TABLE OF CONTENTS

**Computer Terms2**

**Communication With Computers9**

**Operating Systems14**

**Programming Languages23**

**DLLs27**

**System Calls28**

**CPP29**

Geben Sie die Kapitelüberschrift ein (Stufe 2)5

Geben Sie die Kapitelüberschrift ein (Stufe 3)6

**System Calls4**

Geben Sie die Kapitelüberschrift ein (Stufe 2)5

Geben Sie die Kapitelüberschrift ein (Stufe 3)6

**COMPUTER TERMS**

**Signals:** Standardized messages sent to a running program to trigger specific behavior, such as quitting or error handling. They are a form of **inter-process comms** and they are used in POSIX-compliant operating systems. It is an async notification sent to a process or to a specific thread within the same process to notify it of an event. Signals usually deliver the following commands:

* Interrupt
* Suspend
* Terminate
* Kill

When the signal is sent, the OS will interrupt the target process’s normal flow of execution to deliver the signal. Execution is interrupted during non-atomic instructions. If a signal handler is present, the handler’s routine is executed. Keyboard combinations send intuitive signals to the OS like task-manager, drop to desktop, auto-search…

**Interrupt:** this condition alerts the processor and serves as a request for the processor to interrupt currently executing code so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state and execute a function called an interrupt handler. Interrupts are used in multitasking and real-time computing.

* Hardware interrupts
* Software interrupts
* Level-triggered
* Edge-triggered

Interrupts are used for including but not limited to:

* Hardware timers
* Data transfer in storage (disk IO)
* Comms interfaces (UART, Ethernet)
* Keyboard and mouse events

**Clock Interrupt:** aka timer interrupts, they occur on the order of ms and they are used to support preemptive multitasking.

**Inter Process Communication:** refers to the mechanisms an OS provides to allow processes to manage shared data. Applications can use IPC which is categorized as “client and server” where data request and response are present.

**Process:** the instance of a computer program that is being executed by one or more threads. It contains the program code and its activity. A computer program is a passive collection of instructions stored in a file on disk and a process is the execution of those instructions after being loaded from the disk into memory. A common form of multitasking is provided by CPUs **time-sharing** which is a method for interleaving the execution of users’ processes and threads, users are immediately assigned computing resources at the simple pressing of a key or when moving a mouse. Application like music and videos are given higher real-time priority. OSes prevent direct comms between interdependent processes for security reasons and are used as the middle-man between the bare metal CPU and the human-understandable computer actions.

**Time Sharing:** sharing of computing resources among many users at the same time by means of multi-tasking. Users are in the sense of actual humans interacting with a computer at the same time, or users in the sense of processes/programs/tasks demanding computing resources from the OS. In time sharing systems context switches are performed rapidly which makes it seem like multiple processes are being executed simultaneously on the same processor. This has the name of **concurrency.** Time sharing is prevalent in POSIX systems such as windows, mac, linux and unix-based OSes.

**RTOS:** this OS processes data and event which have critically defined time constraints. All processing must occur within defined constraints. RTOSes are event-driven and preemptive, meaning that the OS is capable of monitoring the relevant priority of competing tasks and make changes to the task priority. Event driven systems switch based on priority, while time-sharing systems switch based on clock interrupts. RTOSes have advanced algorithms for scheduling

**Preemption:** the act of temporarily interrupting and executing task, with the intention of resuming it at a later time. This interrupt is done by an external scheduler with no assistance or cooperation from the task. The preemptive scheduler runs in the most privileged protection ring, meaning that interruption and resuming are considered highly secure actions. The change in the currently executing task of a processor is known as **context switching.**

**Context switch**: process of storing the state of a process or a thread so that it can be restored and resume execution at a later point. This allows multiple processes to share a single CPU and is at the core of multitasking OSes. This is computationally expensive, therefore OSes try to optimize this switching so that multi-tasking may be done efficiently. Switching from one process to another requires a certain amount of time for doing the administration, saving and loading registers, memory maps, updating various tables and lists… Multithreading is equivalent to a context switch at the OS level. The difference is that a multithreaded CPU can do a thread switch in one CPU cycle instead of the thousands of CPU cycles a context switch normally requires.

**Thread:** a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler. How threads and processes are implemented differs in OSes. A thread is a component of a process. The multiple threads of a given process may be executed concurrently (multithreading) by sharing resources such as memory.

**Intermediate Representation (IR):** it is the data structure or code used internally by a compiler or virtual machine to represent source code. IRs must represent source code without loss of information, be independent of any particular target language.

**Programming Tool:** It is a computer program which is used in supporting other program and applications. The term refers to relatively simple programs, that can be combined to accomplish a task. The most basic tools are text editors, compilers and interpreters. A non-exhaustive list is as follows:

* Binary compatibility analysis tools
* Bug tracking systems, ticket systems
* Build automation tools
* Call graph
* Code review
* Compilation and linking tools
* Debuggers
* Disassemblers
* Scripting languages
* Grep&find

**Systems Programming:** the primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly, whereas systems programming aims to produce software and software platforms which provide services to other software and must abide by performance constraints.

* Operating systems
* Computational science applications
* Game engines
* Industrial automation
* SaaS applications.

The programmer will exploit hardware while doing systems programming by utilizing hardware specific algorithms.

Used languages in chronological order:

* Assembly
* Pascal
* C
* C++
* ADA

**System Call:** a computer program requests a service from the kernel of the OS. This can include hardware related services, execution of new processes, comms with kernel services. System calls can be made from userspace processes.

**ELF, executable and linkable format:** common standard file format for executable files, object code, shared libs and core dumps. It is the standard binary file format for unix and unix-like systems on x86 processors.

**Event-driven Programming:** the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, message passing from other programs and threads. It is the dominant paradigm used in GUI, device drivers and JavaScript web applications which are centered on performing certain actions in response to user input. A main loop listens for events and then triggers a callback function when an event is detected. In embedded systems, the same is achieved by hardware interrupts. High level languages provide await and closure, which make writing event-driven programming easier. Event-handlers are already present for the user to define, such as KeyPressed() or MouseMoved(). Event handlers have subroutines that may triggers a series of events within the computer.

**Header vs Library:** headers are used to define an interface within an application. It shows the functionality while omitting the technical implementation details. Header files include the following:

* Class description and inheritance hierarchy
* Class data members and types
* Class methods

A library is a collection of code which you want to make available to a program. It includes the implementation of an interface. Code is defined in a library to prevent code duplication and encourage re-use.

* Statically-linked: defines a set of export symbols (method definitions) which are linked into the final executable .exe during the linking stage of the build process. Faster execution but larger binary.
* Dynamically-linked: linked during the execution of the program. Necessary when multiple programs need to re-use the same methods and is mandatory for COM (Microsoft tech).

