Architecture

Simplified registers are instruction register (`ir`), program counter (`pc`) and stack pointer (`sp`).

A simplified *Von-Neumann Cycle* looks like

1. `pc = startValue` (initialize program counter)
2. `ir = mem[pc]` (fetch instruction from memory)
3. `pc = pc + 1` (advance program counter)
4. `execute(ir)` (execute instruction)
5. handle interrupt
6. go to 2)

and is divided into a fetch stage and an execute stage. The „handle interrupt“ step is

1. is interrupt pending? if no continue normally, otherwise continue here
2. save `pc`
3. disable the interrupt
4. `pc = isrAddr`

where `isrAddr` is the Interrupt Service Routine (ISR). This central distribution point is called „interrupt handler“.

Bootup sequence

1. Level 1 bootloader (UEFI/BIOS, ...) initializes basic hardware
2. Level 2 bootloader (grub, Windows Boot Manager), reads drives to initiate booting the OS
3. OS kernel, initializes (almost) all hardware
4. OS usermode (services, applications)

Rings

Usermode subset of instructions, restrictions on memory access

Kernelmode no restrictions, used by OS kernel

From kernel to usermode: write to register. From user to kernelmode: Syscall.

SMP (Symmetric Multiprocessing)

Processors of similar capability can share main memory, I/O components, etc. Used in mulitcore processors.

1st level cache is per-core, 2nd level cache is usually shared.

Processes and Threads

A program is a set of instructions and static data. *Processes* are instances of running programs, they have their own state (registers, resources, ...). A process is either running, ready or blocked.

A process contains at least one but usually many *threads*, which share address space, file handles, etc. Threads are thus usually faster to create and switch between than processes. Threads still have their own execution context (registers, priority, *stack*, ...).

IPC (Interprocess Communication)

Communication between processes (or threads) requires some shared medium. Threads have shared memory, processes can use

- files (in a shared file system)
- named pipes (pseudo files)
- anonymous pipes (shell connecting `stdout` of one process to `stdout` of another)
- sockets (client/server communication)

Race conditions are a common problem. (When results depend on order of execution.)

Scheduling

Short-Term Scheduling is classified to be either

- *non-preemptive* (running tasks cannot be interrupted from outside, except for interrupts)
- *preemptive* (running tasks can be suspended in favor of other tasks, doesn't require cooperation from the processes)

Some important scheduling algorithms are

- First Come First Served (FCFS)
- Shortest Job first (SJF) and Shortest Remaining Time Next (SRTN)
- Priority queuing (optionally with aging)
- Round-Robin (RR)

Shortest Job First does as the name implies. It minimizes *waiting time* of a set of processes. For interative systems this isn't always a high priority (they should optimize for minimal time until the task is started).

Round-Robin A small time interval called *quantum* is defined. After one quantum is elapsed, switch to the next task in the queue and add previously running one to the end.

Priority Scheduling Processes are assigned a priority class. Processes with higher priority are executed first. Execution within a priority class is decided by another algorithm (eg. RR).

A problem with naiive priority scheduling is *starvation*, when processes with low priority are never executed because new processes with higher priority are added. One solution is *aging*, where priority of a process is increased based on its waiting time so far.

Memory

The layers of memory in order of usual capacity are

- Registers
- CPU Caches
- Main Memory (RAM)
- Flash
- Hard disk

Relative and Absolute Addresses

There are two options for address binding

- absolute memory addresses (small microcontrollers, I/O drivers, ...)
- relative memory addresses (translated to absolute addresses at load or run time, stored in the program as if it were loaded at address zero, then adjusted by the loader later)

A loader is part of the OS that...

1. loads a program to be started from disk,
2. adapts addresses if necessary (see below)
3. and starts the program

If the loader needs to adapt addresses (ergo if relative memory addresses are used) there are two options:

Relocatable code has a compiler generated *relocation table*, containing pointers to all relative addresses inside the binary. The loader then changes all those relative addresses to absolute ones by adding the base.

Position-independent code (PIE) is compiled to run at arbitrary memory locations by (eg.) only using relative addressing/jumps or other internal address translation methods. This creates a run time overhead.

Logical and Physical Addresses

There are two ways to view a memory address.

Logical addresses are from the point of view of a program, also called virtual addresses. The logical/virtual address space is the set of all possible addresses that are provided to processes.

Physical address is the real address on system RAM. The physical address space are all available physical addresses.

The mapping between them can happen in the *Memory Management Unit* (MMU) of the CPU. This is essentially hardware support for adding a base to an address given as an offset.

The OS takes care of tracking logical address spaces (protecting them from each other). The MMU can take care of the simple logical to physical mapping. But the OS needs to reconfigure the MMU at run time with the correct base addresses.

