Computer Systems

by dcamenisch

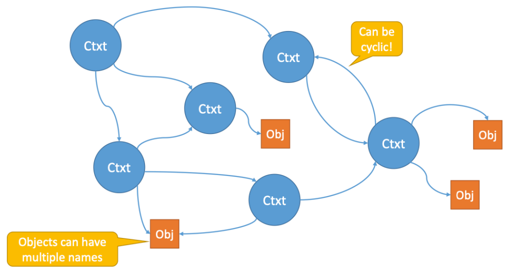
1 Introduction

This document is a summary of the 2022 edition of the lecture *Computer Systems* at ETH Zurich. I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. If you spot any mistakes or find other improvements, feel free to open a pull request at `https://github.com/DannyCamenisch/systems-summary`. This work is published as CC BY-NC-SA.

Operating Systems

2 Naming

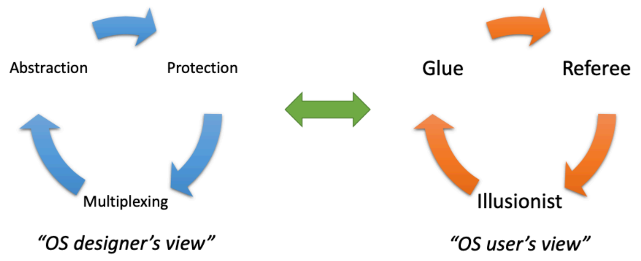
Naming is a fundamental concept, it allows resources to be bound at different times. Names are bound to objects, this is always relative to a context. One example of this would be path names, e.g. */usr/bin/emacs*. Name resolution can be seen as a function from context and name to some object. The resolved object can be a context in itself. This gives us a naming network.



Both synonyms (two names bound to the same object) and homonyms (the same name bound to two different objects) can occur.

3 The Kernel

These are the main functions of any operating system. Commonly they are referred to from a designer's view, but the user's view can be much more helpful to actually understand how it works.



3.1 Bootstrapping

The term bootstrapping refers to pulling himself up from his own boots. In Computer Systems it is what we call the process of starting up a computer (booting). This boot process looks like this:

1. CPU starts executing code at a fixed address (Boot ROM)
2. Boot ROM code loads 2nd stage boot loader into RAM
3. Boot loader loads kernel and optionally initials file system into RAM
4. Jumps to kernel entry point

The first few lines are always written in assembly, but generally we want to switch to C as soon as possible.

3.2 Mode Switch

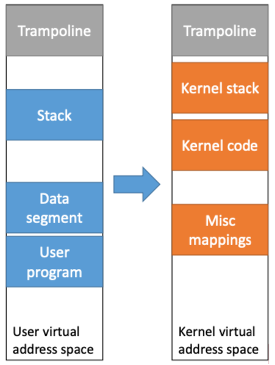
One of our main goals is to protect the OS from applications that could harm it (intentionally or not). For this purpose we introduce two different modes:

- Kernel mode - execution with full privileges, read/write to any memory, access and I/O, etc. Code here must be carefully written
- User mode - limited privileges, only those granted by the OS kernel

These two (or more) modes are already implemented in hardware. The main reason for a mode switch is when we encounter a processor exception (mode switch from user to kernel mode). If this is the case, we want the following to happen:

1. Finish executing current instruction
2. Switch mode from user to kernel
3. Look up exception cause in exception vector table
4. Jump to this address

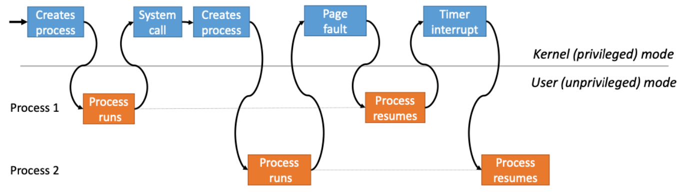
Further we may also want to save the registers and switch page tables. When switching between the modes we also have to change our address space, but we might want to access some informations from the user mode address space. One way of doing this is to use a so called **trampoline**, which is a part of the address space that gets mapped to the same location in user and kernel mode.



Mode switches can also occur the other way around (from kernel mode to user mode). The main reasons for this are:

- New process / thread start
- Return from exception
- Process / thread context switch
- User-level upcall (UNIX signal)

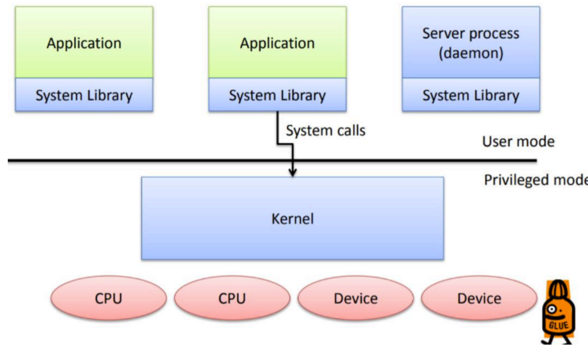
This leads us to the following perspective:



The mode switch is fundamental to modern computers:

- It enables virtualization of the processor
- It creates the illusion of multiple computers
- It referees access to the CPU

3.3 General Model of OS Structure



- Kernel - code that runs in kernel mode
- System Library - interface to kernel, enough for most programs
- Daemon - user space process which provides OS services, varying levels of privileges

We can differentiate between monolithic kernels and microkernels, depending on the amount of code in kernel mode.

3.4 System Calls

System calls are the only way for user mode programs to enter kernel mode. System calls are a type of exception, but they try to look a lot more like a procedure call. Therefore the kernel system call handler has to first locate arguments, copy these arguments into kernel memory, validate these arguments and then copy the results back into user memory after execution. An example of such a system call would be *write()*.

3.5 Hardware Timers

What happens if a user mode program does not cause any exception and does not give control back to the kernel? Hardware timers are a solution for this problem, the hardware device periodically interrupts the processor and returns control to the kernel handler, which sets the time of the next interrupt.

4 Processes

When you run a program, the OS creates a process to execute the program in. A process is an illusion created by the OS. It is an execution environment for a program. This environment gives the program limited rights (access, name spaces, threads, etc.) and therefore it is both a security and a resource principal.

4.1 Creating a Process

There are two main approaches to creating new processes:

- **Spawn** - constructs a running process from scratch
- **Fork / Exec** - creates a copy of the calling process or replaces the current program with another in the same process

Spawn

- Create and initialize the process control block (PCB) in the kernel
- Create and initialize a new address space
- Load the program into the address space
- Copy arguments into memory in the address space
- Initialize the hardware context to start execution at "start"
- Inform the scheduler that the new process is ready to run

Spawn is very complex, we have to specify everything about the new environment. If we omit a key argument a new process might have insufficient rights or resources or it might fail to function due to a security fault.