**Device Driver:** a program which controls a device that is attached to a computer. It provides a software interface between hardware and the OS so that hardware functions are accessed without needing to know mechanical details. Programs who want to execute system calls on the device driver will invoke a routine in the device driver and the device driver than issues this command to the device. Device drivers communicate on computer busses. For example, a high level function like “send data” and “receive data” will be implemented depending on port type, comms protocol, hardware type etc without the user needing to modify his code for each configuration. Drivers work in privileged environment and require low-level access to hardware functions, which need to interact with the OS, therefore writing device drivers require good knowledge of hardware config, OS and the device itself.

**Concurrent Computing:** several computations are executed concurrently during overlapping time periods. there is a separate execution point for each process. One computation can advance without waiting for all others to complete. The goal is to model processes in the real physical world such as multiple clients accessing a server at the same time.

**Parallel Computing:** processes are carried out simultaneously. Forms of parallel computing:

* Bit-level
* Instruction-level
* Data parallelism
* Task parallelism

Parallel computing is the norm when it comes to high-performance because of power concerns. Parallelism is implemented with multi-core processors.

Parallel computing and concurrent computing are not the same and one can be done without the other. Bit level is parallel without being concurrent, multitasking and time-sharing are concurrent without being parallel. Parallelism is impossible on single core CPUs.

**Build Process:** This is for converting source code into standalone software which can be run on a computer. Conversion may take into account the OS, the architecture, bit-ness, platforms etc. build process is managed by build tools, which organize and manage other interdependent tools. Make is a popular build tool for such interdependent functionality.

**Instruction Set Architecture:** aka, computer architecture is the vocabulary, which the CPU can communicate with. A device that can execute the instructions described by the ISA (normally, this would be a CPU) is called an implementation. ISA defines the following:

* Instructions
* Data types
* Registers
* Hardware for the main memory(which is RAM)
* Memory consistency
* Addressing modes
* Virtual memory
* IO Model with the external world

The ISA specifies the behavior of machine code running on the CPU in a fashion that is vendor- and model-agnostic (be it Intel or AMD, be it i7 or i5…) which makes it binary compatible between CPUs. With this approach, ISAs can be run on all CPUs, but their performance varies. Depending on this, various microarchitures of the CPUs can still execute the same ISA, albeit with different performances.

**Microarchitecture:** it is the set of processor design techniques used in CPUs, to implement the instruction set. The uarch of a machine is usually represented as diagrams that describe the interconnections of the various microarchitectural elements of the machine, which can be single gates, registers, ALUs etc… uarch concepts are as follows:

* Instruction cycles
* Execution speed
* Instruction set choices (VLIW, CISC, RISC, EPIC) and their vendor variants
* Instruction pipelining
* CPU caching
* Branch prediction
* Superscalarity
* Out-of-order execution
* Register naming
* Multiprocessing and multithreading

**Computer architecture:** it is the combination of microarchitecture, instruction set architecture, logic design and implementation.

**X86 vs x64:** Both of them are families of ISAs. X86 is an umbrella term used to refer to any 32-bit processor compatible with the x86 instruction set, which is based on the Intel 8086 microprocessor. X64 is 64 bits. X64 is actually the architecture name for the extension to the x86 instruction set that enables 64-bit code. When it was initially developed, it was named x86-64, now it is known as x64.

**Virtual Memory:** It is a memory management technique that provides abstraction of storage resources, which provides the user with a large range of contiguous memory addresses, even though the real memory places are taken up by irrelevant programs. Virtual memory is managed by the operating system. The OS uses hardware and software techniques to map the memory addresses(virtual addresses) used by a program, into physical addresses in computer memory(the literal magnetic pole configuration inside the RAM). Thus, the main storage which is seen by a process appears as a contiguous address space. The hardware of memory management unit which assumes the role of address translation hardware in the CPU, translates virtual addresses to physical addresses. The OS may extend this by software techniques, it may even use disk storage (HDD, SSD) to provide more memory than is physically available.

**COMMUNICATION WITH COMPUTERS**

You have to remember that computers are hierarchical machines, whereby almost 70 years of technology must be learned from the get-go. This makes the new learner feel anxious and overwhelmed, because everything that you learn seems to open another door with more things to learn, since the technology is extremely complex. However, since you already learned the basics over a large period of time, you can categorize computer technology.

The languages that you are familiar with are all **imperative,** meaning that by writing the language, you command the computer to do exactly what you write. These languages spiral from one source and over the years they have produced a lot of variants. Semantically, they build up on the hierarchy of programming, in the following path:

1. Literally manipulating **bare metal** by creating analog circuits. This includes creating logic pathways with transistors to do and/or calculations and the likes.
2. After creating a series of analog circuits, their functionality has become automatic and readily available, this led to the discovery of cascading multiple analog circuits. This is also known as **hardware programming.**
3. Being able to create complex hardware code needed to be delivered in an easy way for computers to process in a fast manner. **machine code** had the ability to send zeros and ones in bundles to cascaded analog circuits to specifically target transistors to open and close their gates. this code, just like the previous ones, cannot be read by humans because there is no indication as to what the code does.
4. Programs were developed to write machine code in an efficient manner so that engineers did not have to rely on physical mediums to write machine code like punching cards. These programs automated the process of writing machine code in which the language could be understood as well. **Assembly** was developed to write basic operations like carrying, jumping, adding of bytes etc.
5. In order to create complex software, a language needed to be developed which required complex thinking as well. **High-level programming languages** are written in basic English, in which complex tasks like doing things over and over again, raising flags when something goes wrong and breaking the procedure of running code were made possible. High level languages have extreme variety, and understanding all of them is not necessary for all engineers.
6. The forefathers of high-level programming languages are **FORTRAN, ALGOL, PASCAL, C** and the likes. Since these languages took a long time to develop, they are industry standards and most software is based on these languages. Remember that these languages were developed in the 60s and 70s, so their reign is still present to this day.
7. An extension of high-level languages is the **object-oriented programming languages** in which data is bundled together so that external manipulation is excluded. This gives extra security to the data with some flexibility. C cannot handle object-oriented approaches, but C++ can. Most hardware runs on C, because most hardware have been designed in the 70s and the 80s.

Forefathers of programming languages can be found below:

* **COBOL**
* **PASCAL**
* **LISP**
* **FORTRAN**
* **ALGOL**
* **C**
* **BASIC**
* **SMALLTALK**

Super popular high-level languages are found below:

* **C (king of them all)**
* **C++ (dominates the market but if it comes to speed and efficiency, especially in embedded tasks, C is preferred)**
* **JAVA (is pseudo-compiled, translated into bytecode, requires a java virtual machine)**
* **Python (same as JAVA, requires its own virtual machine)**
* **Smalltalk (same situation)**
* **FORTRAN (nearly all scientific software talks with FORTRAN in the back stage, because the scientific domains have been written in FORTRAN in the 60s,70s and 80s)**
* **PHP (used for server-side web development)**
* **JS (used for client-side web development)**
* **BASH (unix-like OS terminal language)**
* **POWERSHELL (windows terminal language)**
* **C# (known for collecting garbage, similar to C and C++, but is pseudo-compiled, C and C++ need to manually manage allocated memory for the program)**

Most high-level languages fall in the category of **procedural,** in which loops are created to do things multiple times over and over again in steps and each step is programmed to be efficient. While the program is executed, procedure calls are made to stop at the point of execution, go do the thing that the programmer intended on a different part of bare metal and come back to the point where he left off.

