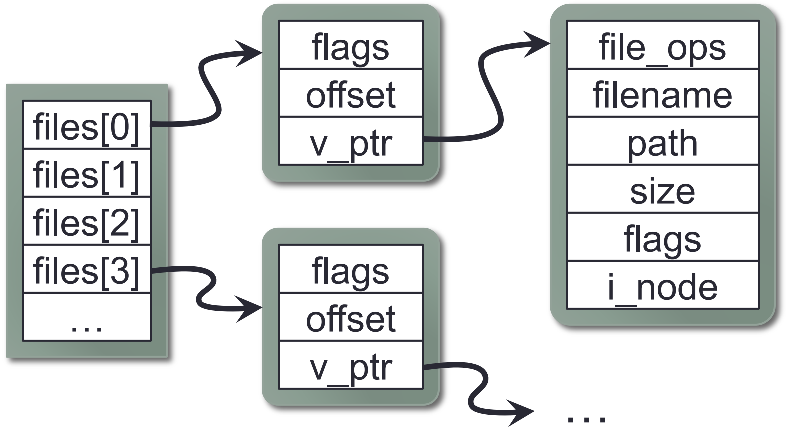
UNIX File/Console IO (2)

- **`filedes`** is a **file descriptor**
  - A nonnegative 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/Console IO (3)

- 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

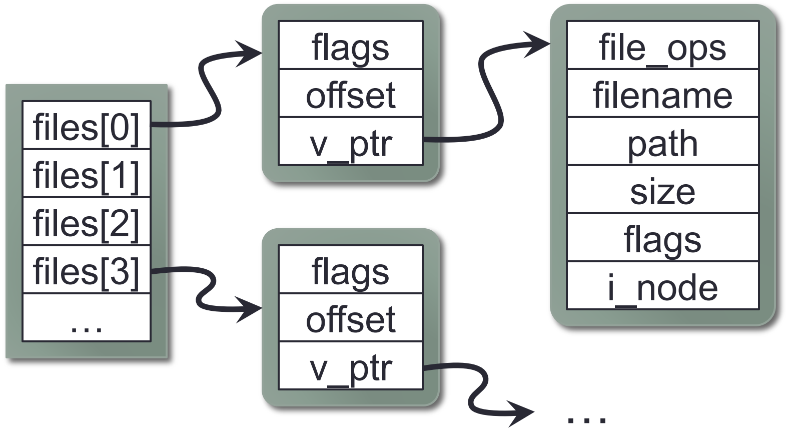


- **`file_ops`** is a struct containing function-pointers for common operations supported by all file types, e.g.

```
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 (4)

- 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

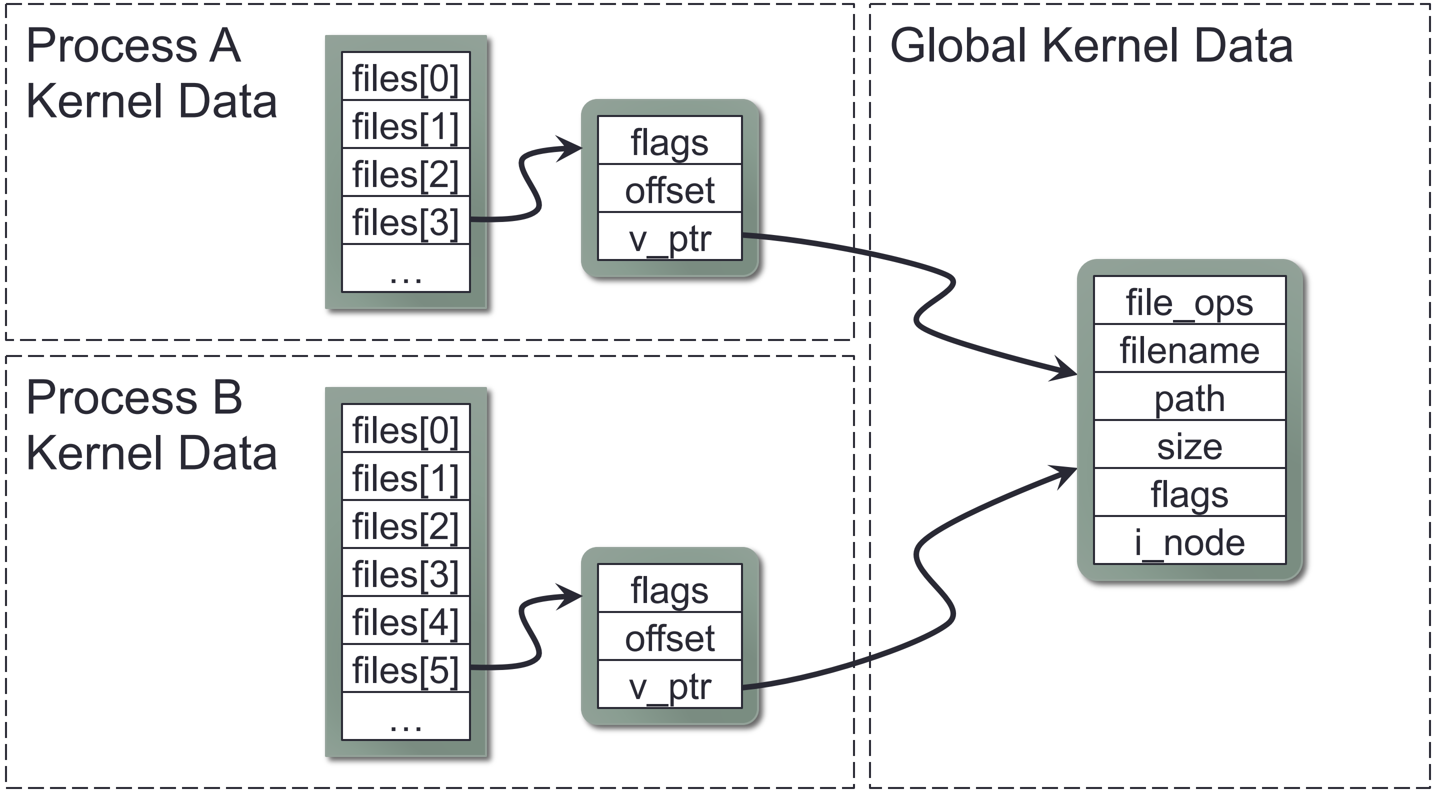


- Kernel can easily read and write completely different file types using indirection

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

UNIX File/Console IO (5)

- 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, <u>always</u> 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, as long as 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}`**

UNIX Command Shell Operation

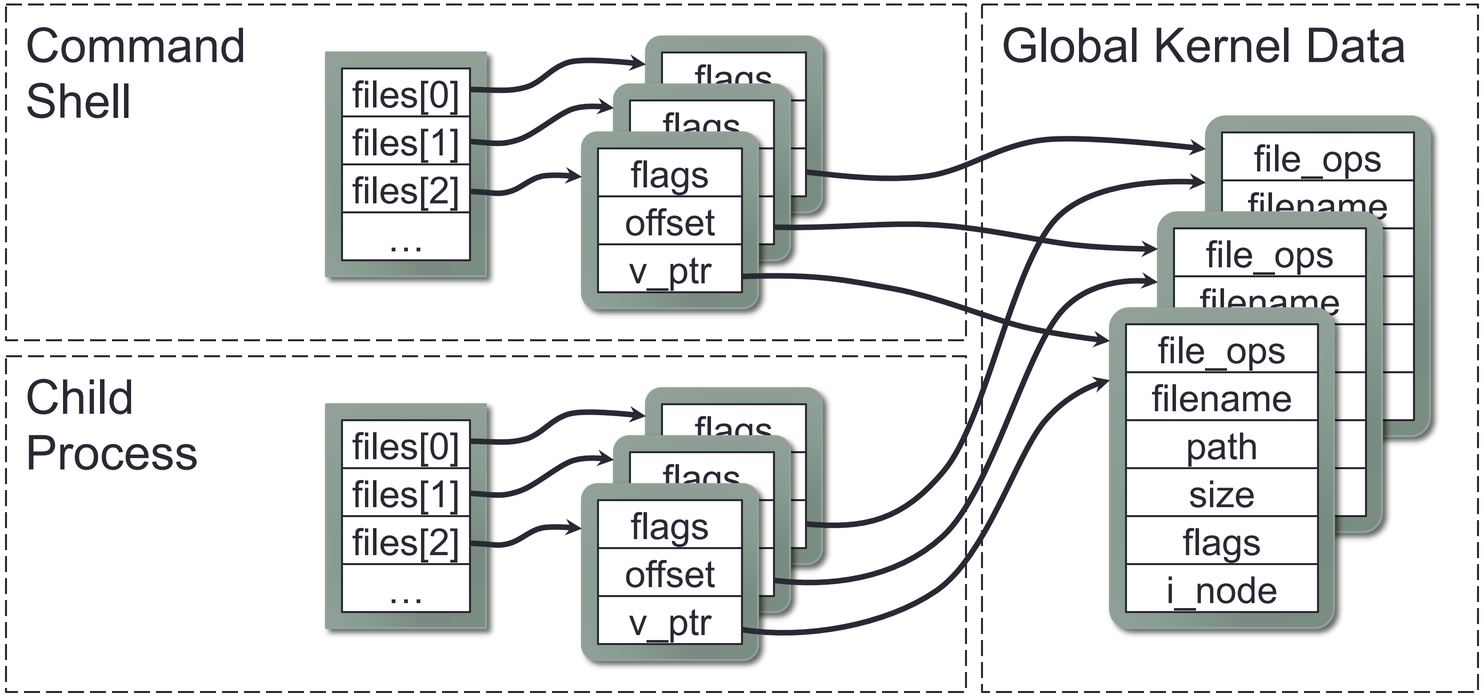