Fork
Fork on the other hand is less complex. The child process is almost an exact copy of the parent, with a different PID. We know which process we are in from the return value of the *fork()* call (0 for child, > 0 for parent, < 0 for error). The complete UNIX process management API also includes:

- *exec()* - system call to change the program being run by the current process
- *wait()* - system call to wait for a process to finish
- *signal()* - system call to send a notification to another process

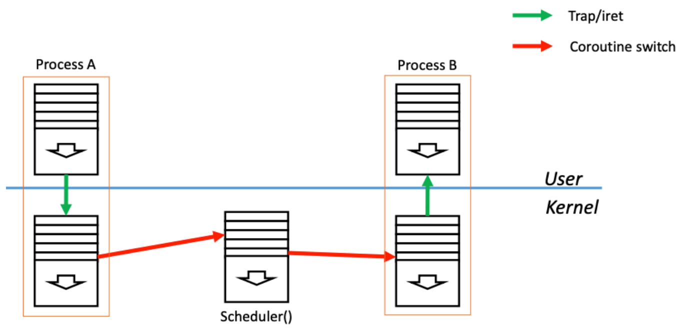
In contrast to *spawn()*, here the child revokes rights and access explicitly before *exec()*, further we can use the full kernel API to customize the execution environment.

4.2 The Process Control Block

The PCB is the main kernel data structure used to represent a process. It has to hold or refer to the page table, trap frame, kernel stack, open files, program name, scheduling state, PID, etc.

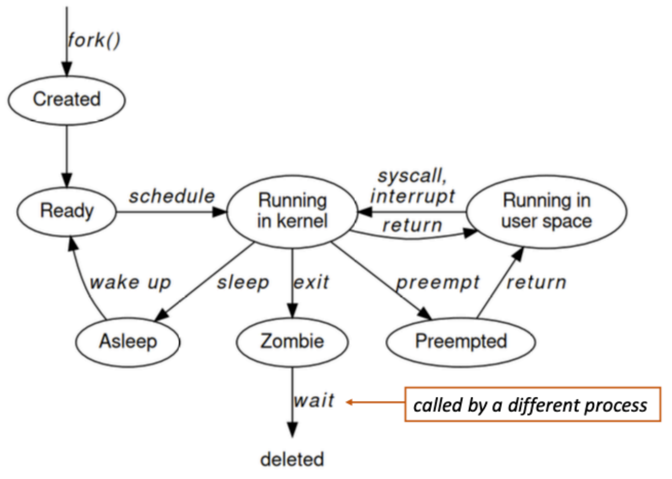
4.3 Process Context Switching

Context switching is the process of switching between different processes running in user mode or kernel mode. It is one of the key elements of the illusion that multiple programs can run in parallel.
There are two main reasons for the kernel to switch processes: either when a process has run for too long and gets interrupted by a hardware timer or when a process blocks. The second case happens when a system call can not complete immediately. The process then often calls *sleep()* and other processes can be executed.



4.4 Process Hierarchy

By forking and spawning new processes we create sort of a hierarchy. If a child process dies, but the parent does not call *wait()*, the child process becomes a zombie - it is dead, but still around since nobody asked for the return code. If a parent dies, but the child does not, the child becomes an orphan and gets reparented to the first process (PID #1, *init*).



The *init* process is basically an infinite loop calling *wait*(NULL), it gets rid of any zombies.

5 Inter-Process Communication

It is often the case, that we want different processes to work together, e.g. DB and web-scraper. For this we need ways to exchange information between processes, we call this inter-process communication or IPC.
One of the most basic ways to exchange information is system calls. We save our data on the stack or in a register and execute the system call, the kernel then switches execution to the other process with the information about the data that should be exchanged. This is a really flawed approach some reasons are that context switches are expensive and it is unsafe to introduce new system calls for each task.

5.1 Shared Memory

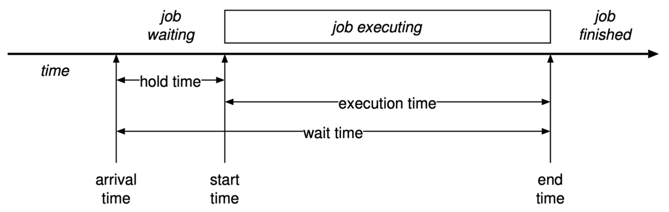
We introduce shared memory that can only be accessed by the processes that communicate with each other. This requires us to define a common interface between the processes. The main building blocks for this are:

- Shared area: registers, memory - define the layout of the memory
- Indicating status: shared variables, signals
- Updating status: changing variables
- Consistency: use synchronization primitives

When checking the state of a shared variable polling can be very inefficient. As an alternative we can use a signal handler. When the first process calls the signal handler, the kernel notifies the second process of the update. When the kernel issues such a system call to a process, we call it an **upcall**. To guarantee consistency, we use synchronization primitives that are already known from previous lectures (spin locks with CAS, TAS, etc.).
A more modern approach is to use transactional memory, hereby we work with transactions that can fail on race condition.

6 Scheduling

First we introduce some terminology, notice how the wait time is defined as the combination of hold time and execution time.



The important metrics for scheduling are throughput (rate of completing jobs) and overhead (time spent without a job executing).
In the following part we will look at various scheduling algorithms, starting from a very simple approach.

6.1 Non-Preemptive Batch Oriented

There are two basic algorithms:

- First Come First Served
- Shortest Job First

Both are algorithms are very simple to implement but their applications are limited.

6.2 Preemptive Batch Oriented

We introduce preemption, meaning that we interrupt the execution after some finite time interval. After such an interrupt we decide which program to run next.
Modified SJF: Go through the sorted list of jobs and execute each until interrupted. Note that the length of a job does not get recomputed after an execution.

6.3 Interactive Scheduling

When running interactive workload, we can encounter events that block the execution (I/O, page-faults, etc.). During this time we would like to run another program instead of wasting execution time.
Round Robin Scheduling: Let R be a queue and q be the scheduling quantum:

1. Set an interval timer for an interrupt q seconds in the future
2. Dispatch the job at the head of R

3. If blocking happens or the timer runs out, return to the scheduler
4. Push the previously running job to the tail of R

Priority Based Scheduling: We assign a priority to each task and dispatch the highest priority task first, if we have ties we use RR to break them. To avoid starvation we might want to use dynamic priorities (increased priority depending on wait time). An even more complex approach is to introduce multi-level queues, meaning that we have a fixed number of queues that assign priorities within each queue. Then we use RR scheduling between the queues to execute tasks.

Priority scheduling runs into problems when a high priority process has to wait for a lock from a low priority process (Priority Inversion). To fix this, we can either introduce inheritance - the holder of the lock acquires the priority of the highest waiting process - or ceiling - the holder of the lock runs at the highest priority.