**CENTRAL PROCESSING UNIT**

Not all computers have been cookie cutters as to what you see around you. Previously, as the hierarchy wasn’t as complicated as it is right now, the granddaddy of all computer related stuff was the CPU. The CPU carries basic arithmetic, logical and I/O operations. If a CPU contains memory and peripheral interfaces, it is called a system on a chip (SoC)

Early days of computing, we had computers with wires that had to be manually changed to perform different tasks. For example, if you wanted to add two numbers you had one configuration, but if you wanted to divide two numbers, you had to manually rewire the computer. This was menial and retarded, so with the additional abstractions and increased technology, CPUs have been developed to not need this rewiring every time we wanted to do different operations. CPUs have the “von Neumann architecture” inside, where a control unit, a memory unit and an ALU are present. Embedded ICs have the Harvard architecture.

CPUs understand Instructions, which are focused on doing basic math. Not all CPUs support every instruction that has been conceived up to this point, hence instructions define the sophistication of CPUs. CPUs have seen a fast advancement with transistors, because transistors could switch faster than relays, which were used in clunky mechanical old computers.

All CPUs follow the fetch – decode – execute steps. These are known as the “instruction cycle”. Execution increments the PC (program counter) so that the CPU knows where it left off. The RISC pipeline describes the barebones of CPU operation, where complexities are omitted for now.

* Fetch: retrieve an instruction from computer memory, where the PC tells you the address of the instruction and the PC is subsequently incremented. Waiting for the next instruction is taken care of by pipelining and caching.
* Decode: the instruction is converted into signals that control the other parts of the CPU. The Instruction Set Architecture of the CPU tells it to decipher the instruction, where the instruction is deconstructed into opcode and the remaining fields.
* Execute: control signals electrically enable and disable various parts of the CPU, the actions are aligned with clock pulses. The results of the completed executions are written to a register, depending on memory type and access priority.

CPUs have been developed over a long period of time, in which parallelism was also discovered to speed up instructions. By running multi instructions at the same time, parallel execution is achieved.

* **Instruction level parallelism:** begin the first steps of instruction fetching and decoding before the prior instruction finishes executing.
* **Task level parallelism:** execute multiple threads in parallel.
* **Data parallelism:** vectors are used instead of scalars.

**HARDWARE ABSTRACTION**

Since at the core of everything you literally manipulate electrons through metal to cause certain voltage changes, you have to know how you can systematically do these tasks. In order to make sense of hardware in which a human can intuitively understand it, it has to be abstracted. Programming interfaces are created, usually in C, so that hardware is meaningfully translated into English. Abstractions provide OS calls which are device independent. The programmer must be able to write high level software without caring about CPU-specific and OS-specific instructions.

**HARDWARE DESCRIPTION LANGUAGES**

A highly specialized language in which you can describe how digital logic circuits can behave. They are used for electric design automation (pcb designs) VHDL is the standard language, with it you can design the behavior of logic gates to understand and process electrical signals, aka highs and lows. Verilog is also an HDL. FPGAs and ASICs are written in hardware description languages, where the logical synthesis workflow is produced.

**MEMORY ADDRESS**

It is a reference to a specific memory location. Memory addresses are fixed-length sequences of digits manipulated as unsigned integers.

**SYSTEM CALL**

It is the way in which a computer program requests a service from the kernel of the OS. These calls can include requests to access hardware, start up new processes and general communication with other elements of the computer, be it hardware or software. The hierarchy or the overall authority of system calls to access hardware and software depend on the security model of the CPU. Depending on the access ring, all system calls have priority and accessing power.

**POINTER**

It is an object that stores a memory address. It is a value located in computer memory. A pointer references a location in memory and obtaining this value stored at that location is called as referencing the pointer.

Assume that a pointer is the page number in a book. Flipping the cover, looking at the page number and going to that page and subsequently reading the text written on that page is the full task of what a pointer provides. By reading the information on that page, you have dereferenced the pointer.

It is computationally cheaper (time wise and traversing the physical location on memory) to copy and dereference the pointer than it is to copy and access the data to which the pointer points.

Pointers are also used to hold the addresses of entry points for called subroutines.

**OPERATING SYSTEMS**

All operating systems originate from the first quasi-OSes that were developed in the 50s and 60s. The primitive OSes focused primarily on the definition of the computer architecture, rather than optimizing the already present and clear-cut protocols. For instance, the OSes focused on the scheduler, the program counter, the interaction with the registers, the protocols of ALUs and so on.

Remember that monitors were not ubiquitous back in the day but computers still needed to function properly. A Window into the monitor was not even a term back then, because at most, people thought in terminals and not in graphical user interfaces. With advanced technology, it was time to program a display system for the monitors. This is called as **windowing system** and this system is controlled by a software called **window manager.**

The first OS that could be described intuitively to the layman originates from the **Berkeley Software Distribution,** which is based on the **Research Unix**, the first ever Unix system conceived. Mostly everything computer related originates from **Bell Labs, General Electric** and **AT&T** which is why the computer technology is an exclusive American invention. More information can be found under the **History of Unix** tab.

Linux-based Operating Systems share a lot of similarities with the FreeBSD, an open source version of the BSD, however there are some major differences that definitively separate the two OSes into two categories.

* FreeBSD maintains a complete system, in which it delivers a kernel, device drivers, userland utilities and a documentation.
* Linux delivers kernel and device drivers and third parties develop the software.

**KERNEL**

It is the portion of the operating system code that is always active. It facilitates interactions between software and hardware. For example, kernels have an overlord control on cache usage, file systems I/O etc…

The Kernel is separately loaded in memory where the other apps cannot reach it. Don’t think of it in terms of your everyday personal computer, but rather in terms of big power plant computers where computing failures lead to catastrophe.

In order to use applications, the user must chart through the “user space” in which a separate memory is presented.

The kernel’s interface is a **low-level abstraction layer**. When a process requests a service from the kernel, it must invoke a **system call**, usually through a **wrapper function.**

Kernel types are varied and can be categorized under the following:

* **Monolithic kernel:** linux and unix-based OSes
* **Hybrid kernel**: Windows NT series aka the white man’s OS
* **Microkernel**

Kernels need to do certain things and the rest can be handled flexibly.

* **RAM usage:** store program instructions and data
* **I/O:** peripheral controls
* **Resource Management:** define execution domain
* **Memory Management:** virtual addressing, paging and segmentation, prevent applications from crashing each other.
* **Device Management:** access to peripherals through device drivers.
* **System Calls:** defines how a process requests a service from the kernel that it doesn’t have permission to run.

