Otázky k bakalářským státnicím FIT VUTBR 2013/2014 v 1.0

autor: Miloslav Číž

Otázky jsou každá na jednu oboustrannou A4 stranu. Neručím za stoprocentní správnost, pokud se vám něco nezdá, raději si to ověřte. Pokud uděláte úpravy, klidně se sem podepište, ale prosím nechte mě uvedeného jako původního autora.

1. Princip činnosti polovodičových prvků (dioda, bipolární a unipolární tranzistor ve spínacím režimu, realizace logických členů NAND a NOR v technologii CMOS). ITO, IPR, INC

Základem polovodičů jsou prvky IV. skupiny periodické tabulky prvků, nejčastěji křemík. Tyto prvky mají ve valenční vrstvě 4 elektrony, kterými mohou vytvářet vazby. Upravením krystalické mřížky atomů (přidáním příměsi) můžeme dostat 2 typy polovodičových materiálů:

- **P** (pozitivní) – Proud vedou díry (místa bez elektronů). Polovodič vznikne přidáním třímocného prvku (např. boru), takže vznikne tzv. **díra**, která se za normální teploty může volně pohybovat (z atomu příměsi se stane nepohyblivý záporný iont, ale celý prvek zůstane elektricky neutrální).
- **N** (negativní) – Proud vedou volné elektrony. Polovodič vznikne přidáním pětimocného prvku (např. fosforu), čímž vznikne přebytečný elektron, který se může za normální teploty volně pohybovat (z atomu příměsi se stane nepohyblivý kladný iont, ale celý prvek zůstane elektricky neutrální).

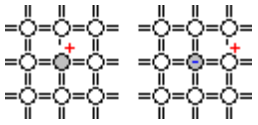


Figure 1: polovodič typu P

Polovodiče typu P a N jsou ale jako celek elektricky neutrální (mají stejný počet elektronů jako protonů). Spojením polovodičů typu P a N vzniká tzv. **PN přechod**. Při spojení dochází k tzv. **difúzi** – některé volné elektrony se přesunou do polovodiče typu P a některé díry do polovodiče typu N, čímž vzniká tzv. **potenciálová bariéra** v místě dotyku, která brání (tzv. difúzním napětím) proudění dalších nosičů náboje. Pomocí PN přechodu můžeme realizovat různé elektronické součástky, jako jsou např.:

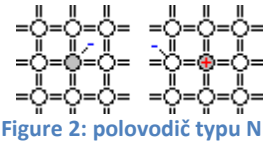


Figure 2: polovodič typu N

- **dioda** – Je tvořena jedním PN přechodem a dokáže propouštět proud jen v jednom, tzv. **propustném**, směru (v opačném směru propouští jen velmi malý, tzv. **saturační** proud).

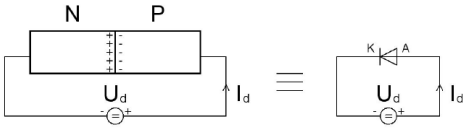
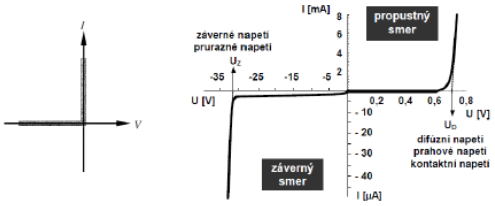


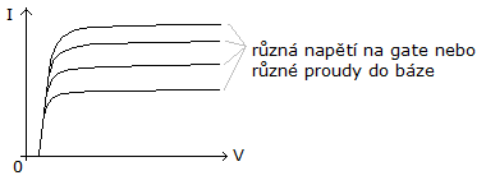
Figure 3: potenciálová bariéra

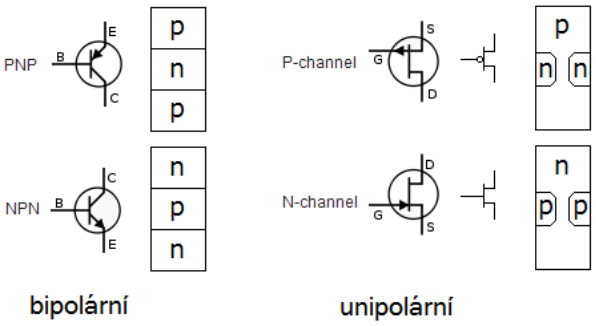
Opačné zapojení (zdroj je pólován stejně jako bariéra) podporuje potenciálovou bariéru a brání elektrickému proudu. Výstupy diody se nazývají **anoda** a **katoda**. Dioda může mít i jinou funkci, např. v závěrném směru může emitovat světlo (tzv. LED dioda).

Obrázek 1: ideální vs reálná voltampérová (přechodová) chaarakteristika diody



- **tranzistor** – Je tvořen dvěma přechody PN. Funguje jako spínač sepínaný elektronicky a z většiny tak nahradil starší relé. Tranzistory dělíme na:
  - **bipolární** – Dříve vyráběné, jsou řízeny proudem tekoucím do báze (vývody označujeme **kolektor**, **báze** a **emitor**). Označují se jako **BJT** (bipolar junction transistor). Dělíme je na PNP a NPN.
  - **unipolární** – Novější, jsou řízeny napětím na gate (vývody označujeme **drain, gate** a **source**). Označují se jako **FET** (field electric transistor). Existují různé druhy, např. MOSFET, JFET apod. Dělíme je na N kanálové a P kanálové.



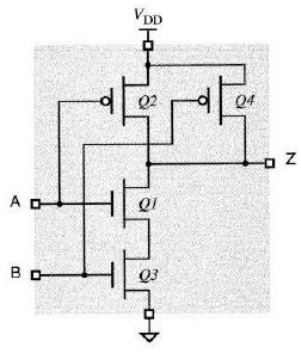


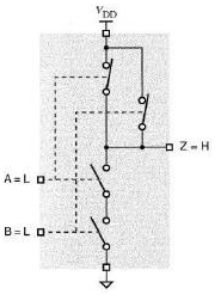
Obrázek 2: V/A charakteristika tranzistoru

CMOS (complementary metal–oxide–semiconductor) je technologie výroby elektronických obvodů, dnes většinou využívaná. Mezi její výhody patří odolnost proti šumu a malá spotřeba ve statickém stavu (energie se spotřebovává z většiny na změnu stavu), nevýhodou je vznik parazitních kapacit a zahřívání. Spotřeba závisí na frekvenci přepínání. Název je odvozen od faktu, že se využívají komplementární FET tranzistory s kanály:

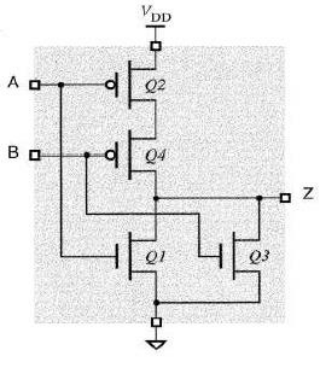
- N – Napětím na gate otevřeme tranzistor (odstraníme odpor mezi source a drain).
- P – Napětím na gate zavřeme tranzistor (vytvoříme velký odpor mezi source a drain).

Některé logické členy v technologii CMOS vypadají následovně:

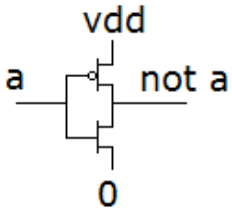




Obrázek 5: NAND



Obrázek 4: NOR



Obrázek 3: NOT

Shottkyho dioda je dioda s malým úbytkem napští a rychlým přepínáním. Zenerova dioda je dioda, u níž není průraz destruktivní (používá se např. pro usměrnění napětí).

Bonus:

Ohmův zákon: $I = \frac{U}{R}$

Kirchhoffovy zákony:

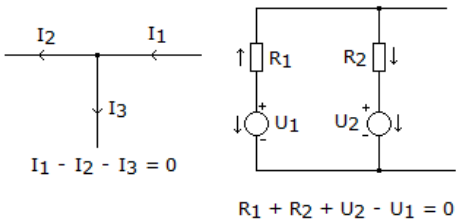
I. Součet proudů v uzlu je nulový (příchozí jsou kladné, odchozí záporné).

II. Ve smyčce se součet úbytků napětí na spotřebičích rovná součtu napětí zdrojů.

Proud ve smyčce je konstantní, napětí ne. Napětí při paralelním zapojení je ve všech větvích stejné, proud ne.

Rezistory se sériově sčítají, paralelně se sečtou převrácené hodnoty a z výsledku se udělá převrácená hodnota. U kondenzátoru je to naopak.

Impedance je obdoba odporu v obvodech se střídavým napětím. Je to komplexní číslo, které má jednak velikost (velikost odporu) a úhel (zpoždění fáze).

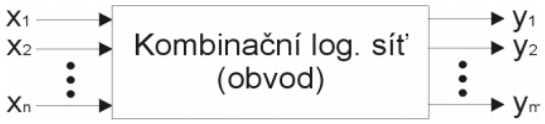


Obrázek 6: Kirchhoffovy zákony

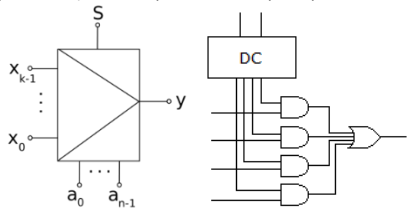
2. Kombinační logické obvody (multiplexor, demultiplexor, kodér, dekodér, binární sčítačka).

INC

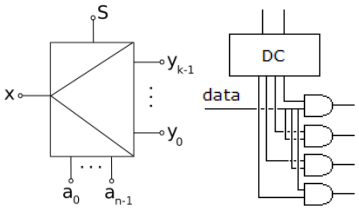
Kombinační logické obvody se liší od sekvenčních tím, že jejich výstup je dán pouze momentálními vstupními hodnotami, nikoliv vnitřním stavem (nemají tedy vnitřní paměť). Patří mezi ně např.:



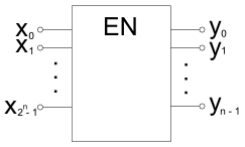
- **multiplexor** (angl. multiplexer) – Přepíná na výstup jeden z několika vstupů na základě dodané adresy. Kromě přepínání vstupů jej lze použít např. pro realizaci libovolné logické funkce (podle pravdivostní tabulky zapojíme na vstupy 0 a 1). Jeho typické využití je pro realizaci **serializátoru** (obvod převádějící paralelní data na sériové, využívá se např. v discích nebo síťových přenosech) – na vstup dáme data a postupně inkrementujeme adresu.



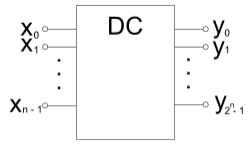
- **demultiplexor** (angl. demultiplexer) – Opak multiplexoru, přepíná vstup na jeden z několika výstupů podle dodané adresy.

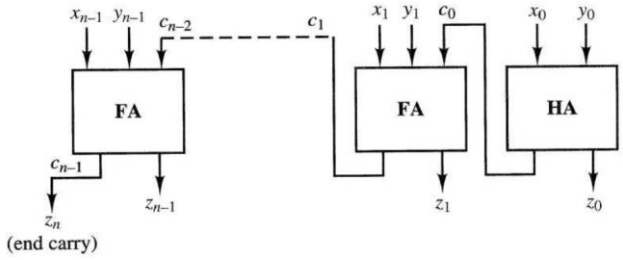


- **kodér** (angl. encoder) – Obecně převádí sekvenci bitů v jednom kódu do jiného kódu, jako klopný obvod většinou z kódu 1 z 2^n do binárního kódu.



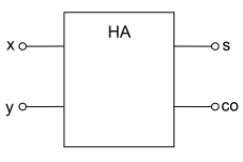
- **dekodér** (angl. decoder) – Opak kodéru, převádí binární kód na kód 1 z 2^n.





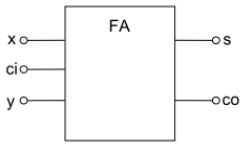
Obrázek 7: vícebitová sčítačka (ripple carry)

- **binární sčítačka** (angl. binary adder) – Obvod pro sčítání dvou binárních číslic, z více sčítaček lze jejich zřetězením sestavit sčítačku vícebitovou. Existují dva druhy binární sčítačky:
  - **poloviční** (**half adder**) – Nemá vstup pro přenos (carry), používá se jako první sčítačka ve vícebitových sčítačkách.

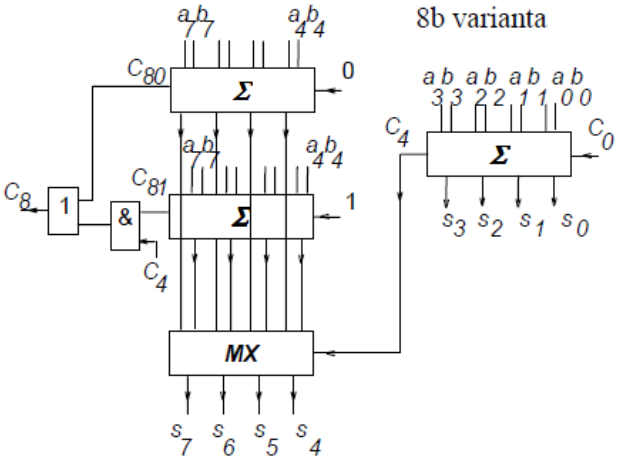


x	y	s	co
1	1	0	1
0	1	1	0
1	0	1	0
0	0	0	0

  - **úplná** (**full adder**) – Má navíc vstup pro přenos z nižšího řádu.



x	y	ci	s	co
1	1	1	1	1
0	1	1	0	1
1	0	1	0	1
0	0	1	1	0
1	1	0	0	1
0	1	0	1	0
1	0	0	1	0
0	0	0	0	0



Obrázek 8: sčítačka s výběrem přenosu

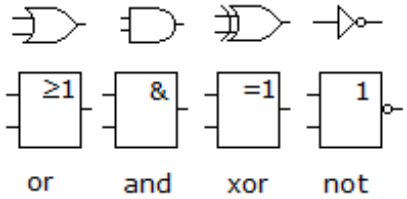
Vícebitovou sčítačku lze realizovat několika způsoby. Klasickým způsobem je kaskádově uspořádat úplné binární sčítačky za sebe (první může být jen poloviční) - doba výpočtu je potom úměrná počtu bitů sčítačky (počtu sčítaček) kvůli nutnosti propagace přenosu. Tato sčítačka se nazývá **ripple carry**.

Pro rychlejší výpočet se používají např. **carry-lookahead** (**CLA**) sčítačky. Ty umožňují rychlý výpočet přenosu díky nově zavedeným pomocným výszupům propagate carry a generate carry. Tato sčítačka se potom skládá z bloků, které nazýváme rozšířená sčítačka. Toto řešení sčítačky je nejrychlejší možné, má konstantní zpoždění, avšak složitost obvodu roste kvadraticky se šířkou sčítačky (pro 32 bitů již nerealizovatelné).

Kompromisem mezi ripple carry a CLA je **sčítačka s výběrem přenosu**. Ta funguje tak, že počítá horní a dolní část výsledku paralelně, přičemž pro horní část máme dvě sčítačky – jedna počítá pro vstupní přes roven 1 a druhá pro 0. Jakmile se vypočítá spodní část výsledku, multiplexor vybere podle přenosu odpovídající horní část výsledku.

Důležitými parametry komb. obvodů jsou cena, příkon a zpoždění.

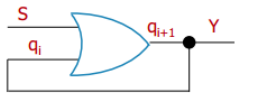
Základní log. hradla jsou:

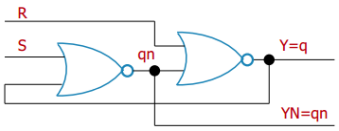


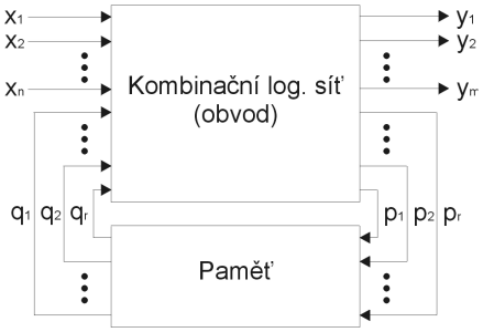
3. Sekvenční logické obvody (klopné obvody, čítače, registry, stavové automaty – reprezentace a implementace). INC

Sekvenční logické obvody jsou obvody, jejichž výstup záleží na aktuálním stavu vstupů a vnitřním stavu obvodu, tzn. mají oproti kombinačním obvodům paměť. Základními sekvenčními obvody jsou klopné obvody:

- **set** – Nejjednodušší klopný obvod, zavedením zpětné vazby u kombinačního hradla (AND nebo OR) dosáhneme možnosti sepnout jedním vstupem obvod tak, že jeho výstup zůstane sepnutý až do resetu obvodu.
- **R-S** (reset-set) – Máme vstupy set (S), kterým nastavíme výstup do jedné úrovně a reset (R), jímž obvod přepneme do úrovně opačné. Při vstupech R = S = 1 je výstup nedefinován.
- **J-K** (Jack-Kilby) – Jedná se o obvod R-S s odpovídajícími vstupy (R = J). Obsahuje vylepšení v tom smyslu, že odstraňuje nedefinovaný stav, R = S = 1 způsobí při každém taktu negaci výstupních hodnot (oproti RS se vyrábí pouze synchronní).
- **T** – Obvod se pouze překlápí s každým taktem hodin.
- **D** – Jde o jednobitovou paměť, v každém taktu si obvod zapamatuje hodnotu na vstupu T a tu potom dává na výstup.







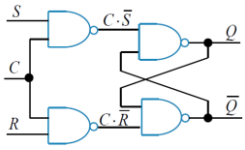
Obrázek 9: schéma sekvenčního obvodu

Klopné obvody rozeznáváme:

- **hladinové** (**latch**) – Nejjednodušší realizace, obvod reaguje na hladinu signálu na vstupu, nemá žádné vstupy navíc.
- **s povolovacím vstupem** (**gated**) – Obvod má navíc povolovací vstup C. Pokud je tento vstup aktivní, obvod reaguje na změnu vstupů, jinak nikoli.
- **dvoufázové** (**master-slave**) – Obvod reaguje na nástupnou hranu. Název master-slave pochází z faktu, že se obvod skládá ze dvou obvodů, první zachycuje hodnotu při nízké úrovni hodin a propaguje ji do výstupního obvodu při změně na vysokou úroveň hodin.
- **derivační** (**edge-triggered**) – Obvod reaguje pouze na hranu hodin (jakoukoli), tzn. při změně hodinového signálu se přepočítá výstup. Toho se dosahuje úmyslným generováním logického **hazardu** (nestálost výstupních hodnot z důvodu dobíhajících přechodových jevů) pomocí nestejné délky logických větví (v jedné je invertor, který vytvoří zpoždění, v druhé nikoliv, hrana tak generuje velmi krátký puls, který je přiváděn jako clock enable vstup do klopného obvodu).

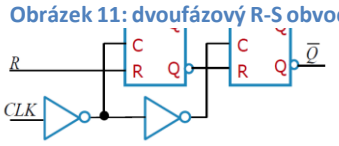
Dále pak:

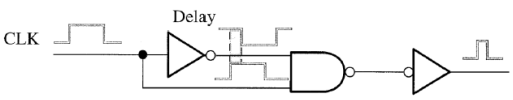
- **synchronní** – Činnost je synchronizována hodinovými signály.
- **asynchronní** – Činnost probíhá bez hodinových signálů.



Obrázek 12: obvod R-S s povolovacím vstupem

Obrázek 11: dvoufázový R-S obvod



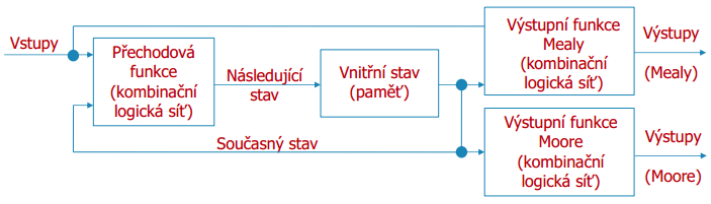


Obrázek 10: realizace detekce hrany

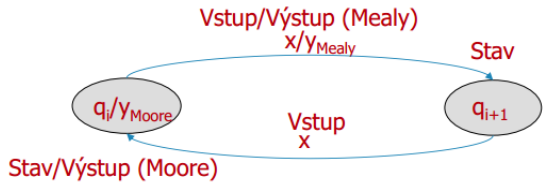
Konečný automat (KA) je šestice $KA = (X, Y, Q, q_0, P, V)$, kde X je vstupní abeceda, Y je výstupní abeceda, Q je množina stavů, $q_0 \in Q$ je počáteční stav, P je přechodová funkce $P: (X \times Q \to Q), q_{i+1} = P(x_i, q_i)$ a V je výstupní funkce. Podle typu výstupní funkce rozdělujeme konečné automaty na dva druhy:

- **Mealyho** – Výstupní funkce je funkcí stavu i vstupu.
- **Mooreův** – Výstupní funkce je funkcí pouze stavu.

KA jsou vhodné pro návrh velkého množství obvodů, kdy máme relativně málo stavů, vstupů a výstupů. KA většinou znázorňujeme grafem zobrazujícím jeho stavy a přechody.

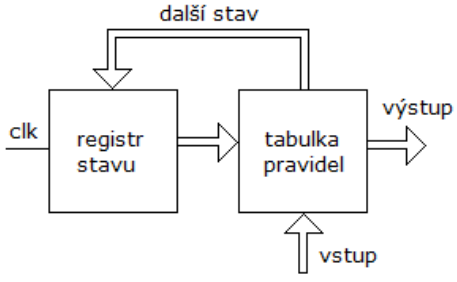


Obrázek 14: schéma konečného automatu



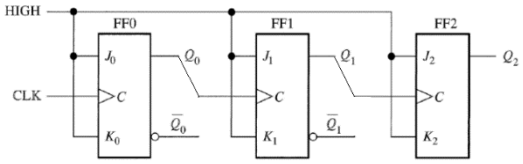
Obrázek 13: znázornění konečného automatu

Konečný automat realizujeme proměnnou s aktuálním stavem (registrem) a tabulkou pravidel:

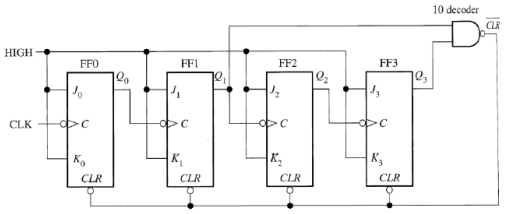


Čítač je obvod, který generuje posloupnost v určitém kódu. Příklady čítačů mohou být:

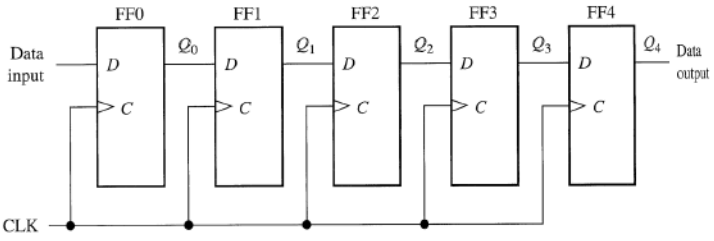
- tříbitový asynchronní binární čítač – Realizován pomocí tří J-K klopných obvodů zapojených jako T obvody. Generuje posloupnost: 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, …



- asynchronní čítač modulo 10 – Pomocí dekodéru detekujeme hodnotu 10 a vynulujeme obvod.
- synchronní čítač – Čítá s každým taktem hodin, může mít vstup ovlivňující např. směr čítání (nahoru/dolů), povolení čítání. Jako výstup má detekci přetečení.



Dalšími sekvenčními obvody jsou **posuvné registry**, které uchovávají a posouvají data. Jsou sestaveny z klopných obvodů a mohou být různě konstruované (sériové, paralelní, ...).



Obrázek 15: posuvný registr

Univerzální posuvný registr je posuvný registr, který umožňuje nastavit směr posunu.

4. Hierarchie paměti v počítači (typy a principy pamětí, princip lokality, organizace rychlé vyrovnávací paměti). INP

Hierarchie pamětí vzniká z důvodu, že rychlé paměti mají vyšší cenu za jednotku kapacity a naopak a jelikož potřebujeme jak rychlé paměti pro potřeby výpočtů procesoru, tak velkokapacitní paměti pro ukládání dat, máme v počítači různé typy pamětí. Typicky jsou to **vnitřní paměť procesoru**, **hlavní paměť** (včetně vyrovnávací paměti) a **vnější paměť**.
Paměti lze popsat různými parametry, jako jsou např.:

- kapacita – kolik jednotek paměti je možné uložit
- přístupová doba – doba od zahájení čtení po získání obsahu paměťového místa
- doba cyklu – doba od zahájení čtení/zápisu do skončení této operace
- přenosová rychlost – počet datových jednotek přenášených do nebo z paměti za jednotku času
- cena za bit
- chybovost
- …

Dále můžeme paměti dělit podle fyzikálního principu na:

- polovodičové – bipolární/unipolární
- magnetické – disketové, diskové, páskové apod.
- optické – CD, DVD apod.
- magnetooptické
- jiné – např. molekulární apod.

Podle způsobu přístupu k datům dělíme paměti na:

- **RAM** (**random access memory**, **paměť s náhodným přístupem**) – Přístupová doba nezávisí na umístění položky.
- **SAM** (**serial access memory**, **paměť se sériovým přístupem**) – Přístupová doba je ovlivněna dobou, než se dostane čtecí hlava k položce.
- **se smíšeným přístupem** – Kombinace obojího, např. u disku s váce záznamovými povrchy určíme číslo povrchu (RAM) a potom čteme sekvenčně (SAM).

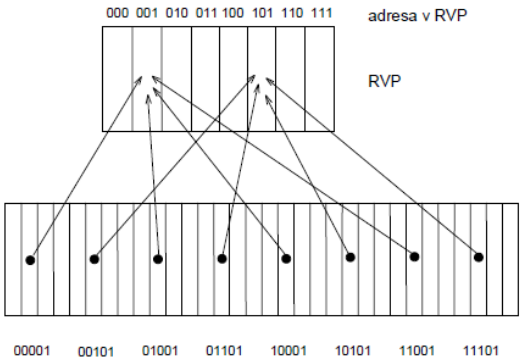


Figure 4: RVP s přímým mapováním

Podle měnitelnosti obsahu dělíme paměti na:

- **RWM** (**read/write memory**) – Paměť umožňující čtení i zápis.
- **ROM** (**read only memory**) – Paměť umožňující pouze čtení, zapisovat nelze.

Můžeme se setkat s různými variantymi pamětí ROM, např.:

- **PROM** – Programovatelná ROM, nenaprogramovaná paměť umožňuje jedno naprogramování, další změna již není možná.
- **EPROM** – Vymazatelná PROM, naprogramovaná paměť se dá vymazat a znovu naprogramovat. Paměti s tímto označením se mažou ultrafialovým zářením.
- **EEPROM** – Elektricky vymazatelná PROM. Vymazat lze pouze celou paměť zaráz, ne jen část (na rozdíl od flash).
- **flash** – Obdoba EEPROM, jsou energeticky nezávislé. Vymazání se provádí elektronickou cestou a může být provedeno přímo z počítače. Flash paměti se dělí na
  - **NAND** – Většinou používány ve flash discích, hlavní paměti nebo SSD. Umožňují mazání po blocích, není nutné mazat celou paměť.
  - **NOR –** Umožňují náhodný přístup po jednotlivých slovech, používají se někdy místo EPROM, např. pro konfigurační data.

Podle doby uchování informace dělíme paměti na:

- **SRAM** (static RAM) – Drží informace libovolně dlouho (při zachování určitých provozních parametrů). Data uchovává klopnými obvody, je dražší a rychlejší.
- **DRAM** (dynamic RAM) – Drží informaci jen po krátkou dobu, musí se neustále obnovovat (tzv. **refresh**). Data uchovává kondenzátory (vybíjí se), je pomalejší ale méně energeticky náročná a levnější.

A podle energetické závislosti na:

- **volatilní** – Potřebuje napájení k udržení informace.
- **nevolatilní** – K udržení informace nepotřebuje napájení.

DDR (**double data rate**) je technologie,která umožňuje dvounásobnou propustnost paměti díky reakci na sestupnou I vzestupnou hranu. Technologie má I další vylepšení nazývané DDR2 a DDR3.
Princip lokality nám umožňuje předvídat čtení z paměti a vychází ze dvou základních principů:

- **časová lokalita** – Pokud procesor používá nějakou položku v paměti, je vysoká pravděpodobnost, že ji bude používat znovu, proto je dobré ji uložit blíž k procesoru.
- **prostorová lokalita** – Pokud procesor pracuje s nějakou položkou v paměti, potom položky, které jsou umístěny v paměti v blízkostí s této položky, budu s vysokou pravděpodobností také použity, proto je dobré je uložit blíž k procesoru.

RVP (**rychlá vyrovnávací paměť**), neboli **cache**, je rychlá SRAM umístěná mezi hlavní pamětí a procesorem. Je rozdělena do bloků o konstantní velikosti (které by měly odpovídat velikosti bloku, s nímž pracuje hlavní paměť), což jsou stejné bloky z nichž se skládá hlavní paměť, avšak v té je bloků mnohem víc, proto mohou být v RVP jenom některé. Většinou je dále organizována jako hierarchie, přičemž se úrovně (od nejrychlejší) označují L1, L2, atd. U RVP nás zajímají tyto parametry:

- **hit rate** (pravděpodobnost úspěchu) resp. **miss rate** (pravděpodobnost výpadku) – Snažíme se maximalizovat, ale vždy závisí I na datech a programu. V praxi se pohybuje kolem 95 – 99%.
- **přístupová doba** (doba potřebná pro nalezení bloku)
- ztrátová doba (miss penalty) – Doba potřebná na přisunutí bloku.

Pokud se zápisem do RVP změní některý blok, data na nižších úrovních (např. v hlavní paměti) se už nesmí použít, vzniká tzv. **datová nekonzistence** (**nekoherence**). K udržování koherence dat slouží tyto strategie:

- **přímý zápis** (write-through) – Při zápisu do RVP se ihned zapisuje i na nižší úroveň. Výhodou je jednoduchost, nevýhodou nižší rychlost, protože se často zapisuje do nižších, tedy pomalejších úrovní.
- **zápis s mezipamětí** (write buffer) – Umožňuje odložit opravné zápisy až do okamžiku uvolnění přístupu k vzdálenější paměti, takže nedochází ke zdržování procesoru.
- **zpětný zápis vždy** – Blok se na nižší úroveň zapíše až při jeho odsouvání z RVP, tato metoda je nepraktická, protože se zapisuje I tehdy, když nedošlo k žádné změně.
- **Zpětný zápis podle příznaku změny** (write-back, copy-back, store on flag) – Prakticky použitelná varianta předchozí metody, máme příznak, tzv. **dirty bit**, který říká, zda došlo ke změně.

RVP může být navržena následujícími způsoby:

- **RVP s přímým mapováním** – RVP má určitý počet bloků očíslovaný binárně – tato čísla představují nejnižší bity adresy. Každý blok má navíc ještě tzv. **tag**, což jsou zbývající bity adresy a **valid bit** – **příznak platnosti**. Nevýhodou je, že dva bloky, které se namapují na stejný blok v RVP, v ní nemohou být současně.
- **vícecestné RVP** – Řeší vzájemné vytlačování položek tzv. zvyšováním **stupně asociativity,** což je snížení počtu bitů číslujících položku v RVP a zvýšení počtu bitů příznaku, přičemž do jedné položky lze uložit až tolik bloků, jaký je stupeň asociativity. Maximální stupeň asociativity znamená, že tag je celá adresa bloku – v praxi je toto však příliš drahé.

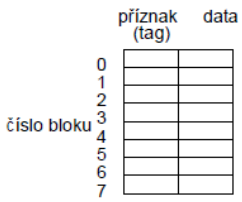


Figure 7: asociativita 1

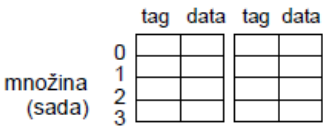


Figure 6: asociativita 2

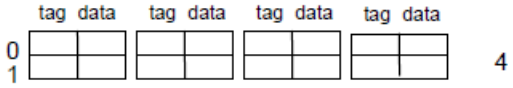


Figure 5: asociativita 4

SSD (solid state disc) je alternativa k pevnému disku, data uchovává elektronicky (v NAND flash), proto je SSD tichý a rychlý, ale je dray. Přístup na pevný disk trvá řádově milisekundy, do paměti řádově nanosekundy.

5. Vestavěné systémy (mikrokontrolér, periferie, rozhraní, převodníky). IMP, IPR

Vestavěný systém (**embedded system**) je počítač, který je zabudovaný do systému, ale pro uživatele je skrytý (vestavěný). Používají se v průmyslu, telekomunikacích, periferiích počítačů, měřících přístrojích apod. Vestavné systémy jsou mnohdy navrženy tak, aby byly funkční i bez lidského zásahu, musí proto splňovat požadavky na reaktivnost (odezvu, musí být určitý garantovaný čas reakce), autonomii (samostatnost) a kritičnost (odchylky od normálu). Je potřeba rozlišovat pojmy:

- **mikroprocesor** – Základní jednotka na čipu, skládá se z **ALU** (**aritmeticko logická jednotka**), registrů atd.
- **mikropočítač = mikrokontrolér** – Mikroprocesor plus příslušné podpůrné obvody, např. paměť, periferní jednotky atd. Příkladem může být 8 bitový HCS08 od firmy Freescale (další např. Siemens, Motorola, ...).

Kompromisem mezi mikroprocesorem a mikrokontrolerem je tzv. **DSP** (digital signal processing) procesor. Ten je určen především pro zpracování signálů. Výhodou je rychlé zpracování číslicových dat, i v plovoucí řádové čárce. Můžou se používat např. pro kompresi obrazu a zvuku.