6.4 Linux o(1) Scheduler

Linux uses 140 multilevel feedback queues, each with a different priority. Multilevel feedback queues penalize CPU bound tasks and prioritize I/O operations, as I/O tasks will eventually block.
The priority range 0-99 is used for high priority, static tasks and it uses FCFS or RR scheduling. The range 100-139 is for user tasks and uses RR together with priority ageing for I/O tasks.

7 Input / Output

Every OS has an I/O subsystem, which handles all interaction between the machine and the outside world. The I/O subsystem abstracts individual hardware devices to present a more or less uniform interface, provides a way to name I/O devices, schedules I/O operations and integrates them with the rest of the system, and contains the low-level code to interface with individual hardware devices.

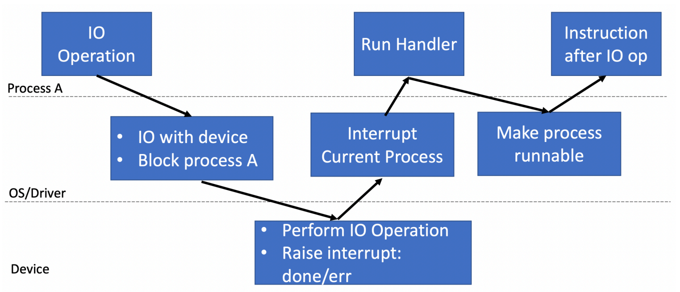
To an OS programmer, a **device** is a piece of hardware visible from software. It typically occupies some location on a bus or I/O interconnect, and exposes a set of hardware registers which are either memory mapped or in I/O space. A device is also usually a source of interrupts, and may initiate Direct Memory Access (DMA) transfers.

The **device driver** for a particular device is the software in the OS which understands the specific register and descriptor formats, interrupt models, and internal state machines of a given device and abstracts this to the rest of the OS.

7.1 Data Transfer

Programmed I/O consists of causing input/output to occur by reading/writing data values to hardware registers. This is the simples form of communication. It is fully synchronous, so the CPU always has to be involved. Further it is polled, the device has no way to signal that new data is ready.

Interrupts can be used to signal the availability of new data and solve the polling problem. But the problem of CPU involvement still persists.



Direct Memory Access or DMA, a device can be given a pointer to buffers in main memory and transfer data to and from those buffers without further involvement from the CPU. A single interrupt is used to signal the end of data transfer. DMA is, for the most part, physical (not virtual) access to memory. Further DMA transfers to and from main memory may or may not be coherent with processor caches.

7.2 Asynchrony

Device drivers have to deal with the fundamentally asynchronous nature of I/O: the system must respond to unexpected I/O events, or to events which it knows are going to happen, but not when.

The **First-level Interrupt Service Routine** (FLISR) is the code that executes immediately as a result of the interrupt. It runs regardless of what else is happening in the kernel. As a result, it can't change much since the normal kernel invariants might not hold.

Since I/O is for the most part interrupt-driven, but data is transferred to and from processes which perform explicit operations to send and receive it. Consequently, data must be buffered between the process and the interrupt handler, and the two must somehow rendezvous to exchange data. There are three canonical solutions to this problem:
A **deferred procedure call**, is a program closure created by the 1st-level interrupt handler. It is run later by any convenient process, typically just before the kernel is exited.

A **driver thread**, sometimes called an interrupt handler thread, serves as an intermediary between ISR and processes. The thread starts blocked waiting for a signal either from the user process or the ISR. When an interrupt occurs or a user process issues a request, the thread is unblocked (this operation can be done inside an ISR) and it performs whatever I/O processing is necessary before going back to sleep. Driver threads are heavyweight: even if they only run in the kernel, the still require a stack and a context switch to and from them to perform any I/O requests.

The third alternative, is to have the FLISR convert the interrupt into a message to be sent to the driver process. This is conceptually similar to a DPC, but is even simpler: it simply directs the process to look at the device. However, it does require the FLISR to synthesize an IPC message, which might be expensive. In non-preemptive kernels which only process exceptions serially, however, this is not a problem, since the kernel does not need locks.

Bottom-half handler - the part of a device driver code which executes either in the interrupt context or as a result of the interrupt.

Top-half handler - the part of a device driver which is called "from above", i.e. from user or OS processes.

7.3 Device Models

The device model of an OS is the set of key abstractions that define how devices are represented to the rest of the system by their individual drivers. It includes the basic

API to a device driver, but goes beyond this: it encompasses how devices are named throughout the system, and how their interconnections are represented as well.

UNIX device model:

- Character Devices - used for unstructured I/O and present a byte-stream interface with no block boundaries.
- Block Devices - used for structured I/O and deal with blocks of data at a time.
- Network Devices - correspond to a network interface adapter. It is accessed through a rather different API.
- Pseudo Devices - a software service provided by the OS kernel which it is convenient to abstract as a device, even though it does not correspond a physical piece of hardware.

7.4 Protection

Another function of the I/O subsystem is to perform protection. Ensuring that only authorized processes can access devices or services offered by the device driver and that a device can't be configured to do something harmful. Unix controls access to the drivers themselves by representing them as files, and thereby leveraging the protection model of the file system. DMA-capable devices are in principle capable of writing to physical memory anywhere in the system, and so it is important to check any addresses passed to them by the device driver. Even if you trust the driver, it has to make sure that its not going to ask the device to DMA somewhere it shouldnt. One approach is to put a memory management unit (MMU) on the path between the device and main memory, in addition to each core having one.

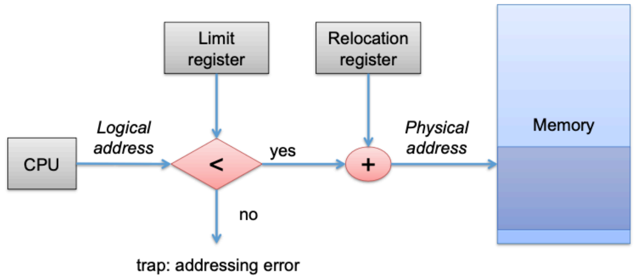
8 Virtual Memory

From previous courses we know MMUs, TLBs and basic paged virtual memory operations. The uses for address translation include:

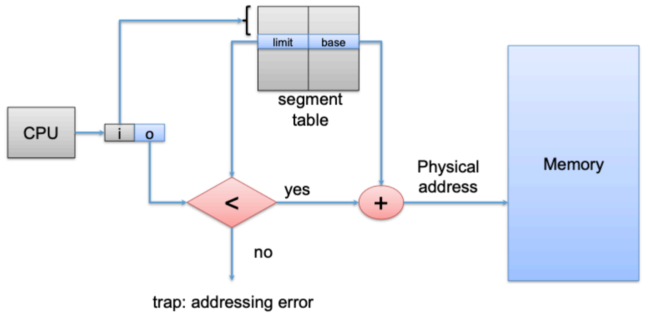
- Process Isolation
- Shared Code Segments
- Dynamic Memory Allocation
- Persistent Data Structures
- etc.

