Computer Systems

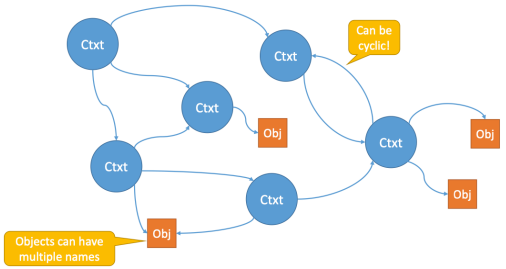
by dcamenisch

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.

2 Naming

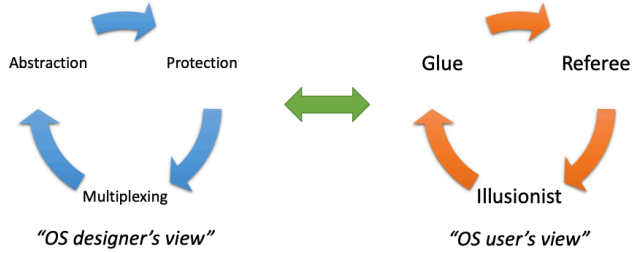
Naming is a fundamental concept, it allows resources to be bound at different times. Names are bound to objects, this is always relative to a context. One example of this would be path names, e.g. */usr/bin/emacs*. Name resolution can be seen as a function from context and name to some object. The resolved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

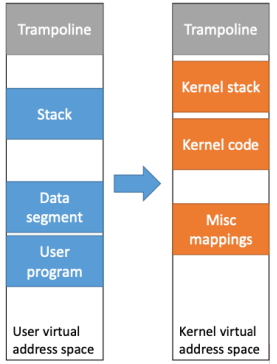
One of our main goals is to protect the OS from applications that could harm it (intentionally or not). For this purpose we introduce two different modes:

- 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

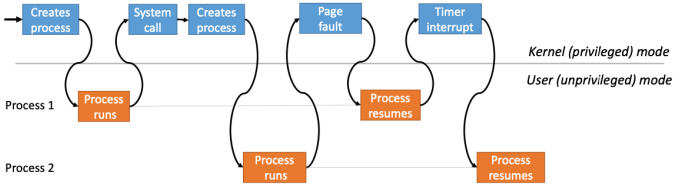
Further we may also want to save the registers and switch page tables. When switching between the modes we also have to change our address space, but we might want to access some informations from the user mode address space. One way of doing this is to use a so called **trampoline**, which is a part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

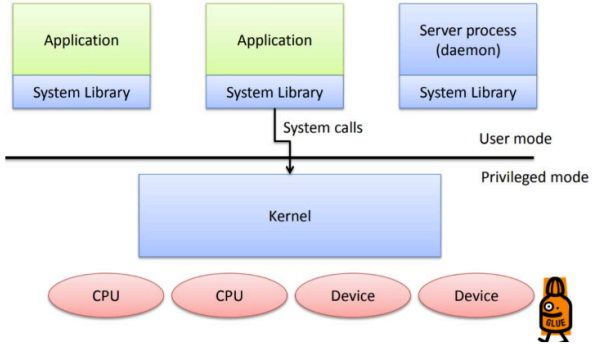
This leads us to the following perspective:



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

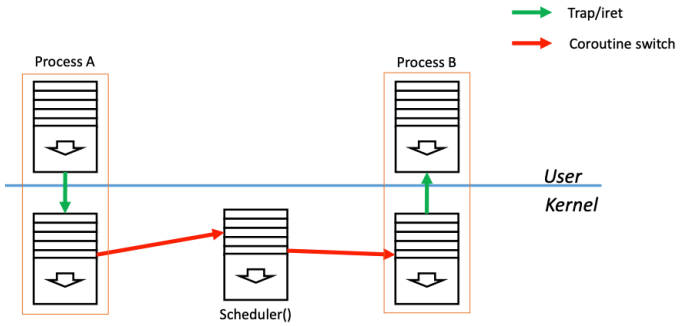
In contrast to *spawn()*, here the child revokes rights and access explicitly before *exec()*, further we can use the full kernel API to customize the execution environment.

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.

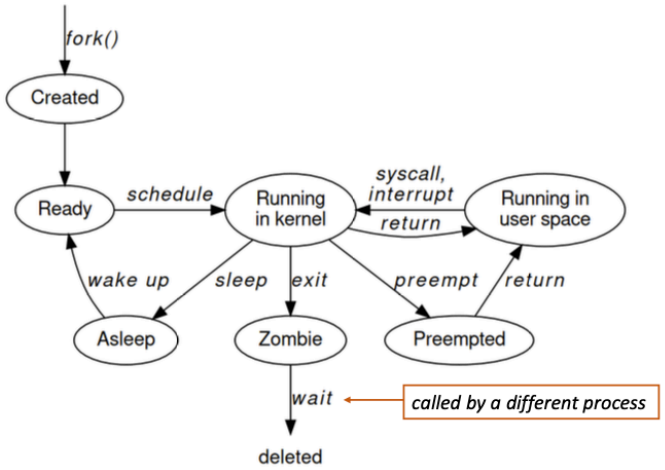
4.3 Process Context Switching

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

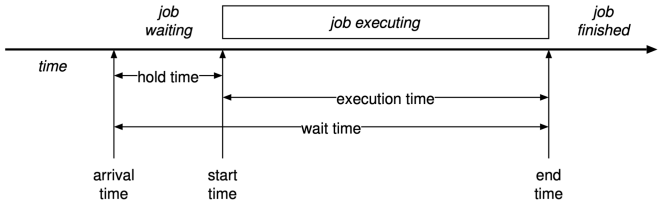
We introduce shared memory that can only be accessed by the processes that communicate with each other. This requires us to define a common interface between the processes. The main building blocks for this are:

- 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.