Rozdíly mezi vestavěným systémem a univerzálním počítačem jsou především:

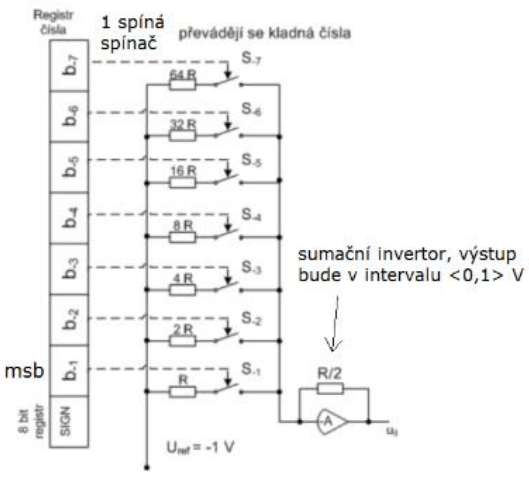
vestavěný systém	univerzální počítač
jeden program pro celý život, specifický pro aplikaci	uživatel spouští různé programy
periferie slouží hlavně pro snímání stavu okolí	periferie slouží hlavně pro komunikaci s člověkem
startuje sám bez zásahu	nutnost operačního systému
uživatel by neměl tušit, že pracuje s počítačem	

Požadovanými vlastnostmi vestavěných systémů jsou:

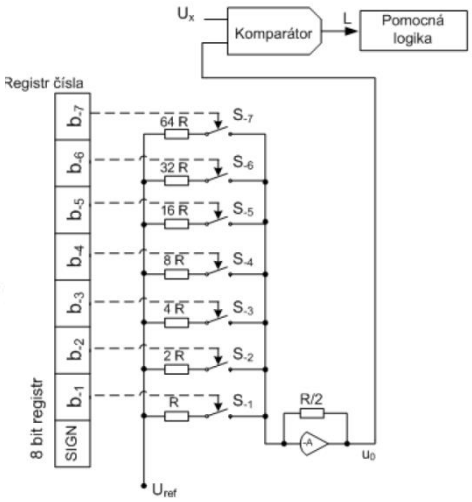
- výkonnost
- odezva
- V/V obvody
- testovatelnost
- nízký příkon
- spolehlivost
- bezpečnost
- udržovatelnost
- zabezpečení
- dostupnost
- cena
- velikost/váha
- životnost

Software pro vestavěný systém se většinou soustřední na interakci s hardwarovými prvky. Musí být spolehlivý a bezpečný, protože jej nelze jednoduše kdykoli vypnout. Může a nemusí být použit některý z operačních systémů pro vestavěné systémy (tzv. **RTOS**, real time operating system, zaručují odezvu bez bufferování požadavků, patří sem např. QNX 4 RIOS, Embedded Linux nebo Windows CE). Při návrhu jsou používány metriky jako jsou náklady, velikost, výkonnost, příkon apod.

Analogově-číslicový převodník (**A/D převodník**, **ADC**) je hardwarový modul pro převod analogového signálu na digitální. Jednou z možných a nějčastějších realizací A/D převodníku je komparační A/D převodník. Ten ke své činnosti využívá **číslicově-analogový převodník** (**D/A převodník**, **DAC**). Komparační D/A převodník funguje tak, že postupně porovnává pomocí komparátoru napětí na vstupu s hodnotami napětí, které generuje a pomocí A/D převodníku převádí na napětí. Porovnávání probíhá v několika krocích a neustále se spřesňuje. Uvažujme 8bitový převodník – v takovém případě se nejdřív nastaví hodnota MSB na 1, ostatní bity se vynulují, budeme tedy porovnávat hodnotu 10000000. Dále se nastaví bit 7 na 1, hodnota MSB zůstane buď 1 nebo 0 podle výsledku předchozího porovnání, tzn. nyní bude,e porovnávat buď hodnotu 11000000 nebo 01000000. Tak postupujeme až k LSB a k výsledné binární hodnotě. Nevýhodou tohoto způsobu je pomalejší převod, protože probíhá v několika krocích. Výhodou je relativně jednoduchá a levná obvodová realizace.



Obrázek 17: D/A převodník



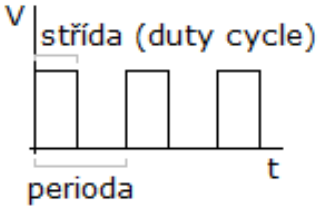
Obrázek 16: komparační A/D převodník

Mikroprocesor je zpravidla konstruován podle jedné ze dvou základních architektur:

- **Harvardská** – Odděluje paměť programu a dat.
- **Von Neumannovská** – Neodděluje paměť programu a dat.

Periferie jsou vnější obvody, jimiž lze mikrokontrolér obohatit. Může jít např. o 7segmentový displej, klávesnici, nebo čítač a časovač. Některými z nich jsou:

- **časovač** – Registr, který se s každým cyklem hodin inkrementuje, jeho účelem je odměřovat čas. Jeho periodu lze zvýšit dělením vstupního signálu (tzv. **předdělička**). Je využíván např. **jednotkou záchytu hrany** pro měření délky impulzu (měří se 2 následující vzestupné hrany).
- **watchdog** – Obvod umožňující zotavení se z programových chyb. Jedná se o čítač, který musí být programem neustále periodicky nulován. Pokud se program např. zacyklí a nevynuluje watchdog, čítač přeteče a spustí se reset obvodu.
- **generátor hodin** – Jedná se o oscilátor (krystalový, může být interní nebo externí) plus doprovodné obvody (např. předdělička).



PWM (**pulzně šířková modulace**) je technika generování pulzů o konstantní periodě a proměnlivé střídě. Používá se především pro ovládání motorů, někdy může nahradit jednoduchý D/A převodník.

Mikrokontrolér má vždy určitá **rozhraní**, přes něž může komunikovat. Může jít např. o:

- **SCI** (Scalable Coherent Interface) – Asynchronní (synchronizuje se start bitem), **plně duplexní** (možná současná komunikace oběma směry) sériové rozhraní. Data se vysílají jednoduše zápisem do registru.
- **SPI** (Serial Peripheral Interface) – Synchronní (rychlejší než asynchronní), plně duplexní sériové rozhraní. Jedno zařízení je master (vysílá hodinový signál) a jedno slave.
- **USB** (**Universal Serial Bus**) – Univerzální sériové rozhraní.

6. Principy řízení a připojování periferních zařízení (přerušení, programová obsluha, přímý přístup do paměti, sběrnice). INP, IPZ, IMP

Periferní zařízení připojujeme na systémovou sběrnici počítače přes jejich **řadič** (neboli **adaptér**), což je jejich hardwarové rozhraní. Řadič a periferní zařízení spolu komunikují pomocí vstupně výstupní sběrnice.
Z programového hlediska může být komunikace se zařízením realizována dvěma způsoby:

- **izolovaný vstup/výstup** – Existují speciální instrukce, většinou *IN* a *OUT*.
- **mapování do paměťového prostoru** – Registry zařízení jsou mapovány do paměti, proto se používají instrukce pro práci s pamětí.

Periferní operace probíhá tak, že počítač vloží přes systémovou sběrnici parametry operace do registrů periferního zařízení a nastaví řadiči zařízení bit **start operace**, který bývá součástí tzv. **stavového registru**, čímž zařízení začne pracovat. Zařízení během operace vytvoří tzv. **stavovou slabiku**, která nese informace o tom, jak operace proběhla. Pokud vznikla chyba, může být dále generována upřesňující **slabika závad**. **Autonomním prováděním** periferní operace rozumíme situaci, kdy procesor operaci pouze inicializuje a dále se jí neúčastní, je uvědomněn (např. přerušením) až ve chvíli, kdy byla operace provedena.

Programová obsluha periferní operace může probíhat jedním z následujících způsobů:

- programově řízený vstup/výstup (**polling**) – Procesor průběžně sám testuje bit konec operace. Tato technika vytěžuje procesor. Používá se např. v USB sběrnici.
- přerušení – Po skončení operace zařízení vyvolá přerušení. Procesor nemusí nic testovat a není vytížen.
- DMA – Další komunikace se zařízením zajišťuje obvod DMA, nikoliv procesor.

Běžně probíhá přenos dat mezi periferním zařízením a pamětí přes střadač procesoru. Jakmile se rozpozná vstupně/výstupní instrukce, převezme řízení řadič sběrnice, který přenos řídí. **Přímý přístup do paměti**, neboli **DMA** (**direct memory access**) je technika, kdy existuje speciální obvod DMA, který je schopný řídit přenos dat mezi datovým registrem periferního pevného disku (u jiných zařízení se nepoužívá) a pamětí bez účasti procesoru, který zatím může provádět užitečné výpočty (data tedy nejdou přes procesor ani řadič DMA). Obvod DMA je tzv. **bus master** – umí řídit sběrnici. Přenos DMA začíná žádostí o DMA od per. zařízení, která se musí potvrdit procesorem.

Přerušení je metoda asynchronní obsluhy událostí. Přerušení dělíme na **vnější** (vyvolané mimo procesor), **vnitřní** (způsobené zařízením na procesoru) a **programové** (generované speciální instrukcí). Dále může být přerušení označeno jako **nemaskovatelné**, což znamená, že má velkou prioritu a nelze jeho obsluhu zakázat.
Pokud někdo vyvolá přerušení, pozastaví se vykonávání aktuálního programu, uloží se kontext a začne se provádět speciální kód, tzv. **obsluha přerušení**. Odkaz na kód přerušení je uložen ve speciální tabulce zvané **vektor přerušení**. Po skončení obsluhy se procesor vrací k vykonávání programu tam, kde byl přerušen. O přerušení se stará **řadič přerušení** (ne přímo procesor), který určuje priority, maskování apod. Systémová (popř. i vstupně/výstupní) sběrnice musí umožnit všem periferním zařízením generovat přerušení. Existují dva způsoby, jak toto realizovat:

- přerušení spouštěné **hranou** – Každé zařízení má na sběrnici jeden vodič pro informaci o přerušení a také svůj vektor přerušení. Po příchodu přerušení se ihned ví, které zařízení jej vyvolalo a je spuštěna jeho konkrétní obsluha.
- přerušení spouštěné **úrovní** – Na sběrnici je pro přerušení jeden vodič pro všechna zařízení společný a existuje jeden vektor přerušení. Po příchodu přerušení se musí v obsluze zjišťovat, které zařízení jej vyvolalo.

Sběrnice je skupina signálových vodičů a definuje pro periferní zařízení rozhraní pro komunikaci s počítačem. Sběrnice dělíme na **sdílené** (jsou společné vodiče pro různá data, neplést si se sdílením sběrnice více zařízeními – tak je tomu vždy!) a **nesdílené** (každý typ dat má vždy svou vlastní skupinu vodičů). Dále sběrnice dělíme na **synchronní** (s hodinovým signálem) a **asynchronní** (bez hodinového signálu) a na **paralelní** (starší, kvůli zvyšující se šířce dat vyvstal problém s rušením) a **sériové** (používané dnes). **Systémová sběrnice** (hlavní sběrnice pro komunikaci zařízení v počítači) musí umožňovat přenos **dat**, **řízení**, **adres** a **stavových informací**. **Šířka sběrnice** je počet bitů, které je možné jedním přenosem přes sběrnici poslat, **rychlost sběrnice** je potom její datová propustnost. Sběrnice je řízena **řadičem sběrnice**. Mezi příklady systémových sběrnic patří:

- **ISA** – Stará 8 nebo 16bitová synchronní paralelní nesdílená sběrnice (1981). Jejím nástupcem byla sběrnice EISA.
- **PCI** – Paralelní, 32 nebo 64 bitová multimaster (může ji řídit více zařízení) sběrnice. Umožňuje plug and play. Základní princip sběrnic na bázi PCI je existence dvou rozhraní mezi sběrnicovými systémy:
  - **northbridge** (**severní most**) – Jsou přes něj připojena zařízení, která musí komunikovat rychle (grafický adaptér, paměť, ...). Transformuje sběrnici procesoru na rozhraní PCI. Někdy obsahuje integrovanou GPU.
  - **southbridge** (**jižní most**) – Jsou přes něj připojena standardní zařízení nevyžadující tak rychlou komunikaci (např. disk, USB, ...).
- **PCI-E** (**PCI express**) – Nástupce sběrnice PCI, na rozdíl od ní je již sériová. Pro komunikaci používá speciální protokol podobný síťovému. Neprobíhá zde přidělování sběrnice, protože spoje jsou point-to-point dedikované, pro každé zařízení existuje jeden, který je možné kdykoli využít.
- …

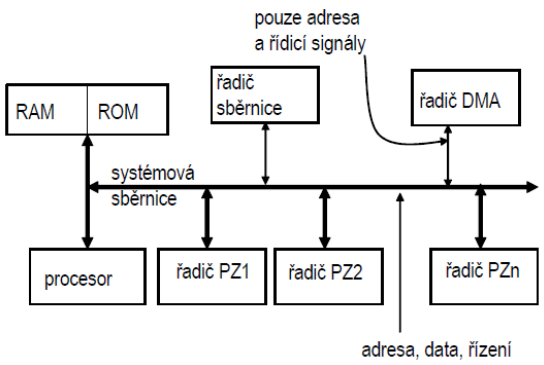
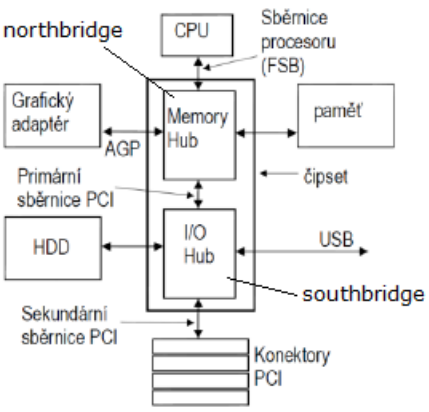


Figure 8: systémová sběrnice

Sběrnici nemůžou v jednu chvíli využívat všechna zařízení, musí se postupně střídat. Před vlastním přenosem dat musí proběhnout tzv. **přidělení sběrnice** zařízením, které spolu budou komunikovat. Přidělování může být:

- **centralizované** – Existuje **arbitr** neboli specializovaná jednotka, která přijímá požadavky pro přidělení sběrnice a rozhoduje o tom, kterým zařízením bude přidělena (budou ji moct využít ke komunikaci).
- **distribuované** – Arbitr neexistuje, o přidělování sběrnice rozhodují samotná zařízení mezi sebou.

Vedle systémové sběrnice rozeznáváme rozeznáváme i další typy sběrnic, např. **vstupně výstupní**, která je umístěna mezi řadičem periferního zařízení a samotným zařízením. Příkladem takových sběrnic je např. centronix (pro tiskárny) nebo **IDE**/**EIDE** (také **PATA**, paralelní **ATA**, rozhraní pro pevné disky, řadič je integrován ve stejné jednotce jako disk, každý disk má tedy svůj řadič, který je mu navíc blízko a tudíž může pracovat rychleji), **SATA** (sériové rozhraní) nebo **SCSI** (paralelní sběrnice pro různé typy periferních zařízení). Příkladem grafické sběrnice je **AGP**.



Obrázek 18: architektura PCI

7. Princip činnosti počítače (řetězené zpracování instrukcí, RISC, CISC). INP

Architektury procesorů dělíme na **RISC** (**Reduced Instruction Set Computing**) a **CISC** (**Complex Instruction Set Computer**). CISC architektura se snaží o poměrně komplexní funkce, které by jinak šly nahradit jednoduššími. Instrukce mají proměnlivou délku a dobu vykonávání. CISC obvody jsou na základní úrovni schopné dělat komplexnější operace. Architektura RISC se naproti tomu snaží o sadu stejně složitých instrukcí, přičemž **CPI** (**cycles per instruction**) je ideálně 1 (toho se snaží dosáhnout pomocí zřetězeného zpracování instrukcí). Pro přístup k paměti existují pouze dvě instrukce – LOAD a STORE. Architektura bývá registrová. Dnešní osobní počítače používají kombinaci obou přístupů, např. základní sada instrukcí IA-32 se snaží o CPI 1, ale máme i komplexní instrukce, které toto nedodržují. CISC má obecně kratší strojový kód než RISC, ale vyžaduje složitější obvodovou implementaci a je u něj problém se zřetězeným zpracováním. Procesory Intel používají kombinaci RISC a CISC.

Zřetězené zpracování (**pipelining**, proudové zpracování) je technika urychlování výpočtů, přesněji zpracování instrukcí procesorem. Principem je mít více zřetězených jednotek (za každou je registr, i za prázdnou) takových, že každá zpracovává instrukci různým způsobem (princip je podobný výrobním linkám v továrnách). Frekvenci zpracování potom určuje nejpomalejší jednotka. Princip zřetězeného zpracování se musí vypořádávat se skoky mezi instrukcemi např. pomocí predikce skoků, při špatné predikci se ztrácí veškerá efektivita, protože se musí celá linka vymazat a znovu naplnit. Zrychlení zřetězené linky oproti běžnému obvodu je:

$$z = \frac{t_{původní}}{t_{zřetězená}} = \frac{N.k.t}{(k+N-1)*(t+d)}$$

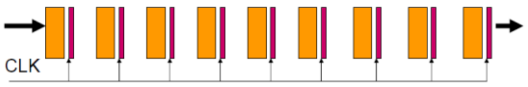


Figure 9: zřetězená linka

kde *N* je počet vstupů, které zpracováváme, *k* je počet stupňů linky, *t* je zpoždění stupně linky a *d* je zpoždění registru.

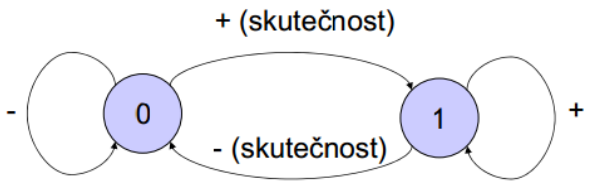
V procesorech typicky rozeznáváme tyto fáze zpracování instrukce:

- **fetch** – Načte se instrukce a inkrementuje programový čítač.
- **decode** – Instrukce se dekóduje.
- **execute** – Instrukce se vykoná.
- **memory access** – Zde se přistupuje do paměti a dokončují se skoky.
- **write back** – Zapisují se výsledky.

Řetězená linka může být zpomalena následujícími konflikty:

- **strukturální** – Obvodová struktura neumožňuje současné provedení dvou akcí (např. současné dva součty, když má pouze jednu sčítačku). Řešení:
- **datový** – Jsou potřeba data z předchozí instrukce, která ještě není dokončena. Řešení: tzv. **baypassing** (forwarding) – poskytnutí mezivýsledku dřív, než je zapsán do registru. Dosahuje se ho přidáním speciálních datových cest. Datové konflikty dělíme na:
  - **RAW** (read after write) **–** B čte zdroj před tím, než je proveden zápis instrukcí A. B přečte starou hodnotu. Řešením je baypassing.
  - **WAW** (write after write) – B zapíše hodnotu dřív než A, zápis je tedy proveden ve špatném pořadí a zapsána zůstane špatná hodnota.
  - **WAR** (write after read) – B zapíše hodnotu dřív, než ji A přečte. A tedy získá novější hodnotu, což je chyba.
  - **RAR** (read after read) – Nejedná se o hazard.
- **řídicí** – Skoková instrukce (podmíněná) mění obsah PC a neví se, z jaké adresy se mají přednačítat instrukce. Řešení:
  - **zpožděný skok** – Přeskládat instrukce tak, aby v době výpočtu adresy linka nečekala a prováděla užitečné výpočty. V nejhorším případě vložit NOPy.
  - **předspracovat obě možnosti** – Vyžaduje složitější hardware.
  - **Predikce skokou** – Předvídá se, kam se skočí. Predikce skoku může být:
    - **statická –** Bit predikce nastavuje kompilátor.
    - **dynamická –** Provádí speciální hardware.

    Nejjednodušším prediktorem je jednobitový prediktor, který si pamatuje výsledek předchozího skoku a ten předvídá pro další skok:



Programátorský model procesoru určuje, jak programátor s procesorem pracuje. Rozlišujeme tyto modely:

- **zásobníkový –** operace se obecně dějí na zásobníku
- **střadačový** – existuje jeden hlavní registr, tzv. **střadač** (**acumulator**)
- **registrový** – existuje více registrů, které jsou na stejné úrovni
- **smíšený** – kombinace výše zmíněných

8. Minimalizace logických výrazů (algebraické metody, Karnaughova mapa, Quine McCluskey).

INC

Minimalizace logického výrazu znamená k danému logickému výrazu najít ekvivalentní, jednodušší výraz, tzn. jeho pravdivostní tabulka musí být stejná. Důvodem je jednodušší obvodová realizace. Pro minimalizaci existuje několik metod několik metod.

Algebraická metoda využívá základních a odvozených vztahů, které ve výrazu hledáme. Patří mezi ně následující:

- komutativita – $x \vee y = y \vee x$, $x \wedge y = y \wedge x$
- distributivita – $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- neutralita 1 a 0 – $x \vee 0 = x$, $x \wedge 1 = x$
- komplementarita – $x \vee \bar{x} = 1$
- nedegenerovanost – $0 \neq 1$
- asociativita –, $(x \vee y) \vee z = x \vee (y \vee z)$, $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
- absorbce – $x \vee (x \wedge y) = x$, $x \wedge (x \vee y) = x$
- agresivita nuly – $x \wedge 0 = 0$
- agresivita jedničky –$x \vee 1 = 1$
- idempotence – $x \vee x = x$, $x \wedge x = x$
- absorbce negace – $x \vee (\bar{x} \wedge y) = x \vee y$, $x \wedge (\bar{x} \vee y) = x \wedge y$
- dvojitá negace – $\bar{\bar{x}} = x$
- **De Morghanovy zákony** – $\bar{x} \wedge \bar{y} = \overline{x \vee y}$, $\bar{x} \vee \bar{y} = \overline{x \wedge y}$

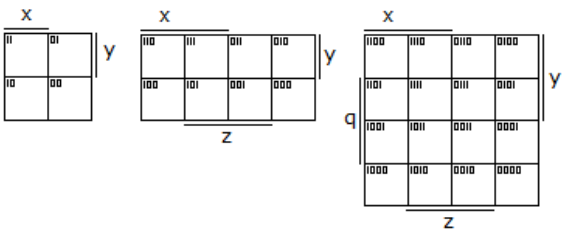
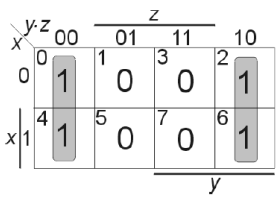


Figure 10: Karnaughovy mapy

Minimalizace pomocí **Karnaughových Map** se provádí tak, že nakreslíme tabulku a každému políčku přiřadíme hodnoty proměnných ve výrazu. Musí být dodrženo pravidlo, že se sousední políčka liší právě v jedné proměnné. Na okraje tabulky můžeme zakreslit pomocné pruhy, pod nimiž má daná proměnná hodnotu 1, jinak má hodnotu 0. Pro více než 4 proměnné je minimalizace touto metodou obtížná – neprovádí se.

s	x	y	z	F(x,y,z)
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0



$$F(x,y,z) = \bar{y} \cdot \bar{z} + y \cdot \bar{z}$$

Figure 11: příklad Karnaughovy mapy (lze zminimalizovat i lépe, přes okraj)

Minimalozovanou funkci zapíšeme do pravdivostní tabulky a najdeme všechny řádky, kde má funkce hodnotu 1. Na těchto řádcích se podíváme na hodnoty proměnných a pro každý zapíšeme na odpovídající místo v mapě 1. Do míst bez 1 nakonec zapíšeme 0. Potom hledáme v mapě tzv. **smyčky**, neboli obdelníky sousedících jedniček (a to I přes okraj). Smyčky potom zapíšeme jako logický součet do výsledku (výsledek je tedy v **ÚNDF – úplné normální disjunktní formě**, tedy sumě součinů, opak je **ÚNKF** – ú**plná normální konjunktní forma,** neboli součin součtů, který lze při alternativním algoritmu Karnaughových map také dostat).

Metoda **Quine-McCluskey** je vhodná i pro funkce více než 5-6 proměnných. Princip fungování je tento: Nejdřív rozdělíme implikanty (řádky) do skupin podle počtu jedniček v binární reprezentaci jejich čísla. Potom se snažíme slučovat řádky ze sousedních skupin, které se liší jenom v hodnotě jedné proměnné. Každou takovou skupinu označíme názvem (Px), pokud už dále nelze takto sjednocovat. Tento postup opakujeme, dokud je to možné, a poté zapíšeme všechny označené řádky spolu s pokrytím do tabulky. V tabulce se snažíme najít tzv. **minimální pokrytí**, tzn. nejmenší množinu řádků, které pokrývají všechny sloupce. Některé řádky jsou tzv. **nesporné implikanty** a ve finálním řešení musí být, protože jako jediné pokrývají některý sloupec. Určení dalších řádků nemusí být jednoznačné.

Př.: Máme minimalizovat funkci: F(w,x,y,z) = 1(2,4,6,8,9,10,12,13,15) (všechny řádky pravd. tabulky, kde funkce nabývá 1).

implikant	wxyz	Jedniček
2	0010	1
4	0100	
8	1000	
6	0110	2
9	1001	
10	1010	
12	1100	
13	1101	3
15	1111	4

implikant	wxyz	označení
2,6	0-10	P2
2,10	-010	P3
4,6	01-0	P4
4,12	-100	P5
8,9	100-	-
8,10	10-0	P6
8,12	1-00	-
9,13	1-01	-
12,13	110-	-
13,15	11-1	P7

implikant	wxyz	označení
8,9,12,13	1-0-	P1

	2	4	6	8	9	10	12	13	15
P1				*x*	*x*		*x*	*x*	
P2	*x*		*x*						
P3	*x*					*x*			
P4		*x*	*x*						
P5		*x*					*x*		
P6				*x*		*x*			
P7								*x*	*x*

nesporné

Výsledek: F(w,x,y,z) = P1 + P3 + P4 + P7 = (1-0-) + (-010) + (01-0) + (11-1) = $w.\bar{y} + \bar{x}.y.\bar{z} + \bar{w}.x.\bar{z} + w.x.z$

Pro nalezení minimálního pokrytí lze použít tzv. **Petrickovu funkci**. Ta funguje takto: Přepíšeme tabulku bez nesporných implikantů:

	2	4	6	8	9	10	12
P2	*x*		*x*				
P3	*x*					*x*	
P4		*x*	*x*				
P5		*x*					*x*
P6				*x*		*x*	

Zapíšeme součin součtů všech řádků, které pokrývají stejný sloupec: C = (P2 + P3) . (P4 + P5) . (P2 + P4) . (P3 + P6)
Přepíšeme na disjunktní formu a zjednodušíme: P2.P3.P5 + P3.P4 + P2.P4.P6 + P2.P5.P6
Vybereme pokrytí s nejnižší cenou (nejméně členů): C_{min} = P3.P4

9. Reprezentace čísel a základní dvojkové aritmetické operace v počítači (doplňkové kódy, sčítání, odčítání, násobení, pevná a plovoucí řádová čárka, standard IEEE 754). IAS, INC, INP

Čísla jsou v počítači reprezentována binárně (pomocí **bitů**) v různých kódech, mezi nejpoužívanější patří:

- **přímý kód** – Nejvýznamnější binární číslice je znaménková (0 = +, 1 = -), následuje binární kódování absolutní hodnoty čísla. Výhodou je jednoduchost a snadný převod mezi kladnými a zápornými čísly, nevýhodou je existence kladné a záporné nuly a nevhodnost pro praktické aritmetické operace.
- **doplňkový kód** – Jedná se o binární negaci čísla zvětšenou o 1 (přetečení se neošetřuje). Nejvýznamnější bit opět určuje znaménko (0 = +, 1 = -). Výhodou je existence už pouze jedné reprezentace čísla nula a vhodnost pro aritmetické operace – lze např. použít stejnou sčítačku pro sčítání kladných I záporných čísel.
- **inverzní kód** – Jedná se o mezikrok mezi přímým a doplňkovým kódem, nejvýznamější bit opět určuje znaménko (0 = +, 1 = -) a následuje binární negace čísla. Vlastnostmi je tento kód podobný kódu přímému.
- **kód s posunutou nulou** – Hodnota čísla je dána hodnotou kódu, od níž odečteme určitou známou hodnotu, např. 128 pro osmibitové číslo (zápis 00000000 pak odpovídá hodnotě -128). Pro sčítání lze použít jednu sčítačku, ale pro násobení musíme od čísel nejdříve odečíst konstantu.Používá se např. pro reprezentaci exponentů.
- **BCD** (**binary coded decimal**) – Dvojkově kódované desítkové číslo, každá desítková číslice je kódována čtyřmi binárními číslicemi.

Při analýze těchto kódů nás typicky zajímá:

- **rozsah zobrazení** – interval <nejmenší zobrazitelné číslo, největší zobrazitelné číslo>
- **rozlišitelnost zobrazení** – nejmenší kladné zobrazitelné číslo
- **přesnost zobrazení** – počet platných dekadických číslic, které je možné zobrazit v daném paměťovém prostoru, pro *n*bitový prostor se vypočítá jako:

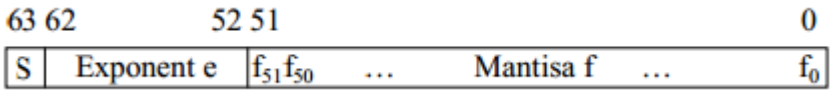
$$n \log_{10}(2)$$

Čísla s řádovou čárkou (tedy aproximaci reálných čísel) reprezentujeme dvěma způsoby:

1) **pevná řádová čárka** (**fixed point, FX**) – Určitý počet nejméně významných bitů se bere jako číslice za řádovou čárkou, máme tedy vždy přesně daný počet celých a číslic (*n*) a číslic za řádovou čárkou (*m*). Celkem máme $k = m + n$ bitů.
   a. rozsah: $<0, 2^n - 2^m>$
   b. rozlišitelnost: 2^{-m}
   c. přesnost: $k \log_{10}(2)$
2) **plovoucí řádová čárka** (**floating point, FP**) – Na pevně daném počtu bitů můžeme reprezentovat čísla extrémně malá i extrémně velká díky možnosti "posunovat" řádovou čárku – lze korigovat poměr mezi přesností a rozsahem. Pro plovoucí řádovou čárku definujeme číslo jako:

$$hodnota = (-1)^{znaménko} . mantisa . základ^{exponent}$$

   Základ se rovná číslu 2, mantisa je v přímém kódu a exponent v (modifikovaném) kódu s posunutou nulou. Standard definuje různé přesnosti, např. single (32b), double (64b) apod. Uložení čísla v přesnosti double potom vypadá např. následovně:



   Pro FP existuje standard IEEE 754. Mimo jiné definuje tyto formáty:

název	celkem bitů	základ	znaménko	exponent	mantisa
single	32b	2	1b	8b (-126 až 127)	23b (explicitních*)
double	64b	2	1b	11b (-1022 až 1023)	52b (explicitních*)
extended	80b	2	1b	různě	různě

   * - mantisa se bere jakože vždy začíná (tzv. implicitní) jedničkou, proto má single, resp. double efektivně 24, resp. 53 bitovou mantisu. Za touto jedničkou je desetinná tečka.

   Standard také definuje speciální hodnoty:

exponent	mantisa	význam
11111111	0	$\pm\infty$
11111111	≠0	NaN (je jich mnoho)
00000000	0	0
00000000	≠0	tzv. **denormalizované** (subnormalizované) číslo, exponent se v tomto případě bere -126 a nepředpokládá se implicitní jednička

Denormalizovaná čísla umožňují reprezentovat čísla menší než by bylo možné reprezentovat klasickým použitím normalizovaného formátu (protože nevyužívají implicitní jedničku, čímž získají možnost hodně snížit řád).

Př. v single precision:

11000000000000000000000000000000

znaménko = 1 => záporné
exponent = 10000000 = 128 => 128 – 127 = 2
mantisa = 000000... => 1.0000... (implicitní jednička)
číslo = -1 . 1.000 . 2^1 = -2

00111110101010101010101010101011

znaménko = 0 => kladné
exponent = 01111101 = 125 => 125 – 127 = -2
mantisa = 0101010... => 1.010101... (implicitní jednička)
číslo = 1.010101 . 2^{-2} = cca 1/3

Základní operace ve dvojkové soustavě (doplňkový kód) jsou:

- **sčítání**, např.:
  3 0011
  +2 0010
  =5 0101
- **odčítání**, např.:
  5 0101
  -6 0110

```
=-1  1111
```

- **násobení** (výsledek násobení dvou *n* bitových čísel je na 2*n* bitech), např.:

```
  11                          00001011
 *14                          00001110
  44                          00000000
+11                           00001011
=154                         00001011
                            00001011
                           00000000
                          00000000
                         00000000
                        00000000
                       000000010011010
```

  Hardwarové násobičky dělíme podle toho, zda násobení provádějí **sekvenčně** (postupně) nebo **kombinačně** (v jednom kroku). Sekvenční násobičky se skládají z malého počtu hradel, ale jsou pomalejší. Proto se většinou používají násobičky kombinační, které se skládají ze sčítaček. Pro násobení dvou čísel se znaménkem v doplňkovém kódu existuje **Boothův algoritmus**, který překódovává čísla pomocí tzv. radixu.

- **dělení**:
  Často dělíme absolutní hodnoty a znamenko určíme až nakonec (**algoritmus SRT** umí pracovat se znaménky přímo). Výsledkem dělení je celá část a zbytek. Častý algoritmus dělení je např následující:

```
Q := 0
zarovnej dělence a dělitele pod sebe doleva
dokud (dělitel je menší než dělenec)
  {
    pokud část dělence nad dělitelem je větší než dělitel:
      odečti dělitele od části dělence nad ním a ke Q připoj zprava 1
    jinak:
      ke Q připoj zprava 0

    posuň dělitele o 1 doprava
  }
celá část := Q
zbytek := dělenec
```

  např.:

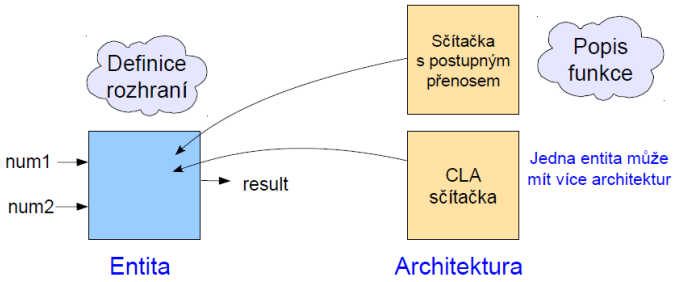
```
11100110 : 110 (230 : 6)

           Q = 100110 <-------- celá část (38)
11100110
110
  100110
  110
  100110
  110
  100110
  110
    110
     111
     110
      10
      110
       10 <-------- zbytek (2)
```

  Hardwarové děličky dělíme podobně jako násobičky na sekvenční a kombinační.

10. Principy VHDL (entita, architektura, proces, příklady kombinačních a sekvenčních obvodů). INC, INP

VHDL ((VHSIC hardware description language) je silně typovaný jazyk pro návrh číslicových systémů (dalším jazykem je např. Verilog, který je novější, má C syntaxi a slabé typování). Umožňuje tyto systémy specifikovat, **syntetizovat** (vytvořit z popisu schéma obvodu, ne vše lze systetizovat) nebo simulovat. Oproti klasickým programovacím jazykům umí modelovat souběžné děje a vlastnosti fyzického hardwaru. VHDL popisuje zařízení a jeho části pomocí **komponent**, kterými jsou:

- **entity** – Definují rozhraní komponent.
- **architektury** – Popisují funkci a chování komponent.



