Intro to Operating Systems

#code #notes

Operating System Topics

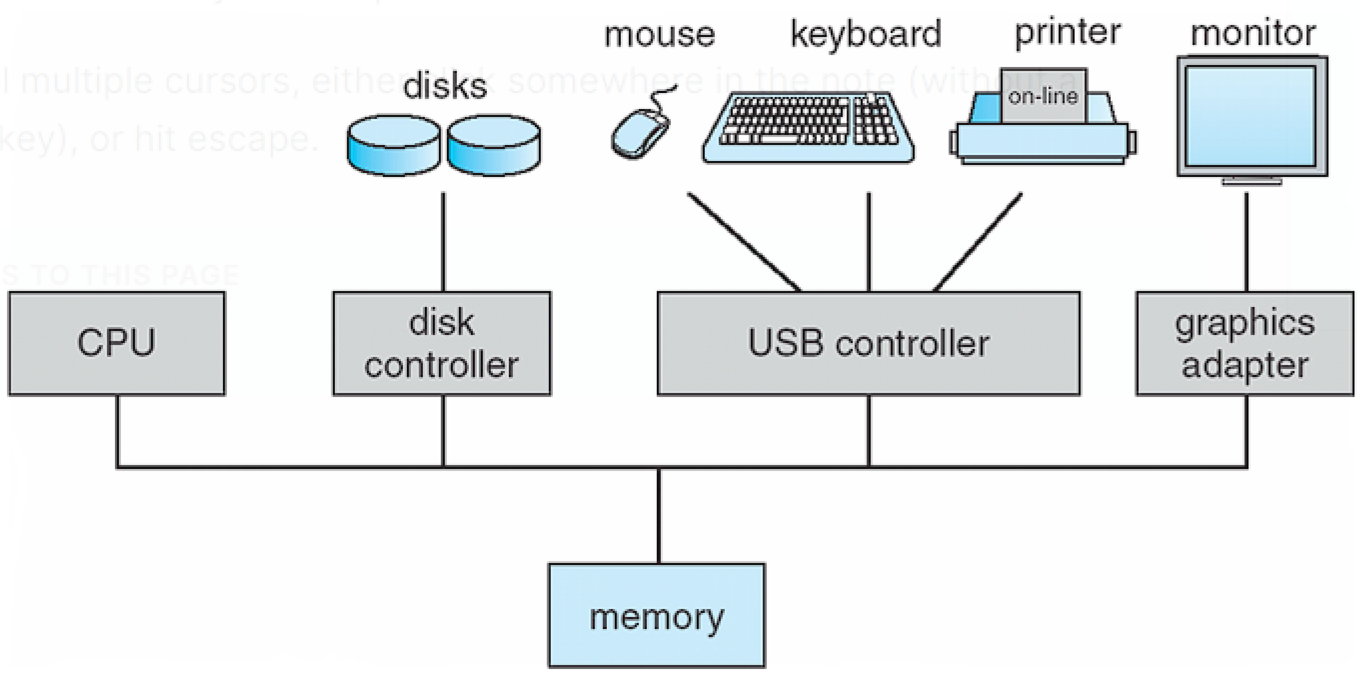
What is an OS?

Main functions of an Operating System:
Resource Allocator
Manages resources (HW) and resolves conflicting requests for *efficient* and *fair* use (e.g. accessing disk)
Control System
Controls execution of programs, preventing errors and improper use (e.g. protects one process from crashing another)

Bootstrapping OS

- Small bootstrap program loaded at boot, typically stored in (EP)ROM, known as firmware (BIOS)
- Inits the system (detects devices, checks mem for errors)
- Loads kernel and starts exec

Computer System Organisation



CPUs and device controllers connect via common bus to shared mem
These devices compete for memory cycles

- IO and CPU execute concurrently
- Each device controller (controller chip) in charge of particular device type

- Device controller has local buffer (memory for data or control registers)
- CPU moves data between main mem and controller buffers (write to screen, read coordinates from mouse)
- IO is from device to controller buffer

Interrupts

- Interrupts transfer control to *Interrupt service routine (ISR)* through interrupt vector which contains addresses of service routines
- Interrupt architecture saves address of interrupted instruction to be able to resume processing
- Incoming interrupts disabled while another is being processed to prevent losing interrupt
- A trap is software-generated interrupt caused from error or user request (i.e. dividing by zero)