- UNIX command shells generally follow this process:
  1. Wait for a command to be entered on the shell's standard input (usually entered by a user on the console, but not always!)
  2. Tokenize the command into an array of **`tokens`**
  3. If **`tokens[0]`** is an internal shell command (e.g. **`history`** or **`export`**) then handle the internal command, then go back to 1.
  4. Otherwise, **`fork()`** off a child process to execute the program. **`wait()`** for the child process to terminate, then go back to 1.
- Child process:
  1. If the parsed command specifies any redirection, modify stdin/stdout/stderr based on the command, and remove these tokens from the tokenized command
  2. **`execve()`** the program specified in **`tokens[0]`**, passing **`tokens`** as the program's arguments
  3. If we got here, execution failed (e.g. file not found)! Report error.

Command Shell and Child Process

- How does the child process output to the command shell's standard output? How does it get the shell's stdin?
- When a UNIX process is forked, it is a *near-identical* copy of the parent process
  - Only differences: process ID and parent process ID
- Specifically, the child process has the same files open as the parent process
  - And they have the exact same file descriptors

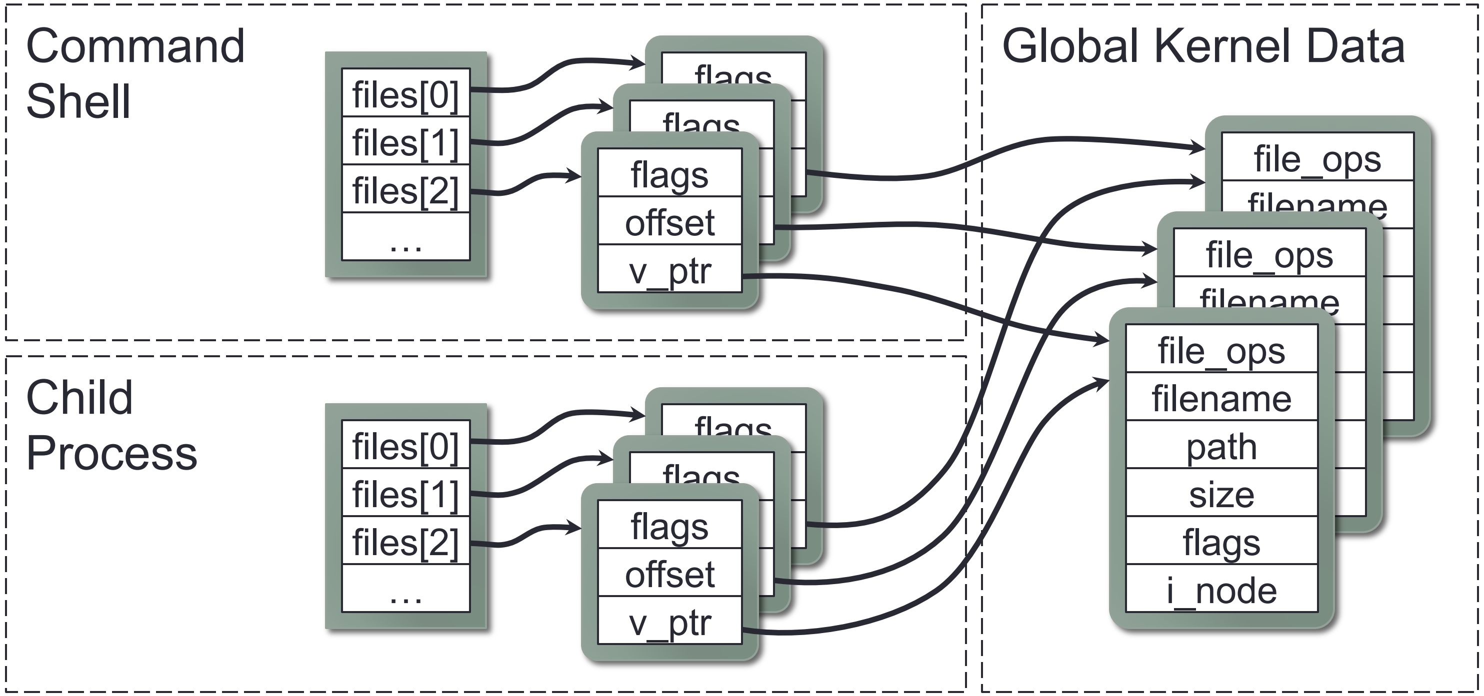
Command Shell and Child Process (2)

- When child process reads stdin and writes stdout/stderr, it writes the exact same files that the command-shell has as stdin/stdout/stderr



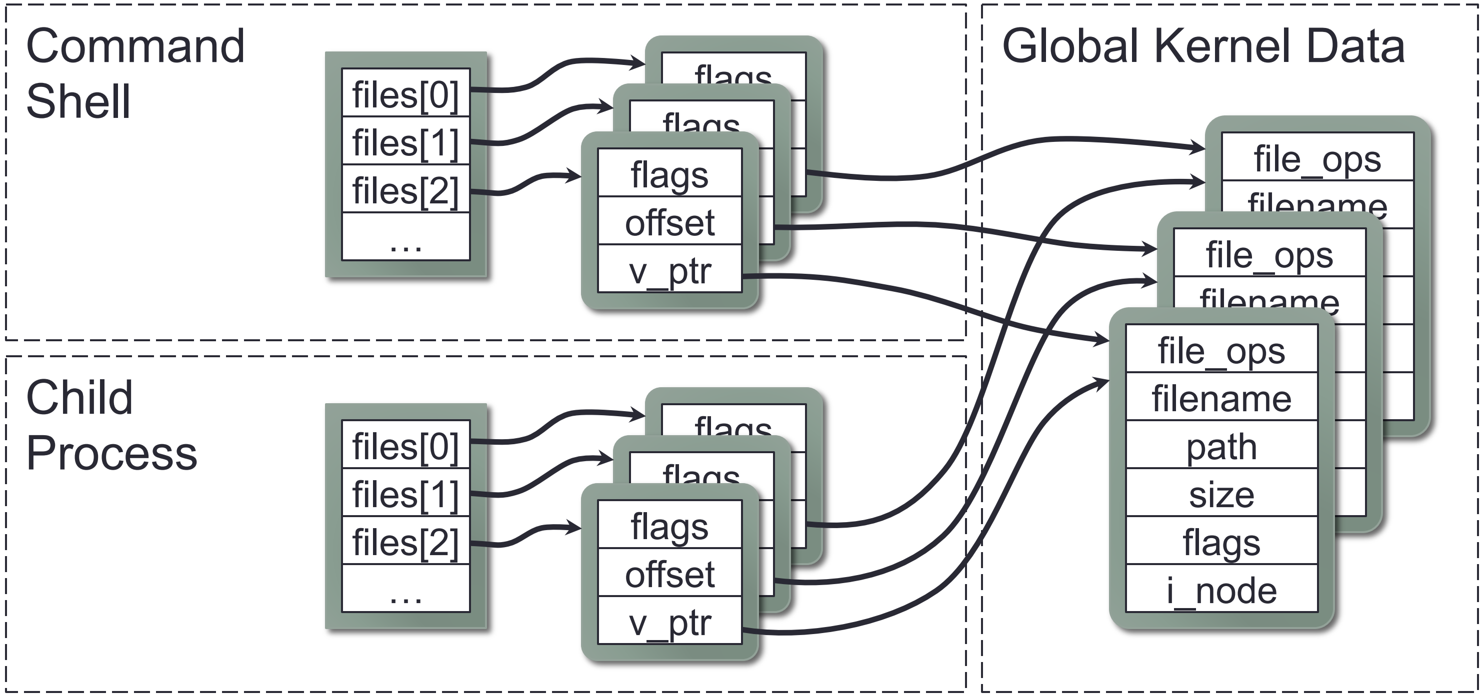
Command Shell and Child Process (3)

- If command redirects e.g. output to a file, clearly can't have the command-shell process do it before forking…
  - Would work fine for the child process, but the command-shell's I/O state would be broken for subsequent commands



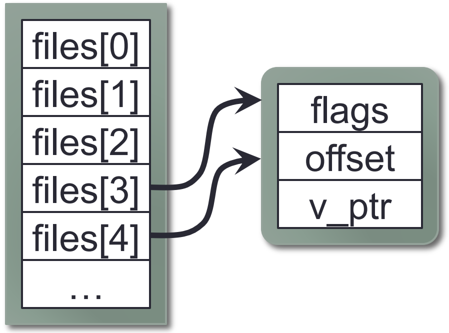
Command Shell and Child Process (4)

- Child process must set up stdin, stdout, and stderr before it executes the actual program
- How does a process change what file is referenced by a given file descriptor?