Paging

Logical address space is split into fixed size blocks called *pages*. Physical address space is split into fixed size blocks called *page frames*. (They have the same size.)

Interpret logical address as `page number` + `offset`. (For an address with m bits, first $m-n$ bits are the page number, following b bits are the offset).

A page table keeps track of the mapping between pages and page frames. It contains pointers to the actual page frames. The mapping logic is implemented in hardware inside the MMU.

Every process appears to have its own private logical address space and has its own page table. Every memory access operations now needs to go through the page table (special hardware cache, translation look-aside buffer).

Page table entries usually have additional bits to keep track of state. For example valid (currently assigned?), modified (dirty?), referenced, protection (read-only, no-exec), ...

Shared Pages allow assigning one page frame to multiple processes (thus to multiple pages). This is useful for sharing code (SO/DLL) and data (IPC).

Copy-on-Write is a situation in which a process attempts to write to a shared data page with the read-only bit set. The OS catches the access, creates a copy of the page and then lets the process write to the new copy. This means that creating process with shared pages cheap as long as they don't diverge.

Virtual Memory is a mechanism for overcommiting memory. Only a subset of memory pages are mapped to physical page frames. Currently inactive pages are stored on mass storage (*swap files*).

A *page fault* occurs when a process tries to access a page which is not in main memory (shown by the *valid* bit in the page table entry). MMU notices this and raises a *page fault*. OS then swaps in (fetches) the page from disk and adjusts the page table entry (set valid bit, set page frame number). Then the instruction is restarted. If necessary the OS might also swap out some other page to make space.

Security Preventing attacks at OS level:

- Data Execution Prevention (no-exec bit on data sections)
- Address Space Layout Randomization (ASLR)

Page Replacement Algorithms

OS sometimes has to choose a page to evict.

Optimal Algorithm Evict the page that will next be referenced at the latest point in the future.

This algorithm cannot be implemented, future cannot be predicted accurately enough.

Least Recently Used Evict the page that hos not been referenced for the longest time period.

Clock Replacemenet Algorithm Make the page table a circular list, can be imagined to be a „clock“, the „hand“ points to a given item. When a page fault occurs: Advance the hand, if the page now being pointed to by the hand has use bit zero, evict and replace that page. Otherwise if the use bit is one, set it to zero and advance the hand.

Thrashing is the situation where there are many processes relative to the available physical memory, thus processes create many page faults because each has a small amount of page frames. OS is then mainly busy with swapping pages in and out.

File Systems

Mass storage devices have two main operations: Reading and writing a block of some hardware-defined block size. File systems are an abstraction over this, they exist to provide a consistent API.

The logical unit of organization is the file, which is a collection of blocks coupled with metadata.

File Metadata will usually include

- Name (depending on charset and locale, *human readable*, at least „8.3“ naming scheme)
- Type (not always)
- Location (pointer to physical or logical address of file, *persistent*)
- Size
- Permissions (owner, groups; who can do what)
- Data & Time (created, last change, last access)
- Flags (hidden, temporary, . . .)

OS Tasks in regards to files are

- Creating and deleting files
- Structuring files in a hierarchy
- Reading and modifying data in files
- Persisting the files on storage mediums (support multiple concurrent mediums, different file systems)
- Controlling access to files (user/process permissions, context, . . .)

Special File Types might include *directories*, *character* and *block device* files (eg. UNIX for device I/O), *named pipes*, *sockets*, hard or soft *links*.

File systems organize data in blocks, comparable to pages for memory. Physical block size (*sector*) depends on storage medium (HDD 4kb, SSD 1MB). Logical block size represents a fixed number of sectors (*quantum*, usually 2^n).

Journaling file systems address the issue of inconsistency on OS crash (file system may be left inconsistent, caches may not be written and lost, . . .). Each action is performed as an atomic *transaction*, the steps are written to a *journal* and removed once performed. If a non-empty journal is found on bootup, just execute the steps.

Virtual file systems are an abstraction over concrete file systems that unify them under a single root tree. Different file systems are mounted at different directories, transparent to the user.

Allocation Strategies

Contiguous file allocation is simple and fast, but files cannot be expanded. Lots of external fragmentation (holes between allocated blocks that won't be filled). Useful for write-once media (optical CD), otherwise not so much.

Allocation with a *linked list* is better, every cluster has a link to the next one. Random access is slow, free clusters have to be tracked in a separate list (or something).

Alternative: *File Allocation Table* (FAT). A linked list of pointers to clusters is kept at the beginning of a file system. Random access is improved because usually the whole FAT will be in RAM. Special FAT entry value denotes an empty cluster, no need for a separate list.