Entita popisuje rozhraní jako soubor **signálů** a **generických parametrů** (např. šířka dat), které mohou být podle směru šíření dat IN, OUT nebo INOUT.
Architektura je vždy svázána s nějakou entitou. Architekturu můžeme popsat těmito způsoby (nebo i jejich kombinací):

- **strukturní popis** – Popis pomocí propojení jiných komponent. Výhodou je částečná kontrola nad tím, jak bude obvod fyzicky sestaven. Z tohozo popisu se provádí tzv. **RTL syntéza**.
- **popis chování** (**behaviorální**) – Popis pomocí sekvenčních příkazů (algoritmem). Hodí se např. pokud nám jde především o simulaci. Elementární obvody (např. AND) jsou vždy popsány již jen behaviorálně. Z tohoto popisu se provádí tzv. **behaviorální syntéza**.
- **dataflow popis** – Popis datových toků pomocí jednoduchých logických funkcí. Z tohoto popisu se vytváří tzv. **logická syntéza**.

Příkladem kódu v jazyce VHDL může být:

```
library IEEE;
use IEEE.std_logic_1164.all;

-- entity
entity ANDGATE is
  port (                -- seznam signalu
    IN1 : in std_logic;
    IN2 : in std_logic;
    OUT1: out std_logic
    );
end ANDGATE;

-- architecture
architecture RTL of ANDGATE is

begin
  -- sekce paralelnich prikazu - nezalezi na poradi!

  OUT1 <= IN1 and IN2; -- dataflow popis

end RTL;
```

Existuje speciální příkaz **process**, který umožní definovat chování komponenty sekvenčními příkazy (záleží na pořadí):

```
[label:] process [(sensitivity list)]
  -- deklaracni cast
begin
  -- sekvencni prikazy
end process [label];
```

V deklarační části lze deklarovat datové objekty, jako např. konstanty nebo proměnné. **Sensitivity list** je seznam signálů, při jejichž změně se proces spustí (toto platí pouze pro simulaci, v reálném hardwaru běží proces stále). Příkaz přiřazení do signálu (<=) je vykonán až po skončení procesu, zatímco přiřazení do proměnné (:=) se provede ihned.

Uveďme příklad sekvenčního prvku – klopného obvodu typu D:

```
library IEEE;
use ieee.std_logic_1164.all;

entity reg is port (CLK : IN std_logic; D : IN std_logic; Q : OUT std_logic);
end reg;

architecture behavioral of reg is
begin

  reg: process (CLK)
  begin
    if CLK'event and (CLK = '1') then - vzestupna hrana
      Q <= D;
    end if;
  end process;
end architecture;
```

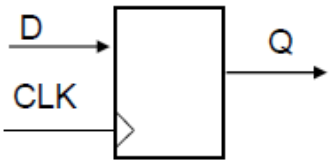


Figure 12: klopný obvod D

VHDL umí deklarovat tyto datové objekty:

- **konstanta** (constant) – Hodnota, která je během simulace nebo syntézy neměnná.
- **proměnná** (variable) – Lze deklarovat pouze uvnitř procesu, používá se typicky pro uchovávání mezihodnot, její hodnota je (na rozdíl od signálu) aktualizována hned po provedení příkazu přiřazení.
- **signál** (signal) – Reprezentuje spojení mezi komponentami systému, po přiřazení není jeho hodnota aktualizována, proto by neměl být používán jako proměnná.
- **další** (file, shared variable, ...) – Není vždy syntetizovatelné.

Datový objekt může být určitého datového typu. VHDL je silně typovaný, tzn. interagovat mohou pouze objekty stejného datového typu. Datový typ může být **skalární** (boolean, integer, character, string, bit, ...) nebo **kompozitní** (array, record, ...).

FPGA (**field-programmable gate array**) je obvod navržený k tomu, aby byl nakonfigurován (naprogramován) zákazníkem po tom, co je vyroben. Obsahuje logická hradla a paměť RAM a je schppný realizovat jakoukoli logickou funkci. Dá se programovat pomocí jazyků pro popis hardwaru, např. VHDL.

11. Metody rasterizace 2D vektorových objektů: úseček, kružnic a křivek. IZG

Rasterizace je proces převodu vektorové reprezentace objektu do rastrové podoby. Existují různé metody rasterizace různých objektů, např.:

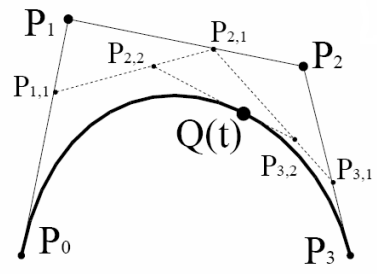
- úsečky – Rasterizují se většinou pouze úsečky ležící v úhlu 0 – 45°, ostatní případy se na tento převádějí.
  - **DDA** (digital diferential analyser) algoritmus – Jeden z prvních a nejjednodušších algoritmů, je intuitivní, ale pomalý, protože používá aritmetiku v plovoucí řádové čárce. Princip spočívá v pohybu po ose X po jednotlivých pixelech (tzv. diferencích) a počítání hodnoty Y přičítáním směrnice úsečky, přičemž výsledek zaokrouhlujeme.
  - **error control DDA** (DDA s hlídáním chyby) algoritmus – Variace DDA algoritmu, opět jdeme po ose X, tentokrát však počítáme v každém kroku chybu $E = E + k$, pokud E přesáhne hodnotu 0,5, přesuneme se o řádek výš a od chyby odečteme 1 (aby zůstala relativní vůči danému řádku).
  - **Bresenhamův algoritmus** – Dnes nejpoužívanější a nejefektivnější algoritmus, využívá pouze celočíselnou aritmetiku, násobení, sčítání, odečítání a porovnávání (proto může být využit i v jednodušších zařízeních). Zavádí tzv. prediktor:

$$p_0 = 2dy - dx$$
$$p_i: \begin{matrix} \geq 0 & y++; & p_{i+1} \coloneqq p_i + 2dy - 2dx \\ < 0 & & p_{i+1} \coloneqq p_i + 2dy \end{matrix}$$

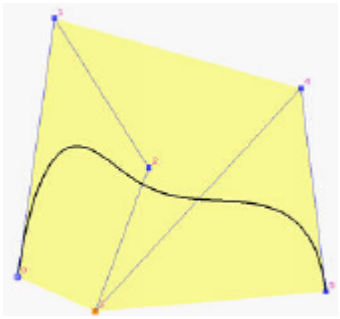
  - …
- kružnice – Počítá se většinou jen 1/8 kružnice, ostatní části se dají díky symetrii zkopírovat.
  - naivní algoritmus – Je intuitivní a jednoduchý, ale neefektivní, protože využívá aritmetiku v plovoucí řádové čárce. Principem je posun po ose X po jednotlivých pixelech a počítání hodnoty *y* z rovnice kružnice ($y = \sqrt{r^2 - x^2}$). Vypočítané hodnoty se kopírují podle výše zmíněné osminové symetrie. Když je *x = y* (pro kružnici se středem v *[0,0]*), algoritmus končí (bylo vykresleno 45°).
  - vykreslování jako n-úhelník – Kružnici vykreslujeme pomocí dostatečně krátkých úseček, např. DDA algoritmem.
  - **Midpoint algoritmus** – Nejpoužívanější algoritmus, nepoužívá čísla v plovoucí řádové čárce (jedná se o obdobu bresenhamova algoritmu pro úsečku). Vykreslování začneme v bodě [0;r] a postupujeme po ose X, dokud není X rovno Y. O posunu ve směru osy Y rozhoduje prediktor:

$$p_0 = 1 - r$$
$$p_i: \begin{matrix} \geq 0 & y--; & p_{i+1} \coloneqq p_i + 2x_i - 2y_i + 5 \\ < 0 & & p_{i+1} \coloneqq p_i + 2x_i + 3 \end{matrix}$$

  - …
- elipsy – Existují algoritmy s prediktorem podobně jako pro kružnici, elipsa je však symetrická pouze po čtvrtinách.
- křivky – Rasterizují se tak, že se vypočítá určité množství bodů ležících na křivce a ty se spojí úsečkami.
  - **Algoritmus de Casteljau** [**de kastelžó**] – Rekurzivní algoritmus pro vykreslování **Beziérových křivek**. Nejdříve zvolíme hodnotu parametru t v intervalu <0, 1> - typicky parametr navyšujeme od nuly po určitém kroku a pro každou hodnotu dostaneme bod ležící na křivce. Bod dostaneme takto: spojíme všechny řídící body (zde P_0 až P_3) a vzniklé úsečky rozdělíme v poměru $\frac{t}{1-t}$. Body rozdělení opět spojíme a opakujeme předchozí akci do té doby, než nám zbyde jeden bod. Ten leží na křivce.
  - …
- …



Obrázek 19: de Casteljau



Obrázek 20: konvexní obálka

Křivka je jednorozměrný geometrický objekt. Je určena konečnou množinou **řídících bodů**. Může mít následující vlastnosti:

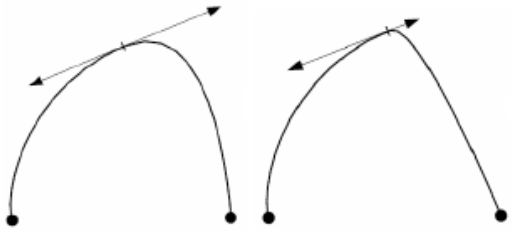
- **invariace k lineárním transformacím** – Lineární transformace (rotace, posun, změna měřítka, ...) nemají vliv na tvar křivky.
- **konvexní obálka** – Křivka leží celá v konvexní obálce svých řídících bodů.
- **interpolace krajních bodů** – Křivka prochází krajními body.
- **lokalita změn** – Poloha jednoho řídícího bodu mění jen část křivky, ne celou.

Křivky dělíme na:

- **interpolační** – Prochází všemi řídícími body.
- **aproximační** – Obecně neprochází řídícími body.

nebo ne:

- **racionální** – Řídící body mají navíc váhové koeficienty. Jsou invariantní vůči perspektivní projekci.
- **neracionální** – Tvar křivky lze měnit pouze změnou polohy řídících bodů. Nejsou invariantní vůči perspektivní projekci.



Obrázek 21: plná vs oslabená podmínka spojitosti

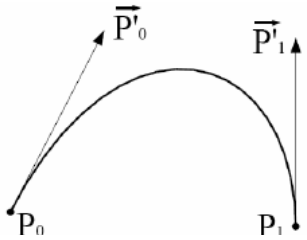
Křivky nevyjadřujeme jako funkci *y = f(x)*, ale parametricky: *Q(t) = [y(t), x(t)]*. Funkce *y(t)* a *x(t)* jsou často kubické polynomy, pak můžeme křivku zapsat jako matici koeficientů polynomu:

$$\begin{matrix} x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \end{matrix} \rightarrow \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix}$$

U křivek spojovaných z více segmentů dále můžeme udávat spojitost:

- **C^0** – Totožnost navazujících bodů.
- **C^1** – Totožnost tečných vektorů (prvních derivací) v navazujících bodech.
- **C^2** – Totožnost druhých derivací v navazujících bodech.
- …

Oslabíme-li podmínku na tvrzení, že stačí, když jsou vektory v navazujících bodech lineárně závislé (tzn. můžou se lišit délkou), udáváme spojitosti:

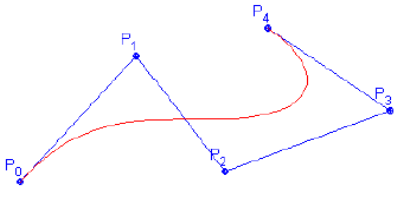


Obrázek 22: Fergusonova kubika

- **G^0** – Totožnost navazujících bodů.
- **G^1** – Lineární závislost tečných vektorů (prvních derivací) v navazujících bodech.
- **G^2** – Lineární závislost druhých derivací v navazujících bodech.
- …

Mezi konkrétní typy křivek patří:

- **spline** – Po částech polynomiální interpolační křivka. Spline stupně *n* má spojitost C^{n-1}.
- **Fergusonova kubika** – Nejčastější interpolační křivka, je určena dvěma koncovými body a dvěma tečnými vektory v těchto bodech.
- **Kochanek-Bartels spline** – Interpolační spline křivka, využívá Fergusonovy kubiky pro své jednotlivé segmenty. Je určena *N* body a koeficienty *a,b* a *c* (tenze, šikmost, spojitost) pro každý z nich.
- **Catmull-Rom spline** – Interpolační spline křivka, která vznikne jako Kochanek-Bartels s nulovými koeficienty. Tečný vektor v bodě P_i je rovnoběžný s vektorem P_{i-1},P_{i+1}.



Obrázek 23: Beziérova křivka

- **Beziérovy křivky** – Aproximační křivka procházející koncovými body. Využívá **Bernsteinových polynomů**. Křivka stupně *n* je určena *n + 1* body.
- **Beziérovy kubiky** – Skládá se ze segmentů Beziérových křivek popsaných čtyřmi body. Posun jednoho bodu způsobuje nelokální změnu tvaru.
- **Coonsovy křivky** (kubiky) – Aproximační křivka, neprochází koncovými body.
- **B-spline** – Zobecnění Coonsových křivek.
- **NURBS** (non uniform rational B-spline) – Zobecnění B-spline křivek, je racionální, invariantní vůči lineárním transformacím, umožňuje vkládat body při zachování tvaru, přesně popisovat kuželosečky apod. Jedná se o velmi používanou křivku.

Bonus:

Mezi vyplňovací algoritmy patří např. řádkové vyplňování nebo Pinedův algoritmus (pouze pro konvexní oblasti, je nejpoužívanější pro trojúhelníky, používá tzv. hranovou funkci k určení, zda daný bod leží uvnitř trojúhelníku nebo ne).

odvození Bresenhamova prediktoru:

Vyjdeme z rovnic pro error control DDA:

```
error[i] + dy/dx:
  <  0,5   error[i + 1] := error[i] + dy/dx
  >= 0,5   error[i + 1] := error[i] + dy/dx - 1, y++
```

Nerovnice vynásobíme hodnotou 2 dx a po úpravách (převedení dx na levou stranu) nám vyjdou finální rovnice:

```
2 * dx * error[i] + 2 * dy - dx:
  <  0   error[i + 1] := error[i] + 2 * dy
  >= 0   error[i + 1] := error[i] + 2 * dy - 2 * dx, y++
```

odvození Midpoint prediktoru:

rovnice kružnice: x^2 + y^2 - R^2 = 0

Zavedeme f(x,y) = x^2 + y^2 - R^2

Prediktor definujeme jako p[i] = f(x[i] + 1, y[i] - 1/2) = (x[i] + 1)^2 + (y[i] - 1/2)^2 - R^2

p[i] vyjde pro bod nad kružnicí kladné nebo nula (v tom případě se posuneme dolů ve směru y), jinak záporné, proto:

p[i] < 0 => y[i + 1] = y[i] // rovnice 1
p[i] >= 0 => y[i + 1] = y[i] - 1 // rovnice 2

Rekurentní vzorce odvodíme takto: podle vzorce pro p[i] platí pro p[i + 1]:

p[i + 1] = (x[i + 1] + 1)^2 + (y[i + 1] - 1/2)^2 - R^2

spočítáme rozdíl p[i + 1] a p[i], čímž nám vyjde hodnota, kterou musíme přičíst k p[i], abychom dostali p[i + 1]. Musíme uvažovat 2 případy:

- p[i] < 0: platí:
x[i + 1] = x[i] + 1 // posunujeme se po ose x po 1
y[i + 1] = y[i] // z rovnice 1

takže:

p[i + 1] - p[i] = ... = 2 * x[i] + 3 => p[i + 1] = p[i] + 2 * x + 3

- p[i] >= 0: platí:
x[i + 1] = x[i] + 1 // posunujeme se po ose x po 1
y[i + 1] = y[i] - 1 // z rovnice 2

p[i + 1] - p[i] = ... = 2 * x[i] - 2 * y[i] + 5 => p[i + 1] = p[i] + 2 * x - 2 * y[i] + 5

Počáteční hodnotu prediktoru získáme dosazením x[0] = 0 a y[0] = R do rovnice pro p[i], vyjde nám 5/4 - R, což zaokrouhlíme na 1 - R.

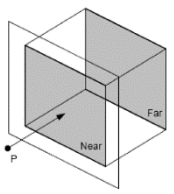
12. Transformace, reprezentace a zobrazení 3D objektů. IZG

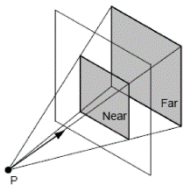
Transformace objektů v prostoru jsou operace, které mění jejich polohu a/nebo tvar. Jedná se např. o posouvání, rotaci, změnu měřítka, zkosení, projekce apod. Transformace jsou velmi frekventované a proto je většinou nutné mít pro ně HW akceleraci.
Transformaci označujeme jako **lineární** v případě, že zachovává lineární kombinace. Tzn. pokud je transformace zobrazení $f: V \rightarrow W$ z jednoho vektorového prostoru do jiného, platí pro libovolný vektor $x = (x_1, x_2, \dots x_n)$ a skaláry $\alpha_1, \alpha_2, \dots \alpha_n$:

$$f(\alpha_1 x_1, \alpha_2 x_2, \dots \alpha_3 x_3) = \alpha_1 f(x_1), \alpha_2 f(x_2), \dots \alpha_3 f(x_3)$$

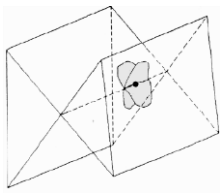
To znamená, že pokud máme množinu bodů, pak nezáleží na tom, jestli je nejdřív vynásobíme a poté transformujeme nebo naopak. Některé transformace, jako např. rotace, jsou lineární, kdežto např. posun nebo perspektivní projekce lineární nejsou!
Důležitým pojmem při práci s transformacemi jsou **homogenní souřadnice**. Ty nám umožňují pracovat s transformacemi, a to i nelineárními, jednotně pomocí matic. Homogenní souřadnice bodu ve 2D s kartézskými souřadnicemi $[x; y]$ je uspořádaná trojice $[X; Y; w]$ pro kterou platí $x = X/w$ a $y = Y/w$. Složku w nazýváme váhou a pro lineární transformace má hodnotu 1. Homogenní souřadnice ve 2D si můžeme představit jako umístění bodů do roviny ve 3D prostoru.
Transformace provádíme pomocí **transformačních matic**, což jsou matice, kterými vynásobíme homogenní souřadnice všech bodů, které chceme transformovat. Příklady transformačních matic základních operací jsou:

- posunutí: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{bmatrix}$
- otočení ve 2D: $\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- změna měřítka: $\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- zkosení: $\begin{bmatrix} 1 & S_{hy} & 0 \\ S_{hx} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$





Obrázek 24: pohledový objem



Obrázek 25: non-manifold objekt

Při provádění více transformací využíváme skládání transformací, které se provádí násobením transformačních matic, jejich výsledkem je jediná matice, která provede všechny transformace. Při násobení matic **záleží** na jejich pořadí – násobíme je zprava (pozpátku).

Ve 3D provádíme transformace obdobně – používáme homogenní 3D souřadnice (tři prostorové + váha – ta je 1 pro bod a 0 pro vektor). Ve 3D je mnohem složitější provádět rotace, pro rotaci okolo každé souřadné osy existuje jiná matice a obecnou rotaci musíme na tyto tři rozkládat. Tento problém řeší tzv. **kvaterniony**.
Pro 3D zobrazení je důležitou transformací **projekce** (promítání) na 2D zobrazovací průmětnu. Existují 2 základní druhy projekcí:

- **perspektivní** (středová) – Promítací paprsky vychází ze středu promítání a proto se zdají vzdálenější objekty menší. Zobrazení odpovídá lidskému vnímání, proto se používá ve hrách, virtuální realitě apod. Je **nelineární**.
- **paralelní** (rovnoběžná) – Promítací paprsky jsou rovnoběžné a proto jsou vzdálené objekty stejně velké jako ty blízké. **Zachovává se rovnoběžnost hran** a proto se využívá především v technickém průmyslu. Je **lineární**.
-

Pohledový objem je část prostoru, která je viditelná pozorovatelem. Ze předu a ze zadu je ohraničen tzv. near a far plochami.
Existuje mnoho způsobů, jak reprezentovat (popsat) 3D objekt. Mezi základní metody patří:

- **hraniční reprezentace** (**B-rep**) – Uchováváme informaci o hranicích (stěnách) objektu. Patří sem např. reprezentace pomocí trojúhelníků nebo splajn ploch. Polygonální modely se uchovávají ve speciální datové struktuře zvané **okřídlená hrana**. Ta uchovává tři lineární seznamy: vrcholů, hran a stěn a rovněž další informace, jako seznam hran pro každou stěnu, sousední vrcholy apod.
- **konstruktivní geometrie** (**CSG**, constructive solid geometry) – Popisujeme objekty pomocí jednoduchých primitiv (válců, krychlí, …). Popis je uložen ve stromu, jehož listy představují primitiva a uzly operace (sjednocení, průnik, …). Používá se především v CAD/CAM systémech (modelování lze snadno parametrizovat).
- **šablonování** – Objekt popisujeme pomocí operace prováděné s primitivem (většinou pohyb 2D profile po trajektorii, např. rotace křivky kolem osy). Může být součástí CSG.
- **dekompoziční modely** – Rozkládáme objekt na základní objemové jednotky (3D pixely, tzv. **voxely**). Jedná se o diskrétní popis. Používá se většinou např. v medicíně, protože nese informace o vnitřní struktuře objektu. Nerovnosti způsobené diskrétní strukturou lze vyhladit metodami **marching cubes** nebo **marching tetrahedra** (nahrazování vzorů voxelů).
- **implicitní plochy** – Množinou elementárních částic, které kolem sebe mají potenciálové pole, popisujeme plochy.
- …

V kontextu reprezentace se bavíme o tzv. **manifold** (vyrobitelných) objektech. Takový object lze zkonstruovat. Musí pro něj platit, že každá hrana sdílí pouze dvě stěny a každý bod stěny má v množině sjednocených stěn otevřené okolí (topologie kruhu). Opakem je **non-manifold** objekt. Zda je objekt manifold nebo non-manifold, lze zjistit pomocí **Eulerových rovnic** pro vrcholy, hrany a stěny objektu.

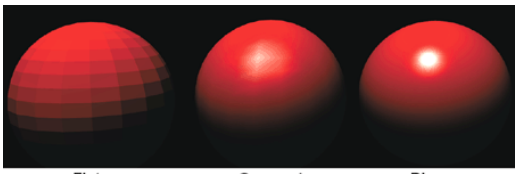
Ať je použita jakákoli reprezentace, při vykreslení se většinou převádí na hraniční trojúhelníkovou reprezentaci. Lze však použít i jiné metody, jako např. ray-tracing bez tohoto převodu. Při zobrazení polygonálních modelů se často využívá tzv. **LOD** (**level of detail**). Jde o techniku, kdy uchováváme více modelů v různé kvalitě (s různým počtem polygonů) a který vybereme, závisí na jeho vzdálenosti od kamery (u vzdálenějších objektů se nižší počet polygonů téměř neprojeví).
Polygonální modely lze zobrazovat několik způsoby, které lze navíc kombinovat. Rychlým způsobem vykreslování je **rasterizace** založená na rasterizaci trojúhelníku. Tato metoda patří mezi **lokální metody** (objekty se vzdájemně neovlivňují) a vyžaduje, aby se ověřovala viditelnost objektů a vzdálenější trojúhelníky se nevykreslily přes bližší. Mezi způsoby řešení viditelnosti patří:

- **malířův algoritmus** – Objekty, popř. trojúhelníky, se seřadí podle vzdálenosti a vykreslují se od nejvzdálenějšího. Problémem je, že pokud se např. objekty protínají, musíme řešit viditelnost na úrovni pixelů, ne pouze objektů nebo trojúhelníků.
- **z-buffer** – Tato metoda je nejpoužívanější a využívá speciální rastrovou paměť, tzv. z-buffer. Z-buffer si pro každý pixel pamatuje nejbližší vykreslenou vzdálenost. Při vykreslování nového pixelu se vždy v z-bufferu ověří, zda je blíž, než doposud nejbližší vzdálenost na dané pozici. Pokud ne, pixel se nevykreslí. Tuto metodu nelze plně využít při částečné průhlednosti objektů.

Pomalejší, ale realističtější metodou zobrazení je **globální metoda** (objekty se vzdájemně ovlivňují) **ray-tracing** (sledování paprsku). Ten se snaží napodobit přirozený způsob vnímání světa pomocí sledování parsků světla. Paprsků je ve skutečnosti nekonečně mnoho a proto je sleduje opačným směrem, tedy od pozorovatele ke zdrojům světla, čímž eliminuje jejich počet. Pro každý pixel zobrazovaného obrazu se vyšle jeden paprsek. Narazí-li primární paprsek na objekt, můžou se vyslat až tři typy dalších paprsků: zrcadlový, lomový a stínový. Stínové paprsky se vysílají ke všem zdrojům světla a zjišťují, kterými z nich je daný bod osvětlen. Zrcadlový a lomový paprsek se vysílá u materiálů, které odráží nebo propouští světlo. Tímto způsobem se může rekurzívně poslat více paprsků až do dané hloubky. Všechny společně potom určí barvu daného pixelu. Nevýhody ray-tracingu jsou:

- pouze ostré stíny
- pouze bodové zdroje světla
- zrcadla odráží obraz, ale neosvětlují okolí
- při změně scény se musí provést celý výpočet znovu

Obrázek 26: stínování



Ray-tracing prvního stupně (bez rekurze) se nazývá **ray-casting**.
Radiozita je globální zobrazovací metoda respektující fyzikální zákony šíření světla, především zákon zachování energie. Scéna se rozdělí na elementární plošky, mezi nimiž se počítá předávání energie (světla). Metoda počítá pouze difúzní osvětlení, proto ji není nutné přepočítávat při pohybu pozorovatele. Umožňuje měkké stíny. Pro její aplikaci musí být scéna energeticky uzavřená a reprezentovaná polygonálně. Neumí řešit průhledné objekty. Lze kombinovat s jinými metodami.

Pro realistické zobrazení se používají **osvětlovací modely**, které aproximují chování světla v reálném světě. Základními jsou:

- **Lambertův** – Počítá s **difůzní složkou** světla, tedy se světlem odraženým do všech směrů rovnoměrně s intenzitou odpovídající kosinu úhlu dopadu paprsku (nezávisí na pozici pozorovatele).
- **Phongův** – Realističtější než Lambertův, odražené světlo je součtem **ambientní** (konstantní, představuje rozptýlené světlo), **difůzní** a **spekulární** složky. Spekulární složka reprezentuje zrcadlové odlesky a závisí jak na úhlu dopadu paprsku, tak na pozici pozorovatele.
-

Stínování je způsob určování barvy pixelů jednotlivých polygonů při rasterizaci (nikoli vrhání stínů!). Většinou zahrnuje osvětlovací model. Mezi základní patří:

- **konstantní** (**flat shading**) – Jednoduchá metoda, každá plocha je vyplněna konstantní barvou určenou osvětlovacím modelem v jejím středu. Plocha se jeví jako po částech lineární.
- **Goraudovo** – Osvětlovací model se vypočítá ve všech vrcholech modelu. Barva každé plochy se potom určí interpolací barev v jejích vrcholech (nedochází k interpolaci normál). Normály ve vrcholech se dají vypočítat jako průměr normál ploch.
- **Phongovo** (neplést s Phongovým osvětlovacím modelem!) – Pomalá ale kvalitní metoda, osvětlovací model se vypočítává v každém rasterizovaném bodě modelu. Je nutné pro každý bod vypočítat normálu interpolací normál ve vrcholech.

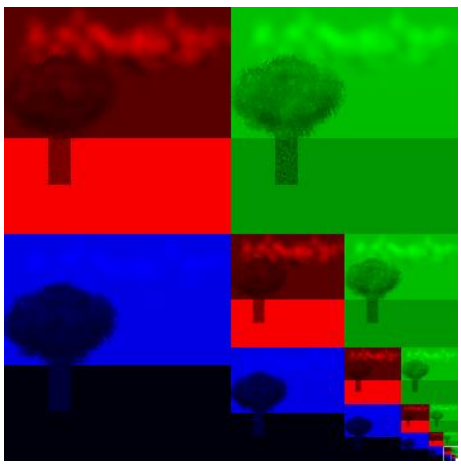
Textury představují způsob, jak přiřadit povrchu objektu nekonstantní vlastnosti. Těmito vlastnostmi jsou např.:

- barva (klasické použití)
- průhlednost
- směr normál (optická změna tvaru povrchu, tzv. **Bump Mapping**)
- skutečná změna tvaru povrchu (tzv. **Displacement Mapping**)
- odraz (obraz vytvořený z okolí, tzv. **Environment Mapping** vhodný pro real-time odrazy)
- odrážení světla (difuze, reflexe)

Slouží k dosažení větší realističnosti. Podle dimenzí dělíme textury na **2D** a **3D** (ty se většinou vytvářejí procedurálně, velmi snadno se mapují na objekt pouze překrytím modelu 3D texturou). Mapování 2D textur na povrch objektu představuje samostatný problém, protože není jednoznačné, často se provádí ručně. Technicky je realizováno přiřazením texturových souřadnic (*u*,*v*) vrcholům modelu.

MIP-mapping je technika uchovávání jedné textury ve více rozlišeních v jedné bitmapě. Důvodem je jednak rychlejší zobrazení při použití textury s menším rozlišením a jednak odstranění aliasu, který může vzniknout, když vzdálený objekt vykreslený malým množstvím pixelů texturujeme bitmapou o vysokém rozlišení.

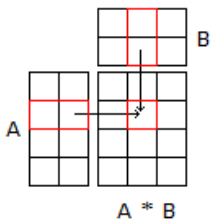
Mezi metody antialiasingu patří supersampling (bere více vzorků na každý pixel) a multisampling (bere více vzorků jen tam, kde je to potřega, např. na hranách).



Obrázek 27: MIP-map textura

Bonus:

Násobení matic: Výsledek součinu matic o rozměrech *m* * *n* (výška * šířka) a *n* * *p* je matice o rozměru *m* * *p*. Záleží na pořadí matic! Provádí se v podstatě skalární součin řádků a sloupců:



Př.:

$$A * B = (1\ 2\ 1) * \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} = (2\ 1\ 1)$$

Rotace kolem obecné osy ve 3D se provádí v několika krocích:

1. posunutí osy do počátku souřadného systému
2. sklopení osy do jedné roviny (např. XY)
3. sklopení roviny do jedné osy (např X)
4. rotace o daný úhel podle této osy
5. navrácení zpět do původní pozice

13. Principy grafických uživatelských rozhraní (komunikační kanály, mody komunikace, systémy řízené událostmi, standardní prvky rozhraní). ITU

Cílem uživatelských rozhraní je zajistit efektivní obousměrnou komunikaci mezi počítačem a člověkem, grafické uživatelské rozhraní (GUI) se snaží usnadnit komunikaci uživateli. Komunikační kanály směrem od stroje k člověku využívají jeho smysly:

- **obraz** (zrak) – Je považován za nejvýhodnější médium pro přenos informace směrem k člověku díky vysoké informační propustnosti a možnosti *náhodného přístupu* člověka k informacím v obraze.
- **zvuk** – Zvuk je vhodné doplňkové médium pro obraz, hodí se pro přenos menšího objemu informací. Nevýhodou oproti obrazu je *sériový přístup* k datům.
- **hmat** – Je využíván jen výjimečně, zejména v aplikacích pro nevidomé. Do budoucna může najít uplatnění ve virtuální realitě.
- **čich**, **chuť** – V současnosti jsou tyto kanály nepoužitelné, protože zatím nelze vůně a chutě syntetizovat v reálném čase. V budoucnu mohou najít uplatnění ve virtuální realitě.

Kamunikace směrem od člověka ke stroji probíhá pomocí těchto kanálů:

- **pohyb** (hmat) – Nejčastější způsob, příkladem je klávesnice a myš.
- **zvuk** (řeč) – Perspektivní, avšak v dnešní době stále ještě málo prakticky použitelný způsob.
- **obraz** (gesta) – Využitelné v některých specifických oblastech, jako např. hry, strojové čtení, apod.

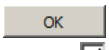
Komunikace může být dále:

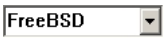
- **aktivní** – Uživatel řídí činnost počítače v širokém slova smyslu, např. pracuje s grafickým editorem (vybírá a používá nástroje, otevírá a zavírá soubory apod.). Tato možnost je efektivní, ale obtížná a může docházet k chybám.
- **pasivní** – Uživatel pouze odpovídá na dotazy počítače (např. instalační průvodce). Tato komunikace je pro uživatele jednoduchá, nenastává mnoho chyb, ale není tak efektivní.

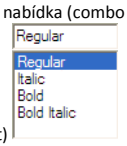
Mod komunikace je stav rozhraní počítače, kdy reaguje jedinečným způsobem na vstupy uživatele. Platí, že čím méně modů se vyskytuje, tím lépe. Příklady modů komunikace jsou: editace textu, zadávání příkazu v příkazovém jazyce, reakce na chybu, odpověď na dotaz počítače apod.

Systémy řízené událostmi jasou takové systémy, které samy z vlastní iniciativy neplní žádnou činnost, ale pouze čekají na výskyt událostí, které daným způsobem obsluhují. Většina moderních operačních systémů funguje na tomto principu, např. systém Windows je založen na systému zpráv, které jsou zasílány oknům při výskytu událostí.

Mezi standardní prvky uživatelských rozhraní patří např.:

- tlačítko (button) 

- zatrhávací box (check box) 
- rozbalovací nabídka (combo box) 

- seznam (list) 

- rádiové tlačítko (radio button) 
- textové pole (text field) 

Dialogové okno je okno, které slouží buď k informování uživatele nebo k výzvě na jeho odpověď (např. potvrzení akce). Dělí se na:

- **modální –** Prioritní, uživatel musí odpovědět, než může pokračovat v práci (zbytek aplikace je blokován).
- **nemodální –** Neprioritní, uživatel může dále pracovat s aplikací a na dialog odpovědět později.
- **systémově modální –** Modální v rámci všech aplikací, bez odpovědi nelze pracovat s žádnou aplikací.

Bonus:

WPF (Windows presentation foundation) je grafický subsystém pro Windowsovské aplikace, používá DirectX, snaží se oddělit popis rozhraní od programu pomocí jazyka podobného XML (XAML).