Kernels had a breakthrough with the name of **time sharing** where multiple users could get small pieces of computer time. This concept comprises the basis of modern computer technology.

**WINDOWING SYSTEM**

Although OSes have not always been synonymous with how the desktop looks like (because the word desktop is in itself built upon so many layers of abstraction, what comes as intuitive has layers of complexity behind it), how a monitor displays information is important.

This system manages different parts of display screens. It is a graphical user interface (which was developed because command line interfaces were too hard for the lay man) that implements **WIMP** paradigm.

**WINDOW MANAGER**

They are divided into 3 classes:

* Compositing window manager
* Stacking window manager
* Tiling window manager
* Dynamic window manager

Examples for window managers are:

* X window manager
* Desktop window manager

Window managers control the placement and appearance of windows within a windowing system in a graphical user interface, aka monitors and touchscreens. Window managers focus primarily on giving an intuitive desktop feeling (desktop environment) Window manager and the windowing system are usually not distinctly separated from each other and understood as a bundle.

**GUI**

It allows users to interact with electronic devices through graphical clickable icons and an audio indicator as notation. Graphical user interfaces dependent on the OSes are listed below:

* Gnome
* Unity
* MATE
* Wayland
* XFCE
* Windows Shell

**LOAD TIME**

A loader is part of an OS that loads libraries and programs, it places programs into memory and prepares them for execution. The time spent in this interval is known as load time. After the loading phase is completed, the OS can give the program the necessary resources to execute its instructions. Embedded systems do not have loaders, the code executes directly from ROM.

**RUN TIME**

It is the final phase of program’s life cycle, in which the program’s code is being executed on the computer’s CPU as machine code. Compile time happens before, where the high level language is transformed into assembly and then into machine code. Errors experienced during runtime are handled differently depending on the OS and the type of error. Before run time begins, the loader (described above) performs the necessary memory setup and program linkage with dynamically linked libraries. After the preliminary steps are done, the program will start executing from the program’s main entry point.

**RUNTIME SYSTEM**

Aka runtime environment, it implements the portions of an execution model. Programming languages have runtime systems in which they provide an environment where the programs can run. This environment handles some tasks like how variables are accessed, how memory is managed, how parameters are passed between procedures…

**RUNTIME LIBRARY**

It is a set of low-level routines used by a compiler to invoke some of the behaviors of a runtime environment, by inserting calls to the runtime library into compiled executable binary.

**SHARED LIBRARY**

Aka shared object, it is a file that can be used by multiple executable files and other shared objects as well. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a monolithic executable file for the program.

**ENVIRONMENT VARIABLE**

A dynamic-named value that can affect how processes behave on a given computer. (this can be a full fledged PC or an embedded system) They are part of the environment in which a process runs. It stems from UNIX derivations but was added to windows and linux and mac variants as well. Windows has different syntax and semantics. When a process is created, it inherits a duplicate run-time environment. From shells, you can directly invoke “env” to change the variables.

**PATH**

Not to be confused with the path of a folder, in which humans intuitively can make analogous comparisons between the location of a folder in physical space, as opposed to the memory address on ROM.

The environment variable PATH specifies the set of directories where executable programs are located. Each user session has its own PATH setting.

POSIX variants have /bin, /usr/bin, /usr/local/bin automatically included in $PATH by default, so each executable program installed on such operating systems look for the entry point in these folders. The list of directories are colon : separated.

**SHELL**

A shell will expose the OS services to the human user or other programs. This exposure usually happens through GUIs or CLIs, depending on human expertise, resources necessary to display graphics and just general need. Shells are special applications that use the kernel APIs just like how a reggie program would use it.

Shells run on local systems, aka the keyboard youre typing on is directly connected to the computer youre controlling. However, you can gain access to remote locations by means of remote access / remote administration. “Secure shell” protocols are used to gain access to remote computers.

GUIs are built on top of windowing systems, which are primarily X and Wayland.

**WINDOWS API (win32 API)**

All windows programs interact with the windows api. On Windows NT line of OSes(the reggie consumer shit), some programs may use the native API. With the usage of the Microsoft SDK, you can have access to header files, documentation, samples and tools to build software that interact with the Windows API. You need to run these through MSVC to compile software for windows. The API provides the following:

* Basic resources like file systems, devices processes, threads, error handling
* Advanced services like the windows registry
* User interface to create and manage screen windows, buttons, scrollbars, mouse and keyboard
* Common dialog box library
* Common control library
* Windows shell
* Network services

**WINDOWS RUNTIME (WinRT)**

A platform agnostic (meaning that it will work on a PC, on embedded chips, on Xboxes etc) component and application architecture, which is implemented in c++.

WinRT is not a runtime in the sense which was explained previously, but rather a language-independent **application binary interface (ABI)** where APIs written in different languages can interact with each other. Apps using the WinRT may run inside a sandboxed environment to allow for greater security and can natively support x86 and ARM. The components are specifically designed by thinking about the interoperability so that multiple dissimilar languages can talk to each other.

WinRT uses the **WINDOWS API (win32 API)** as its base.

**WINDOWS REGISTRY**

Prior to registries, .ini files stored eac program’s settings as a text file or binary file, located in a shared location that did not provide user-specific settings in a multi-user scenario. Win regs store all app settings in one logical standardized repo. Regs contain information, settings, options for the OS and the hardware as well.

**WINDOWS HANDLE**

An application cant directly access object data, nor the system resource that an object represents. Instead, the app must obtain an object handle, which it can use to examine or modify the system resource. Each handle has an entry in an internally maintained table. The entries contain the addresses of resources and the means to identify the resource type.

**WINDOWS NT KERNEL ARCHITECTURE**

Consists of two main components:

* User mode
* Kernel mode

OS category under

* Pre-emptive
* Re-entrant multitasking

Processor compatibility

* Uniprocessor
* Symmetrical multiprocessor (SMP)

Kernel architecture

* Hybrid:
* Comprises a simple kernel, hardware abstraction layer and executive services.