Storage Structure

- Main mem is only large storage that CPU accesses directly
- Secondary storage provides large non-volatile storage
    - Mag disks are an example. Disk surfaces split logically into tracks which are further divided into sectors. Disk controller determines interaction between device and computer
    - Flash memory more recent, same procedure as above

OS Services

User/OS Interaction

- Users interact indirectly through system programs that make up OS interface. Could be GUI, CLI etc.
- Processes interact by using *system calls* to the kernel (though not directly calls)

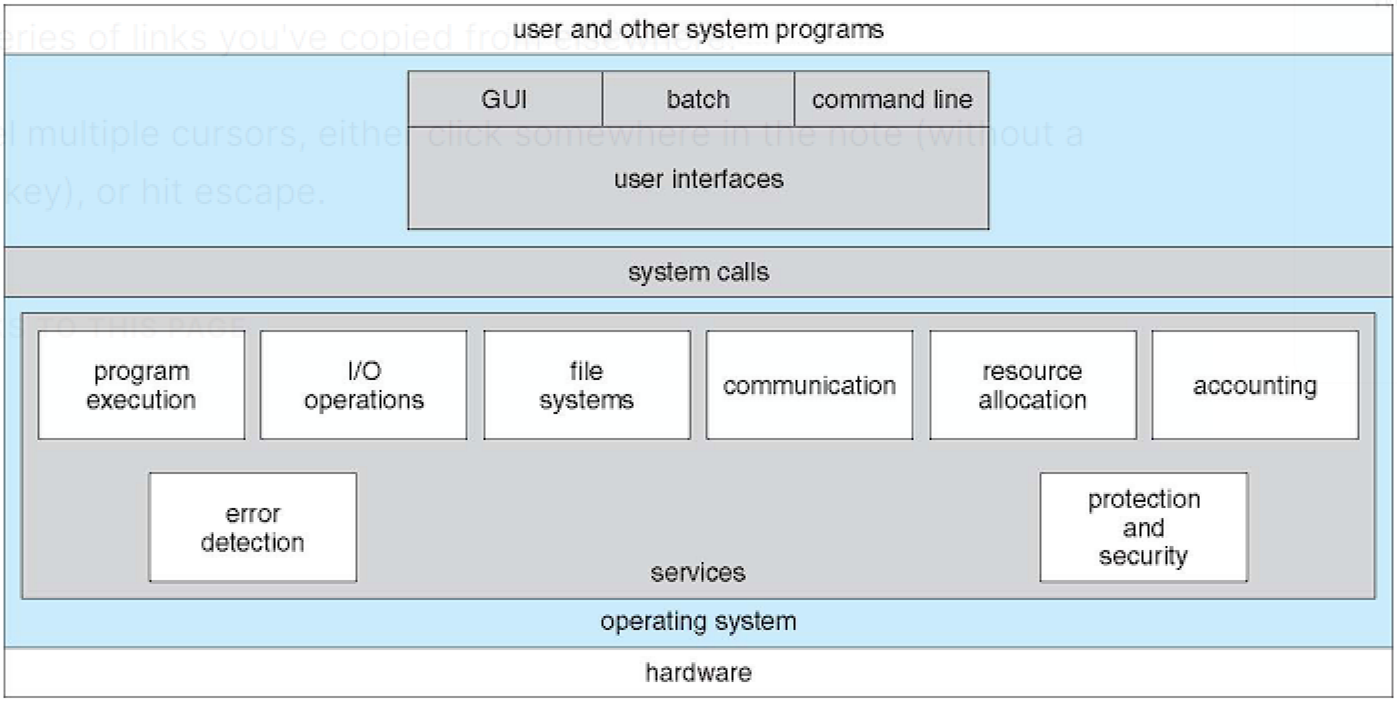
Process Services

- Program execution: System loads into memory and runs programs. Ends them either normally or erroneously
- IO Operations: Running programs may require IO, via file or device
- File system manipulation: Programs may maniupulate files or file directories
- Interprocess Communication (IPC): Allowing shared process mem or message passing

Internal Services

- Error handling: Needs to handle potential process errors
- Resource Alloc: Processes compete for CPU, mem or IO devices

- **Accounting**: Recording what and how much resources processes use
- **Protection and Security**: Multi-user or networked systems need to control information stored. Concurrent processes shouldn't interfere w/e