  - Process must ask the kernel to modify the file descriptors



Manipulating File Descriptors

- UNIX provides two system calls: **`dup()`** and **`dup2()`**
- **`int dup(int filedes)`**
  - Duplicates the specified file descriptor, returning a new, previously unused file descriptor
- Note that the internal **`file`** struct is <u>not</u> duplicated, only the pointer to the **`file`** struct!



- Implication:
  - Reads, writes and seeks through both file descriptors affect a single shared file-offset value
- Even though the one file has two descriptors, should call **`close()`** on each descriptor
  - Remember: each process has a maximum number of open files
  - (Kernel won't free the **`file`** struct until it has no more references)

Manipulating File Descriptors (2)

- **`int dup2(int filedes, int filedes2)`**
  - Duplicates the specified file descriptor into the descriptor specified by **`filedes2`**
  - If **`filedes2`** is already an open file, it is closed before **`dup2()`** duplicates **`filedes`**
    - (Unless **`filedes == filedes2`**, in which case nothing is closed)
- This function allows the command-shell's child process to redirect standard input and output
  - e.g. to replace stdout with a file whose descriptor is in **`fd`**:
    `dup2(fd, STDOUT_FILENO);`
- As before, the file descriptor that was duplicated should be closed to keep from leaking descriptors
  - **`close(fd);`**

Manipulating File Descriptors (3)

- Previous example:
  - `grep Allow < logfile.txt > output.txt`
- After command shell forks off a child process, the child can execute code like this, before it starts **`grep`**:

```
int in_fd, out_fd;

in_fd = open("logfile.txt", O_RDONLY);
dup2(in_fd, STDIN_FILENO);     /* Replace stdin */
close(in_fd);

out_fd = open("output.txt", O_CREAT | O_TRUNC | O_WRONLY, 0);
dup2(out_fd, STDOUT_FILENO);  /* Replace stdout */
close(out_fd);
```

Next Time

- Operating system architectural approaches
- Overview of computer hardware and interface with OS