14. Spektrální analýza spojitých a diskrétních signálů. ISS

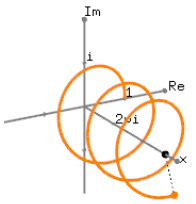


Figure 13: graf komplexní

Spektrální analýza se zabývá spektry signálů, tedy jejich reprezentací ve frekvenční oblasti.
Komplexní číslo můžeme zapsat ve tvaru:

$$z = re^{i\phi}$$

kde *r* je délka vektoru a *Φ* je jeho úhel s reálnou osou. Pokud je jeho velikost 1 a úhel zavedeme jako nezávislou proměnnou, získáme komplexné exponenciálu:

$$y = e^{ix}$$

Dále platí:

$$\cos(x) = \frac{e^{ix}+e^{-ix}}{2}.$$

Pro násobený a posunutý kosinus platí:

$$C_1\cos(\omega_1 t + \phi_1) = \frac{C_1}{2}e^{j\phi_1}e^{j\omega_1 t} + \frac{C_1}{2}e^{-j\phi_1}e^{-j\omega_1 t} \qquad c_1 = \frac{C_1}{2}e^{j\phi_1}, c_{-1} = \frac{C_1}{2}e^{-j\phi_1}.$$

Pomocí **Fourierovy řady** (**FŘ**) můžeme rozložit libovolný spojitý periodický signál na řadu (tj. součet) komplexních exponenciál (které dají pro reálný signál součet funkcí kosinus), tedy:

$$x(t) = \sum_{k=-\infty}^{+\infty} c_k\, e^{ik\omega_1 t}$$

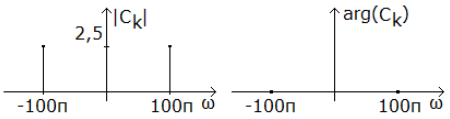
kde c_k je komplexní koeficient dané exponenciály, ω_1 základní kruhová frekvence a k = 0, +-1, +-2, … +-∞. Pro reálné signály jsou koeficienty c_k a c_{-k} vždy komplexně sdružené (zrcadlový obraz podle osy *x*), tzn. $c_k = c^*_{-k}$ (c_0 je reálný, protože je komplexně sdružený sám se sebou). Koeficienty Furierovy řady spočítáme pomocí vzorce:

$$c_k = \frac{1}{T}\int_{T_1} x(t)e^{-ik\omega_1 t}dt$$

Když známe koeficienty, můžeme signál zakreslit v tzv. **frekvenční oblasti** (v závisloti na frekvenci, v závisloti na čase mluvíme o **časové oblasti**). Tento graf nazýváme **spektrem** signálu.

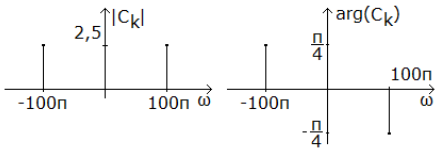
Příklad: $x(t) = 5\cos(100\pi t) \Rightarrow \omega_1 = 100\pi, c_1 = \frac{5}{2}e^0 = 2{,}5, c_{-1} = \frac{5}{2}e^0 = 2{,}5$

$$x(t) = 2.5e^{i100\pi t} + 2.5e^{-i100\pi t}$$



Příklad 2: $x(t) = 5\cos\left(100\pi t - \frac{\pi}{4}\right) \Rightarrow \omega_1 = 100\pi, c_1 = 2{,}5e^{-j\frac{\pi}{4}}, c_{-1} = 2{,}5e^{j\frac{\pi}{4}}$

$$x(t) = 2{,}5e^{-j\frac{\pi}{4}}e^{i100\pi t} + 2{,}5e^{j\frac{\pi}{4}}e^{-i100\pi t}$$

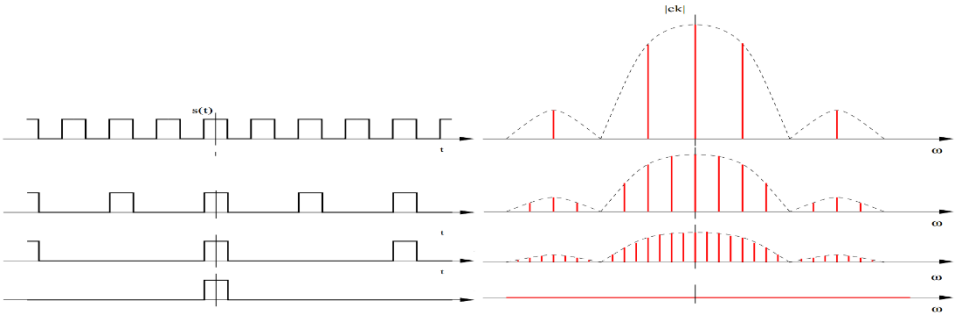


Fourierova transformace (**FT**) transformuje libovolný (ne pouze periodický) signál se spojitým časem *x(t)* na spektrální funkci *X(jω)*, tzv. **obraz**. Jelikož signál není periodický, bude u FT spektrum spojité, narozdíl od FŘ, kde bylo spectrum diskrétní díky tomu, že byl signal periodický. FT se vypočítá jako:

$$X(j\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt$$

Zpětná FT se provádí jako:

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(j\omega)e^{j\omega t}d\omega$$



Obrázek 28: přechod od FŘ k FT

Fourierova transformace s diskrétním časem (**DTFT**) je FT pro vzorkované signály. Jelikož je signal diskrétní, bude spectrum periodické (podobně jako u periodického signálu je spectrum diskrétní). Vypočítá se jako:

$$\tilde{X}(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-k\omega n}$$

Diskrétní Fourierova řada (**DFŘ**) transformuje diskrétní periodický signál s peridodou *N* na koeficienty, kterých je nekonečně mnoho a jsou rovněž periodické s peridodou N. Vypočítá se:

$$\tilde{X}[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}$$

Diskétní Fourierova transformace (**DFT**, neplést s DTFT!) převádí *N* vzorků signálu na *N* vzorků spectra s použitím DFŘ tak, že jednu jednu periodu signálu reprezentuje jednou periodou DFŘ. Praktický postup je:

- vstupní signál periodizujeme: $\tilde{x}[n] = x[mod_N(n)]$
- najdeme koeficienty DFŘ $\tilde{X}[k]$
- výsledek omezíme okénkovou funkcí: $X[k] = R_N[k]\tilde{X}[k]$

operaci značíme $x[n] \overset{DFT}{\rightarrow} X[k]$. Zpětný postup se značí $X[k] \overset{DFT^{-1}}{\rightarrow} x[n]$ a provádí se opětovnou periodizací spektra, inverzní DFŘ a opětovným omezením okénkovou funkcí.

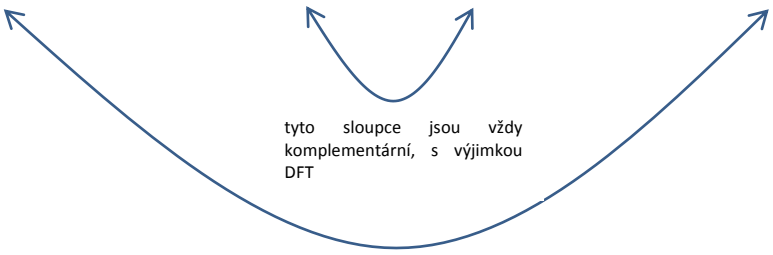
Při vzorkování se chceme zbavit času a počítat pouze s posloupností čísel udávánou číslem vzorku *n*, nikoli přímo hodnotou *nT* (kde *T* je vzorkovací perioda). Proto přecházíme k tzv. **normovanému času** $n' = \frac{nT}{T}$; jakoby určíme vzorkovací periodu rovnu 1. V takovém případě ale musíme přepočítat i frekvence: $f' = \frac{f}{F_s}$ a $\omega' = \frac{\omega}{F_s}$ (kde F_s je vzorkovací frekvence). Čárky u normovaných veličin se často nepíšou a normovaná veličina se ve vzorcích pozná tak, že vedle ní nestojí vzorkovací perioda *T*. Tvar spektra se těmito operacemi nemění, mění se pouze hodnoty na ose *x*.

Rychlá Fourierova transformace (**FFT**) je algoritmus pro rychlý výpočet DFT a zpětné DFT. Zatímco přímé vyhodnocování DFT má kvadratickou složitost, FFT lze počítat se složitostí logaritmickou. Existuje i vícerozměrná varianta.

Shannonův (Nyquistův) **teorém** říká, že aby nedocházelo k aliasingu, musí být vzorkovací frekvence alespoň dvakrát větší, než nejvyšší frekvence ve vzorkovaném signálu.

shrnující tabulka:

transformace	signál		spektrum	
	spojité	periodické	spojité	periodické
FŘ (Fourierova řada)	ano	ano	ne	ne
FT (Fourierova transformace)	ano	ne	ano	ne
DTFT (Fourierova transformace s diskrétním časem)	ne	ne	ano	ano
DFŘ (diskrétní Fourierova řada)	ne	ano	ne	ano
DFT (diskrétní Fourierova transformace)	ne	ne	ne	ne



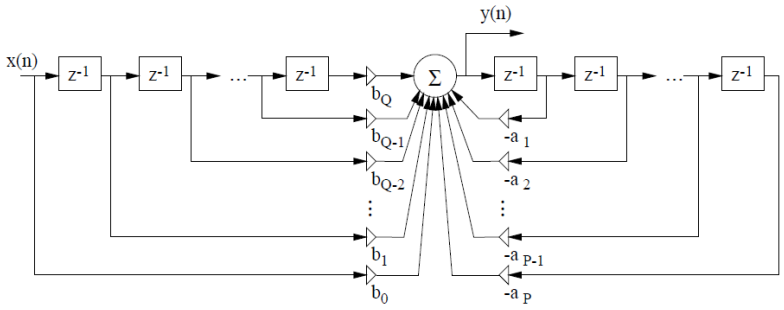
tyto sloupce jsou vždy komplementární, s výjimkou DFT

15. Číslicové filtry (diferenční rovnice, impulsní odezva, přenosová funkce, frekvenční charakteristika). ISS

Filtr je systém, který upravuje spektrum signálu. **Diferenční rovnice** je rovnice, která popisuje n-tý člen posloupnosti jako funkci jiných členů posloupnosti. **LTI** (**lineárně časově invariantní**) číslicové filtry dělíme na:

- **finite impulse response** (**FIR**) – Impulzní odezva je konečná. Tyto filtry se nazývají **nerekurzívní**.
- **infinite impulse response** (**IIR**) – Impulzní odezva je nekonečná. Tyto filtry se nazývají **rekurzívní**. Využívá se zpožděný výstup filtru, který se přivádí na jeho vstup – vzniká **zpětná vazba**.

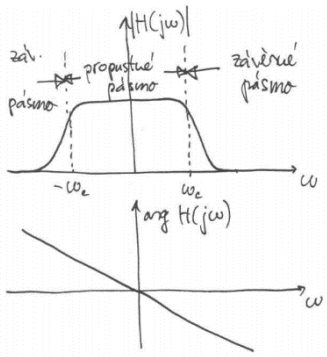
Schéma obecně rekurzivního filtru je následující (využívá bloky zpoždění, násobení a součtu) (nerekurzívní filtr nemá pravou stranu obrázku):



Jeho výstup lze zapsat diferenční rovnicí:

$$y[n] = \sum_{k=0}^{Q} b_k x[n-k] - \sum_{k=1}^{P} a_k y[n-k]$$

kde *x* je vstup a *y* výstup.



Obrázek 29: kmitočtová charakteristika filtru dolní propusť

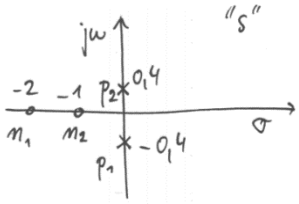
Přenosová funkce je funkce:

$$H(s) = \frac{Y(s)}{X(s)}$$

kde *Y(s)* a *X(s)* jsou funkce získané Laplaceovou transformací ze vstupu a výstupu (*s* je komplexní číslo). Je-li *s* tvaru $j\omega$, pak dostáváme funkci, kterou nazýváme **kmitočtová** (frekvenční) **charakteristika**. Kmitočtová charakteristika definuje, co se stane se spektrem vstupního signálu: $Y(j\omega) = H(j\omega).X(j\omega)$.

Funkce *H(s)* se dá zapsat jako podíl polynomů. Kořeny čitatele tohoto zlomku označujeme jako **nuly** (stahují hodnotu zlomku k 0), kořeny jmenovatele označujeme jako **póly** (stahují hodnotu zlomku k nekonečnu). Příklad:

$$H(s) = \frac{s^2 + 3s + 2}{s^2 + 0.16} = \frac{(s+2)(s+1)}{(s+j0.4)(s-j0.4)}$$



Kauzální systém (nevidí do budoucnosti) je stabilní, pokud všechny póly leží v levé polorovině komplexní roviny.

Laplaceova transformace (**LT**) je transformace převádějící funkci reálné proměnné na funkci komplexní proměnné. Je vhodná např. pro analýzu stability signálu (viz výše). Vypočítá se jako:

$$X(z) \int_{-\infty}^{\infty} x(t) e^{-zt} dt$$

Pokud bereme $z = j\omega$, dostáváme Fourierovu transformaci. Obdobou Laplaceovy transformace pro diskrétní signály je **z-transformace**.

Výstup **LTI** (lineárního, časově invariantního systému) dostaneme konvolucí impulzní odezvy *h[n]* a vstupního signálu *x[n]*:

$$y[n] = h[n] * x[n] = \sum_{k=-\infty}^{\infty} x[k] h[n-k]$$

V oblasti spektra se konvoluce mění na násobení. To je výhodné, protože postup výpočtu koeficientů FŘ, jejich vynásobení a zpětný převod do časové oblasti je rychlejší než konvoluce.

Reakce LTI systému na komplexní exponenciálu je ta samá exponenciála, jenom násobená nějakým číslem.

Filtry lze zobecnit na 2D **konvoluční filtry**. Máme dánu matici (tzv. **konvoluční jádro**, v podstatě odpovídá impulzní odezvě v 1D), kterou přikládáme postupně středem na každý vzorek, vynásobíme hodnoty pod sebou a všechno sečteme, čímž získáme novou hodnotu vzorku. Takto lze realizovat různé operace, jakými jsou např. rozostření (např. matice 3 * 3, kde každý prvek má hodnotu 1/9), zostření, detekce hran apod. Pro zachování jasu obrazu by měla být suma všech prvků konvolučního jádra rovna 1.

Bonus:

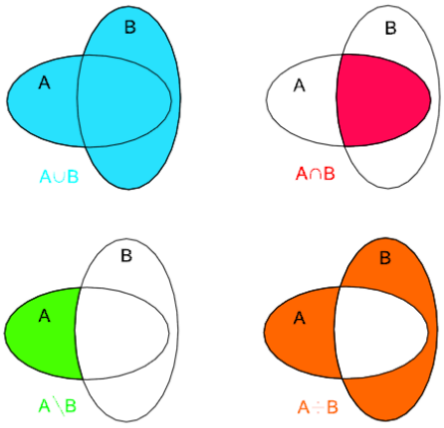
vzorce pro konvoluci:

- diskrétní: $(f * g_N)[n] = \sum_{i=0}^{N-1} f[i]\, g[n-i]$
- spojitá: $(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) h(t-\tau) d\tau$

16. Množiny, relace a zobrazení. IDA

Množina je určitý soubor objektů, které se nazývají **prvky** množiny (tyto prvky množinu jednoznačně určují). Je potřeba si uvědomit, že ne každý soubor objektů je třída – pokud bychom např. chtěli zavést množinu *M* obsahující všechny množiny, které neobsahují samy sebe, pak bychom se vždy dostali ke sporu při rozhohdování, zda množina *M* sama sebe obsahuje – pro tyto případy zavádíme pojem **třídy**, každá množina je třída, ale ne každá třída je množina. S množinami lze provádět operace jako např. **sjednocení**, **průnik**, **rozdíl**, **doplněk**, **kartézský součin** ($X \times Y =$ množina uspořádaných dvojic všech možných kombinací (prvek z *X*, prvek z *Y*)), **doplněk** do jiné množiny apod. Pro operace s množinami platí:

- **komutativnost**: $A \cup B = B \cup A$ $A \cap B = B \cap A$
- **asociativita**: $(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
- **distributivnost**: $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$ $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
- **De Morganovy zákony**: $\overline{A \cup B} = \bar{A} \cap \bar{B}$ $\overline{A \cap B} = \bar{A} \cup \bar{B}$



Obrázek 30: množinové operace

Multimnožina je množina, v níž se prvky mohou vyskytovat víckrát než jednou. Velikost množiny (tzv. **mohutnost**) je u konečných množin dána počtem jejích prvků. Velikost nekonečných množin potom porovnáváme podle vzájemně jednoznačného zobrazení z jedné do druhé. V tomto smyslu existují nekonečně velké množiny:

- **spočetné** – Mají mohutnost množiny celých, přirozených nebo např. racionálních čísel (všechny tyto množiny lze vzájemně jednoznačně zobrazit).
- **nespočetné** – Mají větší mohutnost než množiny spočetné, patří sem např. reálná a komplexní čísla.

Množina s největší mohutností neexistuje, pro každou množinu existuje množina s větší mohutností.

Potenční množina je množina všech podmnožin dané množiny (včetně prázdné množiny a množiny samotné) a počet jejích prvků je $2^{\text{velikost množiny}}$ (značí se 2^M pro množinu *M*).

Teorie množin dále zavádí pojem **relace** jako model vztahů mezi prvky množin, přičemž sama relace je definována jako množina libovolného množství orientovaných vztahů mezi libovolným počtem prvků. Většinou však pojmem relace rozumíme relaci binární, která udává vztahy mezi prvky dvou množin *A* a *B*, přičemž se může jednat o jednu a tu samou množinu. Binární relaci formálně definujeme jako podmnožinu kartézského součinu:

$$[A, B, R]; R \subseteq A \times B$$

Kde *A* je množina nazývaná **definiční obor**, *B* je množina nazývaná **obor hodnot** a *R* je daná relace. Co se týká vlastností relací, rozlišujeme tyto pojmy:

- **symetrická** relace – $(x\ R\ y) \Rightarrow (y\ R\ x)$ (dva prvky buď nejsou v relaci nebo je v relaci první s druhým i druhý s prvním)
- **tranzitivní** relace – $(x\ R\ y) \wedge (y\ R\ z) \Rightarrow (x\ R\ z)$ (pokud je v relaci jeden prvek s druhým a druhý s třetím, pak je v relaci i první s třetím)
- **reflexivní** relace – $\forall x \in X: (x\ R\ x)$ (každý prvek je v relaci sám se sebou)
- **antisymetrická** relace – $(x\ R\ y) \wedge (y\ R\ x) \Rightarrow x = y$ (pozor: relace může být zároveň symetrická i antisymetrická)

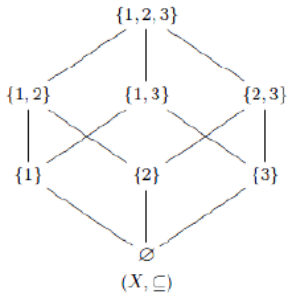
Inverzní relace vznikne prohozením pořadí prvků (podobné inverzní funkci).

Binární relace je výhodné znázorňovat tabulkami.

Pro relaci *R* z *X* do *Y* dále definujeme pojmy:

- $Dom\ R$ (definiční obor) – $\{x | existuje\ y\ \in Y, že\ (x, y) \in R\}$
- $Im\ R$ (obor hodnot) – $\{y | existuje\ x\ \in X, že\ (x, y) \in R\}$

Relace (částečného) **uspořádání** je relace, která je reflexivní, antisymetrická a tranzitivní. Množinu s relací uspořádání nazýváme **uspořádanou** a prvky, které jsou v relaci, např. (*x*,*y*) zapisujeme pomocí operátoru: $x \leq y$ (popř. $x < y$, pokud $x \neq y$). V takovéto relaci nemusí obecně být možné porovnávat jakékoli dva prvky (proto mluvíme o uspořádání částečném)! Příkladem může být uspořádání inkluzí, kdy $x \leq y$ právě tehdy, když je x podmnožinou y. Toto lze znázornit tzv. Hasseovým diagramem:



Úplné (nebo lineární) uspořádání je uspořádání, v němž lze porovnat každé dva prvky, tzn. pro každé *x* a y *z* množiny platí buď $x \leq y$ nebo $y \leq x$. Úplné uspořádání je definováno např. na množině celých čísel.

Relace **ekvivalence** je relace, která rozděluje množinu na vzájemně disjunktní podmnožiny ekvivalentních prvků (tzn. tyto podmnožiny mají prázdný průnik). Tyto množiny nazýváme třídami rozkladu. Relace ekvivalence je relace, která je právě reflexivní, symetrická a tranzitivní. Příkladem může být relace mezi přirozenými čísly, které dávají stejný zbytek po dělení určitým číslem.

Zobrazení je zobecněný pojem funkce a v teorii množin jej definujeme jako relaci f z množiny *X* do množiny *Y*, přičemž pro každé $x \in Dom\ f$ existuje právě jedno $y \in Y$, že $(x, y) \in f$. Speciální druhy zobrazení jsou:

- **prosté** (injektivní) – $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$ (Každý prvek v *Y* má nejvýše jeden vzor.)
- **surjektivní** (na) – pro každé $y \in Y$ existuje $x \in X$, že $f(x) \in Y$(Pro každý prvek *Y* existuje vzor v *X*.)
- **vzájemně jednoznačné** – *Dom f = X* a zároveň *f* je zároveň prosté i surjektivní.

Zobrazení a potažmo i relace můžeme skládat. Máme-li relace $R \subseteq X \times Y$ a $S \subseteq Y \times Z$, pak jejich složení $S \circ R$ (čteme *S* po *R*) definujeme takto:

$$S \circ R = \{(x, z) | \exists\ y \in Y: (x, y) \in R \wedge (y, z) \in S\}$$

17. Diferenciální a integrální počet funkcí více proměnných. IMA

Derivace funkce udává směrnici tečny ke grafu funkce v každém jejím definovaném bodě (neboli rychlost změny funkce). Derivace funkce *f(x)* se značí *f'(x)*. Funkci *f'(x)* nazýváme **primitivní funkcí** k funkci *f(x)*. Základní vztahy pro derivace jsou:

součet: $\big(af(x)+bg(x)\big)' = af'(x)+bg'(x)$
součin: $\big(f(x)g(x)\big)' = f'(x)g(x)+f(x)g'(x)$
podíl: $\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x)-f(x)g'(x)}{g^2(x)}$
složená funkce: $f(x)=h\big(g(x)\big) \Rightarrow f'(x)=h'\big(g(x)\big)g'(x)$

f(x)	*f'(x)*
c (konstanta)	0
x^c	cx^{c-1}
e^x	e^x
$\log_a x$	$\frac{1}{xln(a)}$
$\sin(x)$	$\cos(x)$
$\cos(x)$	$-\sin(x)$
c^x	$c^x\ln(c)$
$tg(x)$	$\frac{1}{\cos^2(x)}$

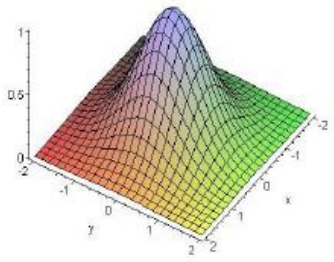


Figure 14: funkce dvou proměnných

Opačnou operací k derivaci je **integrování**. Existují dva typy integrálu: **neurčitý** a **určitý**. Výsledkem neurčitého integrálu dané funkce je množina jejích primitivních funkcí. Určitý integrál provádíme na určitém intervalu a jeho výsledkem je plocha vymezená grafem funkce a osou *x* na daném intervalu (přičemž plocha pod osou *x* je záporná). Určitý integrál se dá počítat přibližně numericky. Analytická integrace je náročnější než integrování, protože neexistuje způsob, jak obecně integrovat součin, složenou funkci apod. Pro integrály platí:

součet: $\int_m^n (af(x)+bg(x))dx = a\int_m^n f(x)dx + b\int_m^n g(x)dx$

f(x)	$\int f(x)dx + C$
c (konstanta)	*cx*
x^n	$\frac{x^{n+1}}{n+1}$
$\frac{1}{x}$	$\ln\lvert x\rvert$
$\sin(x)$	$-\cos(x)$
$\cos(x)$	$\sin(x)$
a^x	$\frac{a^x}{\ln(a)}$

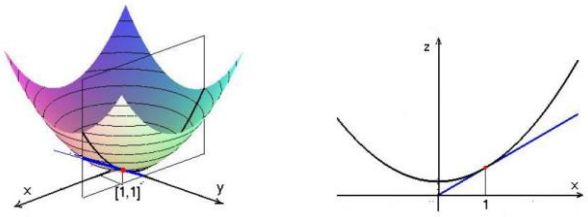


Figure 15: geometrický význam derivace funkce více proměnných

Při analytickém integrování často využíváme metody **substituce** a **per partes**. Substituce je vhodná pro integraci vnořených funkcí a využívá možnosti nahradit část výrazu novou proměnnou dle vzorce:

$$\int f(x)dx = \int f\big(g(t)\big)g'(t)dt$$

Tedy například:

$$\int tan\,(x)dx = \int \frac{\sin(x)}{\cos(x)}dx = \left[\begin{matrix} t=\cos(x) \\ dt=-\sin(x)\,dx \end{matrix}\right] = \int -\frac{\sin(x)}{t}\frac{dt}{\sin(x)} = -\int \frac{1}{t}dt = -\ln|t| + c_t = -\ln|\cos(x)| + c$$

Při počítání určitého integrálu substituční metodou je nutné přepočítat meze jejich dosazením do substituce.
Metoda per partes (po částech) se využívá pro integrály se součinem. Využívá platnosti vztahu:

$$\int u(x)v'(x)dx = u(x)v(x) - \int u'(x)v(x)dx$$

Například:

$$\int xsin(x)dx = \left[\begin{matrix} u=x & v=-\cos(x) \\ u'=1 & v'=\sin(x) \end{matrix}\right] = x(-\cos(x)) - \int -\cos(x)\,.\,1dx = -xcos(x) + \int \cos(x)\,dx = -xcos(x) + \sin(x) + c$$

Limita je hodnota, ke které se funkce v daném bodě blíží. Pokud je funkce v daném bodě definovaná, pak je funkční hodnota zároveň limitou, limita však nemusí vždy existovat, pokud funkce v bodě definovaná není nebo může záležet na směru, kterým se k bodu přibližujeme (tzv. limita zleva nebo zprava).

L'Hospitalovo pravidlo je pravidlo pro výpočet limit, které za určitých podmínek umožňuje vypočítat limitu podílu funkcí jako limitu podílu jejich derivací:

$$\lim_{x\to a}\frac{f(x)}{g(x)} = \lim_{x\to a}\frac{f'(x)}{g'(x)}$$

Tento vztah lze použít, pouze pokud je limita typu $\frac{0}{0}$ nebo $\frac{cokoliv}{\pm\infty}$.

Funkce více (*n*) reálných proměnných je zobecnění funkce jedné proměnné takové, že se jedná o zobrazení $\mathbb{R}^n \to \mathbb{R}$ (tedy z množiny uspořádaných *n*-tic do množiny reálných čísel).
Pokud se bavíme o derivacích funkce o více proměnných, mluvím o tzv. **parciálních derivacích**. Parciální derivaci vztahujeme k určité proměnné, u funkce *f*(*x*,*y*) tedy mluvíme o parciální derivaci podle proměnné *x* nebo *y*. Parciální derivaci funkce *f*(*x*,*y*) v bodě (x_0,y_0) definujeme jako limitu:

$$f'_x(x_0,y_0) = \frac{\delta f}{\delta x}(x_0,y_0) = \lim_{h\to 0}\frac{f(x_0+h,y_0)-f(x_0,y_0)}{h}$$

Analogicky se definují parciální derivace funkce více než dvou proměnných. Prakticky se výpočet provádí jako běžná derivace funkce jedné proměnné s tím, že proměnnou, podle níž derivujeme, bereme jako proměnnou a ostatní proměnné bereme jako konstanty.
Z geometrického hlediska je parciální derivace směrnicí (tangens úhlu) tečny ke grafu funkce, která vznikne řezem roviny, která je množinou bodů, kde je hodnota proměnné, podle níž derivujeme, konstantní a odpovídající bodu, v němž derivujeme (když např. derivujeme podle proměnné *x* v bodě (1,2), jde o rovinu *x* = 1).
Jako příklad uveďmě všechny parciální derivace funkce:

$$f(x,y,z) = xy^2 + 3x^3z + z^4 + 2xyz$$

jimiž jsou:

$$f(x,y,z)' = (y^2 + 9x^2z + 2yz, 2xy + 2xz, 3x^2 + 4z^3 + 2xy)$$

Předchozí tvar zápisu se nazývá **gradient** a jedná se o vektor udávající nejrychlejší směr růstu funkce.

Zobecněním integrálu na více dimenzí dostáváme **dvojný**, **trojný** atd. integrál určující objem pod grafem funkce. Platí: $\int_a^b\int_c^d f(x,y)dx\,dy = \int_a^b\left(\int_c^d f(x,y)\right)dy$.

První derivace funkce jedné proměnné udává, zda je funkcu rostoucí (> 0), klesající (< 0) nebo konstantní (= 0). Druhá derivace udává, zda je funkce konvexní (< 0, tvar U), konkávní (> 0) nebo zda má v daném bodě inflexní bod (změna mezi konvexní a konkávní). Takto lze zjišťoval lokální extrémy – minima (f' = 0 a f'' < 0) a maxima (f' = 0 a f' > 0).

Integrály a derivace mohou být nevlastní resp. v nevlastním bodě, pokud je jejich výsledkem $\pm\infty$ resp. pokud jsou prováděny v bodě $\pm\infty$. Takovéto situace se řeší pomocí limit.

Diferenciál určuje změnu funkce v bodě a vypočítá se jako: $dy = f'(x)\,dx$.

18. Číselné soustavy a převody mezi nimi. INP, IAS

Číselná soustava je způsob zápisu čísla. Rozlišujeme dva hlavní typy soustav:

- **polyadické** (**radix**) – Mají určitý základ, tzv. radix), který udává počet číslic v ní použitých. Základ musí být celé číslo větší nebo rovno jedné. Číslo vyjadřujeme jako součet násobků mocnin základu. Mezi nejvýznamější soustavy patří nejpoužívanější desítková soustava. V informatice se nejvíce uplatňují dvojková (binární), šestnáctková (hexadecimální) a osmičková (oktalová) soustava, protože korespondují s binárním modelem počítače. Obecný vztah mezi zápisem čísla v soustavě o základu *z* a hodnotou čísla je:

$$(\dots a_2 a_1 a_0, a_{-1} a_{-2} a_{-3} \dots)_z = \dots + a_2 . z^2 + a_1 . z^1 + a_0 . z^0 + a_{-1} . z^{-1} + a_{-2} . z^{-2} a_{-3} . z^{-3} + \dots$$

- **nepolyadické** – Nemají základ, patří sem např. římské číslice nebo soustava zbytkových tříd.

Převod z desítkové soustavy do soustavy o základu *n* provádíme rozdílně pro celou a necelou část. Celou část převádíme tak, že dané číslo celočíselně dělíme číslem *n* tak dlouho, dokud nám nevyjde 0. V každém kroku si při tom zapísujeme zbytek po dělení. Zápis zbytků je po ukončení algoritmu zápisem čísla v soustavě o základu *n*, přičemž nejméně významná je první zapsaná číslice a každá další je významější než předchozí zapsaná. Jako příklad uveďme převod čísla $(173)_{10}$ do dvojkové soustavy:

krok	výsledek dělení dvěma	zbytek
0	173	
1	86	1
2	43	0
3	21	1
4	10	1
5	5	0
6	2	1
7	1	0
8	0	1

Výsledkem je tedy číslo $(10101101)_2$. Necelou část převádíme naopak násobením základem. Při každém překročení hodnoty 1 vezmeme celou část čísla jako převedenou číslici a pokračujeme opět pouze s necelou částí. Převod není obecně konečný! Příkladem může být převod čísla $(0{,}453)_2$ do dvojkové soustavy:

krok	výsledek násobení dvěma	číslice
0	0,453	0
1	0,906	1
2	0,812	1
3	0,624	1
4	0,248	0
5	0,496	0
6	0,992	1
7	...	...

Výsledkem je číslo 0,0111001, avšak toto číslo je pouze aproximací na daném počtu bitů, není přesné! Chyba při tomto převodu se nazývá **truncation error**. Převod ze soustavy o základu *n* do soustavy desítkové můžeme provést jednoduše dle výše zmíněné definice. Jako příklad ukažme převod čísla 10101101 zpět do desítkové soustavy:

$$(10101101)_2 = 1 + 0 \times 2 + 1 \times 4 + 1 \times 8 + 0 \times 16 + 1 \times 32 + 0 \times 64 + 1 \times 128 = 173$$

V informatice navíc můžeme využívat výhodný převod mezi dvojkovou a šestnáctkovou soustavou. Jedna hexadecimální číslice odpovídá čtyřem binárním číslicím, proto když se naučíme nazpaměť tabulku převodu šestnácti hexadecimálních číslic na binární a zpět, můžeme snadno převádět mezi těmito soustavami z hlavy. Podobnou výhodu lze využít i u soustavy osmičkové, v níž jedna číslice odpovídá třem binárním číslicím. Pro ilustraci uveďme příklad převodu:

$$\left(\underline{1\,0\,1\,0}\,\underline{1\,1\,0\,1}\right)_2 = (\underline{A}\,\underline{D})_{16} = 173$$

Soustava zbytkových tříd (RNS, residue number system) je uspořádaná *k*-tice vzájemně různých prvočísel (základů) $z_0, \dots z_{k-1}$. Mějme např. tyto základy:

z0 = 2 zbytkové třídy: 0, 1
z1 = 3 zbytkové třídy: 0, 1, 2
z2 = 5 zbytkové třídy: 0, 1, 2, 3, 4

Číslo 5 potom vyjádříme jako trojici zbytků po dělení základy jako (1, 2, 0). Jednoznačně lze vyjádřit pouze čísla, která jsou menší než součin všech základů (tzv. peridoa), v našem případě $2 . 3 . 5 = 30$. Soustava zbytkových tříd umožňuje rychlé sčítání, odčítání a násobení (jednoduše s operací modulo), protože se neuplatňují přenosy. Dělení však není jednoznačně definované a rovněž není možné jednoduše čísla porovnávat. Převod z této soustavy zpět může probíhat např. lineárním testováním všech čísel, zda vyhovují danému zápisu.