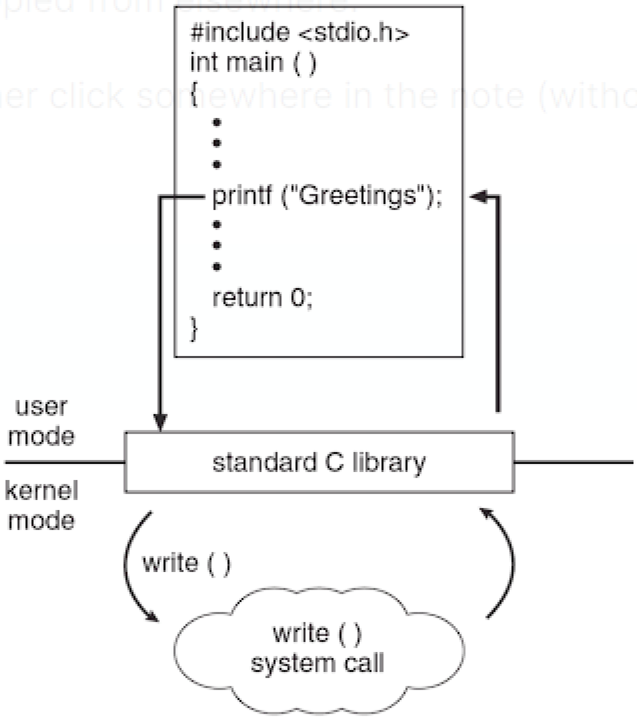
System Calls

Programs written in high-level langs access OS services via API calls rather than direct system calls

Used since system calls execute privileged kernel code and unwise to let user processes execute in privileged mode, so use hardware trap instruction though cumbersome

User process runs trap instruction to switch to priviliged mode and jump to pre-defined kernel address of system call. Hence transition controlled by kernel

API allows backward compatibility if system calls change with new OS release



System calls for file ops

Open

Register file with OS. Called prior to ops. Returns file descriptor (integer) which is an index to list of open files

Read

Read data from file. Returns number of bytes read or 0 for EOF

Write

Write data to file. Returns # bytes written

Close

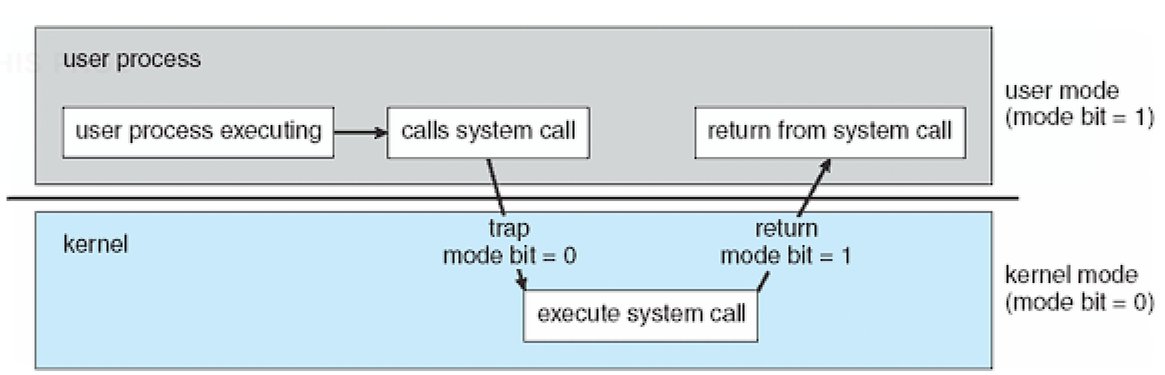
De-registers file with OS. No further ops possible