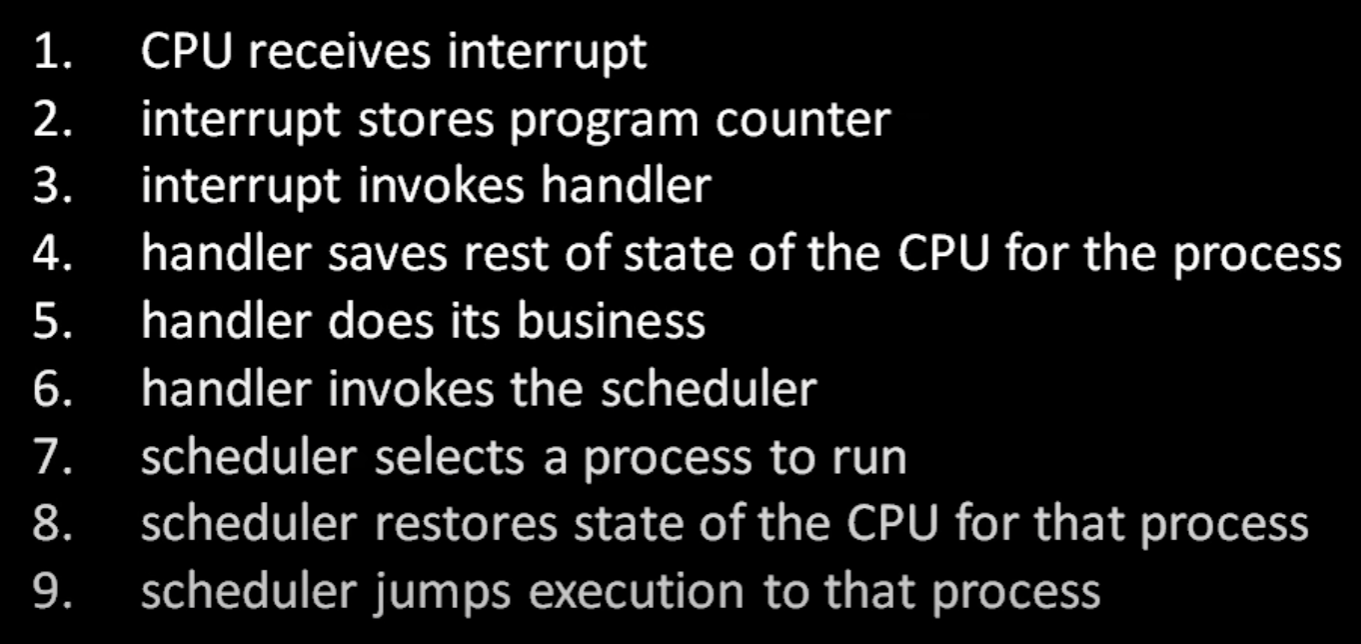
The CPU can’t run processes and the OS code at the same time, so it has to run them one by one, usually one after another, in the simplest case of one-core CPUs. In multi-core CPUs, this process may be optimized.

The CPU has access to hardware, like the monitor, the speakers, the physical memory aka magnetically charged 1s and 0s. This access has to follow some kind of order to run multiple processes at the same time, like showing the desktop environment while producing sound. (the basic way to visualize this concept, but in reality a lot of processes are run at the same time).

However, it is impossible to run things at the same time. Therefore, the illusion of simultaneity must be provided. Each CPU core can only execute the code of one process at a time, and the OS’s own code cant run on the core while a process code is being executed. Therefore, the core must alternate between running the code of a process and the code of the OS. This is the bare minimum approach to single core CPUs, multi-core CPUs can run it differently, albeit the concept of simultaneity is the same.

The portion of the OS, called as the Scheduler, runs after each process to decide what OS work should be done and which process should run next. When any hardware interrupt is triggered, the interrupt handler passes off control to the scheduler rather than handing the processor core back to the interrupted process.

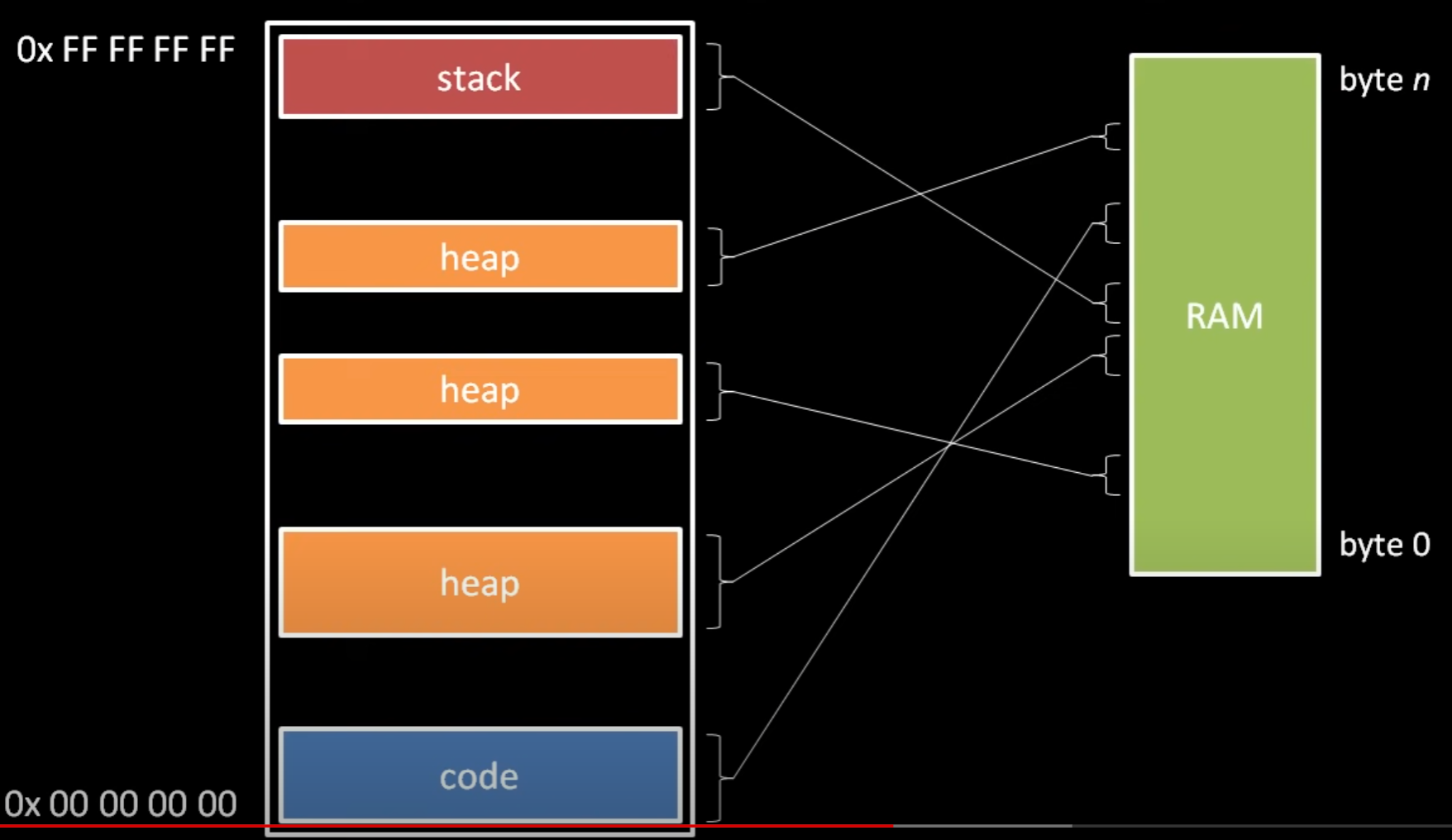
The scheduler then decides what OS code should be run next. This is called pre-emptive multitasking