19. Boolovy algebry. IDA

Algebra je n-tice:

$$(A, \omega_1, \omega_2, \dots \omega_k)$$

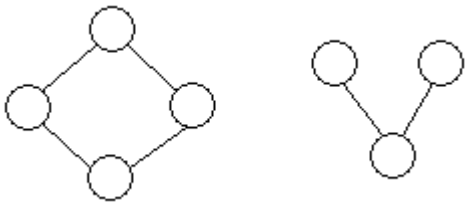
kde *A* je množina a ω_1 až ω_k jsou operace s četnostmi (aritami) n_1, n_2, … n_k (tedy zobrazení z množiny *A* na danou četnost do množiny *A*). **Boolova algebra** je algebra:

$$(X, \oplus, \odot, ', 0, 1)$$

kde $\oplus$ a $\odot$ jsou binární operace, $'$ je unární operace (vrací komplementární prvek) a 0 a 1 jsou nulární operace (vrací minimální resp. maximální prvek). Booleova algebra zachycuje podstatné vlastnosti množinových a logických operací. Booleova algebra musí splňovat následující axiomy:

1. $(x \oplus y) \oplus z = x \oplus (x \oplus z), \quad (x \odot y) \odot z = x \odot (x \odot z)$ (**asociativita**)
2. $x \oplus y = y \oplus x, \quad x \odot y = y \odot x$ (**komutativita**)
3. $x \odot (y \oplus z) = (x \odot y) \oplus (x \odot z), \quad x \oplus (y \odot z) = (x \oplus y) \odot (x \oplus z)$ (**distributivita**)
4. $x \oplus 0 = x, \quad x \odot 0 = x$ (**neutralita 0 a 1**)
5. $x \oplus x' = 1, \quad x \odot x' = 0$

Lze taky zavést **Booleovský svaz**, což je distributivní (platí distributivita) komplementární (pro každý prvek existuje komplement splňující axiom č. 5) svaz. V booleovském svazu představují operace $\oplus$ a $\odot$, značené $\vee$ a $\wedge$, operace infima a suprema (zobecnění nejmenšího/největšího prvku množiny, např. otevřený interval nemá největší a nejmenší prvek, ale má supremum a infimum). **Supremum** podmnožiny je nejmenší prvek takový, že je větší nebo roven všem prvkům této podmnožiny (jde o zobecněný pojem maximálního prvku). **Infimum** je obdobně největší prvek takový, že je menší nebo roven všem prvkům z podmnožiny. Svaz je množina uspořádaná relací (částečného) uspořádání, která pro každou dvouprvkovou podmnožinu obsahuje i její superemum a infimum.



Obrázek 31: svaz vs nesvaz

Počet prvků Boolovy algebry je 2^n. V informatice je důležitá především Boolova algebra se dvěma prvky, protože reprezentuje binární logiku (operace algebry odpovídají operacím *OR*, *AND*, *NOT*, *FALSE* a *TRUE*). V takovéto algebra lze odvodit platnost dalších rovnic:

- komplementarita – $x \vee \bar{x} = 1$
- nedegenerovanost – $0 \neq 1$
- absorbce – $x \vee (x \wedge y) = x, x \wedge (x \vee y) = x$
- agresivita nuly – $x \wedge 0 = 0$
- agresivita jedničky – $x \vee 1 = 1$
- idempotence – $x \vee x = x, x \wedge x = x$
- absorbce negace – $x \vee (\bar{x} \wedge y) = x \vee y, x \wedge (\bar{x} \vee y) = x \wedge y$
- dvojitá negace – $\bar{\bar{x}} = x$
- **De Morghanovy zákony** – $\bar{x} \wedge \bar{y} = \overline{x \vee y}, \bar{x} \vee \bar{y} = \overline{x \wedge y}$

Z této algebry potom vychází logické funkce využívané v hardwaru:

x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

x	y	OR
0	0	0
0	1	1
1	0	1
1	1	1

x	NOT
0	1
1	0

x	y	XOR
0	0	0
0	1	0
1	0	0
1	1	0

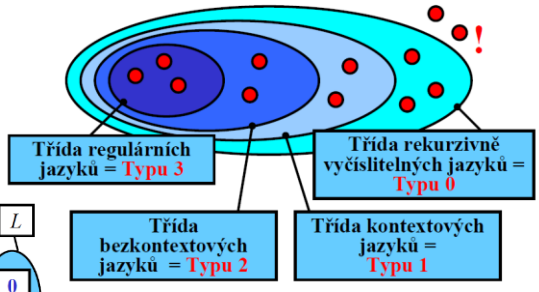
x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

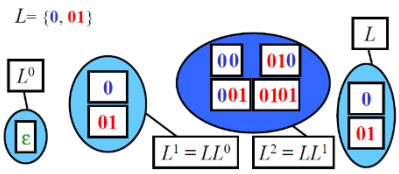
20. Regulární jazyky a jejich modely (konečné automaty, regulární výrazy). IFJ

Základními pojmy v oblasti formálních jazyků jsou:

- **abeceda** – neprázdná množina symbolů
  - Množina všech řetězců nad abecedou *V* se značí *V**.
- **slovo** – $\omega \in \Sigma^*$, je to konečná posloupnost znaků abecedy, prázdné slovo se značí ε
  - **délka slova** $|w|$ = je počet jeho symbolů
  - **prefix slova** *xy* = *x*
  - **suffix slova** *xy* = *y*
  - **mocnina slova** x - $x^0 = \varepsilon$, $x^i = xx^{i-1}$
- **jazyk** – jazyk *L* nad abecedou *V*: $L \subseteq V^*$ (∅ a $\{\varepsilon\}$ jsou jazyky nad libovolnou abecedou), může být **konečný** nebo **nekonečný** (je to množina)
  - **mocnina jazyka** L : $L^0 = \{\varepsilon\}$, $L^i = LL^{i-1}$
  - **iterace jazyka** L : $L^* = \bigcup_{i=0}^{\infty} L^i$
  - **pozitivní iterace jazyka** L : $L^+ = \bigcup_{i=1}^{\infty} L^i$



Obrázek 16: Chomského hierarchie jazyků



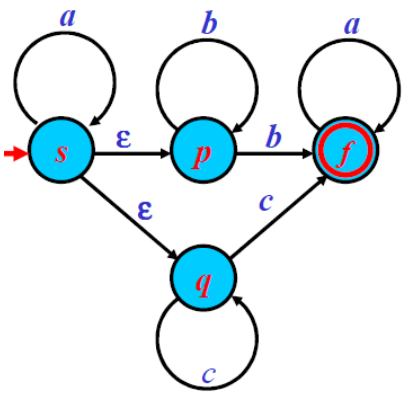
Obrázek 32: mocnina jazyka - příklad

Regulární jazyky jsou jazyky, které lze popsat:

- regulárním výrazem
- konečným automatem (deterministickým i nedeterministickým)
- regulární gramatikou

Regulární výrazy (regular expressions) jsou výrazy, které umožňují popsat potenciálně nekonečné jazyky, ač jsou samy konečné. V praxi se jedná o textové řetězce umožňující v textu snadné vyhledávání, nahrazování apod. Formálně jsou regulární výrazy nad abecedou Σ definovány následovně:

- ∅ je reg. výraz značící prázdnou množinu (prázdný jazyk)
- ε je reg. výraz značící jazyk $\{\varepsilon\}$ (prázdný řetězec)
- a, kde $a \in \Sigma$, je reg. výraz značící jazyk $\{a\}$
- pokud *r* a *s* jsou reg. výrazy značící jazyky L_r a L_s, potom
  - *(r.s)* je reg. výraz značící jazyk $L = L_r L_s$ (konkatenace)
  - *(r + s)* je reg. výraz značící jazyk $L = L_r \cup L_s$ (alternativa)
  - *(r*)* je reg. výraz značící jazyk $L = L_r^*$ (iterace)



Obrázek 33: příklad konečného automatu

Regulární výtazy lze zjednodušit pomocí dodatečných pravidel:

- priority operátorů (méně závorek): * > . > +
- *r.s* může být zapsáno jako *rs*
- rr^* nebo r^*r může být zapsáno jako r^+

Nedeterministický konečný automat je pětice:

$$A = (Q, T, \delta, s, F)$$

kde *Q* je množina stavů, *T* je abeceda vstupních symbolů, $\delta: Q \times T \to 2^Q$ je přechodová funkce, $s \in Q$ je počáteční stav a $F \subseteq Q$ je množina koncových stavů. **Deterministický** konečný automat se liší pouze přechodovou funkcí, která je pro něj ve tvaru $\delta: Q \times T \to Q$ (každá kombinace stavu a vstupu má jednoznačně určený stav, do něhož se přejde).
Přechodovou funkci zapisujeme ve formě pravidel:

$$stav_1\ vstup \rightarrow stav_2$$

Konečný automat většinou znázorňujeme graficky (viz. obr.) nebo tabulkou. **Konfigurace** automatu je dvojice *(q,x)*, kde *q* je aktuální stav a *x* je řetězec znaků, které zbývá přečíst ze vstupu. Existuje algoritmus pro převod jakéhokoli nedeterministického konečného automatu na deterministický konečný automat.

Gramatika je čtveřice

$$G = (N, T, P, S)$$

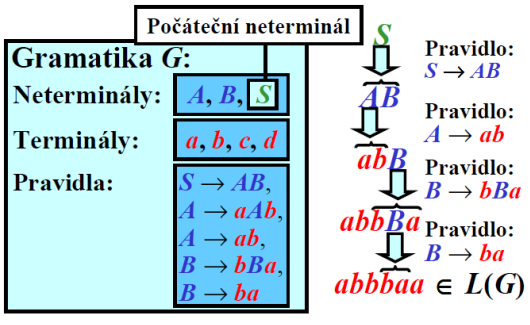
kde *N* je abeceda neterminálů, *T* je abeceda terminálů ($N \cap T = \emptyset$), *P* je konečná množina přechodových pravidel tvaru $x \to y$ a $S \in N$ je počáteční symbol. Gramatika funguje na principu přepisování řetězců. Podle daných pravidel mohou být neterminály přepisovány na řetězce terminálů a neterminálů. Gramatika definuje, tzv. **generuje**, určitý jazyk. Terminály jsou symboly, které se už dál nepřepisují. **Derivační krok** je jedno uplatnění pravidla ke změně řetězce. **Nejlevější derivace** je derivační krok, při němž je přepsán nejlevější neterminál v řetězci. Obdobně je definována **nejpravější derivace**. Jazyk generovaný gramatikou se nijak nezmění, povolíme-li jakékoli nebo pouze nejlepší nebo nejpravější derivace. **Větná forma** je každé slovo, které lze gramatikou odvodit z počátečního symbolu. **Věta** je větná forma složená pouze z terminálů (nelze již dále rozvíjet). **Ekvivalentní gramatiky** generují stejný jazyk.

Regulární gramatika (také **pravá lineární gramatika**) je gramatika, která má pravidla ve tvaru:

$$X \to wY$$
$$Y \to w$$

Kde *X* a *Y* jsou neterminály a *w* je řetězec terminálů.

Pumping Lemma umožňuje dokázat, že jazyk není regulární.



Obrázek 34: ilustrace gramatiky

21. Bezkontextové jazyky a jejich modely (zásobníkové automaty, bezkontextové gramatiky).

IFJ

Bezkontextové jazyky jsou nadtřídou jazyků regulárních (tzn. jsou stejně složité nebo o něco složitější) a lze je popsat pomocí:

- bezkontextových gramatik
- zásobníkových automatů

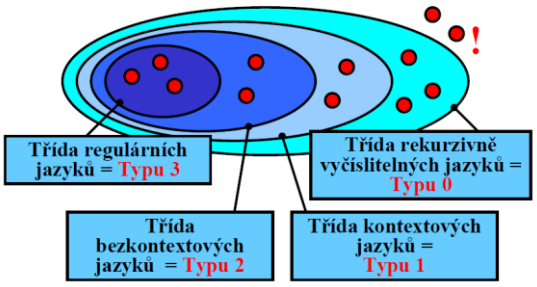
Bezkontextová gramatika je gramatika, která má pravidla ve tvaru:

$$A \rightarrow \beta$$

kde A je neterminál a β je řetězec terminálů a/nebo neterminálů. Jméno bezkontextová pochází z faktu, že neterminál na levé straně nemůže mít okolo sebe jiné symboly, tzn. kontext. Formálně je bezkontext. gramatika čtveřice:

$$G = (N, T, P, S)$$

kde *N* je abeceda neterminálů, *T* je abeceda terminálů, přičemž $N \cap T = \emptyset$, *P* je konečná množina pravidel tvaru neboli relace $N \rightarrow (N \cup T)^*$ a $S \in N$ je počáteční neterminál. **Derivační krok** je použití jednoho pravidla k přepsání některého neterminálu (pokud se takto dostaneme z jednoho řetězce k jinému, říkáme, že je přímo derivován, zapisujeme *řetězce1* ⇒ *řetězec2*). Přímá derivace za pomoci **nejlevější derivace** znamená, že prepíšeme nejlevější neterminál v řetězci. Bez újmy na obecnosti můžeme uvažovat pouze nejlevější derivace. Gramatika je **nejednoznačná**, pokud existuje alespoň jeden řetězec s různými derivačními stromy, jinak je jednoznačná.



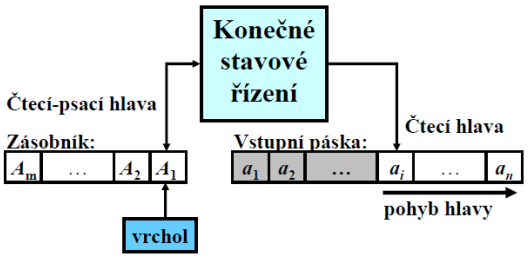
Obrázek 17: Chomského hierarchie jazyků

Zásobníkový automat je neformálně řečeno konečný automat rozšířený o zásobník. Formálně je to sedmice:

$$M = (Q, \Sigma, \Gamma, R, s, S, F)$$

kde *Q* je konečná množina stavů, Σ je vstupní abeceda, Γ je zásobníková abeceda, $s \in Q$ je počáteční stav, $S \in \Gamma$ je počáteční symbol na zásobníku a *R* je konečná množina pravidel ve tvaru:

$$Apa \rightarrow wq$$

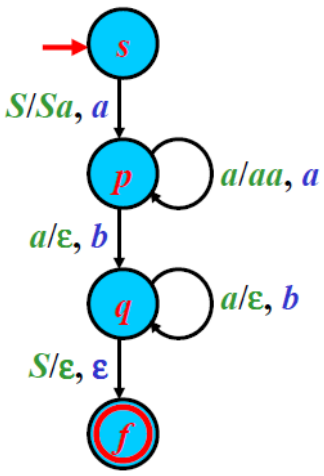


Obrázek 36: zásobníkový automat

kde $A \in \Gamma$, $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ a $w \in \Gamma^*$. Pravidlo tedy znamená, že pokud je automat ve stavu *p*, na vstupu symbol *a* a na vrcholu zásobníku symbol *A*, pak může automat přejít do stavu *q* a přepsat symbol *A* na vrcholu zásobníku na řetězec *w*. Konfigurace zásobníkového automatu je řetězec $x \in \Gamma^* Q \Sigma^*$ (řetězec zásobníku, stav automatu a zbývající vstup). Jazyk přijímaný zásobníkovým automatem může být přijímaný:

- **přechodem do koncového stavu**
- **vyprázdněním zásobníku**
- **přechodem do koncového stavu a vyprázdněním zásobníku**

Existují algoritmy pro převod jakéhokoli výše zmíněného typu na jakýkoliv jiný.



Obrázek 35: příklad zásobníkového automatu

Pro každý jazyk generovaný bezkontextovou gramatikou existuje i zásobníkový automat, který jej generuje, a naopak. Zásobníkové automaty se používají při bezkontextové syntaktické analýze shora dolů.

příklad bezkontextové gramatiky:

S → aSa
S → bSb
S → ε

počáteční symbol: S

S ⇒ aSa ⇒ aaSaa ⇒ aabSbaa ⇒ aabbaa

Bonus:

příklad nejednoznačné gramatiky:

S → aA
S → Aa
A → aa

Řetězec aaa nemá jednoznačný derivační strom.

22. Struktura překladače a charakteristika fází překladu (lexikální analýza, deterministická syntaktická analýza a generování kódu). IFJ

Překladač je program, který transformuje program z jednoho jazyka do jiného tak, že je nový program funkčně ekvivalentní své předloze. Překladač se typicky skládá z těchto částí:

- **lexikální analyzátor** (**scanner**) – Rozděluje vstupní text na tzv. **lexémy** (**tokeny**) – logické lexikální jednotky (např. klíčová slova, jména proměnných, operátory atd.). Lexémy mohou mít atributy. Lexikální analyzátor se realizuje jako konečný automat.
- **syntaktický analyzátor** (**parser**) – Jedná se o hlavní část překladače. Z řetězce tokenů sestavuje tzv. **derivační strom** (v praxi se většinou nevytváří, generuje se rovnou abstraktní synt. strom). Parser kontroluje, zda řetězec tokenů reprezentuje syntakticky správný program a provádí další akce, jako např. sestavování kódu.
- **sémantický analyzátor** – Provádí kontrolu sémantiky jazyka, tzn. např. kontrolu datových typů nebo zda byla proměnná již deklarována. Jeho vstupem je derivační strom vytvořený syntaktickým analyzátorem a výstupem je tzv. **abstraktní syntaktický strom**.
- **generátor vnitřního kódu** – Z abstraktního syntaktického stromu generuje instrukce vnitřního kódu (také mezikódu). Tento kód většinou ještě není spustitelný, ale je platformě nezávislý a dobře optimalizovatelný. Často se používá tzv. **tříadresný kód**, v němž má každá instrukce maximálně dva operandy, jde tedy o uspořádanou trojici: operátor, operand 1, operand 2.
- **optimalizátor** – Provádí optimalizace nad vnitřním kódem, aby byl rychlejší a/nebo kratší a/nebo paměťově úspornější.
- **generátor cílového kódu** – Z vnitřního kódu generuje spustitelný kód pro danou platformu.

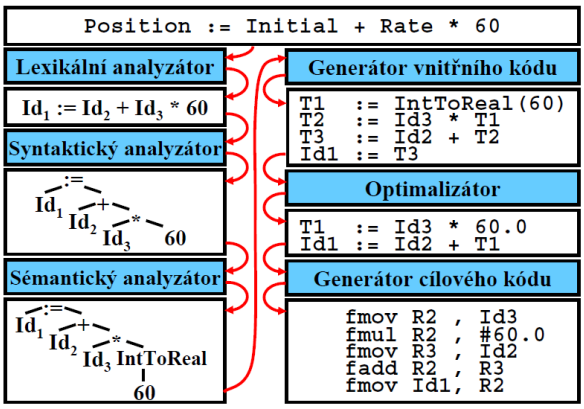


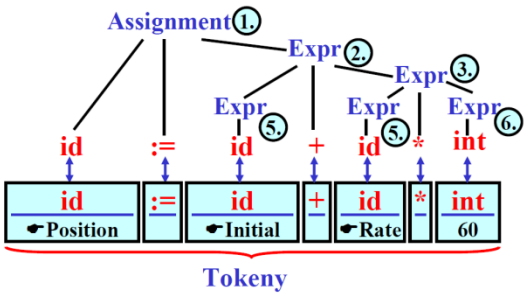
Figure 18: logická struktura překladače

Syntaxí řízený překlad – překlad je řízený syntaktickým analyzátorem, který je jádrem překladače a volá metody ostatních jednotek.

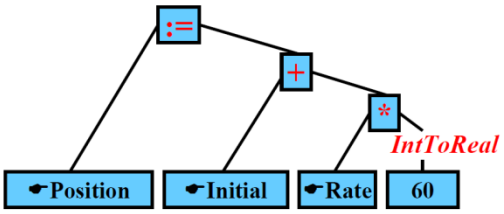
Derivační strom je strom, který přesně reprezentuje řetězec generovaný nějakou gramatikou, kterými jsou jazyky nejčastěji popsány. Vnitřní uzly se nazývají **neterminály**, listy **terminály**. Abstraktní syntaktický strom oproti němu některé věci vynechává nebo nahrazuje.

Syntaktickou analýzu můžeme provádět těmito způsoby (v praxi se používá kombinace obou):

- **shora dolů** – Parser začíná počátečním symbolem a snaží se jej převést na vstup (derivační strom konstruujeme od kořene). Příkladem je LL analýza, kdy vždy hledáme levou derivaci, tzn. přepisujeme nejlevější neterminál. Může se stát, že existuje více aplikovatelných pravidel a my použijeme nesprávné pravidlo, v takovém případě se musíme navracet a vyzkoušet postupně ostatní. Analyzátor používající tuto metodu se nazývá **LL syntaktický analyzátor** (čte vstup zleva a hledá vždy levou derivaci). **LL gramatika** (typu 1) je gramatika, kterou lze bez navracení analyzovat LL analyzátorem při znalosti maximálně jednoho následujícího symbolu, proto jsou vhodné pro použití s tímto typem analyzátoru. Implementace této analýzy se dělá buď pomocí zásobníkového automatu nebo **rekurzívním sestupem** (máme sadu procedur, které rekurzí nahradí zásobník). Při implementaci LL analyzátoru se používá tzv. **LL tabulka**, která určuje pomocné množiny (např. množina všech terminálů, které se mohou vyskytovat vpravo od *A* ve větné formy) a pomáhá analyzátor sestavovat.
- **zdola nahoru** – Parser nejdříve identifikuje terminální symboly a potom se snaží zkonstruovat strom směrem nahoru (tzn. od listů ke kořeni). Příkladem je LR analýza. Pro tuto analýtu existují různé analyzátory:
    - **precedenční syntaktický analyzátor** – Nejslabší, ale snadno se implementuje. Vyžaduje precedenční tybulku.
    - **LR syntaktický analyzátor** – Nejsilnější, ale složitý, je založen na rozšířeném zásobníkovém automatu.



Obrázek 37: derivační strom



Obrázek 38: abstraktní syntaktický strom

Překladač používá různé pomocné tabulky, z nichž nejtypičtější je **tabulka symbolů**. V ní se ukládají veškeré symboly a informace o nich, tzn. proměnné, funkce, jejich jména, datové typy a adresy, popř. i další pomocné informace. Díky této tabulce může např. sémantický analyzátor provádět typovou kontrolu. Tabulka symbolů je často implementována jako binární vyhledávací strom nebo hashovací tabulka pro rychlé vyhledávání symbolů.

Polská (prefixová) notace je zápis výrazu ve tvaru *operátor operand1 operand2* (např. 3 4 + místo 3 + 4), výhodou je, že nepotřebuje závorky a dá se snadno zpracovávat zásobníkem. Existuje i postfixová (reverzní polská) notace ve tvaru *operand1 opearand2 operátor*.

23. Numerické metody a matematická pravděpodobnost (numerické řešení algebraických a obyčejných diferenciálních rovnic, rozložení pravděpodobnosti, generování pseudonáhodných čísel). INM, IMS

Numerické metody jsou metody hledání přibližných výsledků rovnic, které je časově náročné počítat přesně, nebo je přesně spočítat neumíme.

Numericky lze řešit např. velké soustavy lineárních rovnic. Zatímco běžné soustavy lineárních rovnic řešíme **Gaussovou eliminační metodou** (klasická úprava matice na trojúhelníkový tvar), popř. **Cramerovým pravidlem** (podíl determinantů) (tzv. přímé metody), u větších soustav musíme použít **iterační metody**, které nám dají pouze přibližný výsledek. Klasickými iteračními metodami jsou:

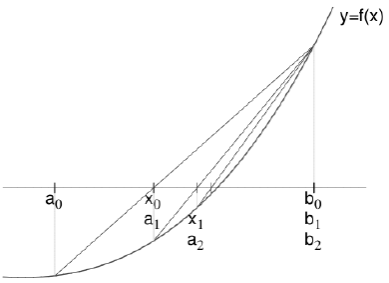
- **Jacobiho iterační metoda** – Soustavu rovnic převedeme na tvar $x_1 = ..., x_2 = ..., x_3 = ..., ...$ Potom zvolíme počáteční aproximaci (např. $x_1 = 0, x_2 = 0, x_3 = 0, ...$), kterou dosadíme do pravých stran rovnic. Tím nám vyjde nová aproximace, se kterou stejným způsobem pracujeme dál, dokud nemáme dostatečně přesné řešení. Podmínky konvergence: matice soustavy rovnic musí být řádkově nebo sloupcově diagonálně dominantní (v každém řádku (sloupci) je absolutní hodnota prvku na diagonále větší než součet absolutních hodnot všech ostatních prvků v řádku (sloupci), diagonála se bere z levého horního rohu k dolnímu pravému).
- **Gauss-Seidelova iterační metoda** – Metoda velmi podobná Jacobiho metodě, rozdíl je v tom, že vždy počítáme s nejaktuálnějšími hodnotami, tzn. pokud během iterace vypočítáme novou aproximaci jedné proměnné, použijeme ji ještě během té samé iterace u zbývajících rovnic. Podmínky konvergence: stejné jako u Jacobiho metody (asi ?).

Dále lze numericky řešit jednotlivé nelineární rovnice. Máme dánu funkci *f(x)* a hledáme, kde platí *f(x) = 0* (jiné případy, např. *f(x) = g(x)*, převádíme na tento). Tyto kořeny hledáme s přesností ε (kořen leží v dosahu vypočítaná hodnota $\pm\,\varepsilon$ na ose *x*). Nejprve vždy provedeme **separaci kořenů** – určíme intervaly, v nichž kořeny leží. To provádíme např. za pomoci nákresu grafu funkce nebo věty:

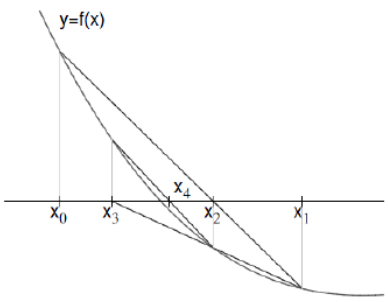
> *f(x)* je spojitá na *<a,b>* a *f(a) f(b) < 0* $\Longrightarrow$ v intervalu *<a,b>* leží alespoň jeden kořen *f(x)*

Přesnější určení polohy kořenů provádíme následujícími metodami:

- **půlení intervalu** (**bisekce**) – Nejjednodušší metoda, provádí se na intervalu z výše zmiňované věty. Nejdřív nalezneme střed intervalu <a,b> ($\frac{a+b}{2}$) a pokračujeme v té polovině intervalu, v níž kořen zaručeně leží. Tento interval poznáme podle znaménka funkční hodnoty v polovině intervalu. Končíme tehdy, když nalezneme kořen, nebo když po určení středu platí pro poslední vyšetřovaný interval $b - a < 2\varepsilon$ (výsledkem je střed intervalu). Nevýhodou je pomalejší konvergence.
- **regula falsi** – Velmi podobná metodě bisekce. Rozdíl je v tom, že střed interval nepůlíme v jeho středu, ale vedeme úsečku z *[a,f(a)]* do *[b,f(b)]* a její průsečík s osou x bereme jako střed. Podmínkou ukončení je, že rozdíl dvou po sobě vypočtených středů na ose x je menší než ε. Metoda konverguje většinou rychleji než bisekce, avšak ne ve všech případech.
- **metoda sečen** – Podobná metodě regula falsi. Nejprve vedeme úsečku z *[a,f(a)]* do *[b,f(b)]* a určíme průsečík *c*, avšak dále neurčujeme, ve které polovině intervalu leží kořen, ale vedeme novou úsečku z *[b,f(b)]* do *[c,f(c)]*, určíme průsečík *d*, vedeme úsečku z *[c,f(c)]* do *[d,f(d)]* atd. V *k*-tém kroku určíme aproximaci kořene vzorcem:
  $$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k)$$
  kde $x_0 = a$ a $x_1 = b$. Výpočet ukončíme, když $|x_k - x_{k-1}| < \varepsilon$. Metoda je rychlejší než metoda sečen, ale nekonverguje vždy. Konvergenci je obtížné zjistit, proto metodu většinou používáme s nastaveným maximálním počtem kroků.
- **Newtonova metoda** (**metoda tečen**) – Velmi efektivní metoda, avšak nemusí vždy konvergovat. Na začátku zvolíme počáteční aproximaci x_0, pak sestrojíme tečnu v bodě $[x_0, f(x_0)]$ a v jejím průsečíku s osou *x* získáme novou aproximaci x_1. Tak pokračujeme, dokud neplatí $|x_k - x_{k-1}| < \varepsilon$. Podmínka konvergence: v intervalu musí ležet jediný kořen *f(x) = 0*, dále *f'(x)* a *f''(x)* na něm musí být spojité a neměnit znaménko a musí platit $f(x_0)f''(x_0) > 0$.
- **metoda prosté iterace** – Vyřetřovanou rovnici *f(x) = 0* převedeme na tvar *g(x) = x*. Zvolíme počáteční aproximaci x_0 a iterujeme podle vztahu: $x_{k+1} = g(x_k)$. Podmínky konvergence: $|g'(x)| < 1$.

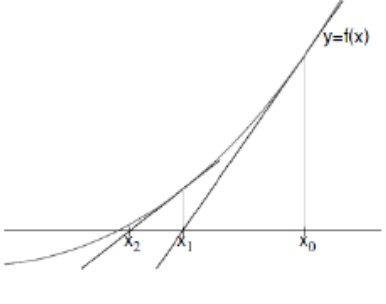


Obrázek 39: regula falsi



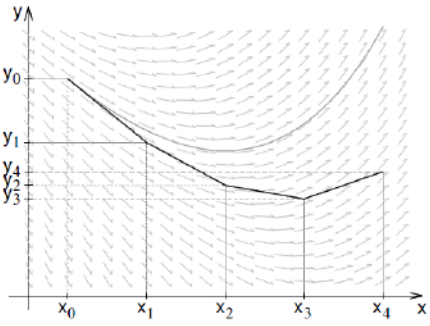
Numerické metody pomáhají řešit rovněž diferenciální rovnice. **Diferenciální rovnice** je rovnice, v níž jako neznámé vystupují funkce a jejich derivace a jejich řešením je tedy funkce (ne pouze množina čísel, jako je tomu u **algebraických rovnic**). **Řádem rovnice** rozumíme stupeň nejvyšší derivace v rovnici. K řešení rovnice musíme mít zadány počáteční podmínky pro všechny derivace (i když se v rovnici některá nevyskytuje) kromě nejvyšší (např. pro rovnici $f(x) + f'(x) - 2f''(x) = 0$ musíme mít zadáno $f(x_0) = a_0$ a $f'(x_0) = a_1$). Pro různé počáteční podmínky můžeme dostat různé průběhy výsledné funkce. Tyto rovnice dále dělíme na **obyčejné** (funkce má jednu proměnnou) a **parciální** (funkce má více proměnných). Výsledkem numerického řešení diferenciální rovnice není celá spojitá funkce, ale pouze množina bodů na určitém intervalu proložená polynomem, která přibližně aproximuje analytické řešení. Metody dělíme na **jednokrokové** (vycházejí pouze z aktuálního stavu) a **vícekrokové** (používají historii stavů, problém bývá na začátku, kdy ještě nemáme dostatek hodnot). Dále metody dělíme na **explicitní** (pouze dosazujeme do vzorce) a **implicitní** (v každém kroku musíme řešit soustavu rovnic). **Řád metody** určuje řád polynomu, jímž funkci aproximujeme. Mezi nejpoužívanější metody patří:

- **Eulerova metoda** – Jednoduchá jednokroková metoda prvního stupně. Ve směru *x* se posunujeme po kroku *h* a derivace v aktuálním bodě nám určí směr, kterým se posuneme dál:
  $$f(x_{i+1}) = f(x_i) + hf'(x_i)$$
  Pro tuto metodu existují modifikace, které ji dokáží vylepšit (jsou 2. řádu, zavádí pomocné proměnné).
- **Runge-Kutta metody** – Jednokroková metoda. Modifikace Eulerovy metody patří mezi tyto metody. Nejznámější je Runge-Kutta 4. řádu, která používá vzorec:
  $$f(x_{i+1}) = \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$
  Hodnoty k_1 až k_4 (jejich počet se rovná stupni metody) jsou určeny dalšími vztahy a počítají se z hodnot x_i a $f(x_i)$.



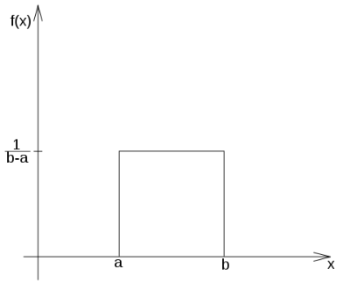
Obrázek 41: Newtonova metoda

Rozložení (**rozdělení**) **pravděpodobnosti** je charakteristika pravděpodobnosti hodnot **náhodné veličiny** (veličina, jejíž hodnota je dána pravděpodobností). **Pravděpodobnostní funkce *P(x)*** je funkce přiřazující každé hodnotě diskrétní náhodné veličiny určitou pravděpodobnost a musí platit $\sum_{x \in D(P)} P(x) = 1$. Obdobou pravděpodobnostní funkce pro spojitou náhodnou veličinu je **funkce hustoty pravděpodobnosti $\rho(x)$**, pro kterou musí platit $\int_{x \in d(\rho)} \rho(x)dx = 1$ (u této funkce nelze vyšetřovat jen jednu hodnotu, ale musí se vždy pracovat s integrálem na intervalu!). Dále definujeme **distribuční funkci *F(x)*** pro diskrétní a spojitou náhodnot veličinu: $F(x) = P[X \leq x]$. Tato funkce udává pravděpodobnost, že náhodná veličina bude menší nebo rovna dané hodnotě. Důležitými pravděpodobnostními rozloženími jsou:



- **rovnoměrné rozložení** – Všechny hodnoty náhodné veličiny mají stejnou pravděpodobnost výskytu.
- **normální** (**Gaussovo**) **rozložení** – P(x) je Gaussova křivka. Toto rozložení mají veličiny, které jsou ovlivněny velkým množstvím nezávislých faktorů (např. výška stromu v lese). Součet velkého množství jakýchkoli rozložení konverguje k normálnímu rozložení. Značí se $X \sim N(\mu, \rho^2)$, kde μ je střední hodnota a ρ^2 rozptyl.
- **binomické rozložení** – Náhodná veličina udává počet úspěchů v řadě nezávislých pokusů. Značí se $X \sim Bi(N, p)$, kde *N* je počet pokusů a *p* pravděpodobnost úspěchu. Jedná se o diskrétní rozložení. Při určitých parametrech (velký počet pokusů, malá pravděpodobnost) se dá nahradit Poissonovým rozložením.
- **Poissonovo rozložení** – Náhodná veličina vyjadřuje počet výskytů události v určitém časovém intervalu. Značí se $X \sim Po(\lambda)$, kde λ je průměrný počet výskytů události v daném intervalu. Jedná se o diskrétní rozložení.

- **exponenciální rozložení** – Náhodná veličina vyjadřuje čas mezi náhodně se vyskytujícími událostmi. Značí se $X \sim Exp(\lambda)$, kde λ je průměrná doba mezi výskyty událostí. Jedná se o spojité rozložení.