Indexed allocation keeps data about file allocation combined per file. Eg. the first cluster of a file will contain pointers to the clusters of that file (in order). High overhead for small files (each file needs at least one index block). Free cluster management is easy, just assign them all to a special file or keep track in a bitmap (one bit for every cluster).

A variation on indexed allocation (*RUNs*) stores `(cluster pointer, size)` pairs in the index block. This is useful if many blocks will be allocated continuously, data in index blocks will be smaller. Used by NTFS.

Disk Layout

Most mass storage devices are divided into *partitions*. (Tracked in the MBR for old BIOS implementations or GUID Partition Table, GPT, for UEFI.)

A bootup sequence with an MBR disk layout will look like

1. BIOS loads MBR, which
2. analzes the Partition Table and starts the Partition Boot Record, which starts
3. OS bootstrap, which loads the OS

Concurrency

Process are executed either sequentially or concurrently. Concurrent processes are either *independent* (if they use different data regions) or *interacting* if they use shared data. Coordination between concurrent processes is called *synchronization*.

Producer-Consumer Problem Several producers and consumers operate on a shared object. Particular example: Shared counter that is increased by producers and decreased by consumers. Concurrent access leads to a race condition.

This motivates the concept of *mutual exclusion*: Define *critical regions* which only one process can enter at a time. Requirements are

- mutual exclusion (obviously, only one process can be inside a CR at any time)
- no assumptions (order of execution, ... are arbitrary)
- progrss (no process not currently in a CR may block another process from entering that CR)
- fairness (each process must be able to enter a CR in finite time)

Test and Set Lock reads a memory word and writes another value to it in a single atomic CPU instruction. During this, the memory bus must be locked.

Spin Lock Test and Set Lock is used in a spinlock. Between the spinlock and setting the bit to false is the CR. Relies on atomic writing. Inefficient in case the lock takes a while to acquire.

This section could be expanded significantly.

Security

Security is preventing losses due to intentional actions by malovelent actors. *Safety* is preventing losses due to unintentional actions, usually by benevolent actors.

Basic Security Requirements are *confidentiality* (prevention of unauthorized disclosure), *integrity* (prevention of information or system modification) and *availabilty* (ensuring access to and use of information). Also called CIA triad.

When security and *usabilty* collide, usabilty always wins.

Computer System Layers are Applications > Kernel > Hypervisor/UEFI > Firmware > Hardware.

Threats Attacking running processes with higher privileges (defended against with eg. ASLR), accessing memory not assigned to own process (MMU, paging), accessing files without permission, ...

Access Control Policies are

Discretionary Access Control (DAC) based on the identity of the requestor and on access rules set by the owner of the entity. The controls are discretionary in the sense that a subject can pass it's permissions on to any other subject. Typical subject classes are: world (all subjects), group (of subjects) and owner.

Mandatory Access Control (MAC) based on comparing security labels with security clearances; mandatory because owner may not be able to delegate access (as opposed to DAC).

Role-Based Access Control (RBAC) based on roles that users/processes have within a system and rules based on those roles.

Access control matrices are often stored as an

- *Access Control List* (ACL), a column of the access control matrix which contains subjects and their access rights on a given object
- *Capability Vector*, a row of the access control matrix which contains objects and the access rights a given subject has on them

Trusted computing base (TCB) is a portion of a system that enforces a particular policy, must be resistant to tampering and circumvention. Informally, these are the components that one *has* to trust, for the system to be trustworthy.

Virtualization

... is an abstraction of the resources used by some software which runs in a virtual machine.

- better efficiency in the use of physical system resources (supposedly)
- support for multiple distinct OS
- additional security concerns

An additional layer, a *hypervisor* (virtual machine manager, *VMM*) should guarantee safety, fidelity and effieciency.

Different possibilities for a VMM implementation

- on top of host OS (*type 2*, eg. VirtualBox, VMWare Workstation) vs. bare-metal (*type 1*, eg. Hyper-V, L4)
- *full virtualization* (unmodified guest OS, worst performance) vs. *paravirtualization* (guest OS aware, special syscalls) vs. compartmentalization within a single OS kernel (*containers*, not real virtualization, eg. Docker)
- *emulation* is possible but slow

Embedded and Real-Time

The correctnes of a real-time OS depends not only on logical computation results but also on the time in which the results are produced.

Important requirements for RTOS are being deterministic, responsive, reliable and fail-soft opertaion (fail in a way that minimizes consequences).

Deadlines for some actions need to be kept: either guarantee to finish a task within a time period or guarantee that the system reacts within a time period. With *hard real time* it is critical that deadlines are met at all times, with *soft real time* it's tolerable if some are missed.

Priority inversion example: Process with low priority holds a resource which a process with high priority wants to access.