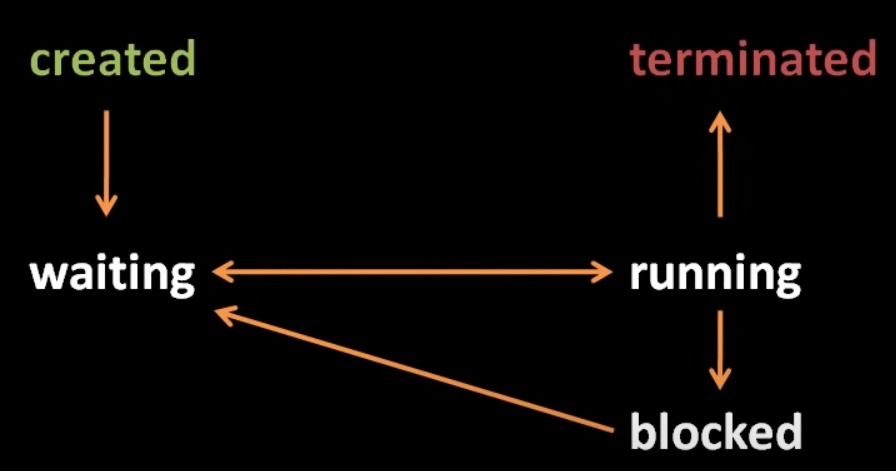
OS may access any portion of memory, processes can only access their own portion of memory, which is enforced by hardware. Processes can reach outside of their portion of memory, by invoking system calls to the OS’s portion of memory. After this call is received by the OS, the OS can grant access to the process.

How do the OS and the hardware restrict the process to only access its own portion of memory? By enforcing two different privilege levels. OS has the ring 0 privilege, which allows it to access the IO devices and any address of memory. When a process runs, the privilege level is revoked and reverted by to ring 3 which has the least amount of privilege. The rings of protection are done in hardware.

When the process runs, the CPU is put into a privileged level that triggers a hardware exception when the code attempts to directly access the IO devices or addresses not allowed for that process.



How the OS allocs a process’ internal organs to physical RAM, it will use virtual memory techniques.



The blocked state is triggers when the process invokes certain system calls such as reading files, because reading a hard drive is much slower than the operations of the CPU. Once the process completes its task that made it blocked in the first place, it can be placed back to the waiting state so that the scheduler can come to it. The most common reason for blocking is because the process is waiting for some slow device.

IPC: an umbrella term for any mechanism of the CPU that facilitates comms between processes. The simplest form of IPC are files. others are

* Pipes
* Networks
* Sockets
* Signals
* Shared memory

**PROGRAMMING LANGUAGES**

**machine code**: you literally change tubes before the advent of transistors. Each time you want to calculate one equation you need to change the tube configuration.

Later on, punch cards were invented to do the same thing which would take care of the handy work required to change tubes for each calculation. These were fed into the machines.

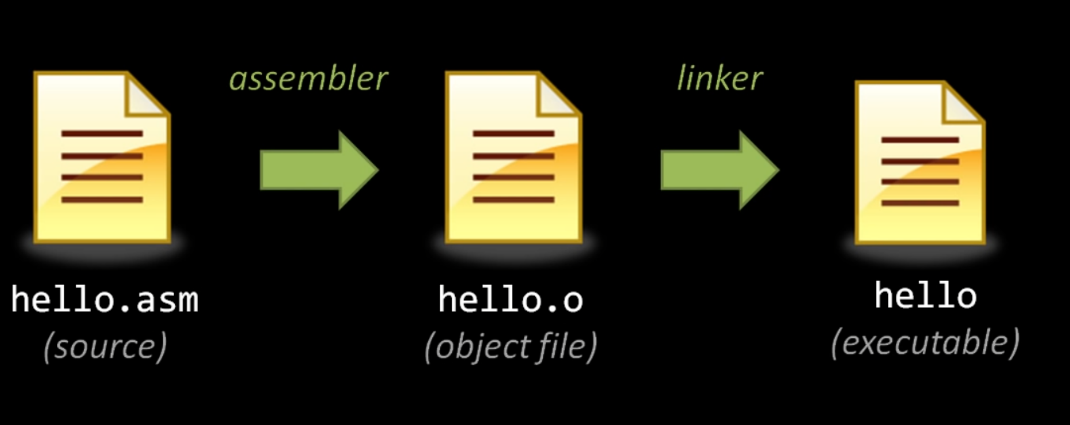
**Terminal:** computers can display text on displays, which were really primitive, no 3D graphics is possible.

**Assembly language:** instead of writing binary code (which is what machines understand due to transistor gates only being open or closed) assembly was developed which humans could understand. The text must be translated into what the machines understand, this process is called assembly.

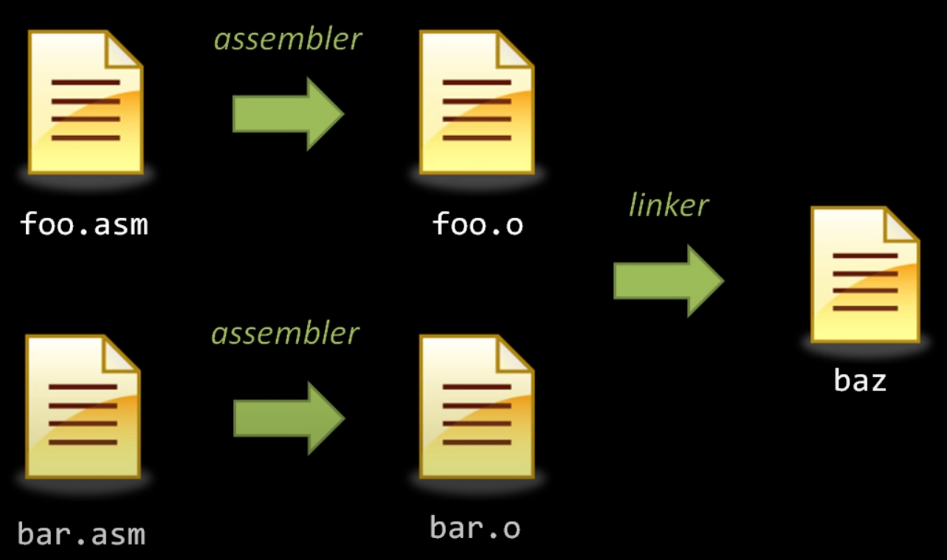
There are different assembly languages for different processors. (asm for x86, asm for ARM…)

In assembly, system calls are used extensively, which are OS specific, therefore asm written for one OS wont work for the other.

Asm written as text by the programmer is known as source code. The source code must be translated into machine code by the program called assembler. Every assembler is unique, they may not be compatible for each other due to processor, architecture and OS differences.



The reason why this approach is preferred is because we write multiple source files due to readability/portability/modularity reasons. This way they can be merged together later.