All return negative number on error. `perror` will display error

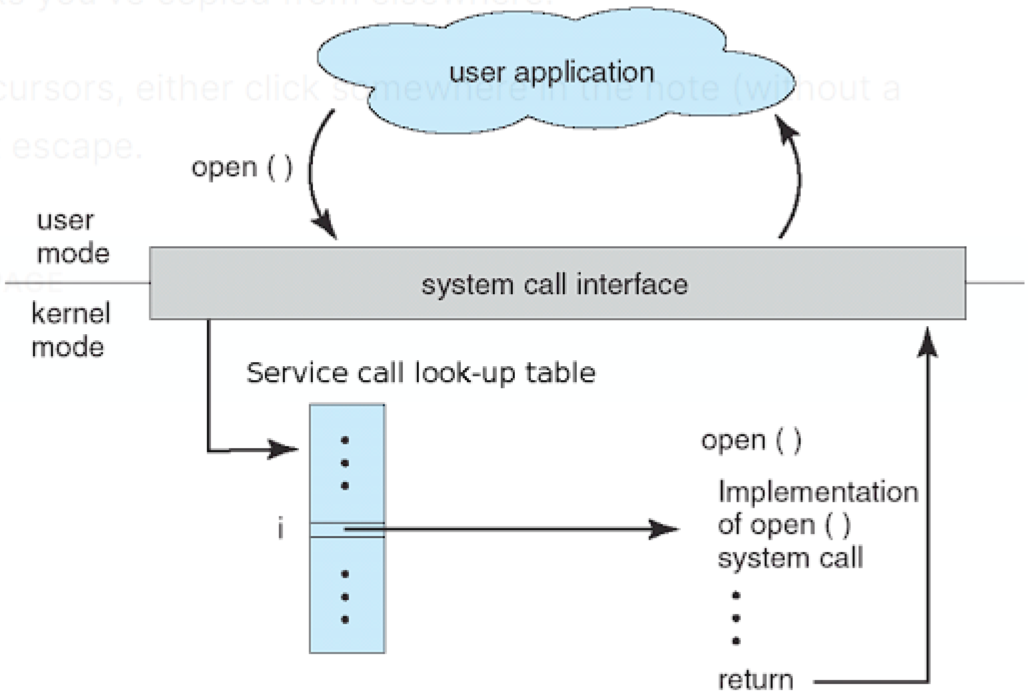
Trapping to Kernel

User process calls system call wrapper function from C std lib

Wrapper function issues *trap* instruction to switch from user to kernel mode



To get around problem that calls cannot be directly made from user to function in kernel
Before trap instruction, index stored in known location (CPU register, stack)
Once switched to kernel space, index used to look up kernel function
Functions that take arguments may have them passed as ptr to structures via registers.



OS Architecture

Traditional UNIX

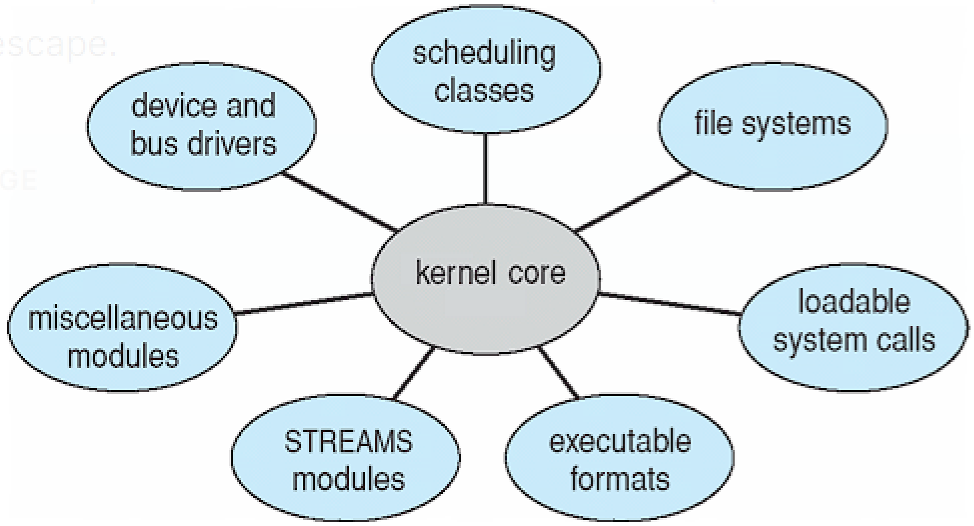
UNIX - one big kernel
Consists of everything between system call interface and hardware
Provides file system, CPU scheduling, mem management etc; many functions for one level
Limited HW support compiled in kernel

Modular Kernels

Modern OSs implement modules
Uses OO-like approach
Core components separate
Each talks to others over interfaces
Each loadable when needed in kernel, so new device drivers can be loaded when necessary

Similar to layered architecture with more flexibility since modules don't need to be compiled with kernel binary

Modularity only logical, all kernel code runs in same privileged space (monolithic), module can technically wipe OS