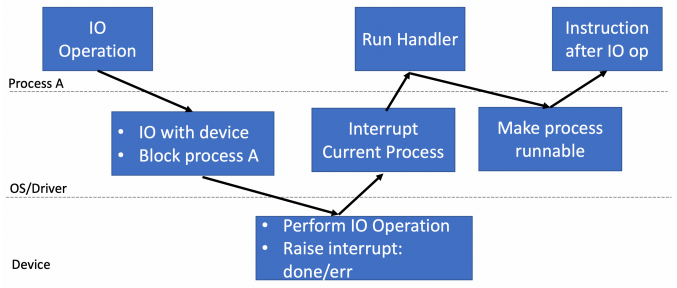
To an OS programmer, a **device** is a piece of hardware visible from software. It typically occupies some location on a bus or I/O interconnect, and exposes a set of hardware registers which are either memory mapped or in I/O space. A device is also usually a source of interrupts, and may initiate Direct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

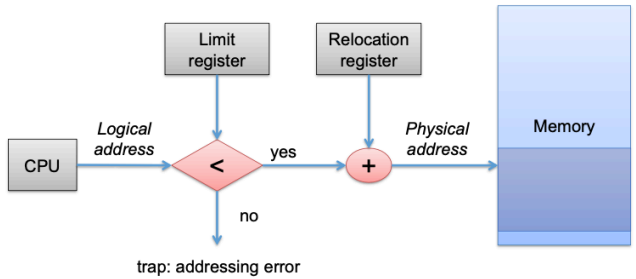
8 Virtual Memory

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

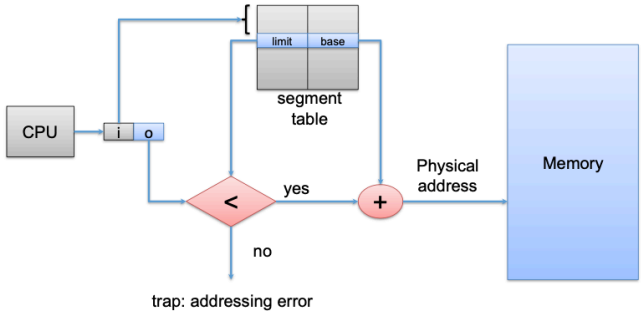
- Process Isolation
- Shared Code Segments
- Dynamic Memory Allocation
- Persistent Data Structures
- etc.

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).



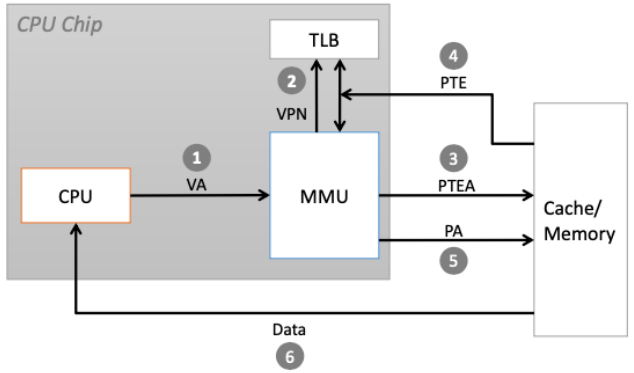
A segment is a triple (I, B_I, L_I) of values specifying a contiguous region of memory address space with base B_I, limit L_I, and an associated segment identifier I which names the segment. Memory in a segmented system uses a form of logical addressing: each address is a pair (I, O) of segment identifier 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.



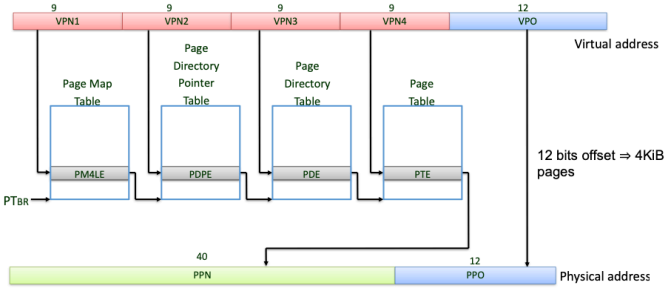
This enables sharing code/data segments between processes, further it adds protection and allows transparently growing stack/heap as needed. The principal downside of segmentation is that segments are still contiguous in physical memory, which leads to external fragmentation.

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):



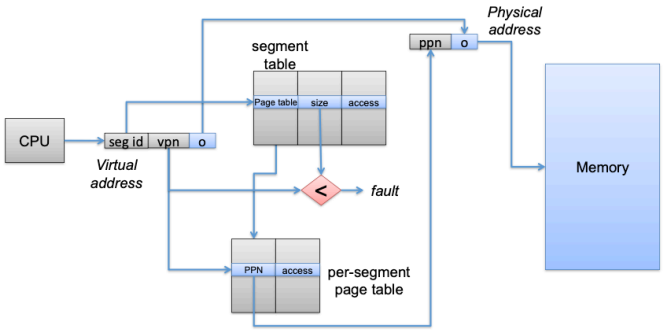
We have also seen how multi-level page tables work. Multi-level translation allows us to allocate only page table entries that are in use and makes memory allocation simpler.



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.

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.

9 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 1: 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 2: 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.

9.1 Two-Phase Protocol

Algorithm 3: 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?

9.2 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 4: 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 5: Paxos Server / Acceptor

```
   /* Initialization                          */
1  T_max = 0        /* largest issued ticket */
2  C = ⊥                  /* 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.