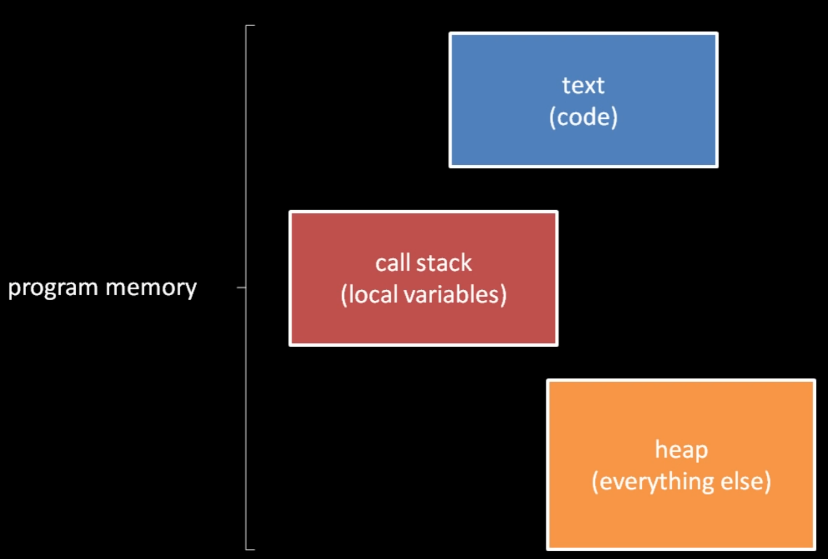
Linkers must resolve the links in code in object files, because the code written in separate source files is interdependent (they may invoke functions residing in each other etc…) since asm cant know this, they will leave a stub address where the function is invoked. The linker fills these stubs with actual addresses.

**Dynamic Linking:** modern OSes allow for processes to use code coming from other processes’ address space during execution. This is done for memory management reasons. Instead of copying the code every time it may be used, it is copied once and accessed later on if need be.

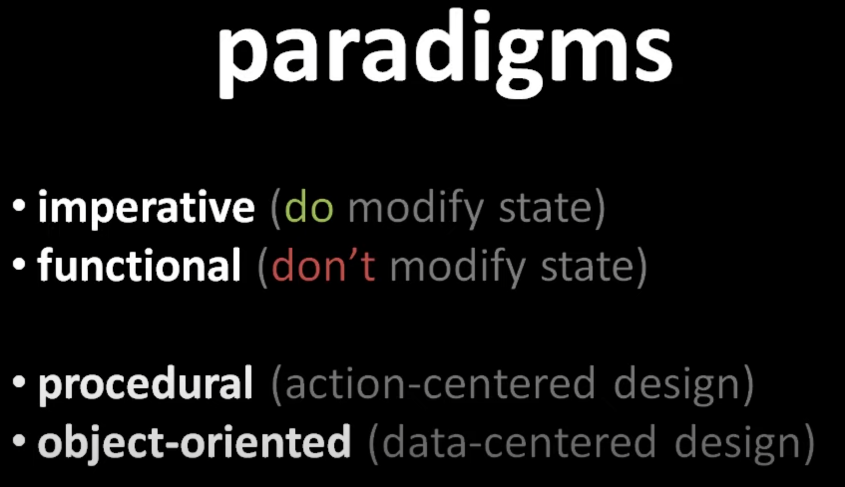
**High and Low Level:** you can combine object code generated from different levels such as one coming from asm and one coming from c. they will still be linked by the linker to produce an executable at the end.

Nothing in the semantics of C and C-derived languages can invoke system calls. System calls are literally what comms with the hardware to flip transistor gates. C-code is ultimately reliant on assembly code to invoke these system calls, which can create the specific machine instructions necessary to comms with hardware.

**Interpretation:** reads the source code and does what it says to do. No executable file is created when the source code goes through interpretation. Multiple source files can be fed into the interpreter the same way. The source code is complied into bytecode which is then fed to the interpreter by a virtual machine. Interpreter code allows to run the code in multiple platforms without first compiling for each platform. The platform must only have the virtual machine. Machine code is directly executed by CPUs, whereas bytecode must be parsed first, hence performance overheads.



Stack is allocated by the OS automatically, whereas heap must be managed by the programmer, when finished with the chunk, the programmer must release this memory specifically by demanding it from the OS. Java and Python have auto garbo collection, which does the demanding process automatically. Memory leaks are hard to pinpoint because they don’t show at the first execution but later on.



**Library:** It is a body of pre-existing code. When we write a new program, we don’t want to write everything from scratch, we use library code for common functionality. For example, by using a library, we can read and write files in the memory by invoking the functions provided in the library. All compilers come with the standard library depending the language, a c-compiler will come with the standard library. Whatever is not covered by the standard library, third party software may be used.

**HOW CAN YOU RUN CODE ON MULTIPLE DEVICES?**

You can run code on multiple platform with varying characteristics.

* The different makes of a processor, the different instruction sets a CPU might use and the machine code written specifically for that instruction set which uses a unique assembler are all different characteristics that one needs to take into account.
* Different OSes have different system calls and different APIs for invoking these system calls.
* Not all computers have the same OS and the same IO devices capable of running code which was meant for something else.
* Because of all of the above, the library code does not run on all platforms, hence the code dependent on the library will not work on the platform.

Solving these are as follows:

* Using a language higher than assembly will avoid directly using system calls, instead we invoke library code which invokes the system calls for us.
* The libraries have different code for each operating system, but our code can be the same for all platforms since our code outsources this problem to the library.
* If the target platform doesn’t support hardware, youre fucked my nigga zmaooo.
* Some differences, like file systems on Windows and Linux, can be worked around by avoiding the features which both systems don’t have in common.

**DLL**

DLLs are shared libraries; whereby multiple processes can access them at the same time. Applications are linked to libraries during runtime, not during compilation time. Libraries may also be linked to other libraries during runtime.

Dll files don’t get loaded into RAM with the main program. As long as the user does not need the library, it does not occupy space. When the user demands an application that needs the library, it is loaded and then after the activity is finalized, it will be destroyed.

Libraries create modular programs where the modules are separated from each other. Using one module does not need the inclusion of the other but the program can reach the modules whenever it wants to.

OS related DLLs reside inside the kernel, where it may be accessed from outside if the library needs to be loaded depending on the process. This way, the library doesn’t get duplicated each time a process demands it. It will be called from the source, used during runtime and then destroyed after use. If another program wants to access the library, it can access the memory which was assigned to the library when the first process wanted to use it.

Dlls are separated into 2 linking types:

* Load time dyn linking: the process makes explicit calls to the exported DLL functions like local functions. Reggie headers and libraries are imported. Linker will know that you demand the DLL at load time.
* Run time dyn linking: the process calls the LoadLibrary function (OS related funcs residing within the kernel) to load the DLL. After the DLL is loaded, use GetProcAddress function to obtain the address of the exported DLL function that the user wants to call. No importing is required.

Depending on performance and logic demands, you can choose between the two types.