Obrázek 47: rovnoměrné rozložení

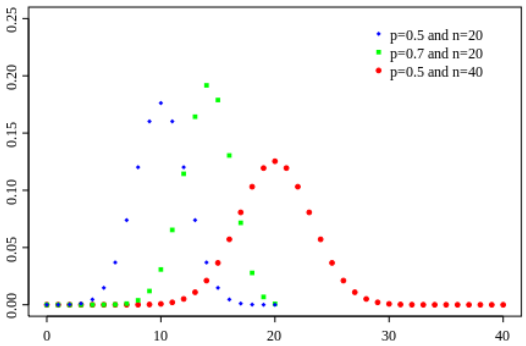
Pseudonáhodná čísla jsou čísla, která mají charakter náhodných čísel, ale jsou generována deterministickým způsobem (tedy počítačem). Čistě náhodná čísla lze generovat také, ale je to časově náročné a málo kdy potřeba, proto se budeme zabývat pseudonáhodnými čísly. Základem generování je většinou tzv. **kongruentní generátor** (alternativou je např. Mersenne twister), který pracuje na základě posloupnosti:

$$x_0 = seed$$
$$x_{n+1} = (ax_n + c) mod\ m$$

Kde a, c a m jsou vhodně zvolené konstanty. Různé počáteční hodnoty (***seed***) nám zjevně dají různé posloupnosti. Nevýhodou kongruentního generátoru je, že nejpozději po m krocích začne generovat opakující se sekvenci čísel. Také se může stát, že při nevhodně zvolených konstantách dostaneme posloupnost, která neprojde některými testy náhodných generátorů, protože členy posloupnosti budou vykazovat určitou provázanost.

Kongruentní generátor generuje rovnoměrné rozložení pravděpodobnosti. Pro jiná rozložení musíme použít některou metodu transformace:

- **inverzní transformace** – Vygenerované číslo v rovnoměrném rozložení předáme jako argument inverzní distribuční funkci daného rozložení. Inverzní funkce musí samozřejmě existovat.
- **vylučovací metoda** – Náhodně generujeme body se souřadnicemi *[x,y]*, dokud nenajdeme takový, který leží pod grafem funkce hustoty pravděpodobnosti daného rozložení. Výsledným číslem je potom souřadnice *x*. Metoda není vhodná pro neomezená rozložení a rozložení, jejichž graf svým tvarem umožňuje generovat spoustu bodů mimo – metoda je potom pomalá.
- ...

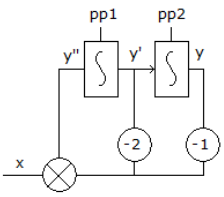


Obrázek 46: binomické rozložení

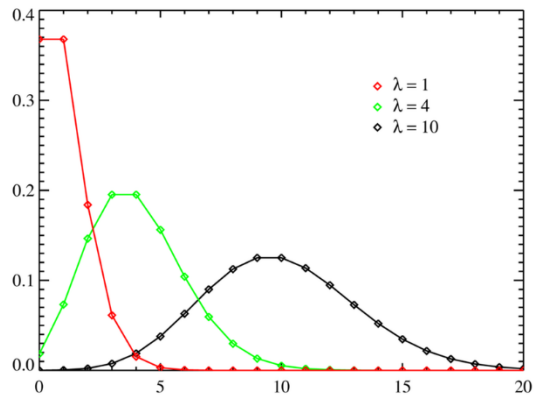
Střední hodnota náhodné veličiny je vážený průměr všech jejích možných hodnot, přičemž váhou je pravděpodobnost dané hodnoty.

Při numerických výpočtech diferenciálních rovnic musíme rovnice převést na první řád, abychom je mohli předat integrátorům. Metodami převodu jsou:

- **snižování řádu derivace** – Osamostatníme nejvyšší derivaci a vytvoříme soustavu rovnic prvního řádu. Pro tuto metodu musíme mít rovnici bez derivací vstupů. Př.:
  y'' – 2y' + y = x => y'' = 2y' – y + x
  y' = ∫ y''
  y = ∫ y'



- **postupná integrace** – Převádí derivování a integrování na násobení a dělení, mohou se vyskytovat i derivace vstupů, koeficienty však musí být konstantní. Při výpočtech zavádíme pomocné proměnné, např.:
  $p^2y + 2py + 3y + px = 0$
  $p^2y = -2py - 3y - px$ /p
  $py = -2y - x - w_1$ $w_1 = 3y / p$
  $y = 1/p\ (-2y - x - w_1)$
  $y = w_2$ $w_2 = 1/p\ (-2y - x - w_1)$



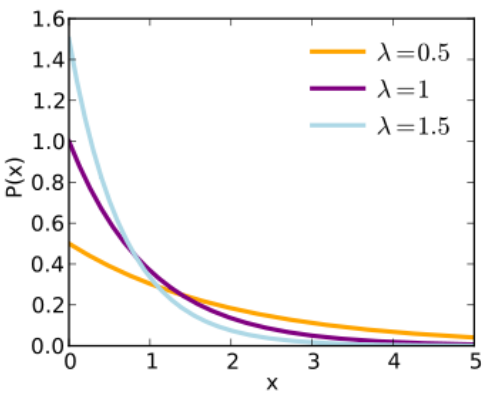
Obrázek 43: Poissonovo rozložení

Bonus:

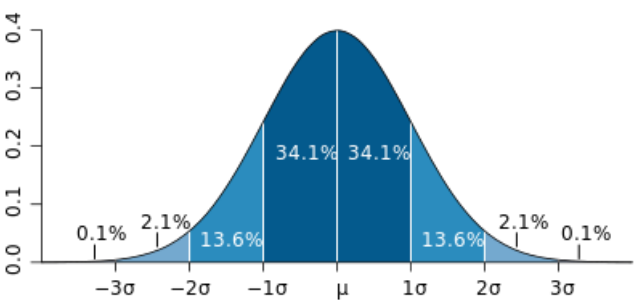
výpočet determinantů:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = (aei + bfg + cdh) - (ceg + bdi + afh)$$



Obrázek 44: exponenciální rozložení



Obrázek 45: normální rozložení

24. Řešení úloh (prohledávání stavového prostoru, rozklad na podúlohy, metody hraní her). IZU

Umělá inteligence je věda o tom, jak konstruovat stroje, jejichž činnost, kdyby ji vykonávali lidé (a kdybychom nevěděli, že ji vykonávají stroje), bychom považovali za projev jejich inteligence. **Úloha** je definována svým počátečním stavem, množinou koncových stavů a operátory, které slouží ke změně stavů. Cílem řešení úloh je nalezení posloupnosti operátorů, jejichž aplikací se dostaneme z počátečního stavu do některého ze stavů koncových. To často vyžaduje rozdělení úlohy na podúlohy.

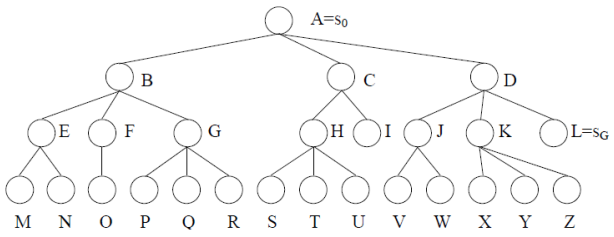
Stavový prostor je dvojice:

$$(S, O)$$

kde *S* je množina stavů a *O* množina operátorů. **Úloha** je definována jako dvojice:

$$(S_0, G)$$

Kde $S_0 \epsilon S$ je počáteční stav a $G \subseteq S$ je množina cílových stavů úlohy.
Stavový prostor si tedy můžeme představit jako orientovaný strom nebo graf, kde každý uzel představuje jeden stav a každá hrana jeden přechod mezi stavy. Hrany mohou být ohodnocené cenou přechodu.



Obrázek 48: příklad stavového prostoru

Řešení úloh metodou **prohledávání stavového prostoru** je založeno na procházení stavů v určitém pořadí a hledání cesty k cílovému stavu. **Operátor** je symbol, kterým lze měnit stav úlohy (provést přechod do jiného stavu), je to akce, kterou lze provést v určitém stavu. Řešením úlohy je tedy posloupnost operátorů. Metody prohledávání stavového prostoru dělíme na:

- **neinformované** (**slepé**) – Nemají k dispozici žádnou pomocnou informaci.
- **informované** – Mají k dispozici pomocnou informaci (**heuristiku**), která dokáže napovědět, kde řešení leží.

Metody dále hodnotíme různými kritérii, metoda může být:

- **úplná –** Existuje-li řešení, metoda jej vždy nalezne.
- **optimální** – Nalezne-li metoda řešení, řešení je optimální.

Mezi neinformované metody patří:

- **prohledávání do šířky** (**BFS**, **breadth first search**) – Je úplný a optimální, časová a paměťová složitost je exponenciální (konkrétně S^N, kde S je počet operátorů a N faktor větvení, algoritmus je proto prakticky nepoužitelný), jedná se o základní metodu. Prohledává stavový prostor "po řádcích". Algoritmus je následující:
  1. Sestroj frontu OPEN (bude obsahovat uzly určené k expanzi) a umísti do ní počáteční uzel.
  2. Fronta OPEN je prázdná => Úloha nemá řešení (konec).
  3. Vyber z OPEN první uzel.
  4. Vybraný uzel je cílový => Máme řešení (vrať posloupnost operátorů).
  5. Expanduj vybraný uzel, všechny bezprostřední následovníky umísti do OPEN a jdi na bod 2.

  Abychom mohli vrátit posloupnost operátorů, musíme si u každého uzlu pamatovat posloupnost stavů, kterými jsme se do něj dostali.
  U některých problémů může nastat problém zacyklení. Máme-li stav A, jehož expanzí se dostaneme do stavu B a expanzí stavu B se dostaneme opět do stavu A (přímo nebo po více krocích), budou se stále dokola ověřovat stavy A a B. Tento problém se řeší přidáním fronty CLOSED, do níž se ukládají již ověřené uzly. Navíc takto dosáhneme toho, že si u každého uzlu stačí pamatovat pouze bezprostředního předchůdce, nikoli všechny (ti se dají rekonstruovat z fronty CLOSED). Modifikovaný algoritmus vypadá takto (modifikace červeně):
  1. Sestroj frontu OPEN (bude obsahovat uzly určené k expanzi) a umísti do ní počáteční uzel a prázdnou frontu CLOSED.
  2. Fronta OPEN je prázdná => Úloha nemá řešení (konec).
  3. Vyber z OPEN první uzel a dej ho do CLOSED.
  4. Vybraný uzel je cílový => Máme řešení (vrať posloupnost operátorů).
  5. Expanduj vybraný uzel, všechny bezprostřední následovníky, kteří nejsou v OPEN ani v CLOSED, umísti do OPEN a jdi na bod 2.
- **prohledávání do hloubky** (**DFS**, **depth first search**) – Není úplný ani optimální, časová náročnost je exponenciální, paměťová lineární. Stavový prostor se prohledává do hloubky. Algoritmus je následující:
  1. Sestroj zásobník OPEN a umísti do něj počáteční uzel.
  2. Zásobník OPEN je prázdný => Úloha nemá řešení (konec).
  3. Vyber z OPEN první uzel.
  4. Vybraný uzel je cílový => Máme řešení (vrať posloupnost operátorů).
  5. Expanduj vybraný uzel, všechny bezprostřední následovníky umísti do OPEN a jdi na bod 2.
- **omezené prohledávání do hloubky** (**DLS**, **depth limited search**) – Metoda DFS s omezenou hloubkou. Není úplný ani optimální, časová náročnost je exponenciální, paměťová lineární. Modifikace spočívá v tom, že se uzly expandují pouze tehdy, pokud nejsou níže než je zadaná hloubka.
- **metoda stejných cen** (**UCS, uniform cost search**) – Podobný jako BFS, pracuje s cenami přechodů. Je úplný a optimální, časová i paměťová náročnost jsou exponenciální. Algoritmus je následující:
  1. Sestroj seznam OPEN a umísti do ní počáteční uzel s jeho ohodnocením (0) a prázdný seznam CLOSED.
  2. Fronta OPEN je prázdná => Úloha nemá řešení (konec).
  3. Vyber z OPEN uzel s nejlepším ohodnocením a dej ho do CLOSED.
  4. Vybraný uzel je cílový => Máme řešení (vrať posloupnost operátorů).
  5. Expanduj vybraný uzel, všechny bezprostřední následovníky, kteří nejsou v OPEN ani v CLOSED, umísti do OPEN spolu s jejich ohodnocením (součet cen přechodů). Z uzlů, které jsou v OPEN vícekrát, ponech jenom ty s nejlepším ohodnocením. Jdi na bod 2.

  ohodnocení uzlu *n* se tedy vypočítá jako:

  $$cena(n) = cena\ cesty(n)$$
- **postupné zanořování do hloubky** (**IDS**, **iterative depth first search**) – Metoda spočívá v opakovaném použití DLS s postupně se zvyšující hloubkou. Je optimální a úplná. Ač se zdá tato metoda neefektivní, má stejnou časovou složitost jako BFS, ale pouze lineární prostorovou. Algoritmus je následujcí:
  1. Nastav maximální hloubku na 1.
  2. Použij DLS s maximální hloubkou. Skončí-li metoda úspěchem, vrať nalezenou cestu.
  3. Inkrementuj hloubku a jdi na bod 2.
- **metoda zpětného navracení** (**backtracking**) – Metoda není optimální ani úplná, časová náročnost je exponenciální, avšak vyznačuje se extrémně nízkou paměťovou náročností, která je konstantní. Jedná se o metodu DFS s tím, že se neexpandují vždy všichni následnící uzlu, ale jen ten, který se má ověřovat, další se expandují až v případě potřeby. Algoritmus je následující:
  1. Sestroj zásobník OPEN a umísti do něj počáteční uzel.
  2. Zásobník OPEN je prázdný => Úloha nemá řešení (konec).
  3. Vyber z OPEN první uzel. Jde-li na něj aplikovat první/další operátor, aplikuj jej a pokračuj se vzniklým uzlem bodem 4. Jinak uzel odstraň ze zásobníku a jdi na bod 2.
  4. Vybraný uzel je cílový => Máme řešení (vrať posloupnost operátorů). Jinak ulož uzel do OPEN a jdi na bod 2.
- **obousměrné prohledávání** (**BS**, **bidirectional search**) – Pokud máme k dispozici inverzní operátory, můžeme aplikovat BFS zároveň z počátečního a koncového stavu, čímž se sníží časová i paměťová složitost.

Mezi informované metody patří:

- **best first search** (**best FS**) – Algoritmus je stejný jako UCS, jenom ohodnocovací funkce zohledňuje cenu uzlu, která je dána heuristickou funkcí. Cena uzlu *n* se vypočítá jako:

  $$cena(n) = cena\ cesty(n) + heuristika(n)$$

  Heuristická funkce je odhad vzdálenosti k cíli (pro cílový uzel tedy musí dávat hodnotu 0). UCS je speciálním púřípadem této metody, kdy je heuristická funkce vždy nulová.
- **greedy search** – Stejný jako UCS, ale ohodnocení uzlů počítá pouze na základě heuristické funkce:

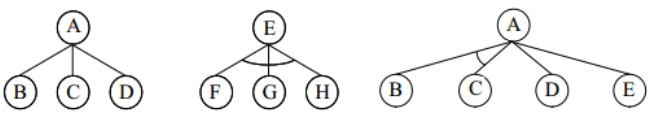
  $$cena(n) = heuristika(n)$$

- **A*** – Optimální metoda, složitost závisí na heuristice. Je to nejznámější a nejpoužívanější informovaná metoda. Jedná se o metodu best FS s tím, že heuristická funkce musí být **spodním odhadem** skutečné ceny cesty od ohodnocovaného uzlu k cíli (tzn. odhad musí být menší nebo roven skutečné ceně). Příkladem heuristické funkce může být například vzdálenost vzdušnou čarou.

Metody lokálního prohledávání jsou metody, které používáme tehdy, potřebujeme-li najít pouze cílový stav, nikoli cestu k němu (např. rozmístění čipů na desce). Mají zanedbatelnou paměťovou složitost. Mezi tyto metody patří např.:

- **Hill-climbing** – Greedy search s tím, že heuristika ohodnocuje stav, ne vzdálenost k cíli. Algoritmus je následující:
  1. Označ počáteční uzel i s jeho ohodnocením jako CURRENT.
  2. Expanduj CURRENT a vyber nejlépe ohodnoceného následníka (NEXT).
  3. Ohodnocení NEXT > ohodnocení CURRENT => vrať CURRENT jako řešení.
  4. Označ NEXT jako CURRENT a jdi na krok 2.
- **simulované žíhání** – Stochastická metoda odtraňující některé nedostatky metody Hill-Climbing. Algoritmus je následujcící:
  1. Vytvoř funkci klesání teploty v závislosti na kroku výpočtu.
  2. Označ počáteční uzel i s jeho ohodnocením jako CURRENT.
  3. Teplota = 0 => vrať CURRENT jako řešení.
  4. Expanduj CURRENT a z následníků vyber náhodně uzel NEXT.
  5. Ohodnocení NEXT > ohodnocení CURRENT => CURRENT := NEXT, jinak CURRENT := NEXT s pravděpodobností závisející na rozdílu ohodnocení a teplotě.
  6. jdi na bod 3.

Úloha s omezujícími podmínkami (**CSP**, **constrait satisfactory problem**) je úloha, v níž cílový stav není jednoduše označen jako cílový, ale musí splňovat určité podmínky. Úloha má proměnné $X_1, X_2, \ldots X_n$ a podmínky $C_1, C_2, \ldots C_n$ na tyto proměnné. V počátečním stavu jsou všechny proměnné volné – nemají přiřazené žádné hodnoty. Použití operátoru přiřadí hodnotu některé proměnné. Řešením je stav, ve kterém mají všechny proměnné přiřazené hodnoty vyhovující všem podmínkám. Některé úlohy můžou vyžadovat navíc hledání extrému hodnot některých proměnných. Příkladem CSP je problém n dam (rozestavení n dam tak, aby se vzájemně neohrožovaly). CSP lze řešit všemi metodami prohledávání stavového prostoru, ale existují i mnohem efektivnější metody, mezi které patří např. **backtracking for CSP**, **forward checking** (po každém přiřazení hodnoty proměnné eliminuje zbývající možné hodnoty proměnných, které jsou s ní v konfliktu) nebo **min-conflict** (vychází z libovolného, i nelegálního stavu, a snaží se snižovat počet konfliktů).



Obrázek 49: podproblémy OR, AND a smíšený

Metody rozkladu úloh na podproblémy jsou metody, které při obtížných úlohách využívá i člověk. Snažíme se problém rozložit na elementární řešitelné podproblémy. Existují 3 typy podproblémů:

- **OR –** Podproblém je řešitelný, je-li řešitelný alespoň jeden z jeho podproblémů.
- **AND –** Podproblém je řešitelný, jsou-li řešitelné všechny jeho podproblémy.
- **smíšený –** Ostatní, lze vždy převést na čistě AND nebo OR (pomocí pomocných uzlů).

Dále budeme uvažovat pouze OR a AND uzly. Základním neinformovaným algoritmem pro řešení těchto úloh je:

- **AO –** Algoritmus je následující:
  1. Sestroj seznam OPEN a CLOSED, do OPEN dej počáteční uzel (problém).
  2. Vyjmi zleva ze seznamu uzel X.
  3.
     a) Je-li X elementární řešitelný problém, přenes informaci o řešitelnosti výš. Je-li počáteční uzel řešitelný, máme řešení.
     b) Je-li X elementární neřešitelný problém, přenes informaci o neřešitelnosti výš. Je-li počáteční uzel neřešitelný, řešení neexistuje.
     c) Jinak expanduj X a všechny jeho následníky ulož do OPEN.
  4. Ulož X do CLOSED.
  5. Je-li seznam OPEN prázdný, ukonči hledání jako neúspěšné. Jinak jdi na bod 2.

Metody hraní her se zabývají výběrem tahu hráče ve hře. Budeme uvažovat pouze hry pro dva hráče, kdy se hráči pravidelně střídají a mají úplnou informaci o hře. Hráče nazveme A a B, tahy určujeme pro hráče A. Využívá se princip rozkladu na podproblémy – stav, kdy je na tahu hráč A, je řešitelný, vede-li alespoň jeden jeho tah k výhře hráče A (OR problém), zatímco stav, kdy je na tahu hráč B, je řešitelný, vedou-li všechny jeho tahy k výhře hráče A (AND problém). Hledání tahů vedoucích k výhře je tedy procházením AND/OR grafu. Hry dělíme na:

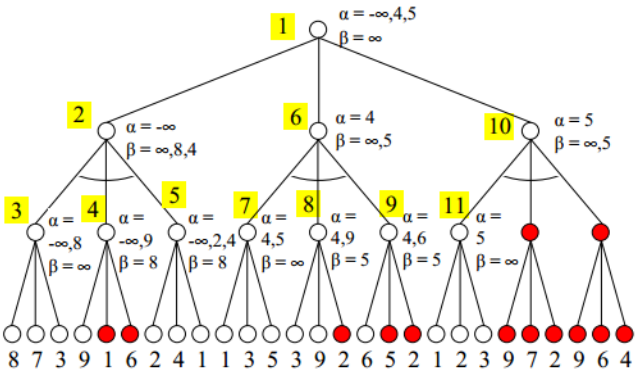
- **jednoduché** – V konečném čase lze prohledat celý AND/OR graf (např. odebírání zápalek). Lze použít klasický AO algoritmus.
- **složité** – V konečném čase nelze prohledat celý AND/OR graf (např. šachy). Prohledávání je proto omezené na určitou hloubku a musí existovat **ohodnocovací funkce**, která hodnotí stav hry. Není-li v dané hloubce nalezena možnost výhry/prohry, snaží se algoritmus dostat do stavu, který ohodnocovací funkce určí jako nejlepší.

Základní algoritmus hraní (složitých) her je:

- **MiniMax –** Název je odvozen od faktu, že se střídavě hledá minimum a maximum ohodnocovací funkce (hráč A se snaží o minimum, hráč B o maximum). Základem algoritmu je procedura MiniMax:
  1. Vstupní uzel nazvěme X.
  2. Je-li X v maximální hloubce nebo jedná-li se o koncový stav (výhra/prohra), vrať ohodnocení uzlu.
  3.
     a) Je-li na tahu hráč A, volej proceduru MiniMax pro všechny možné tahy a vrať maximum z navrácených hodnot.
     b) Je-li na tahu hráč B, volej proceduru MiniMax pro všechny možné tahy a vrať minimum z navrácených hodnot.

Algoritmus MiniMax lze zrychli použitím tzv. **Alfa/Beta ořezávání**. Alfa řezy zabraňují zbytečnému vyšetřování tahů hráče A, Beta řezy zase zbyteřnému vyšetřování tahů hráče B. Upravený algoritmus je následující:

1. Vstupní uzel nazvěme X.
2. Je-li X počáteční uzel, $\alpha := -\infty, \beta := \infty$
3. Je-li X koncový stav nebo je-li v maximální hloubce, vrať ohodnocení uzlu X.
4.
   a. Je-li na tahu hráč A (OR uzel), pak:
      i. Dokud je $\alpha < \beta$, volej postupně pro každý další tah MiniMax s aktuálními hodnotami α a β. Po každém vyšetřeném tahu nastav α na maximum z aktuální a navrácené hodnoty.
      ii. Vrať hodnotu α.
   b. Je-li na tahu hráč B (AND uzel), pak:
      i. Dokud je $\alpha < \beta$, volej postupně pro každý další tah MiniMax s aktuálními hodnotami α a β. Po každém vyšetřeném tahu nastav β na maximum z aktuální a navrácené hodnoty.
      ii. Vrať hodnotu β.

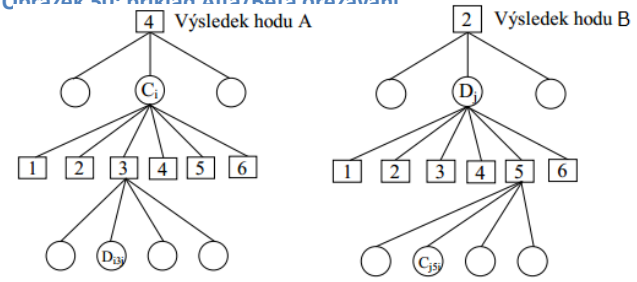


Obrázek 50: příklad Alfa/Beta ořezávání

Hry s neurčitostí jsou hry, v nichž se uplatňuje náhoda (hrací kostka). Jako algoritmus se u těchto her používá MiniMax s modifikací výpočtu minima a maxima na:

$$expectmin(C_i) = \sum_k P(h_k) \min(D_{ijk})$$

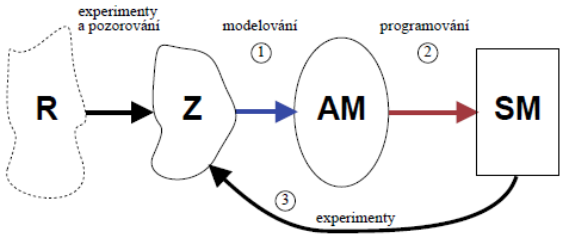
$$expectmax(D_i) = \sum_k P(h_k) \max(C_{ijk})$$



25. Principy modelování a simulace systémů (systémy, modely, simulace, algoritmy řízení simulace). IMS

Systém je soubor elementárních prvků, které mezi sebou mají určité vazby. Dělíme je např. na **reálné** a **nereálné** (existující nebo fiktivní) nebo **statické** a **dynamické** (podle toho, zda mění svůj stav v čase, pro simulace jsou zajímavé především dynamické systémy). **Model** je napodobenina systému jiným systémem, např. počítačovým programem. Model by měl dostatečně přesně napodobovat všechny důležité aspekty své předlohy. **Modelování** je process vytváření modelů na základě znalostí. Jde o náročný process vyžadující spolupráci více oborů. **Simulace** je metoda získávání znalostí o systému experimentováním s jeho modelem.

Formálně je systém dvojice (U,R). U je množina prvků ve tvaru (X,Y), kde X resp. Y je množina vstupních resp. výstupních proměnných. R je množina relací $R_{ij} \subseteq Y_i \times X_j$ (propojení prvku i s prvkem j).



Obrázek 51: princip modelování a simulace

Principem modelování a simulace je:

1. Vytvoření tzv. **abstraktního modelu** systému na základě našich znalostí. Abstraktní model je matematický model systému, z nějž následně vznikne simulační model. Tento model zanedbává a zjednodušuje určité pro simulaci málo důležité aspekty původního systému.
2. Vytvoření tzv. **simulačního modelu** na základě abstraktního modelu. Tento model dál již nic nezjednodušuje, pouze převádí abstraktní model do podoby, s níž se dají provádět experiment (např. počítačový program).
3. Provádění experimentů se simulačním modelem, na jejichž základě získáme informace a jejich analýzou nové znalosti o systému. Celý proces můžeme opakovat s novými znalostmi.

Systémy můžeme klasifikovat na:

- **spojité** – Všechny prvky mají spojité chování.
- **diskrétní** – Všechny prvky mají diskrétní chování.
- **kombinované** – Existují spojité i diskrétní prvky.

nebo na:

- **deterministické** – Všechny prvky jsou deterministické.
- **nedeterministické** – Existuje alespoň jeden nedeterministický prvek.

Modely můžeme klasifikovat na:

- **konceptuální** – Neformální schémata a slovní popis modelu, většinou je výchozím popisem pro sestavení modelu.
- **deklarativní** – Popisují změnu stavu systému na základě událostí, jsou popsány grafem podobným stavovému automatu. Patří sem např. Petriho sítě.
- **funkcionální** – Popisují systém pomocí grafů obsahujících fukce a proměnné. Používají se tam, kde se vyskytuje přímý tok mezi entitami systému. Patří sem např. logické obvody.
- **rovnicové a grafové** – Popsané např. diferenciálními rovnicemi, jsou vhodné pro reprezentaci přírodních zákonů.
- **prostorové** – Vyjadřují prostorovou dekompozici systému, patří sem např. celulární automaty.
- **multimodely** – Model vzniklý propojením různých typů modelů.

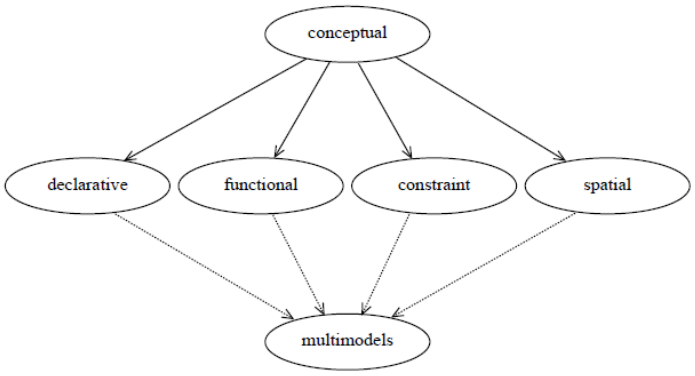
Alternativou k simulačnímu přístupu je analytické řešení problem, které je přesné, avšak většinou příliš složité nebo nemožné. Modely můžeme **validovat** (dokazovat, že pracujeme s modelem adekvátním modelovanému systému) nebo **verifikovat** (ověřovat korespondenci simulačního a abstraktního modelu). Simulace (a modely) dělíme na **diskrétní** a **spojité** podle toho, jak reprezentují časovou dimenzi.

Čas bereme jako množinu T všech okamžiků, ve kterých jsou definovány hodnoty vstupních, výstupních a stavových proměnných. Při simulaci rozlišujeme tři druhy času:

- **reálný** – Čas, v němž probíhá skutečný děj v reálném světě.
- **modelový** – Časová osa modelu. Může být reálný (ekvivalentní reálnému času), zrychlený nebo zpomalený.
- **simulační** – Čas CPU spotřebovaný na výpočet simulace (závisí na složitosti modelu, nesouvisí přímo s modelovým časem).

Základní datovou strukturou diskrétní simulace je tzv. **kalendář událostí** (anglicky next event). Jde o uspořádanou prioritní frontu záznamů naplánovaných událostí. O každé události se uchovávají následující informace:

- čas aktivace
- priorita (pro případ vice současných událostí)
- odkaz na událost



Obrázek 52: možná klasifikace modelů

Celá diskrétní simulace se potom řídí následujícím **algoritmem řízení diskrétní simulace**:

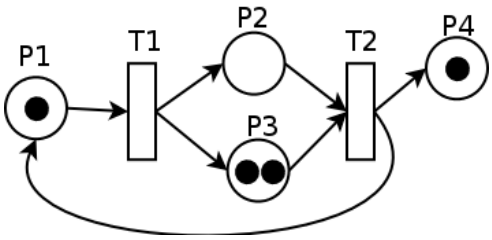
```
čas = T_START
inicializace kalendáře událostí a modelu

while (kalendář je neprázdný)
  {
    vyjmi první aktivační záznam (AZ) z kalendáře

    if (čas aktivace v AZ > T_END)
      break                          // ukončení cyklu

    nastav čas na aktivační čas v AZ
    proveď popis chování události odkazované v AZ
  }

čas = T_END                          // konec simulace.
```

Spojitá simulace se řídí následujícím **algoritmem řízení spojité simulace**:

```
inicializace // nastavuje počáteční stav stavových proměnných a simulátoru

while (čas < T_END)
  {
    if (vhodný čas)
      výstup sledovaných hodnot

    // krok numerické integrace:

    vyhodnocení derivací (vstupů integrátorů)
    výpočet nového stavu integrační metodou
    posun modelového času
  }

výstup sledovaných hodnot // konec
```

Systém hromadné obsluhy (SHO) je systém obsahující **transakce** (procesy), **obslužné linky** (různých typů) a **fronty** (různých typů). Při simulaci SHO sledujeme informace o průběhu transakce, doby čekání ve frontách a vytížení obslužných linek. SHO můžeme klasifikovat tzv. Kendallovou klasifikací, při níž uvádíme tři vlastnosti systému zapsané ve formátu $X/Y/c$, kde X je způsob příchodu požadavků, Y rozložení doby obsluhy a c počet obslužných linek. Parametr c zapisujeme číslem, zbylé dva pak symbolem:

symbol	X	Y
M	Poissonův proces příchodů.	Exponenciální rozložení doby obsluhy.
E_k	Erlangovo rozložení intervalů mezi příchody.	Erlangovo rozložení doby obsluhy.
K_n	Rozložení x^2 intervalů mezi příchody.	Rozložení x^2 doby obsluhy.
D	Pravidelné (deterministické) příchody.	Konstantní doba obsluhy.
G	Žádné předpoklady o době mezi příchody.	Jakékoliv rozložení doby obsluhy.
GI	Rekurentní proces příchodů.	



Obrázek 53: příklad Petriho sítě

Příkladem může být systém *M/M/1*, který má jednu obslužnou linku, do níž chodí procesy podle Poissonova rozložení a jsou obsluhovány po dobu danou Exponenciálním rozložením. Některé typy systémů, jako např. právě *M/M/1* lze řešit analyticky.

Obslužné linky dělíme na **zařízení** (**facility**, kapacity využívajících procesů je 1, příkladem je např. tiskárna, kterou může využívat maximálně jeden proces) a **sklady** (**store**, kapacity je větší než 1, příkladem je půjčovna aut).

Petriho síť je matematická reprezentace diskrétních distribuovaných systémů. Jedná se o graf obsahující **místa**, **přechody** a **hrany**. Hrany jsou pouze mezi místy a přechody, nikoliv mezi dvěma místy nebo dvěma přechody. Místa mohou obsahovat libovolný počet tzv. **tokenů**. Přechody mohou tokeny tzv. **vystřelovat** (přemisťovat) do jiných míst. To lze provést jen tehdy, je-li v každém ze vstupních míst přechodu alespoň jeden token. Vystřelením se odebere jeden token z každého vstupního místa a přesune se do místa výstupního.

Metoda **Monte Carlo** je třída algoritmů pro simulaci systémů. Jde o stochastické (na náhodě založené) metody používající pseudonáhodná čísla. Mají široké využití, používají se např. pro výpočet vícerozměrných integrálů, kde jiné metody nemusí být efektivní. Typickou úlohou je výpočet čísla π tak, že generujeme náhodné body ve čtvercové části roviny, v níž se nachází kruh. Pomocí poměru bodů, které padnou mimo kruh a do kruhu lze přibližně číslo π vypočítat. Chyba metody je rovna $\frac{1}{\sqrt{N}}$, kde *N* je počet provedených pokusů.

26. Datové a řídicí struktury. IZP, IAL

Datová struktura je způsob uložení dat tak, aby se dala efektivně využít. Dělíme je na:

- **statické** – Nemění svou velikost ani strukturu, jsou to např. pole nebo struktury apod.
- **dynamické** – Mění svou velikost a/nebo strukturu, jsou to např. lineární seznamy, stromy apod. S dynamickými datovými strukturami pracujeme pomocí **ukazatelů,** neboli odkazů do paměti. Programovací jazyk nám většinou nabízí možnost alokovat a uvolňovat místo v paměti, díky čemuž můžeme dynamicky měnit velikost struktur.