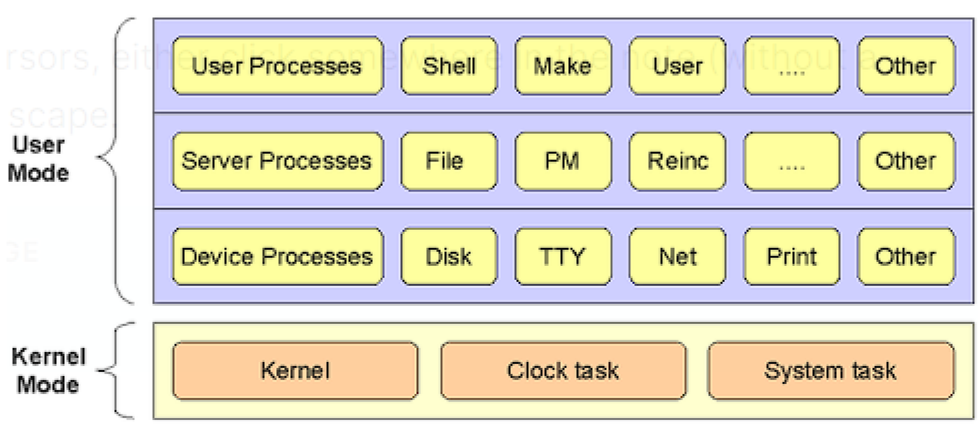
Microkernel

Moves as much as possible from kernel into user space (file system, device drivers)

Communication between modules uses message passing
Device driver can run logic in user space (e.g. queuing sectors to read next)
When HW communication required (say via IO port instructions), passes request to kernel

Benefits
Easier to develop extensions
Easier to port OS to new architectures
More reliable (less kernel mode code), if device failed, it can be reloaded
More secure. Kernel less complex and thus less likely for vulnerabilities
System can recover from failed device driver which normally causes BSOD or kernel panic

Drawbacks
Performance overhead of communication between user and kernel space



The MINIX 3 Microkernel Architecture

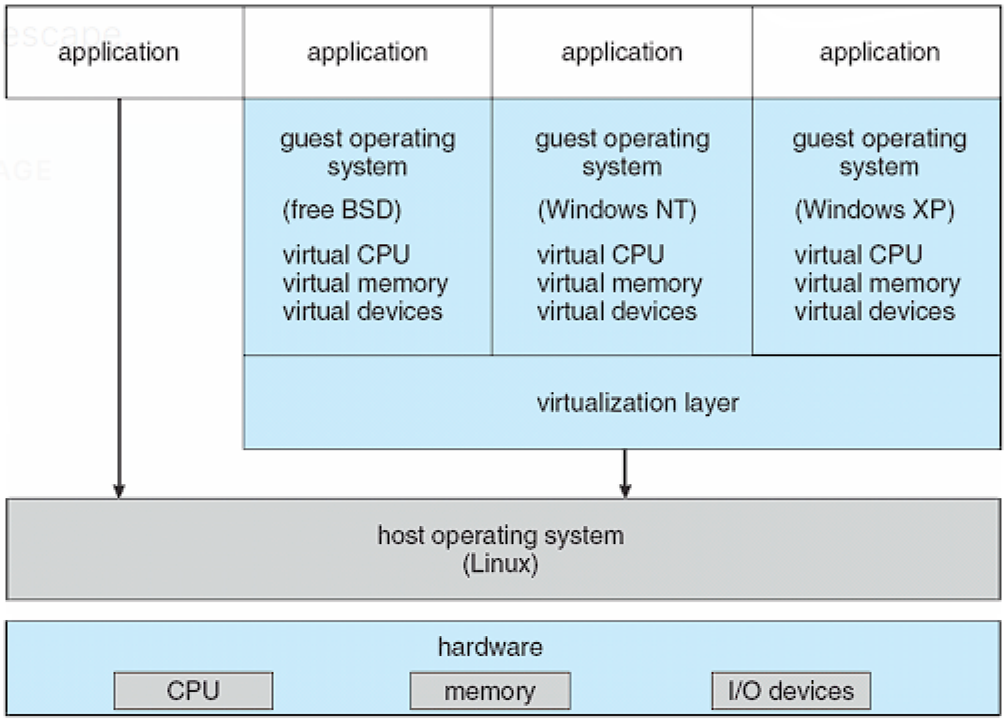
Virtual Machines

Virtual machines (VMs) allow running one OS (guest) on another (host)

A VM provides interface identical to underlying bare hardware

Host creates illusion that process has own processor (and virtual memory)

Each guest is provided with (virtual) copy of underlying computer so possible to install Win 10 on Linux



Benefits

Multiple execution environments share same hardware
Protected from one another, no interference
Sharing of files can be permitted and controlled
Communicates with each other and with other systems via networks
Useful for (OS) development + testing, where it is trivial to revert from accidentally destroyed OS to stable snapshot