8.1 Segments

Before paging, there were segments. Before segments there where base and limit registers. These contained two addresses B and L. A CPU access to an address a is permitted iff $B \leq a < L$. Relocation registers are an enhanced form of base register. All CPU accesses are relocated by adding the offset: a CPU access to an address a is translated to $B + a$. This allows each program to be compiled to run at the same address (e.g. 0x0000).



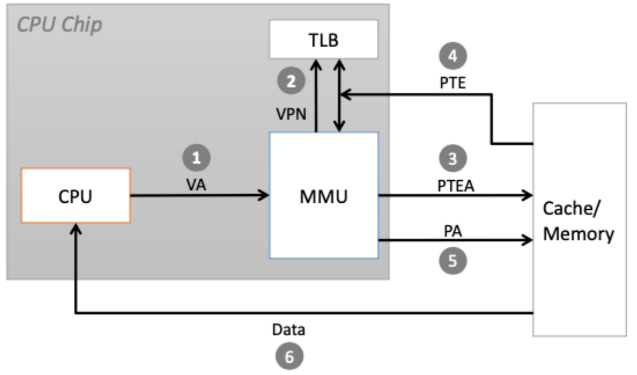
A segment is a triple (I, B_I, L_I) of values specifying a contiguous region of memory address space with base B_I, limit L_I, and an associated segment identifier I which names the segment. Memory in a segmented system uses a form of logical addressing: each address is a pair (I, O) of segment identifier and offset. A Segment Table is an in-memory array of base and limit values (B_I, L_I) indexed by segment identifier, and possibly with additional protection information.



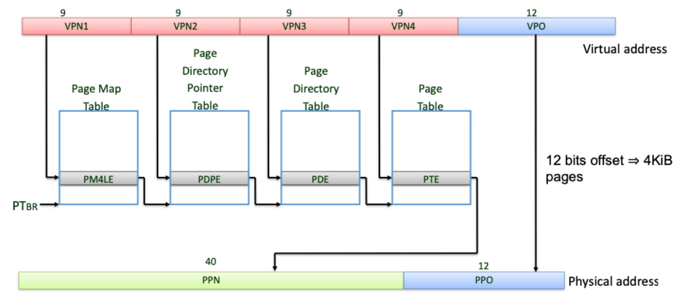
This enables sharing code/data segments between processes, further it adds protection and allows transparently growing stack/heap as needed. The principal downside of segmentation is that segments are still contiguous in physical memory, which leads to external fragmentation.

8.2 Paging

This is a short recap of paging. Virtual memory is divided into (virtual) pages of the same size having a VPN, physical memory gets divided into frames / physical pages having a PFN / PPN. Then a page table gets used to map VPNs to PFNs. This is implemented in hardware as the MMU. To speed up translation the TLB is used. Getting data from memory can work as follows (in this case we have a TLB miss):



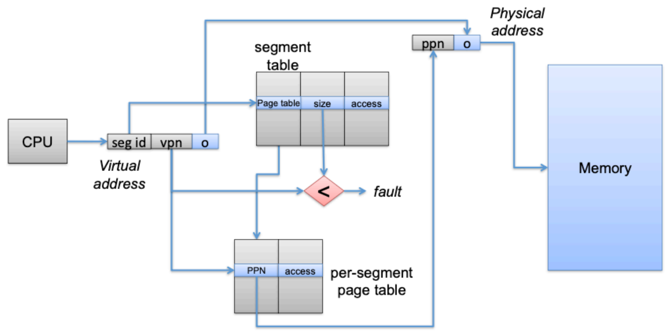
We have also seen how multi-level page tables work. Multi-level translation allows us to allocate only page table entries that are in use and makes memory allocation simpler.



On a context switch we need to store/restore the pointer to the page table and its size. One of the downsides of paging is if our page is very small and we do not need all of the space, this leads to internal fragmentation.

8.3 Paged Segmentation

It is possible to combine segmentation and paging. A paged segmentation memory management scheme is one where memory is addressed by a pair (segment.id, offset), as in a segmentation scheme, but each segment is itself composed of fixed-size pages whose page numbers are then translated to physical page numbers by a paged MMU.



One of the main benefits here is that each segment can have its own size of page table.

8.4 Zero-on Reference

The question how much physical memory is needed, is difficult to answer. So when a program runs out of memory, the system does the following:

1. Page fault into OS kernel
2. Kernel allocates some memory
3. Zeroes the memory
4. Modify page table
5. Resume process

8.5 Fill On Demand

Most programs do not need all their code to start running and might never use some code in its execution. Therefore we want to do something similar, we want to start a program before its code is in physical memory:

1. Set all page table entries to invalid
2. On first page reference, kernel trap
3. Kernel brings page in from disk
4. Resume execution
5. Remaining pages can be transferred in the background while the program is running

8.6 Copy On Write

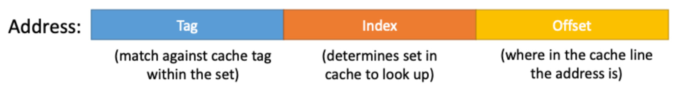
Remember that we said *fork()* copies the entire address space. This can be expensive and might not feasible in performance. On *fork()* we copy the page table and set all mappings to read-only in both address spaces. Reads are now possible for both processes. If a write in either process causes a protection fault the kernel allocates a new frame and copies the referenced frames content into it. The faulting process now maps to the new copy and the protection changes to read/write (also for the non-faulting process).

8.7 Managing Caches and the TLB

8.7.1 TLBs

A problem with TLBs on context switches is that they can hold content inaccessible to the new process. To avoid having to flush the TLB, we introduce tags. Each TLB entry has a 6-bit tag and the OS keeps track of a mapping between processes and tags.

8.7.2 Caches


Address:
Tag
Index
Offset
(match against cache tag within the set)
(determines set in cache to look up)
(where in the cache line the address is)

Remember the different types of caches:

- Virtually indexed, virtually tagged - simple and fast, but context switches are hard
- Physically indexed, physically tagged - can only be accessed after address translation
- Virtually indexed, physically tagged - overlap cache and TLB lookups
- Physically indexed, virtually tagged

Also remember the different write (write through, write back) and allocate (write allocate, non-write allocate) policies. In virtually tagged caches we can encounter homonyms, the same virtual address maps to multiple physical address spaces. To avoid this we can use physical tags, add address space identifiers, try to ensure disjoint address spaces or flush the cache on a context switch.

