Processor controls hardware and executes instructions.

Memory stores data and code.

I/O components read and write from I/O devices.

System bus interconnects different components and provides communication between them.

OS must make **efficient** use of available resources and must **share** resources (CPU cores, caches, RAM, GPUs, clocks, timers, persistent storage) among multiple users.

Providing clean interfaces allows us to convert raw hardware into a usable computer system by keeping **details** hidden from user-level software.

Must support several simultaneous parallel activities, and may switch activities at arbitrary times, so must ensure **safe concurrency**.

MULTIPROCESSORS (Windows, macOS, Linux) have many CPU cores and CPUs.

SERVER OS (Solaris, FreeBSD, Linux) shares hardware/software resources.

MAINFRAMES have bespoke hardware, limited workload.

SMARTPHONES have power-efficient CPUs.

EMBEDDED OSs (QNX, VXWorks) are used for home utilities, and only run trusted software.

REAL-TIME OSs are time-oriented (not performance/IO).

SENSOR NETWORK OS (Tiny OS) is resource efficient.

The OS **kernel** is a privileged program that implements OS functionality.

Monolithic kernels are a single black box with all functionality. One executable with its own address space. Easier to write kernel components but overall design is complex, with no protection between them.

Microkernels have as little functionality in the kernel as possible, functionality passed to a user-level server. Kernel does IPC between servers - separate ones for I/O, file access, scheduling, but this leads to high overhead. Less error-prone and servers have clean interfaces.

Hybrid kernels combine features of both.

LINUX is a variant of Unix. **Interrupt handlers** are the primary means to interact with devices. I/O scheduler orders disk operations. Supports static in-kernel components and dynamically loadable **modules**.

NTOS provides **system calls** in **WINDOWS**. Programs are built on top of **dynamic code libraries**. NTOS loaded from *ntoskrnl.exe* at boot, consists of **executive** and **kernel** layers. **Device drivers** loaded into kernel memory. NT structured like a **microkernel**, most components run in same address space. **Hardware Abstraction Layer** abstracts out BIOS config, CPU types, etc...

PROCESSES

allow a single processor to run multiple programs "**simultaneously**". Provides concurrency, isolation between programs, allows better resource utilisation, simplifies programming.

Pseudo concurrency — single physical processor is switched between processes by interleaving.

Real concurrency utilises multiple physical processors.

On a **context switch**, the processor switches from executing process A to executing process B. May occur in response to events/interrupts, or as a result of periodic scheduling. Process may be **restarted later**, so all information needed should be stored in **process descriptor** / **control block**, kept in the **process table**.

Process Control Block — process has its own VM, should store PC, page table register, stack pointer, ..., process management info, file management info.

Context switches are expensive, so avoid unnecessary usage. *Direct cost* — save/restore process state. *Indirect cost* — perturbation of memory caches.

^ Flushing TLB & pipeline flushing

UNIX allows processes to create hierarchies. Windows has no notion of this.

int fork(void) — Creates new child process, to be executed concurrently, by making exact copy (same resources) of parent process image. Returns child PID in **parent** process and 0 in **child** process. No child created, and -1 returned to parent on error.

int execve(char *path, char *argv[], char *envp[]) — Changes process image and runs new process. Lots of useful wrappers.

int waitpid(int pid, int* status, int options) — Suspends execution of calling process until process *pid* terminates normally or signal received. Can wait for more than one child.

Simple design philosophy, so process API made up of basic blocks that can be easily combined.

CreateProcess() (equivalent of *fork()* & *execve()*) has 10 parameters!

Termination — normally process completes execution of body, calls *exit()*. **Abnormal exit** — runs into error or unhandled exception. **Aborted** — another process has overruled execution. Some run in an endless loop.

void exit(int status) — Terminates a process. Never returns in the calling process.

int kill(int pid, int sig) — sends signal sig to process pid.

`SIGINT`	Interrupt from keyboard
`SIGABRT`	Abort signal from `abort`
`SIGFPE`	Floating point exception
`SIGKILL`	Kill signal
`SIGSEGV`	Invalid memory reference
`SIGPIPE`	Broken pipe: write to pipe with no readers
`SIGALRM`	Timer signal from `alarm`
`SIGTERM`	Termination signal

SIGNALS are an IPC mechanism — **delivery** similar to delivery of hardware interrupts. Process can send signal to another process, if it has permission. Generated when an exception occurs, when kernel wants to notify process of an event, when certain key combinations typed in terminal, or programmatically using *int kill()*.

Default action for most signals is to terminate process, but receiving process may ignore/handle it instead (unless *SIGKILL* or *SIGSTOP*).

PIPES — method of connecting stdout of one process to stdin of another. **Named** or **unnamed**. Sender should close the read end, receiver should close the write end.

Named Pipes / **FIFOs** are persistent pipes that outlive processes which created them. Stored on file system. Any process can open it like a regular file.

int pipe(int fd[2]) — returns two file descriptors: read end *fd[0]*, write end *fd[1]*.