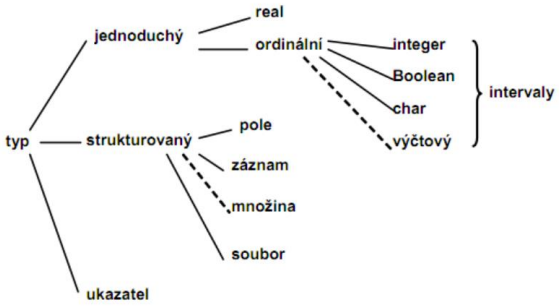
Nebo na:

- **homogenní** – Všechny komponenty struktury jsou téhož typu (např. pole).
- **heterogenní** – Komponenty struktury nejsou obecně téhož typu (např. záznam).

Datový typ je množina hodnot, kterých mohou data nabývat, spolu s množinou operací, které s nimi lze provádět. **Abstraktní datový typ** (**ADT**) je totéž, pouze s tím, že hodnoty a operace jsou definovány matematicky, omezujícími podmínkami na jejich výsledek, nikoli příkazy. ADT tak odstiňuje konkrétní implementační detaily. Pojem ADT se často používá pro uživatelem definované typy. Příkladem datových typů jsou: celé číslo, reálné číslo, řetězec, pole znaků apod.

Datové typy se dělí na:

- **jednoduché** (**primitivní**, **skalární**) – Základní, dále nedělitelný datový typ, jako např. číslo, logická hodnota apod.
- **strukturované** – Datový typ složený určitým způsobem z jiných datových typů. Příkladem je např. pole čísel, nebo zásobník stromů.
- **ukazatel** – Uchovává adresu v paměti.



Obrázek 54: datové typy v jazyku Pascal

Datový typ může být dále **ordinální**. Potom platí, že každá hodnota (s výjimkou první a poslední) má hodnotu následníka a předchůdce (např. celá čísla).

Příklady datových struktur jsou:

- **pole** (**array**) – Homogenní statická struktura, skládá se z prvků adresovaných celočíselným indexem. Může být vícerozměrné (adresované více indexy).
- **seznam** (**list**) – Homogenní, lineární, dynamická struktura. Jde o množinu prvků propojených bez větvení, z nichž každý obsahuje data (libovolný, i strukturovaný dat. typ). Seznam může být dále:
  - **lineární –** Každý prvek má jednoho předchůdce a následníka, s výjimkou prvního (nemá předchůdce) a posledního (nemá následníka).
  - **dvojsměrný** – Prvky obsahují navíc zpětné ukazatele pro jednodušší zpětný průchod.
  - **kruhový** (**cyklický**) – Následníkem posledního prvku je první, předchůdcem prvního prvku je poslední.
  - **s aktivitou –** Zahrnuje možnost aktivity prvku – maximálně jeden prvek může být aktivní.
- **zásobník** (**stack**) **–** Homogenní, lineární, dynamická struktura, která vznikne z lineárního seznamu, omezíme-li přístup k prvkům pouze na první prvek. Jde o strukturu typu **LIFO** (last in first out) – prvke, který byl do zásobníku uložen naposled, je z něj vybrán jako první. Základními operacemi jsou **Push** (vložení prvku na vrchol zásobníku), **Pop** (zrušení prvku na vrcholu zásobníku) a **Top** (přečtení prvku na vrcholu zásobníku). Dalšími operacemi jsou např. inicializace nebo zjitění prázdnosti.
- **fronta** (**queue**) **–** Homogenní, lineární, dynamická struktura podobná zásobníku s tím rozdílem, že prvky se vkládají na jednom konci (konec) a vybírají na druhém (začátek). Jde o strukturu typu **FIFO** (first in first out) – prvek vložený nejdříve bude také nejdříve vybrán. Základními operacemi jsou **QueUp** (vložení prvku na konec fronty), **Remove** (zrušení prvku na začátku fronty) a **Front** (čtení prvku na začátku fronty). Dalšími operacemi jsou např. inicializace nebo zjitění prázdnosti.
- **vyhledávací tabula** (**look-up table**, **search table**) – Jedná se o pole, které nahrazuje výpočet hodnoty nějaké funkce tabulkou dvojic vstup:výstup. Příkladem může být např. barevná paleta mapující šedotónové hodnoty na konkrétní barvy.
- **strom** (**tree**) **–** Dynamická struktura skládající se z uzlů. Každý uzel může mít *n* potomků (jiných uzlů). Počáteční uzel se nazývá **kořen**, koncové uzly **listy**. Pro binární strom existují následující způsoby rekurzívního průchodu:
  - **preorder –** Nejdříve se uzel zpracuje, potom se postupuje do levého následníka a potom do pravého.
  - **inorder –** Nejdřív se postupuje do levého následníka, potom se uzel zpracuje a potom se postupuje do pravého následníka.
  - **postorder –** Nejdřív se postupuje do levého následníka, následné do pravého a potom se uzel zpracuje.
- ...

Řídící struktury jsou konstrukce vyšších programovacích jazyků, které umožňují ovlivňovat vykonávání programu. Dijkstra dokázal, že jakýkoli program se dá zapsat pomocí tří základních konstrukcí:

- **sekvence** (**posloupnost**) – příkazy vykonávané v předem daném pořadí
- **selekce** (**větvení**) – struktura umožňující na základě podmínky vykonat buď jeden nebo druhý blok příkazů
- **iterace** (**cyklus**) – struktura umožňující opakované provádění bloku příkazů, dokud platí určitá podmínka

Strukturované programovací jazyky zavádí vždy alespoň tyto možnosti, avšak většinou nabízí více řídících struktur z důvodu usnadnění práce, jsou jimi např.:

- **multivětvení** (**case**, **switch**) – větvení s více než dvěma větvemi, konkrétní větev se provede podle hodnoty určitého daného výrazu
- **cyklus for** – cyklus s předem známým počtem provedení
- **cyklus while** – cyklus s podmínkou na začátku
- **cyklus do while** – cyklus s podmínkou na konci
- **goto** – skok na konkrétní řádek programu, tento příkaz však nemá téměř nikdy využití a ve strukturovaném programování se bez něj lze zcela obejít, navíc výrazně snižuje přehlednost programu a proto se silně doporučuje nepoužívat

Ve většině případů je vhodnější volit složitější datové struktury a jednoduché algoritmy.
Při **rekurzívním volání** procedura volá sama sebe. Aby nedošlo k zacyklení, je nutné mít ukončovací podmínku.

Selektor je operace, která umožňuje přístup k položkám datové struktury. **Iterátor** je operace, která se provede nad každou položkou homogenního datového typu.

27. Vyhledávání a řazení. IAL

Vyhledávání a řazení jsou v informatice velmi frekventované operace a proto existuje velké množství algoritmů pro jejich realizaci. Vyhledáváním rozumíme proces, při němž máme dán vyhledávaný klíč a snažíme se v určité datové struktuře nalézt prvek s tímto klíčem. Vyhledávací klíč může mít různé vlastnosti, může být **jednoduchý** (řetězec, …) nebo **složený** (jméno a rok narození, …), s **relací uspořádání** (čísla, …) nebo pouze **ekvivalence** (objekty, …) apod. Rovněž nás u vyhledávání zajímají různé atributy jako např. minimální/maximální/průměrná doba úspěšného/neúspěšného vyhledání apod. Následují vybrané metody vyhledávání:

- **sekvenční vyhledávání v poli** – Procházíme pole od prvního prvku k poslednímu a porovnáváme klíč aktuální položky s vyhledávaným klíčem. Výhodou je jednoduchá realizace datové struktury i algoritmu, klíč musí mít definovanou pouze relaci ekvivalence. Časová složitost je evidentně lineární. Tuto metodu lze mírně vylepšit na **vyhledávání se zarážkou** (také **rychlé sekvenční vyhledávání**) tak, že před vyhledáváním se nejdříve podíváme na poslední prvek pole a pokud tam nenalezneme hledaný klíč, vložíme na toto místo prázdnou položku s hledaným klíčem – tzv. **zarážku**. Tím v algoritmu nemusíme testovat konec pole - vyhledávání vždy skončí úspěšně, avšak pokud nalezneme prvek na posledním místě pole, vyhledávání bereme jako neúspěšné (v původním poli prvek na tomto místě nebyl).
- **sekvenční vyhledávání v seřazeném poli** – Stejný princip jako klasické sekvenční vyhledávání, pouze s rozdílem, že skončíme procházení, pokud narazíme na klíč větší než vyhledávaný klíč – urychlí se pouze neúspěšné vyhledání! Klíč nyní musí mít definovanou relaci uspořádání a pole musí být seřazeno od nejmenšího prvku k největšímu (musí se řešit při vkládání prvků).
- **sekvenční vyhledávání s adaptivní rekonfigurací pole podle četnosti** – Vyhledávání probíhá opět sekvenčně, ale při každém úspěšném vyhledání se nalezený prvek vymění se svým levým sousedem (pokud je to možné) – tím se častěji vyhledávané prvky dostanou blíže k začátku pole a budou rychleji vyhledávány.
- **binární vyhledávání v seřazeném poli** – Vyhledávání je s náhodným přístupem (tj. nikoli sekvenční). Pole musí být seřazeno. Principem je půlení pole v každém kroku – nejdříve porovnáme vyhledávaný klíč s klíčem uprostřed pole a pokud je hledaný klíč větší resp. menší, provádíme to samé s pravou resp. levou polovinou pole. Tak postupujeme stále dál. Časová složitost je proto **logaritmická** – v nejhorším případě se prvek nalezne v $log_2 n$ kroků, což je výrazně lepší než při vyhledávání sekvenčním.
- **Dijkstrova varianta binárního vyhledávání** – Jedná se o variantu binárního vyhledávání pro vyhledávání v poli, kde se potenciálně mohou vyskytovat stejné klíče. V takovém případě většinou požadujeme nalezení např. nejpravějšího prvku. Vyhledávání v tomto případě neskončí nalezením prvku s odpovídajícím klíčem, ale dál se půlí, až se nalezne klíč, pro který platí, že je shodný s vyhledávaným klíčem a zároveň je menší než klíč o jeden prvek vpravo.

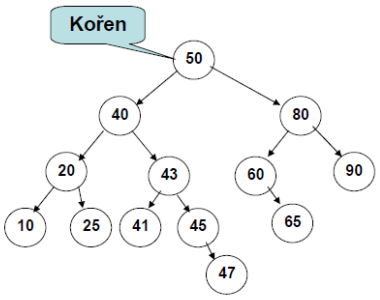


Figure 19: BVS

- **Uniformní vyhledáván –** Varianta binárního vyhledávání pro systémy, v nichž je operace půlení intervalu (dělení dvěma) časově náročná (např. systém pracující v BCD kódu). Systém má předpočítanou sadu odchylek od středu pro různé úrovně vyhledávání. Např. pole o s indexy 1 až 17 má odchylky 8, 4, 2, 1. Na začátku se spočítá střed jako první index + první odchylka a každý další střed se spočítá jako aktuální střed + nebo – aktuální odchylka.
- **Fibonacciho vyhledávání –** Podobně jako uniformní vyhledávání pomáhá v systémech, kde je půlení časově náročné. Používá **Fibonacciho stromy**.
- **vyhledávání v binárním vyhledávacím stromu (BVS)** – Binární vyhledávací strom je speciálním případem stromu, u nějž každý uzel může mít maximálně dva následníky, levého a pravého, a klíč každého uzlu je větší resp. menší než klíče všech prvků v levém resp. pravém podstromu. Tato datová struktura je pro vyhledávání velmi vhodná, neboť počet kroků potřebný k vyhledání kteréhokoli prvku je v nejhorším případě roven výšce stromu. Binarní strom lze vylepšit tak, aby se sám vyvažoval pomocí tzv. rotací, čímž si udržuje minimální výšku a maximální efektivitu. Takový strom nazýváme **AVL** strom.
- **vyhledávání v hashovací tabulce** – **Tabulka s rozptýlenými hodnotami** (hashovací tabulka) je podobně jako BVS datová struktura určená k rychlému vyhledávání. Jedná se o pole, které je klasicky schopné ukládat v každé položce zřetězený seznam. Při vkládání/hledání položek se počítá tzv. **hash** (číslo) daného klíče pomocí **hashovací funkce** (která by měla být rychlá a i pro málo odlišné hodnoty by měla dávat velmi odlišný výsledek). Hash potom určuje index, na němž se má položka nacházet. Při konfliktech se uloží více položek na jeden index do seznamu (hledání je potom **indexsekvenční**), popř. se použije tzv. otevřené adresování, kdy pro konfliktní položku (tzv. synonymum) hledáme volné místo dál v tabulce nějakým předem daným způsobem. Operace s hashovací tabulkou mají teoreticky konstantní časovou složitost (lepší než BVS), avšak kvůli konfliktům se může změnit až na lineární (horší než BVS). Hashovací tabulka také neumožňuje vyhledávání v určitém intervalu nebo jen částečně specifikovaný klíč.

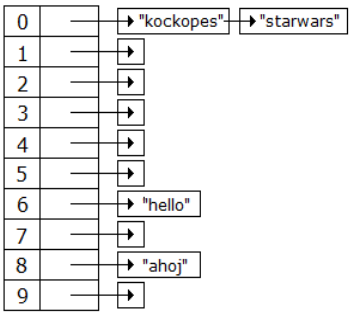


Figure 20: hashovací tabulka s konfliktem

K vyhledávání v řetězcích se používá **KMP algoritmus** nebo **Boyer Mooreův** algoritmus.

Řazení (ordering) je uspořádání položek lineární homogenní datové struktury podle relace uspořádání nad zadanou vlastností (klíčem) položek. Formálně jde o nalezení permutace prvků tak, aby platilo, že $x_i < x_{i+1}$ pro všechna $i < n$. Často se nesprávně zaměňuje s pojmem třídění (sorting), což je rozdělování do tříd. Často nás u těchto algoritmů zajímají tyto vlastnosti:

- **stabilita** – Je-li v neseřazené posloupnosti více položek se stejným klíčem, pak budou po seřazení mít vůči sobě stejné relativní uspořádání.
- **přirozenost** – Doba řazení již seřazeného pole je menší než doba řazení náhodně uspořádaného pole a tak je menší než doba řazení opačně seřazeného pole.
- **řazení na místě** (**in situ**) – K řazení není potřeba žádná významná paměť navíc, vše probíhá v paměti řazené datové struktury.
- **řazení bez přesunu položek** – Neseřazuje se samotné pole hodnot, ale pouze pole indexů udávající pořadí těchto hodnot. To je výhodné tehdy, zabírají-li položky řazeného pole hodně místa a při řazení by docházelo k velkým přesunům dat.

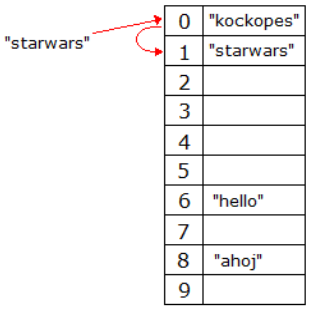


Figure 21: hash. tabulka s konfliktem a otevřeným adresováním

Důležitými metodami řazení jsou:

- **select sort** (**řazení výběrem**) – Nestabilní metoda s kvadratickou složitostí. Je velmi jednoduchá a pro člověka přirozená, spočívá v cyklu, který prochází postupně polem a pro danou pozici vždy hledá extrém (např. minimum).
- **bubble sort –** Stabilní, přirozená metoda s kvadratickou složitostí. Princip spočívá ve dvou cyklech, jeden postupně prochází polem od začátku ke konci a druhý prochází od aktuální položky ke konci a vždy testuje dvojici sousedních prvků. Je-li levý větší než pravý, prvky se prohodí, jinak se nedělá nic.
- **heap sort** (**řazení hromadou**) **–** Nestabilní, nepřirozená metoda s linearitmickou složitostí. Řadí pomocí datové struktury **hromada** (**heap**). Hromada je strom, u nějž má otcovský uzel a uzel potomka vždy stejnou relaci uspořádání (např. otec je vždy větší než syn). Binární hromada je hromada, která je binárním stromem. Principem je opakování akcí: vytvoření hromady (v poli), odebrání horního (maximálního) prvku a jeho zařazení do pole.
- **insert sort** (**řazení vkládáním**) – Metoda s kvadratickou složitostí. Postupně bere prvky v poli zleva doprava a hledá pro ně správné místo v již seřazené levé části pole.
- **bubble insert sort**
- **vkládání s binárním vyhledáváním –** Insert sort s modifikací dohledávání správného místa pro prvek, které se provádí binárním vyhledáváním místo lineárního. Časová složitost se však příliš nezlepší, protože sice urychlíme dohledání pozice pro prvek, ale počet přesunů nesnížíme.
- **quick sort –** Patří mezi nejrychlejší a nejpopulárnější metody řazení. Je nestabilní, nepřirozená a má linearitmickou časovou složitost. Pracuje na pricipu rekurzívního rozdělování pole na poloviny. V každé iteraci se vždy určí prvek, tzv. **pivot**, a ostatní prvky se potom přesunou tak, že nalevo od pivota jsou jen menší (nebo stejné) prvky a napravo jen větší. To samé se potom dělá s každou ze vzniklých polovin, až dokud v dané části není méně nebo rovno než dva prvky.
- **shell sort –** Nestabilní, rychlejší než heap sort ale pomalejší než quick sort. Metoda bývá těžko srozumitelná. V podstatě pracuje na principu vkládání s tím, že se prvky přibližují k místu, kam patří, s vyšším krokem.
- **merge sort –** Nestabilní, nepřirozená metoda s linearitmickou složitostí. Postupuje současně zleva i zprava a seřazuje dvě posloupnosti proti sobě.
- **radix sort –** Počítačová obdoba řazení tříděním na děrnoštítkových strojích, pracuje bez přesunu položek.

28. HTML a Javascript (z pohledu návrhu webových stránek). IIS

HTML (**hyper text markup language**) je **značkovací jazyk** (obsahový text převažuje nad formátovacími prvky), který původně vznikl jako podmnožina jazyka **SGML** (metajazyk pro popis jazyků jako svých podmnožin pomocí tzv. **DTD** (**document type definition**), dnes je nahrazován např. XML schématy), zatímco v poslední době je k dispozici verze HTML jako XML jazyka, tzv. **XHTML**. Rozdíl XHTML oproti klasickému HTML je např. ve vyžadování párovosti všech tagů, case-sentive syntaxi apod. Jazyk HTML slouží k popisu struktury dokumentu (nadpisy, odstavce, obrázky, seznamy, …) a jeho obsahu. V minulosti byl určen taky k definici vzhledu dokumentu, ale v dnešní době tuto úlohu přebírá jazyk **CSS** (kaskáda stylů: prohlížeč, uživatel a stránka) a vzhled tvořený pomocí HTML je považován za zastaralou a nevhodnou praktiku. Příkladem jednoduchého dokumentu ukazující strukturu dokumentu v HTML může být tento:

```
<!DOCTYPE html>              <!-- deklarace typu dokumentu, zde jenom pojmenování -->

<html>                       <!-- kořenový element -->
  <head>                     <!-- hlavička dokumentu -->
  </head>

  <body>                     <!-- tělo dokumentu -->
    <h1> Nadpis </h1>
    <p> Toto je odstavec. </p>
  </body>
</html>
```

Tag je text uzavřený v loměných závorkách (< a >). **Element** je vše, co se nachází mezi počátečník a koncovým tagem včetně tagů (např. *" abc "*). Element může mít:

- **obsah** – To, co je mezi tagy, tedy např. text nebo vnořené elementy.
- **atributy** – Upřesňují vlastnosti elementu a jsou zapsány v počátečním tagu (např. href v <a href="abc">).

Značky v HTML dělíme na:

- **strukturální** – Rozvrhují strukturu dokumentu (např. *<h1>, , ...*).
- **popisné** – Popisují povahu obsahu (např. *<title>, <address>, ...*).
- **stylistické** – Popisují vzhled (např. *<b>, <i>, ...*), od tohoto druhu značek se upouští a využívá se CSS.

Speciální znaky v HTML zapisujeme pomocí tzv. **entit**, např.   (nezlomitelná mezera), &lt (<), &#60 (<) atd.

Pojem **dynamické HTML** (**DHTML**) je zastřešující pojem pro techniky vytváření interaktivních, animovaných webových stránek. Tyto techniky využívají HTML, Javascript, CSS a DOM.

Hypertext je text obsahující **hypertextové odkazy**, což jsou odkazy na jiné dokumenty, ke kterým lze takto ihned přistoupit.

Značkování dělíme na:

- **procedurální** – Značky udávají přesnou sekvenci příkazů, které se provedou. Většinou nemohou být hierarchické. Patří sem např. jazyk Latex.
- **univerzální** – Značky slouží k popisu struktury dokumentu nebo mají určitou sémantiku, mohou být hierarchické. Patří sem např. jazyky XML a SGML.

Jazyk **Javascript** slouží k programování skriptů, které běží na straně klienta, tedy ve webovém prohlížeči a kromě názvu nemá nic společného s jazykem Java. Syntaxí se podobá jazykům C, C++ nebo Java. Napomáhá interaktivitě webových stránek, neboť nabízí možnosti zobrazovat zprávy, dialogy, dynamicky měnit vzhled i strukturu HTML dokumentu apod. Jedná se o prototypově orientovaný, dynamicky typovaný, interpretovaný, case-sensitive objektový jazyk a k HTML dokumentu jej lze připojit několika způsoby:

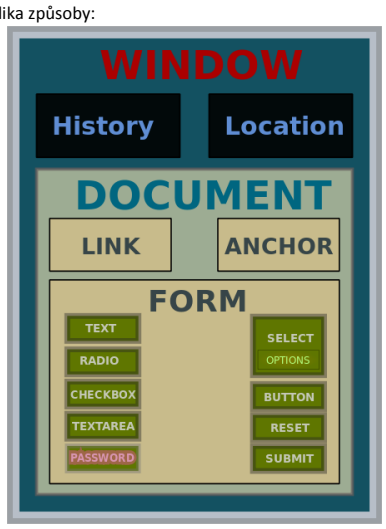
- Pomocí párového tagu *<script>*, do nějž zapíšeme kód. Tento kód se vykoná ihned, jakmile na něj prohlížeč narazí (může být v části *head* i *body*).
- V parametru události některých tagů, např. *onlick*, *onload* apod. Tento kód se vykoná, jakmile daný element vyvolá konkrétní událost.
- V externím souboru, na nějž v HTML dokumentu uvedeme odkaz tagem *<script src="soubor.js" />*.

Skript se spustí ihned, jakmile se na něj v dokumentu narazí. Dále je možné využít událostmi řízené programování, tzn. můžeme asociovat určitou funkci Javascriptu s událostí, která může nastat (např. stisknutí tlačítka, načtení stránky, událost časovače, ...). Implementace jazyka je závislá na konkrétním prohlížeči a uživatel jej může vypnout. Javascript neumí ukládat a načítat data v podobě souborů s výjimkou **cookies** (malý objem dat ukládaný v proxlížeči za účelem stavového chování).

Javascript pracuje s **DOM** (**document object model**) modelem. Jedná se o objektový model reprezentující dokument v HTML nebo XML. Globálním objektem v DOM je objekt **window** představující okno prohlížeče. Základní třídou DOM modelu je Node představující obecný uzel, který má operace pro určení pozice, charakteristiky nebo jeho modifikaci. Od Node potom dědí např. třídy Element, Entity, Attribute nebo Document.

Konkrétními objekty v DOM jsou potom např. Window (představuje okno prohlížeče), Navigator (obsahuje historii apod.) nebo Document (obsahuje odkazy na uzly dokumentu, seznam formulářů, odkazů apod.).

Pro navigaci v DOM slouží jazyk **XPATH**, který adresuje uzly ve tvaru krok1/krok2/krok3/... . Výsledkem příkazu je obecně množina uzlů např. příkaz A/B/C (ekvivalentní příkazu child::A/child::B/child::C) vybere všechny uzly C, které jsou v B, které jsou v A. Kromě příkazu ::child existují i další, jako např. following-sibling::, self:: apod. K použití XPATH v JavaScriptu slouží funkce getElementByXpath.



Obrázek 55: příklad DOM hierarchie

Javascript podporuje specifikaci objektů pomocí tzv. **JSON** (Javascript Object Notation) syntaxe. Jedná se o způsob, alternativní např. k XML, jak popsat objekt výčtem párů atribut:hodnota. Takto lze např. jednoduše vytvořit nový objekt nebo poslat objekt mezi klientem a serverem v textové podobě. Datové typy podporované JSONem jsou:

- číslo – číslo s desetinnou tečkou (např. *6.58*)
- řetězec – uzavřený v dvojitých uvozovkách (např. *"hello"*)
- logická hodnota – *true* nebo *false*
- pole – hodnoty oddělené čárkou uzavřené v hranatých závorkách (*např. ["abc", 3.5, false]*), může obsahovat jiná pole nebo objekty
- objekt – dvojice atribut : hodnota oddělené čárkou a uzavřené ve složených závorkách (např. *{name: "John", age: 20}*), může obsahovat jiné objekty nebo pole
- prázdná hodnota – *null*

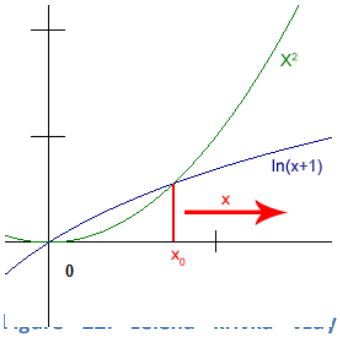
Ajax (**Asynchronous Javascript and XML**) je soubor technik programování klienta umožňující vytvářet asynchronní webové aplikace (tj. aplikace, které dokáží přijímat a posílat data v pozadí bez narušení načtené stránky). Zahrnuje technologie (X)HTML, CSS, DOM, XML, **XSLT** (jazyk pro transformaci XML jazyka na jiný XML jazyk), Javascript apod.

XSLT je jazyk pro transformaci jednoho XML jazyka na jiný.

29. Hodnocení složitosti algoritmů (paměťová a časová složitost, asymptotická časová složitost, určování časové složitosti). IZP, IAL

Časová resp. **pamětová** (či **prostorová**) **složitost** jsou kritéria hodnocení algoritmů, která udávají, jak se se zvyšujícím se objemem dat (*N*) zvyšuje potřebný čas resp. paměť pro vykonání algoritmu. **Asymptotická složitost** je nejčastější způsob klasifikace algoritmů podle jejich složitosti do tříd, u kterých platí, že od určité velikosti dat je algoritmus dané třídy vždy pomalejší než algoritmus třídy předchozí a to nezávisle na výkonnosti počítače, na kterém každý z nich spouštíme. Pokud spadají dva algoritmy do různých tříd asymptotické složitosti, pak vždy existuje takové množství dat, od kterého je asymptoticky lepší algoritmus vždy rychlejší bez ohledu na to, kolikrát je některý z počítačů výkonnější.
U většiny algoritmů záleží doba jejich běhu nejenom na velikosti dat, ale i na jejich povaze (např. zda je vstupní pole už seřazené nebo ne). Proto používáme různé notace značící různé třídy:

- $O(f(x))$ (**omikron**, big O) – Algoritmus probíhá asymptoticky stejně rychle nebo rychleji než *f(x)*. Jde o horní hranici potřebného času (nejhorší případ). Matematicky:
  $f(x) \in O(g(n)) \Rightarrow \exists c > 0, n_0 > 0: \forall n \geq n_0: 0 \leq f(n) \leq c * g(n)$
- $\Omega(f(x))$ (**omega**) – Algoritmus probíhá asymptoticky stejně rychle nebo pomaleji než *f(x)*. Jde o spodní hranici potřebného času (nejlepší případ).
  $f(x) \in \Omega(g(n)) \Rightarrow \exists c > 0, n_0 > 0: \forall n \geq n_0: 0 \leq c * g(n) \leq f(n)$
- $\Theta(f(x))$ (**theta**) – Algoritmus probíhá asymptoticky stejně rychle jako *f(x)* (tzn. je zároveň v $O(f(x))$ i $\Omega(f(x))$).
  $f(x) \in \Theta(g(n)) \Rightarrow \exists c_1 > 0, c_2 > 0, n_0 > 0: \forall n \geq n_0: 0 \leq c_1 * g(n) \leq f(n) \leq c_2 g(n)$



nakonec přeroste modrou

U asymptotické složitosti nás tedy nezajímá přesná hodnota složitosti, ale pouze její třída. Aditivní a multiplikativní konstanty nemají na třídu složitosti žádný vliv, tzn. algoritmus s funkcí složitosti X^2 bude patřit do stejné třídy jako algoritmus se složitostí $5X^2 + 10^{80}$.

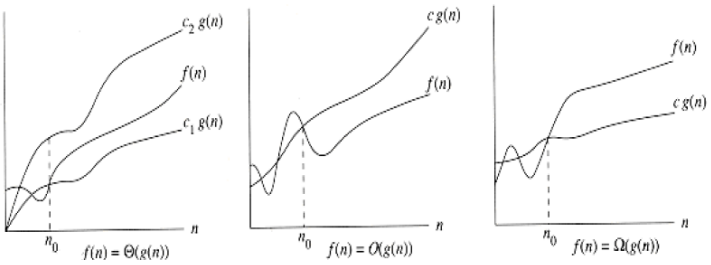


Figure 23: různá značení

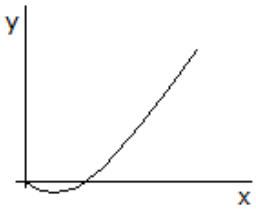
Třídy složitosti mají následující pojmenování:

$\Theta(1)$	**konstantní**
$\Theta(\log(n))$	**logaritmická** (základ není podstatný)
$\Theta(n)$	**lineární**
$\Theta(n \log(n))$	**linearitmická**
$\Theta(n^2)$	**kvadratická**
$\Theta(n^3)$	**kubická**
$\Theta(polynom)$	**polynomiální** (třída P)
$\Theta(k^n)$	**exponenciální**

Pokud pro určitý problém neexistuje polynomiálně omezený algoritmus, ale existuje polynomiální algoritmus, který ověří řešení problému, patří tento problém do tzv. třídy NP (nedeterministicky polynomiální – v polynomiálním čase jej řeší jen nedeterministický počítač). Tyto problémy jsou považovány za prakticky neřešitelné, protože čas potřebný k jejich vyřešení s rostoucími daty neúnosně roste.

Časovou složitost lze většinou do určité míry zlepšit na úkor paměťové složitosti a naopak (např. předpočítat si často používané hodnoty do paměti, abychom je nemuseli později počítat znovu).

Časová složitost se určuje buď exaktně nebo neformálně. Neformální analýza probíhá bez konkrétního matematického aparátu a je vhodná pro určování řádu složitosti. Při určování složitosti se typicky díváme na zanořené cykly v kódu a na to, jak jejich počet provedení závisí na vstupních datech.



Obrázek 56: linearitmická funkce

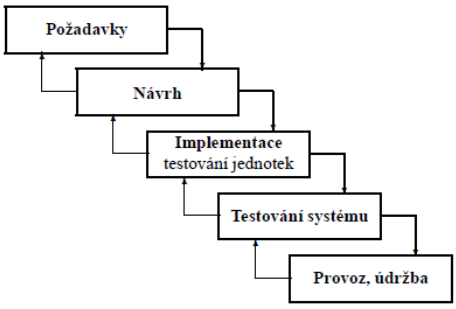
30. Životní cyklus softwaru (charakteristika etap a základních modelů). IUS

Softwarové inženýrství je systematický přístup k vývoji, nasazení a údržbě softwaru. Zabývá se např. **modely životního cyklu softwaru**. Každý model představuje přístup k vývoji softwaru a definuje etapy, v nichž bude vývoj probíhat, a jejich vstupy a výstupy. Etapy jsou následující:

- **analýza a specifikace požadavků** (cca 8%) – Počáteční fáze, v níž analyzujeme většinou nejasné požadavky zákazníka a přetváříme je na jasně definované požadavky na software. Pozornost by měla být věnována jen požadavkům a ne realizaci. Součástí etapy může být analýza rizik. Nezbytnou součástí je naproti tomu definice **akceptačních testů výsledného** produktu.
- **architektonický a podrobný návrh** (cca 7%) – Ujasnění koncepce systému a jeho dekompozice. Součástí by měly být plány na testování systému jako celku. Při podrobném návrhu se již zaměřujeme na konkrétní výběr algoritmů, způsobu implementace a testy jednotlivých součástí systému.
- **implementace** (cca 12%) – Zahrnuje vytvoření jednotlivých součástí systému programátory a také jejich testování. Vytváří se dokumentace.
- **integrace a testování** (cca 6%) – Spojení součástí do výsledného systému a jeho testování jako celku.
- **provoz a údržba** (cca 67%) – Řeší problémy spojené s nasazením systému, opravováním chyb, rozšiřováním softwaru a podporou pro uživatele.

Konkrétní modely životního cyklu softwaru jsou např.:

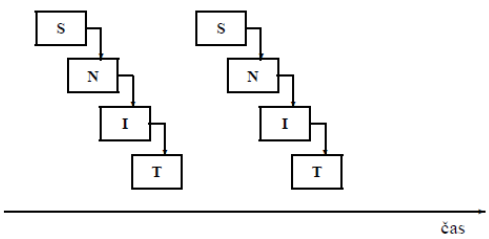
- **vodopádový model –** Základní model, v němž etapy následují za sebou a lze se vracet vždy o jednu zpět. Nevýhodou je absence uživatele během vývoje, tzn. zadavatel uvidí až finální produkt, se kterým nemusí být spokojen.
- **v-model –** Varianta vodopádového modelu s větším důrazem na testování.
- **Iterativní model** – Tento model se snaží vyřešit hlavní nevýhodu vodopádového modelu a zapojit uživatele do vývoje. Jeho jednotlivé fáze se skládají z vodopádů, na jejichž konci je vždy do jisté míry funkční verze systému, kterou může uživatel vyzkoušet a upřesnit své požadavky.
- **inkrementální model** – Velmi podobný iterativnímu modelu. Stanoví se ucelené části systému, které se postupně odevzdávají uživateli.
- **spirálový model** – Klade důraz na analýzu rizik a zavádí tzv. **prototypování**. Jednotlivé etapy vývoje se opět provádí opakovaně, avšak na rozdíl od iterativního modelu dostává uživatel na konci každé iterace pouze **prototyp**, ne funkčně omezenou verzi softwaru. Prototyp se liší od funkčně omezené verze systému tím, že se po otestování zahodí a software se vyvíjí znovu. Chyby se díky prototypům dají odhalit velmi brzo.