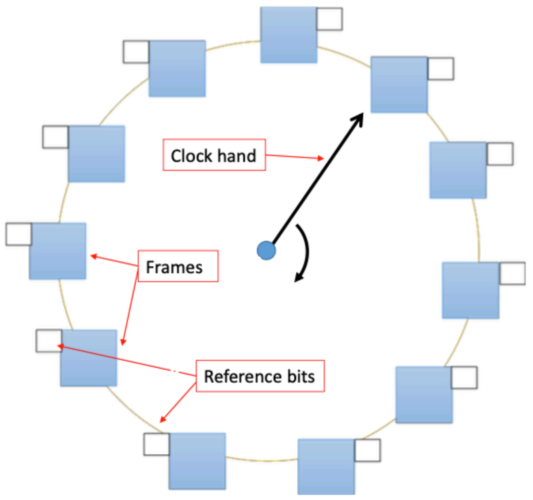
There is also a synonyms problem, where two virtual addresses map to the same physical address. This leads to inconsistent cache entries. The solutions to the homonym problem do not help here. To solve this problem we restrict VM mappings, so that synonyms map to the same cache set.

8.8 Demand Paging

Demand paging solves the problem how to find a frame to use for missing pages. The goal is to minimize the page fault rate p $(0 \leq p \leq 1)$. The metric we are interested in is the effective access time $(1-p)*l_m + p*l_f$, where l_m is the latency of a memory access and l_p the latency of a page fault handling. Generally speaking the performance of a paging system depends on how many frames it has, but there is a diminishing return on adding new frames.

8.8.1 Page Replacement Policies

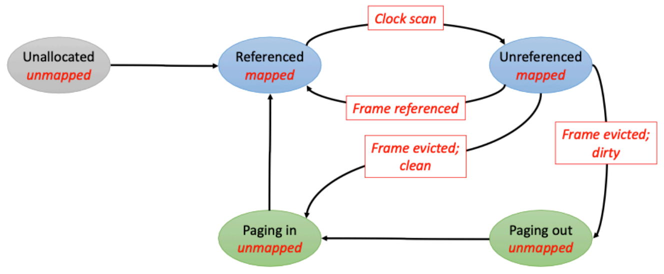
If a page has to be evicted, which one should be choose? We have already seen Least Recently Used (LRU) and First In First Out (FIFO). Both of these are not optimal, LRU is too expensive and FIFA works poorly for most workloads. A good compromise is the **Clock** or **2nd Chance** algorithm. It approximates LRU but much cheaper. It requires a linear table of all PFNs, with associated referenced bits. This can be best visualized as a circular buffer.



It works as follows:

- Mark each frame when referenced for the first time
- If we want to replace a frame, process as follows:
  - If marked, unmark and advance the clock hand
  - If unmarked, allocate this frame and mark it

8.8.2 Frame States



This only works if the corresponding bits are present in the page table and are properly set, in RISC-V for example these bits are present but do not have to be set. We can emulate these bits using faults.

9 File System

9.1 Abstractions

File systems are a very important concept and it makes sense to define an abstract file system. The main task of a file system is to virtualize, this means it should allow for multiplexing, have the same abstraction for multiple file systems implementation.

9.1.1 Access Control

We want to enforce some access control for our files. We call user or group of users a **principal** and files **objects**. One of the simplest way of representing this information would be in a matrix format. Each row would be a principal and columns correspond to objects. A entry in this matrix describes the rights the principal has in regards to the object. The problem with this idea is that such a matrix would become too large.

Another idea would be access control lists, a more compact representation. For each file you need to specify user and its access right, so you do not have to save any information on principals that do not have any rights to a file. If a new file gets created we use mandatory access control (MAC) to set the rights to a file. MAC defines the access policy centrally, so the principals cannot make policy decisions. This is easy to implement but hard to customize if you need special permissions.

A third idea would be to use capabilities, this means that there are tokens that give the holder of the token specific rights.

POSIX access control uses a hybrid approach. As principals there exist users but also groups. Access rights are divided into read, write and execute (rwx).

9.1.2 File Abstraction

Files consist of two parts. The first part is the **data**, consisting of unstructured bytes or blocks. The second part is the **metadata**, containing informations like type (structures or unstructured), time stamps, location, permissions, etc. The filename is not part of its metadata.

The filename is part of the **namespace**. Depending on the OS there are different restrictions on the filename, e.g. allowed characters. Filenames can be thought of as pointers to a file, meaning a file does not need a name or it could even have multiple names. The POSIX namespace results in a tree like structure, where each filename is preceded by the directories it is contained in. There are other location based bindings '.' meaning the current location and '..' being the parent directory. The challenge with this tree structure is to prevent cycles.

9.1.3 File Descriptors

File descriptors are used to open files and then perform operations on it. A file descriptor is basically an id that the OS give to a process to access a file (a FD comes with some metadata, e.g. type of access). Each processor has its own file descriptors. This concept is similar to capabilities.

There are different access types:

- Direct - unrestricted access to the file without offset, curser starts at zero
- Sequential - access without rights to move the curser, writing happens at the end of the file
- Structured - defined agreement on how data gets written

9.1.4 Memory-Mapped Files

Alternatively, we can open files by mapping its content to virtual memory. This allows for anonymous memory, meaning we can treat memory regions as files even though this is not the case. It also allows us to have shared memory, by loading the same file on different processes. But if we use memory-mapped files, we have to take care of synchronisation between the memory and the file system.

9.1.5 Executable Files

Executing a file creates a new process, checks if the file is valid and then uses the ELF format to load the file and start execution.

9.2 Implementations

After having specified how an interface for a file system might look like, we know want to have a look at the implementations.

First we introduce an additional abstraction, **volumes**. A volume is a generic name for a storage device, consisting of a contiguous set of fixed-size blocks. To access blocks, we need **logical block addresses** (LBA), these are numbers for blocks on a volume. Closer to the hardware, there are **disk partitions**. Partitions are physical volumes divided into contiguous regions. For this to work we need to store a partition table at the start of the physical volume.

A file system implementation consists of a set of data structures. These are stored on a volume and allow for naming and protection. These things together form the **FS API**.

This API allows us to mount several file systems on top of each other. At the top we have the root file system and all other mount points are under the root (e.g. */dev/sda1*).

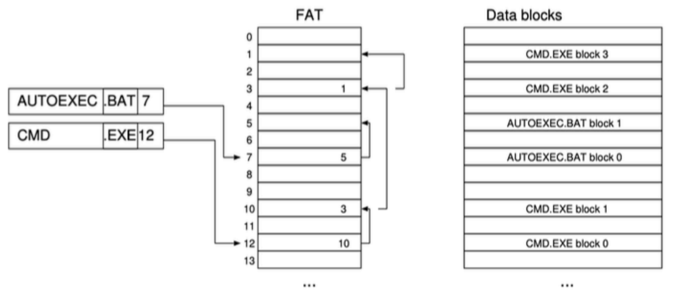
From the OS point of view, it implements a **virtual file system** layer in the kernel. This layer tries to resolve the type of file system that is being used. This allows for different file system implementations to coexist.