Consolidates many low resource systems into fewer busier systems

"Open Virtualization Format" (OVF): standard format of VMs, allows VM to run within many different hosts

Emulation is different, guest instructions are run within a process that pretends to be CPU. In virtualisation, goal is to run host instructions directly on host CPU, so guest CPU must run on the CPU architecture of host. aka emulation mimics guest CPU instructions via software, virtualisation does not

Para-virtualisation

Presents guest with system similar to HW
Guest OS modified to run on paravirtualized HW
E.g kernel recompiled with all code for privileged instruction replaced with hooks to virtualisation layer

After OS successfully modified, para-virtualisation is very efficient, often used for low-cost inertnet servers

VMWare Architecture

VMWare implements full virtualisation, so guest does not require modification to run on virtualised machine
VM and guest run as user process on host

VM must get around problems to convice guest that it is running in privileged CPU mode when it is not
Consider when guest process raises divide-by-zero error
Without intervention, cause host OS to immediately halt VM rather than just the guest process
WMWare looks troublesome instructions and replaces them at run-time with alternatives achieving same effect in user space but less efficient
Since these occur occasionally, many instructions from guest can run unmodified on host CPU

Linux kernel programming

Structure of kernel

SImplified kernel

```
initialise data structures at boot;
while(true)
{
        while(!timer)
        {
                assign CPU suitable process;
                execute process;
        }
        select next process;
}
```

Kernel accesses all resources
Kernel programs not subject to constraints for memory or HW access
Faulty kernel programs can crash system

Kernel and user program interaction

Kernel provides functions via system calls (which can be wrapper functions to actual system calls)
C stdlib provides them
Strict separation of kernel and user data
Need to explicitly copy between program and kernel
aka `copy_to_user()` and `copy_from_user()`

Additionally has interrupts:
kernel asks HW to perform action
HW sends interrupt to kernel which performs the action

Interrupts processed quickly
Any code called from interrupts mustn't sleep

Linux kernel modes

Two main modes for kernel code:
Process context:
kernel code for user programs executed via system call
Has access to user data
Code may be pre-empted at any time by interrupt
Interrupt context:
kernel code handling interrupt (eg by device)
Lower priority interrupts can be pre-empted by higher priority interrupts

Kernel modules

Can add code to running kernel
Useful for providing device drivers on demand
`modprobe` inserts modules and its dependencies into kernel
`insmod` inserts modules without dependencies into kernel
`rmmod` removes module from kernel (if unused)
`lsmod` lists currently running modules

Concurrency in kernel

Correct concurrency handling is important:
Manipulation of data structures which are shared between:
Code in process and interrupt mode
Code in interrupt mode
Must only happen within critical regions
In multi-processor system, manipulation of data structures shared between code in process context must happen in critical regions

Achieving mutual exclusion

Two ways,

Semaphores/Mutex
When entering critical section fails, current process put to sleep until region available
Usable only if **all** critical sections are in process context
Functions: `DEFINE_MUTEX()`, `mutex_lock()`, `mutex_unlock()`
Spinlocks
When processor repeatedly tries to enter critical section
Usable anywhere
Has "busy" waiting (repeated attempts require CPU usage/clock cycles)
Functions: `spin_lock_init()`, `spin_lock()`, `spin_unlock()`

Semaphore Use

Two kinds
Normal semaphores

Read-Write semaphores
Useful if some critical sections may only read shared data and occurs often

Data transfer between user and kernel

Linux maintains directory called `proc` to interface between kernel and user
Files here do not exist on disk
Read-write operations on files here act as data exchange between user and kernel

Tour of Linux kernel

Major parts:
Device Drivers - in subdirectory `drivers`, sorted via category
File systems - in subdirectory `fs`
Scheduling and process management - in subdirectory `kernel`
Memory management - in subdirectory `mm`
Networking - in subdirectory `net`
Architecture specific code (including assmebly) - in subdirectory `arch`
Include files - in subdirectory `include`

Summary

OS as *Resource manager*, particularly for HW
OS accesses all resources
API to interact with OS from programs (system calls)
Monolithic kernel in UNIX, Microkernels (interaction with HW only in kernel space, rest can be user space)
In Linux, can modify kernel via kernel modules, separate programs but part of kernel
VMs make possible running one OS on top another. Careful use of *privileged OS instructions*