DLLs will export variables, functions and resources by name. a client app will import the names to use those variables, functions and resources. Unlike a statically linked library, windows connects the imports in your app to the exports in a DLL at load time or at run time, instead of connection them at link time. Windows will require some info regarding these connections, which isn’t found in the usual C++ compilation cycle, thus MSVC is required to create DLLs.

DLLs use C calling conventions and this means that apps written in other programming languages can call these DLLs, as long as the platform, calling conventions and linking conventions match.

The client app uses implicit linking, where Windows links the app to the DLL at load-time, where the app can call the DLL-supplied functions just like the functions from a statically linked library. Explicit-linking would be loading DLLs at run-time.

To call the funcs or access data exported by a DLL, the client source code must have the declarations available at compile time. The DLL contains information about how to find the functions and data.

The client app needs to find the following:

* Headers that declare the DLL exports
* Import libraries for the linker
* The DLL itself

You can copy these into the client project if you are using third party DLLs. If you are writing the DLL yourself, store it somewhere else where you know how the version control is taken care of as to avoid duplication and mismatches.

Set the include path in the client project to include the DLL header files directly from your DLL project.

To avoid out-of-sync code, we recommend you set the include path in your client project to include the DLL header files directly from your DLL project. Also, set the library path in your client project to include the DLL import libraries from the DLL project. And finally, copy the built DLL from the DLL project into your client build output directory. This step allows your client app to use the same DLL code you build.

**SYSTEM CALLS**

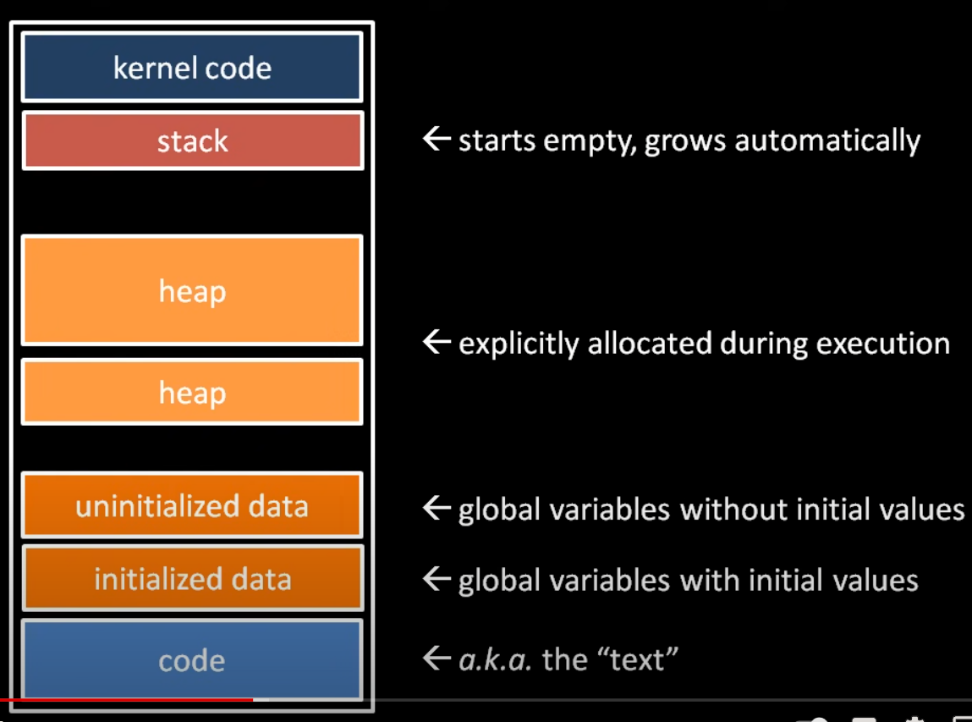
Everything descended from UNIX, but none of them are compatible with each other. In order to make them comms with each other, POSIX has been developed.

PORTABLE OPERATING SYSTEM INTERFACE FOR UNIX

Now, a program which is created for a variant of UNIX can run on all variants of UNIX.

For every process currently in the system, the OS keeps a data structure that keeps track of all the things associated with that process. These things include:

* Address space (memory table which is loaded when that process is running)
* User ID files
* File descriptors
* Environment
* Current directory and root directory



In modern OSes, the kernel code for the system calls and the table for the system calls inside the address space of each process. These pages cannot be accessed by the process, only when the process invokes a system call, execution jumps to the kernel code. When a syscall is invoked, it uses the stack of the process to place a stack frame for that system call, just like any other function. This avoids context switching and helps with suspending processes.

In compiled languages, the uninit and init data chunks have no need to either shrink or grow, because the compiler already knows from the get-go how many vars the program will use within its runtime.

**CPP**

**CPP STANDARD LIBRARY**

A collection of classes and functions which are written in cpp and conform to C++ ISO standards.

The library incorporates C standard library, these libraries end with “.h”.

The library is based upon the “Standard Template Library”. They share many features, with minor differences in between. These libraries are two distinct entities.

The library places performance, syntax and semantics requirements on generic algorithms. Generic algorithms are required to have linear 0(n) time or 0(nlogn) time constraints. Such a requirement can be seen as an example in the sorting algorithms.

CPP Standard Library is available pre-built in most if not all compilers. Each compiler has a specific implementation of the library. Compilers may have proprietary/unique libraries, but that’s their business. You don’t need to include the path of the library, the compiler will find it automatically.

* GNU C++
* LLVM C++
* NVIDIA C++
* MS C++
* Electronic Arts STL
* HPX C++ for parallelism and concurrency

With each iteration of the C++ library, new features are added. The latest is the C++20 version. Not all programs, not all hardware can use higher versions of the library, therefore, some features must be implemented manually even though this specific feature may be present pre-built in a later library.

Some of the barebone header files are as follows.

* *Chrono*: time elements
* *Functional*: provides function objects
* *Memory*: memory management in c++
* *Stdexcept*: exception handling
* *Exception*: exception handling
* *Limits*: fundamental numeric types
* *New*: operators of new and delete for c++ memory management
* *Typeinfo*: runtime type information
* *Array*: container class template for fixed sized arrays.
* *Tuple*: class template for tuples
* *Bitset*: bit arrays
* *Deque*: double ended queues
* *List*: container class template for doubly linked lists.
* *Queue*: single-ended queue and priority queue.
* *Stack*: stacks
* *Vector*: container class template for dynamic arrays.
* *Algorithm*: various algorithms
* *Execution*: parallelized algos
* *Iterator*: iteration
* *String*: c++ standard string classes and templates.
* *Regex*: reggie expr.
* *Fstream*: file io
* *Iomanip*: manipulate output formatting
* *Ios*: iostream operations
* *Iostream*: c++ io fundamentals
* *Mutex*: mutual exclusion, locks and call once.
* *Shared*\_mutex: shared exclusion.
* *Thread*: class and namespace for working with threads
* *Bit*, *complex*,*random*,*numeric*: numerics library

Each header from the C lib is included in c++, the name of the library changes from/to i.e. <time.h> --- <ctime>