The main goals of concrete file system implementations depend on the device it is used on. Often they are a mixture of performance and reliability. In a typical file system implementation we will see the following things:

- Directories and Indexes - where on the disk is the data for each file?
- Index Granularity - what is the unit of allocation for files?
- Free Space Maps - how to allocate more sectors on the disk?
- Locality Optimizations - how to make it go fast in the common case?

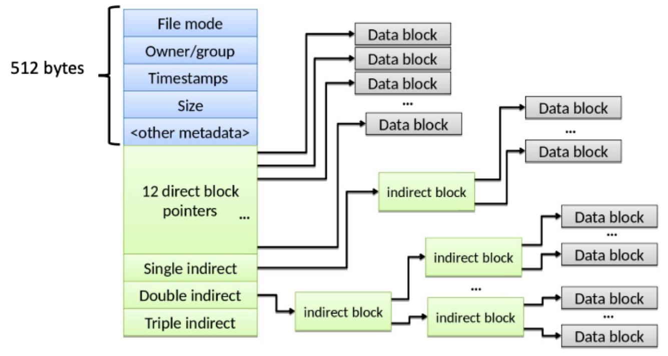
9.2.1 The FAT File System

FAT (file allocation table) is a very basic file system.



For naming, the FAT system uses the filename and the extension. It then remembers the first block of that file. The file allocation table itself works like a linked list of blocks, where at the end of each block there is a pointer to the next block. To allocate new space, we simply need to scan the table. This can lead to very poor locality (fragmentation).

9.2.2 The Berkeley Fast Filing System

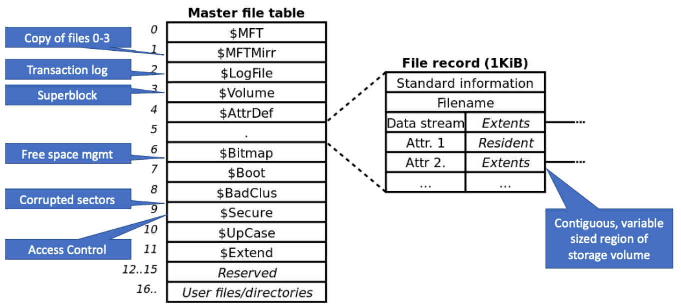


This system uses an index node (inode) for each file in the file system. The inode consists of the metadata and block pointers (or data directly if it fits). To allocate blocks it uses a bitmap indicating if a block is free or not.

Block groups are continuous subset of disk track where related inodes, directories, free space map and file data blocks are gathered together. A superblock then holds all the informations about the overall layout and where the block groups are.

9.2.3 Windows NTFS

NTFS treats everything as a file, e.g. files system and file metadata. The master file table contains file entries for each file and is itself a file with the very first entry in the table. A MFT entry consists of metadata and a list of variable length attributes. One attribute is all the names for a file. If the data is small enough, it gets stored as part of the attributes, else it is stored in an extent. If works very similar to Berkeley FFS.



In NFTS file descriptors are basically pointers to extents. Additionally there are basic file descriptors.

10 Network Stack

In this part we want to look at what happens in the OS when we do networking on a low level. The role of the OS network stack is to handle all network related I/O. This includes delivering and transmitting packets, multiplex and demultiplex packets, and processing protocols.

The following parts play a role in the networking stack:

- the NIC (networking interface card)
- first-level interrupt handlers
- NIC driver bottom half, e.g. deferred procedure calls
- NIC driver top half, kernel code
- app libraries, daemons, utils

Together they provide the following functionalities:

- Multiplexing - taking packets from the user space, deciding the protocol to use and sending them out
- Encapsulation - taking raw data from an application and encapsulate it in a packet

- Protocol State Processing - advance the state of a protocol to process packets arriving
- Buffering and Data Movement - store data until the application decides to process it

10.1 Header Space

The header space is the set of all possible packet headers. The OS needs to process these headers as fast as possible and deliver them to the applications. Part of processing the header is to decide if the header is valid.

10.2 Protocol Graphs

Some OS maintain a protocol graph. The protocol graph of a network stack is the directed-graph representation of the forwarding and multiplexing rules. Nodes in the graph represent a protocol acting on a communication channel and perform de-/encapsulation and possibly de-/multiplexing.

If a node has multiple outgoing edges, its probably demultiplexing packets (vice versa for multiplexing). Note that this graph can well be cyclic due to tunnelling.

10.3 Network I/O

If the NIC receives a packet it copies it to a OS buffer, enqueues it on a descriptor ring and returns the buffer to the OS. If the OS wants to send a packet it enqueues it on a descriptor ring, notifies the NIC and after processing the NIC returns the buffer to the OS. A simplified implementation of the first-level- interrupt handler for packet receive looks as follows:

Algorithm 1: First-level interrupt handler for receiving packets

```
  /* Inputs                                  */
  /* rxq:  the receiver description queue    */
1 Acknowledge interrupt
2 while not(rxq.empty()) do
3     buf = rxq.dequeue()
4     sk_buf = sk_buf_allocate(buf)
5     enqueue(sk_buf) for processing
6     post a DCP (software interrupt)
7 end
8 return
```

Note that this only copies the packet and does not process it. This allows the OS to free the space for the NIC to receive new packets with deferring the processing the packet to later.

10.4 Top-Half Handling

In the top half of the stack, a socket interface is used: we can call *bind(), listen(), connect(), send(), recv()* etc. from user space.

Some protocol processing happens in the kernel directly as a result of top half invocations, but for the most part the top half is concerned with copying network payload data and metadata between queues of protocol descriptors in the kernel and user-space buffers.

10.5 Polling

To increase performance, instead of the conventional interrupt-driven descriptor queues, a network receive queue can be serviced by a processor polling it contiguously. This eliminates the overhead of interrupts, context switches and maybe even kernel entry/exit. It requires a dedicated processor spinning, waiting for packets.

But even polling is insufficient to handle modern high-speed networks. Its not clear how to scale this approach to multiple cores. Thus one relies on hardware support.

10.6 Hardware Acceleration

Today many operations are accelerated using dedicated hardware, e.g. by smart NICs. This works by having multiple physical queues for each connection. The OS then can bind an application to a hardware queue when needed. These smart NICs have dedicated hardware to perform operations on these queues, e.g. calculate checksums or applying filtering rules.

RDMA devices allow for direct memory accesses between different devices. This is extremely fast and specially useful in a datacenter context. On a downside allowing a remote devices to access a devices memory can be dangerous.

10.7 Routing and Forwarding

An OS also implements the core functionality of routers (forwarding and routing). For forwarding a set of hardware tables is used. Forwarding a packet on a receive queue essentially involves reading its header, then using this information to transfer the packet to one or more transmit queues to be sent. The forwarding tables are a result of the routing calculations, which mostly happen in user space.

11 Virtual Machines

A **virtual machine monitor** (VMM) virtualizes an entire system. The execution environment of the VMs well look at provide a simulation of the raw machine hardware.

While a VMM is the functionality required to create the illusions of real hardware, the **hypervisor** is the software that runs on real, physical hardware and supports multiple VMs (each with its associated virtual machine monitor). There is one hypervisor on which many VMMs can run. We call a hypervisor running on bare metal a type-1 (native) hypervisor and one running on a real OS a type-2 (hosted) hypervisor.

OS-level virtualization uses a single OS to provide the illusion of multiple **containers** of that OS. Code running in a container have the same system call interface as the underlying OS, but cannot access any device. This is achieved by limiting the file system namespace (by changing the root for each container) and the process namespace, so processes can only see processes which share their container. In general, using containers is more efficient than using hypervisors.

11.1 The Uses of Virtual Machines

When multiple applications contend for resources the performance of one or more may degrade in ways outside the control of the OS. **Resource isolation** guarantees to one application that its performance will not be impacted by others, this is done by running the application in a VM.

Cloud computing is the business of renting computing resources as a utility to paying customers rather than selling hardware. They are primarily based on renting computing resources in the form of a VM (similar to resource isolation).

The term **server consolidation** refers to taking a set of services, each running on a dedicated server, and consolidating them onto a single physical machine so that each runs in a VM.

Backward compatibility is the ability of a new machine to run programs (including OSes) written for an old machine.

11.2 Virtualizing the CPU

To run an OS inside a VM, we need to completely virtualize the processor, including the kernel (else we simply could use threads). The processor in the VM clearly cannot execute a privileged instruction "for real". Instead, the default result is a trap/fault:

Trap-and-emulate is a technique for virtualization which runs privileged code in non-privileged mode. Any privileged instruction causes a trap to the VMM, which then emulates the instruction and returns to the VM guest code.

The problem is that there might be some instructions which do not cause a trap when run in non-privileged mode but have a different behavior when executed in kernel mode (e.g. *POPF* in x86). This cannot happen with a strictly virtualizable ISA.

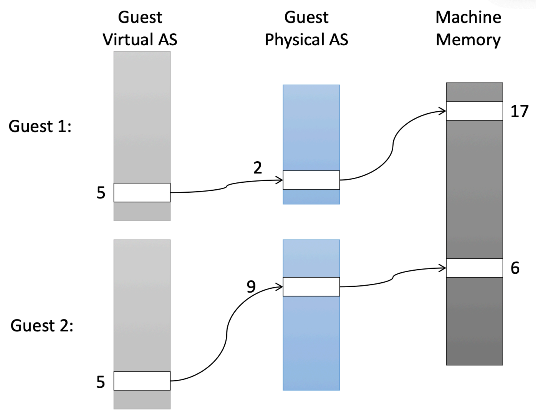
An ISA is **strictly virtualizable** iff it can be perfectly emulated over itself with all non-privileged instructions executed naively and all privileged instructions emulated via traps.

There are different approaches to dealing with non strictly virtualizable ISA:

- **Full software emulation**: Creates a virtual machine by interpreting all kernel-mode code in software. This is very slow, especially for many I/O operations.
- **Paravirtualization**: A paravirtualized guest OS is one which has been specifically modified to run inside a VM. Critical calls are replaced with explicit trap instructions.
- **Binary rewriting**: Scans compiled kernel code for unvirtualizable instructions and rewrites them essentially patching the kernel on the fly. This is done on demand: All kernel pages are first protected and when first accessed (i.e., the pro- tection trap occurs), they get scanned and rewritten.
- **Virtualization extensions**: Convert ISA by adding virtualization extensions. This typically takes the form of a new processor mode. Today, both ARM and x86 do have hardware support for virtualization.

11.3 Viratualizing the MMU

With virutalization, there is a second level of indirection with memory addresses. Now, a physical memory address is not unique in the machine, but in one VM (guest OS thinks it is physical). Thus, we define the **machine address** to be a real address on the machine which gets translated from the guest OSs physical address. From the view of the hypervisor, the machine address is the physical address.



The hypervisor thus needs to translate a guest virtual address not to a guest physical address but to a machine address instead. There are several ways to do so:

- Directly writable tables: The guest OS creates the page tables that the hardware uses to directly translate guest virtual to machine addresses. This requires paravirtualization. The VMM needs to check all writes to any PTE in the system. To change a PTBR, a hypercall is needed.
- A shadow page table is a page table maintained by the hypervisor which contains the result of translating virtual addresses through the guest OSs page tables, and then the VMMs physical-to-machine page table. The guest OS thus sets up its own PT, but they get never used. The VMM maintains the shadow PT which maps directly from guest VAs to machine addresses.

  The VMM must keep the shadow table consistent with both the guests PT and the hypervisors own physical-to-machine table. It achieves this by write-protecting all the guest OS's PT and trapping writes on them. When a write happens, it applies the update to the shadow page as well.
- Nested Paging/2nd level page translation is an enhancement to the MMU hardware that allows it to translate through two page tables (guest virtual to guest physical and guest physical to machine), caching the result (virtual to machine) in the TLB. This can be fast, but a TLB miss is costly.

11.4 Viratualizing the Physical Memory

How can the hypervisor allocate memory to a guest OS? The guest OS expects a fixed area of physical memory which does not change dynamically. In theory, this problem can be solved with paging. However, there is a phenomenon called **double paging**. Consider the following sequence of events:

1. The hypervisor pages out a guest page P to storage
2. A guest OS decides to page out the virtual page associated with P and touches it.
3. This triggers a page fault in the hypervisor, hence P gets paged back in memory.
4. The page is immediately written out to disk and discarded by the guest OS.

So to throw away a page in a guest OS, there are three I/O operations and one extra page fault! We could solve this problem with paravirtualization, but this introduces more complexity.

Memory ballooning is an elegant solution to this problem. It allows hypervisors to reallocate machine memory between VMs without incurring the overhead of double paging. A device driver, the balloon driver, is installed in the guest kernel. This driver is VM-aware, i.e. it can make hypercalls and receive messages from the VMM. The principle is to block a large area of physical memory in the guest OS, which then can be allocated to the OS by unblocking it.

Memory can also be reclaimed from a guest OS (inflating the balloon):

1. The VMM asks the balloon driver to return n physical pages from the guest OS to the hypervisor.
2. The balloon driver notifies the OS to allocate n pages of memory for its private use.
3. It communicates the guest-physical addresses of these frames to the VMM using a hypercall.
4. The VMM unmaps these pages from the guest OS kernel and reallocates them elsewhere.

Reallocating machine memory to the VM (deflating the balloon) can be done similarly:

1. The VMM maps the newly allocate machine pages into guest-physical pages inside the balloon.
2. The VMM then notifies the balloon driver that these pages are now returned.
3. The balloon driver returns these guest-physical pages to the rest of the guest OS.

11.5 Viratualizing the Devices

To software, a device is something that the kernel communicates using memory mapped I/O registers, interrupts from the device to the CPU and DMA access by the device to/from main memory. The hypervisor needs to virtualize all of this, too.

A device model is a software model of a device that can be used to emulate a hardware device, using trap-and-emulate to catch CPU writes to device registers. Interrupts from the emulated device are simulated using upcalls from the hypervisor into the guest OS kernel at its interrupt vector.

A **paravirtualized device** is a hardware device design which only exists as an emulated device. The driver of the device in the guest OS is aware that it is running in a VM and can communicate efficiently with the hypervisor using shared memory buffers and hypercalls.

For the device drivers talking to the real devices, we have the option to put them in the hypervisor kernel. Alternatively, one could use device passthrough, mapping a real hardware device into the physical address space of a guest OS. However, this does not solve the problem of sharing a real device among multiple virtualized guest OSes.

A **driver domain** is a virtual machine whose purpose it is to provide drivers for devices using device passthrough. With this, we can share devices across multiple VMs, by exporting a different to these devices using inter-VM communication channels. They are great for compatibility, but can be very slow, due to the communication overhead.

A **self-virtualizing** device is a hardware device which is designed to be shared be- tween different VMs by having different parts of the device mapped into each VMs physical address space. **SR-IOV** is one form of this.

Single-Root I/O Virtualization (SR-IOV) is an extension to the PCIe standard which is designed to give VMs fast, direct but safe access to real hardware. An SR-IOV capable device appears initially as a single PCI device. This device can be configured to make further virtual functions appear in the PCI device space: each of this is a restricted version, but otherwise looks like a completely different, new device.

11.6 Viratualizing the Network

A soft switch is a network switch implementation inside a hypervisor which switches network packets sent from paravirtualized network interfaces in VMs to other VMs and/or one or more physical network interfaces.

The soft switch can be quite powerful but it needs to be fast. We can address a network interface inside a VM by giving each virtual network interface a MAC address on its own and letting DHCP do the rest.

Distributed Systems

Today almost all computer systems are distributed, for different reasons:

- Geography
- Parallelism - speed up computation
- Reliability - prevent data loss
- Availability - allow for access at any time, without bottlenecks, minimizing latency

Even though distributed systems have many benefits, such as increased storage or computational power, they also introduce challenging coordination problems.

12 Fault Tolerance and Paxos

In this section we want to create a fault-tolerant distributed system. We start out with a simple approach and improve our solution until we arrive at a system that works even under adverse circumstance, Paxos.

A **node** is a single actor in the system. In the message passing model we study distributed systems that consist of a set of nodes, where each node can perform local computations and send messages to every other node. Message loss means that there is no guarantee that a message will arrive safely at the receiver. This leads us to the first algorithm

Algorithm 2: Naive Client-Server Algorithm

```
1 Client sends commands one at a time to server
2 Server acknowledges every command
3 If the client does not receive an acknowledgment
    within a reasonable time, it resends the command
```

This simple algorithm is the basis of many reliable protocols, e.g. TCP. The algorithm can easily be extended to work with multiple servers: The client sends each command to every server, and once the client received an acknowledgment from each server, the command is considered to be executed successfully.

In practice, messages might experience different transmission times, even if they are being sent between the same two nodes. A set of nodes achieves **state replication**, if all nodes execute a sequence of commands in the same order. Since state replication is trivial with a single server, we can desig- nate a single server as a serializer.

Algorithm 3: State Replication with a Serializer

```
1 Client sends commands one at a time to the serializer
2 Serializer forwards commands one at a time to all other servers
3 Once the serializer received all acknowledgments, it notifies the client about the success
```

The downside of this algorithm is that the serializer is a single point of failure.

12.1 Two-Phase Protocol

Algorithm 4: Two-Phase Protocol

```
  /* Phase 1                                              */
1 Client asks all servers for the lock
  /* Phase 2                                              */
2 if client receives lock from every server then
3     Client sends command reliably to each server and gives the lock back
4 else
5     Clients gives the received locks back
6     Client waits, and then starts with Phase 1 again
7 end
```

Instead of directly establishing a consistent order of commands, we can use a different approach: We make sure that there is always at most one client sending a command; i.e., we use mutual exclusion, respectively locking.

Still there are quite some problems with this algorithm. What happens if the node holding the locks crashes or it only gets part of the locks?

12.2 Paxos

A **ticket** is a weaker form of a lock, with the following properties:

- Reissuable: A server can issue a ticket, even if previously issued tickets have not yet been returned.
- Ticket expiration: If a client sends a message to a server using a previously acquired ticket t, the server will only accept t, if it is the most recently issued ticket.

There is no more problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected. (At this point the naive ticket protocol is left out)

Algorithm 5: Paxos Client / Proposer

```
   /* Initialization                                 */
1  $c$                             /* command to execute */
2  $t = 0$                         /* ticket number to try */
   /* Phase 1                                        */
3  $t = t + 1$
4  Ask all servers for ticket $t$
   /* Phase 2                                        */
5  if a majority answers ok then
6      Pick($T_{store}, C$) with largest $T_{store}$ if $T_{store} > 0$ then
7          $c = C$
8      end
9      Send propose($t, c$) to same majority
10 end
   /* Phase 3                                        */
11 if a majority answers success then
12     Send execute($c$) to every server
13 end
```

Algorithm 6: Paxos Server / Acceptor

```
   /* Initialization                                 */
1  $T_{max} = 0$             /* largest issued ticket */
2  $C = \bot$                /* stored command */
3  $T_{store} = 0$           /* ticket used to store $C$ */
   /* Phase 1                                        */
4  if $t > T_{max}$ then
5      $T_{max} = t$
6      Answer with ok($T_{max}, C$)
7  end
   /* Phase 2                                        */
8  if $t = T_{max}$ then
9      $C = c$
10     $T_{store} = t$
11     Answer success
12 end
```

Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. This has the advantage that we do not need to be careful about selecting good values for timeouts, as correctness is independent of the decisions when to start new attempts. The performance can be improved by letting the servers send negative replies in Phase 1 or 2 if the ticket expired. Using randomized backoff we can eliminate contention between clients.

Theorem: If a command c is executed by some servers, all servers (eventually) execute c.

Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.

For state replication we need to be able to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once the 1st command is chosen, any client can decide to start a new instance and compete for the 2nd command. If a server did not realize that the 1st instance already came to a decision, the server can ask other servers about the decisions to catch up.