Obrázek 57: vodopádový model

Softwarový produkt dělíme na:

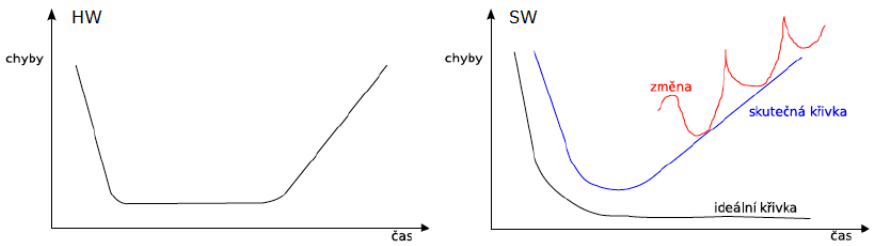
- **generický** – Krabicový software.
- **zákaznický** – Vytvářený na míru konkrétního zákazníka, je dražší, protože je prodán jen jednou.



Obrázek 58: iterativní model

Hodnotíme u něj tyto vlastnosti:

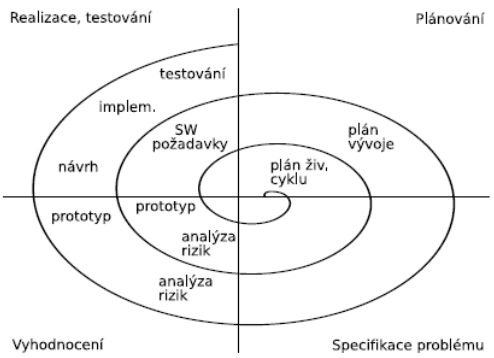
- správnost – Míra, do jaké software vyhovuje specifikaci.
- spolehlivost – Pravděpodobnost, že software bude v daném čase pracovat správně.
- efektivnost – Kritéria na využití zdrojů.
- použitelnost – Úsilí, které je nutné vynaložit na používání softwaru.
- bezpečnost – Míra odolnosti vůči neoprávněnému zásahu do systému.
- přenositelnost – Úsilí, které je nutné pro přenos softwaru z jedné platformy na jinou.
- znovupoužitelnost – Míra, do jaké lze software použít v jiných aplikacích.
- interoperabilita – Úsilí, které je potřebné k zajištění spolupráce softwaru s jiným systémem.
- udržovatelnost – Úsilí, které je nutné vynaložit na další vývoj a údržbu softwaru.
- pružnost (modifikovatelnost) – Úsilí nutné pro modifikaci výrobku.
- testovatelnost – Úsilí nutné pro testování softwaru.
- dokumentovanost – Míra, do jaké jsou všechna rozhodnutí během vývoje dokumentována.



Obrázek 59: chybová křivka hardwaru a softwaru

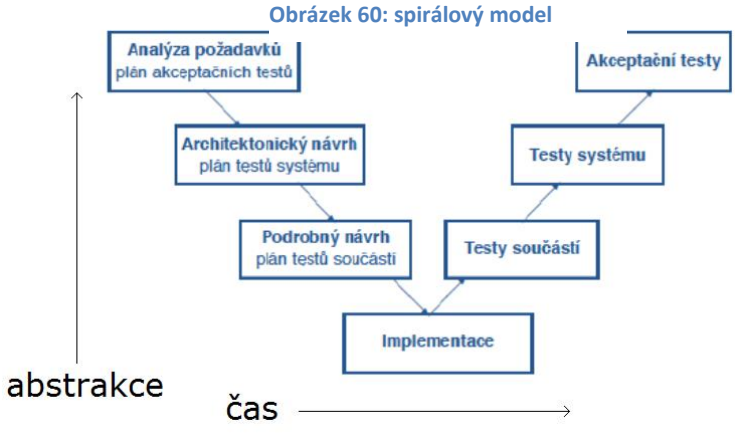
Proces vývjoje softwaru definuje, kdo, kdy a co má dělat, aby bylo dosaženo požadovaného cíle.

RUP (**rational unified process**) je konkrétní metodika vývoje softwaru. Je vhodněší pro větší projekty, protože klade důraz na analýzu, návrh a dokumentaci. RUP je založen na iterativním modelu (po každé iteraci je k dispozici spustitelný kód).



Obrázek 60: spirálový model

Pro menší projekty je vhodnější použít **agilní metodologie**. Jsou to alternativní, odlehčené metodologie ke klasickým (např. RUP) metodologiím, které se staly za dobu své existence poměrně složitými. Klade se důraz především na člověka jako nenahraditelného jednotlivce (oproti člověku jako nahraditelnému zdroji). Mezi agilní metodologie patří např. **extrémní programování**, které provádí běžné úkony dovedené do extrému (neustálé testování, revize kódu, párové programování, extrémní jednoduchost, extrémně krátké iterace, ...).



Obrázek 61: v-model

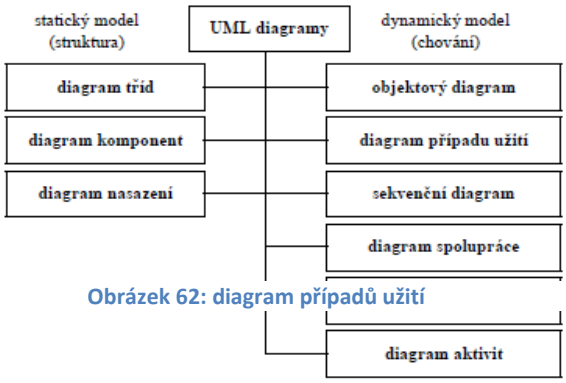
31. Jazyk UML. IUS

UML (**Unified Modeling Language**) je standardizovaný grafický jazyk využívaný v softwarovém inženýrství pro navrhování softwarových systémů. Návrhářům umožňuje využívat výhody objektového přístupu a díky jeho rozšířenosti existuje mnoho nástrojů, které umožňují např. automatické generování kódu z diagramů. UML je nástroj, ale nikoli metodika (neříká, jak se má systém navrhovat). Jazyk je vyvíjen od roku 1990 a od té doby vzniklo několik jeho verzí (mezi poslední patří 2.4.1).

UML pracuje s tzv. **pohledy** (**views**) – zaměřením se na určitý aspekt systému při ignorování ostatních. V základu rozlišujeme pohledy:

- **strukturální** (structural) – Zachycuje třídy, objekty, jejich vazby a komunikaci.
- **chování** (behavioral) – Zobrazuje interakce a reakce objektů.
- **datový** (data) – Zachycuje stavy systému a jejich vazby.
- **rozhraní** (interface) – Zaměřuje se na zapouzdření komunikaci systému s okolím.

Nástroji UML jsou diagramy, např. tyto:



Obrázek 62: diagram případů užití

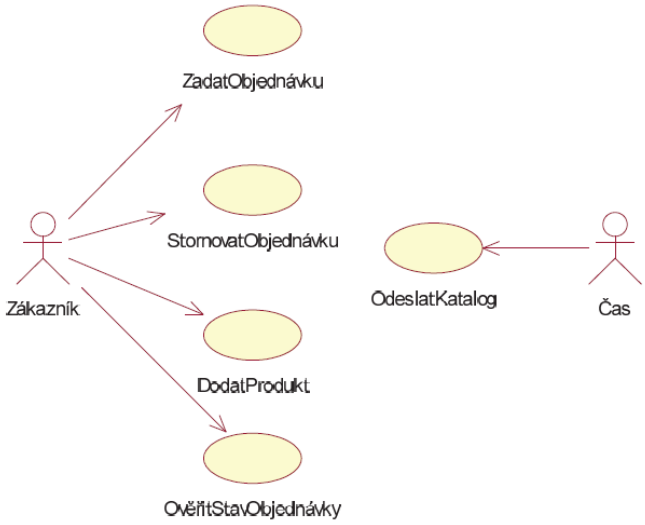


Diagram **případů užití** (**use case diagram**) zachycuje funkce, které systém vykonává jménem účastníků nebo v jejich prospěch. Zobrazuje hranice systému, účastníky, případy užití a interakce mezi nimi. Diagram samotný zobrazuje pouze to, jaké akce je možné vykonávat, proto se k němu navíc dodávají detaily jednotlivých případů užití. Pro tyto detaily neexistuje standard, většinou se však používá tabulka. Ta popisuje vstupní podmínky, tok událostí a výstupní podmínky. Tok událostí může obsahovat podmínky, cykly apod.

Existují pokročilé techniky, které umožňují zpřehlednit a zkvalitnit diagram případů užití. Patří mezi ně:

- **zobecnění účastníka** – Vyjádření generalizace/specializace účastníků, podobně jako v ER diagramu (nebo dědičnost v OO), naznačuje se šipkami mezi účastníky. Specializovaný účastník dědí vlastnosti generalizovaného.

název	**Případ užití: Platit daň z přidané hodnoty**
identifikátor	**ID: UC1**
účastníci	**Účastníci:** Čas finanční úřad
stav před	**Vstupní podmínky:** 1. Je konec fiskálního čtvrtletí?
kroky	**Tok událostí:** 1. Případ užití začíná na konci fiskálního čtvrtletí. 2. Systém určuje výši daně z přidané hodnoty, kterou je třeba odvést státu. 3. Systém odesílá elektronickou platbu finančnímu úřadu.
stav po	**Následné podmínky:** 1. Finanční úřad přijímá daň z přidané hodnoty ve spr

Obrázek 63: detail případu užití

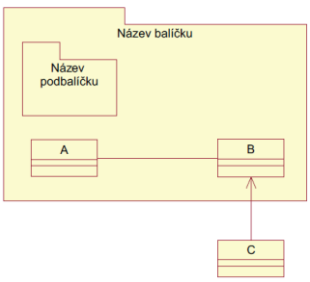
- **zobecnění případů užití** – Generalizace/specializace pro případy užití.
- **relace <<include>>** – Umožňuje naznačit (šipkou s <<include>>), že určité případy užití zahrnují jiné (např. změnit údaje o zákazníkovi zahrnuje vyhledání údajů o něm).
- **relace <<extend>>** – Umožňuje naznačit, že některý případ užití rozšiřuje jiný. Oproti <<include>> se tento případ nemusí provést vždy (např. uložení pokuty při vrácení knihy do knihovny).

Diagram tříd zobrazuje statickou strukturu systému ve formě tříd. U tříd uvádí jejich atributy a metody (ne všechny, pouze důležité). Může obsahovat i jiné elementy jako např. rozhraní.

ER (**entity relationship**) **diagram** zachycuje perzistentní data a používá se při konceptuálním modelování databází.

Objektové diagramy zachycují konkrétní instance (objekty) tříd a jejich vazby v určitém čase a podmínce (oproti např. diagramu tříd, který zobrazuje jen třídy). Umožňuje analyzovat konkrétní situace, které mohou v systému nastat.

UML rozlišuje pojmy **analytická** a **návrhová třída**. Analytická třída zachycuje jen nejpodstatnější atributy, operace a vazby. Slouží k identifikaci entit v řešeném problému a ve fázi návrhu je potom upřesněna do jedné nebo více návrhových tříd.

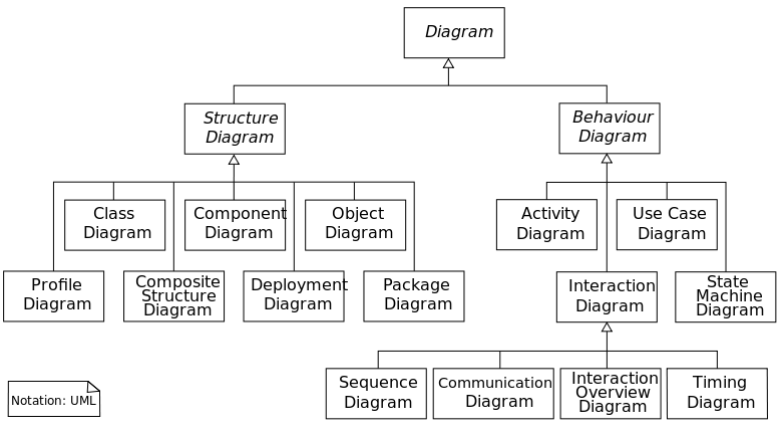


Obrázek 64: analytické balíčky

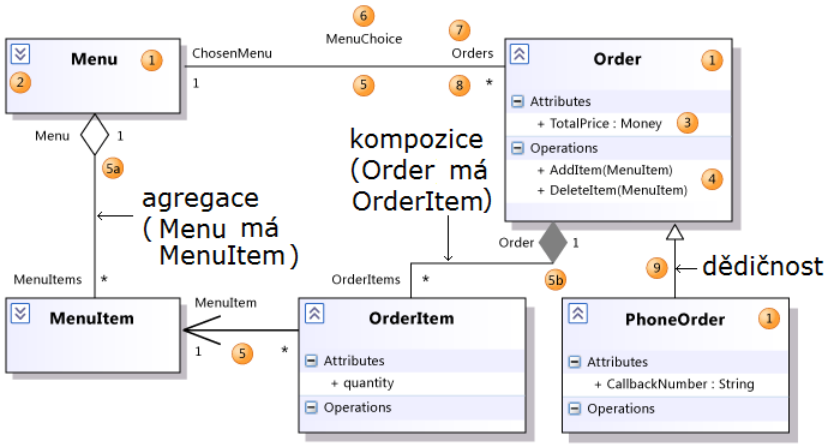
Analytické balíčky jsou nástrojem, jak dekomponovat problém. Balíček může obsahovat případy užití, analytické třídy a další balíčky. Dále definuje viditelnost a zapouzdření svých elementů (public, private a protected) a prostor jmen. Na různých balíčcích lze pracovat souběžně.

Dalšími UML diagramy jsou:

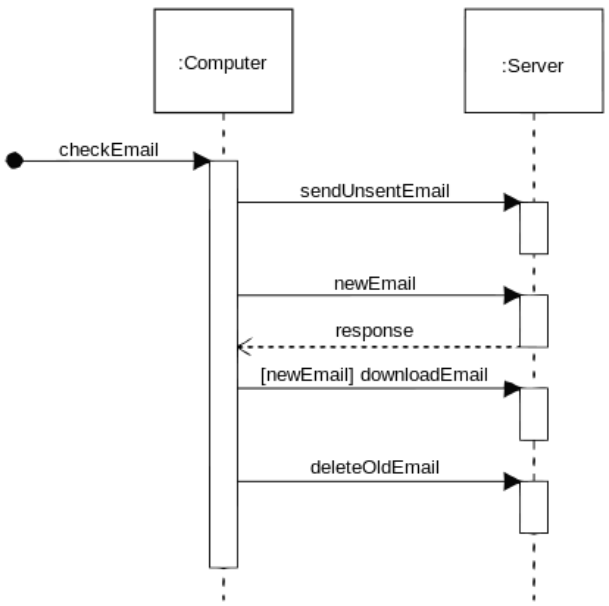
- **diagramy spolupráce** – Reprezentuje strukturální vazby mezi objekty.
- **diagram konkrétní spolupráce** – Zobrazuje konkrétní objekty a konkrétní vazby mezi nimi.
- **sekvenční diagramy** – Zobrazují časovou posloupnost předávání zpráv mezi objekty.
- ...



Obrázek 65: UML diagramy

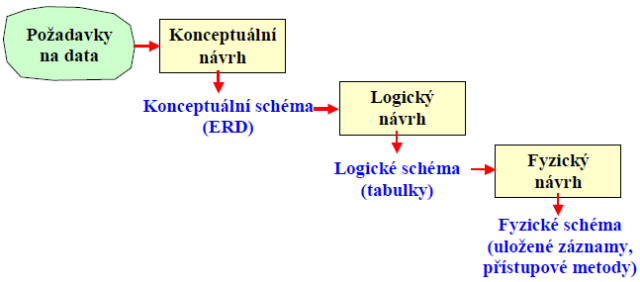


Obrázek 66: diagram tříd



Obrázek 67: sekvenční diagram

32. Konceptuální modelování a návrh relační databáze. IDS, IUS

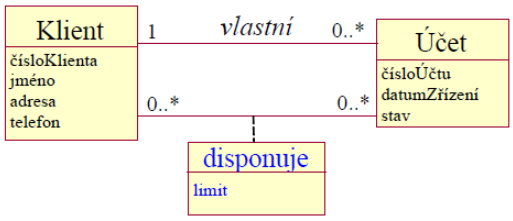


Návrh databáze probíhá na různých úrovních abstrakce:

- **konecptuální návrh** – Probíhá na grafické úrovni, výstupem je **ER** (**entity relationship**) **diagram** nebo diagram tříd (při objektovém přístupu).
- **logický návrh** – Navrhuje se struktura databáze, výstupem je schéma databáze.
- **fyzický návrh** – Navrhuje se fyzické uložení tabulek za využití konkrétního SŘBD tak, aby bylo co nejefektivnější.

ER diagram je založený na chápání světa jako množiny základních objektů – **entit** a **vzathů**. Popisuje data v klidu, nikoliv operace s nimi. Entity jsou konkrétní objekty (jako např. konkrétní auto) náležící do tzv. **entitních množin** (např. množina auto), často se však slovo entita používá ve významu entitní množina. Entity mají různé parametry, mezi něž patří speciální parametr zvaný **primární klíč**, který jednoznačně určuje danou entitu. Atributy mohou být **jednoduché** (jedna atomická hodnota, např. věk) nebo **složené** (více atomických hodnot, např. adresa). Atributy, jež lze spočítat z jiných, se nazývají **odvozené** a entitě je přiřadíme, chceme-li zdůraznit jeho důležitost. Takový atribut může a nemusí být ve výsledném schématu databáze.

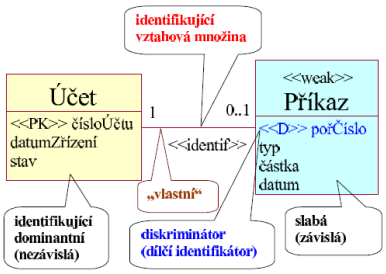
Mezi entitami mohou být vztahy. Počet entit ve vztahu se nazývá **stupeň vztahu** (nejčastěji 2), mluvíme o binárním, ternárním atd. vztahu. **Reflexivní** (unární) vztah je vztah, který má entita sama se sebou. Vztah by měl být pojmenovaný. Dále by měl mít pro všechny konce uvedenou tzv. **kardinalitu**, neboli počet entit vstupujících do vztahu. Rozlišujeme kardinality 1 : *N*, *N* : 1, *N* : *N*, apod. Kardinalita 1 : 1 by se neměla vyskytnout. Pro návrh struktury databáze je důležitá především maximální kardinalita (horní hranice). Vztah může mít své atributy, je-li to třeba.



Obrázek 68: atribut vztahu

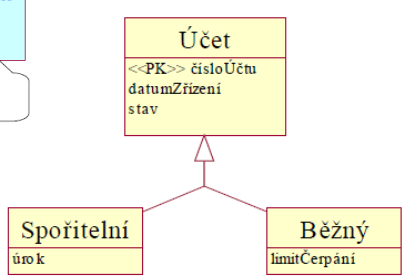
Entity můžeme dále dělit na:

- **slabé** (**weak**) **–** Entita nemůže existovat samostatně, jen spolu s jinou entitou (např. transakce nemůže existovat bez účtu). Její identifikátor je vždy složený a skládá se z identifikátoru identifikující entity a identifikátoru unikátního mezi všemi ostatními entitami závislými na stejné identifikující entitě. Identifikátor transakce by byl např. číslo účtu + pořadové číslo (může tedy existovat více transakcí s pořadovým číslem 1, ale musí mít různé číslo účtu).
- **silné** (**strong**) **–** Ostatní entity (mají identifikátor nezávislý na jiných entitách).



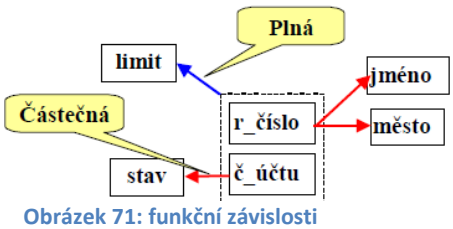
Obrázek 69: slabá entitní množina

Dále je v ER diagramu možné využít generalizaci/specializaci. Jedná se o obdobu dědičnosti u objektového přístupu, kdy můžeme entitu *A* označit jako potomka entity *B*, přičemž *A* zdědí všechny vlastnosti *B* a může navíc přidat nějaké navíc.
Modelovaný systém neuvádíme jako entitu (v systému banky tedy nebude entita *banka*, která by ji představovala). Informace o systému ve výsledné databázi mohou být, ale do konceptuálního modelu nepatří.
Jakmile máme zhotovený ER diagram, můžeme provést transformaci konceptuálního modelu na schéma relační databáze. Při něm dodržujeme tato pravidla:



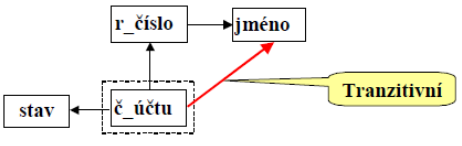
Obrázek 70: generalizace/specializace

- Entitní množinu představuje tabulka.
- Složené atributy převedeme na jednoduché (např. adresu na město, ulici a č.p., v některých případech může složený atribut být samostatnou tabulkou).
- Vztahy 1 : *N* a *N* : 1 reprezentujeme cizím klíčem (odkazem).
- Vztahy *N* : *N* reprezentujeme samostatnou tabulkou, v níž jsou uloženy dvojice řádků, které jsou ve vztahu.
- Ternární a vyšší vztahy reprezentujeme také samostatnou tabulkou.
- Slabou entitní množinu reprezentujeme tak, že její primární klíč je atribut + cizí klíč identifikující entity.
- Generalizaci/specializaci realizujeme jedním ze způsobů:
  - tabulka pro nadtyp + pro podtyty s primárním klíčem nadtypu
  - pouze tabulky podtypů i s atributy nadtypu
  - tabulka pro nadtyp a i pro oba podtypy
  - všechno v jedné tabulce (rozlišení atributem nebo prázdnými hodnotami)



Obrázek 71: funkční závislosti

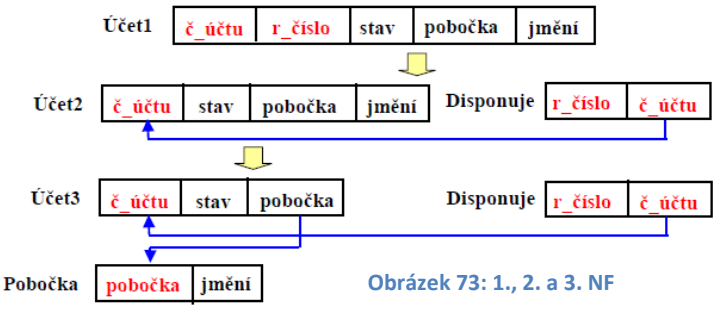
Teorie závislostí se zabývá závislostmi mezi hodnotami atributů v databázi. Rozeznává několik druhů závislostí, přičemž důležitou je především **funkční závislost**. Jedná se o závislost, kdy hodnota atributu *X* jednoznačně určuje hodnotu atributu *Y*, zapisujeme $X \rightarrow Y$. Pokud je např. rodné číslo primární klíč v tabulce člověk, pak platí $rodné\ číslo \rightarrow jméno$, $rodné\ číslo \rightarrow ulice$ apod. **Plná funkční závislot** znamená, že je určitý atribut funkčně závislý právě na všech atributech složeného atributu. Dále může být **závislost tranzitivní** (atribut je funkčně závislý na jiném funkčně závislém atributu).



Obrázek 72: tranzitivní závislost

Proces **normalizace** je založený na teorii závislotí a umožňuje vyvarovat se opakujícím se informacím (redundanci), nemožnosti reprezentovat určitou informaci a složité kontrole integritních omezení. Při normalizaci postupně dekomponujeme (bezeztrátově) tabulky tak, aby měly určité vlastnosti. Normalizace uvádí tzv. **normální formy** (NF), které určují konkrétní vlastnosti návrhu a odráží jeho kvalitu. Nejčastěji se uvádějí tyto NF:

- **1. NF** – Jednoduché atributy (domény) obsahují pouze atomické hodnoty.
- **2. NF** – Schéma je v 1. NF a každý neklíčový atribut (není součástí kandidátního klíče) je plně funkčně závislý na každém (celém) kandidátním klíči.
- **3. NF** – Schéma je v 2. NF a neexistuje neklíčový atribut, který je tranzitivně závislý na některém kandidátním klíči.
- **Boycd – Coddova normální forma** (**BCNF**) – Splňuje 3. NF a pro každou netriviální funkční závislost $X \rightarrow Y$ je *X* superklíčem (nadmnožinou kandidátního klíče). BCNF nelze vždy dosáhnout – pokud ne, pak stačí 3. NF.
- **4. NF –** Týká se vícehodnotových závislostí (splňuje BCNF).
- **5. NF** – Týká se závislostí na spojení (splňuje BCNF).



Obrázek 73: 1., 2. a 3. NF

33. Relační datový model a jazyk SQL. IDS

Relační databázový model vychází z pojmu matematické relace (nikoliv ze vztahů mezi tabulkami). Pro bližší objasnění musíme zavést následující pojmy:

- **doména** – Pojmenovaná množina skalárních hodnot (nejmenší sémantická jednotka dat, dále nestrukturovaná), např. doména názvů měst.
- **složená doména** – Doména složená z několika jednoduchých domén (např. jméno a příjmení).

Nyní můžeme zavést pojem **relace**. Relace na doménách D_1, D_2, … D_n je dvojice

$$R = (R, R^*)$$

kde $R = (A_1{:}D_1, A_2{:}D_2, \ldots A_n{:}D_n)$ je **schéma relace** (pojmenování sloupců tabulky) a $R^* \subseteq D_1 \times D_2 \times \ldots D_n$ je **tělo relace** (z této relace pochází jméno relačního modelu). Počet atributů *n* relce se nazývá **stupeň** (**řád**) relace, kardinalita těla relace $|R^*|$ se nazývá **kardinalita** relace. Relace znázorňujeme **tabulkami**, kde sloupce odpovídají atributům a řádky nticím. Relace má tedy následující vlastnosti:

- Neexistují duplicitní ntice.
- Ntice jsou neuspořádané.
- Hodnoty jednoduchých atributů jsou atomické – relace je tzv. **normalizovaná**.

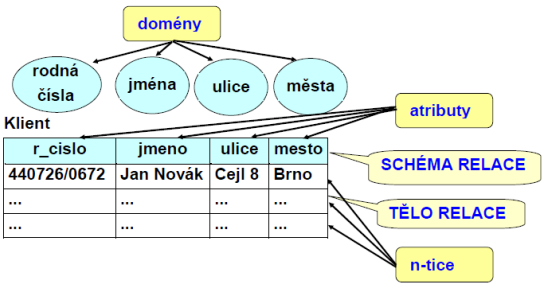


Figure 26: pojmy relačního modelu

Integritní pravidla jsou pravidla, která určitým způsobem omezují data v databázi. Dělíme je na:

- **obecná** – Platí v každé databázi daného typu. V relačním modelu jsou to pravidla týkající se primárních a cizích klíčů.
- **specifická** – Jsou určena konkrétní aplikací.

Obrázek 75: cizí klíč a referenční integrita

Primární klíč je atribut (i složený), který jednoznačně identifikuje n-tici v relaci. Atribut, který může být primárním klíčem nazýváme **kandidátní klíč** (pokud není primárním klíčem, nazývá se alternativní nebo sekundární). Kandidátní (a tedy i primární) klíč musí být:

- **unikátní**
- **minimální** (**neredukovatelný**) – Pokud jde o složený atribut, nelze žádnou složku vypustit bez ztráty unikátnosti.

Pravidlo integrity entit říká, že žádný atribut primárního klíče nesmí být NULL (toto neplatí pro alternativní klíče).
Cizí klíč (**CK**, **foreign key**) je atribut realizující odkaz na nějakou n-tici. Musí pro něj platit:

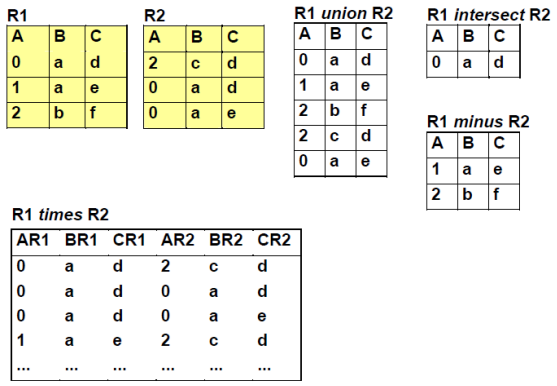
- Všechny položky jsou buď plně vyplněné, nebo nejsou vyplněné vůbec.
- N-tice, na kterou CK odkazuje, musí existovat.

Nutnost existence odkazované ntice se nazává **referenční integrita** (je to integritní pravidlo, vynucuje jej SŘBD).
Relační algebra je dvojice:

$$RA = (R, O)$$

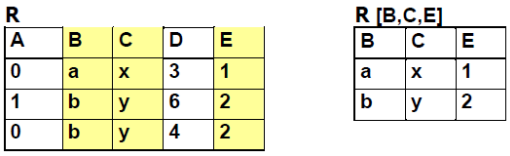
kde *R* je množina relací a *O* množina operací skládající se z klasických množinových operací (sjednocení, průnik, ...) a speciálních relačních operací, kterými jsou:

- **projekce** – Vybírá sloupce tabulky. Selekce atributů *X* a *Y* nad relací *R* se značí *R*[*X*,*Y*].
- **selekce** (restrikce) – Vybírá řádky tabulky s určitou hodnotou atributu. Selekce atributu *X* s operátorem *O* s hodnotou *Y* nad relací *R* se značí *R* where *X O Y*. Podmínku lze rozšířit o logické spojky.
- (**přirozené**) **spojení** – Spojuje dvě tabulky do jedné tak, že spojí jejich schémata a zachová řádky, které mají v tomto spojení stejnou hodnotu atributů. Spojení relací R_1 a R_2 se značí R_1 join R_2. Existují i jiné, jinak definované druhy spojení než přirozené.



Obrázek 74: množinové operace nad relacemi

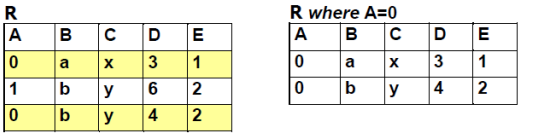
Minimální množinu operací rel. algebry tvoří sjednocení, průnik, rozdíl, kart. součin, projekce a selekce. Relační algebra slouží jako základ pro relační dotazovací jazyky, jakým je např. SQL nebo QBE.



Obrázek 76: projekce

SQL je case-insensitive relační dotazovací jazyk, který je deklarativní a není výpočetně úplný. Existují tři kontexty použití jazyka SQL:

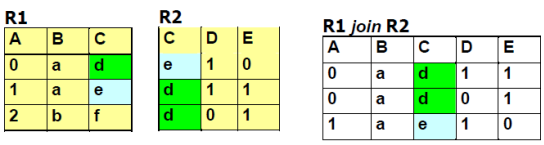
- **přímý** (direct) – Dotazy se zadávají z terminálu.
- **hostitelský** (embedded) – Dotazy se dějí z programovacího jazyka, obecně je tento kontext mocnější než přímý. Dotazy začínají *EXEC* a končí dle zvyklosti jazyka (např. středníkem).
- **jazyk modulů** – Existují SQL moduly pro konkrétní vyšší programovací jazyky.



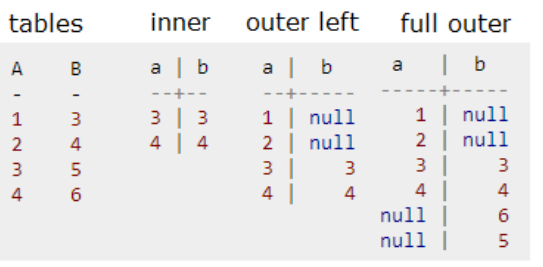
Obrázek 77: selekce

Dotazy vrací jako výsledek tabulky. Dotazy se dělí do těchto kategorií:

- **definice dat a pohledů** (DDL, data definition language)
  - vytvoření tabulky – *CREATE TABLE jméno (sloupec, ..., [integritní omezení sloupců])*
  - zrušení databázového objektu – *DROP objekt*
  - změna vlastností databázového objektu – *ALTER objekt změna*
  - ...
- **manipulace s daty** (DML, data manipulation language)
  - selekce – *SELECT co FROM odkud WHERE podmínka*
  - aktualizace existujícího záznamu – *UPDATE tabulka SET co WHERE podmínka*
  - smazání záznamu – *DELETE FROM tabulka WHERE podmínka*
  - vložení záznamu – *INSERT INTO tabulka VALUES hodnoty*
  - spojení tabulek – *JOIN*
    - vnitřní (inner) – spojí schémata tabulek a ponechá řádky, které mají v obou tabulkách stejnou hodnotu.
    - vnější (outer)
      - left – spojí schémata tabulek a ponechá řádky z první tabulky a řádky, které mají v obou tabulkách stejnou hodnotu.
      - full outer – spojí schémata tabulek a ponechá řádky obou tabulek.
  - **agregační funkce** – Vrací číselnou hodnotu, např. průměr (SUM), minimum (MIN) apod. Používají se s příkazem *GROUP BY* (atribut, z něhož se hodnota počítá).
  - ...
- **autorizace** (DCL, řízení přístupových práv)
- **integrita dat**
- **řízení transakcí** (TCL)



Obrázek 78: spojení



Obrázek 79: SQL join

Pohledy jsou pojmenované virtuální tabulky odvozené od jiných tabulek, nemusí v databázi fyzicky existovat. Vytváří se příkazem *CREATE VIEW*. Z pohledů lze především vybírat data, někdy je lze i vkládat.

Kurzory jsou prostředky pro zpracování víceřádkových výsledků dotazů, používají se v embedded SQL. Každý kurzor má jméno. Může existovat více kurzorů najednou. Příkaz, jako je např. *DELETE* se typicky provede nad určitým kurzorem, který ukazuje na konkrétní záznam.

Dynamické SQL je možnost vytváření SQL příkazů jako textových řetězců za běhu programu.

Uložená procedura je sada SQL příkazů uložených na serveru a zkompilovaných pro rychlejší použití. Mohou obsahovat smyčky, podmínky apod. Jejich výhodou je především možnost je optimalizovat a mají bezpečnostní výhody (mohou např. umožnit konkrétní zápis uživatelům bez práva k zápisu).

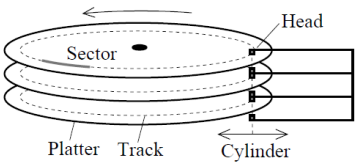
Trigger je speciální případ uložené procedury, která se automaticky volá po provedení určitého dotazu. Triggerům nelze narozdíl od obecných uložených procedur předávat parametry. Jejich využití je především pro zachování integrity databáze, např. když se přidá záznam "auto", automaticky se vytvoří i záznam "majitel".

Index je prostředek (konkrétně datová struktura), jak zrychlit vyhledávání v databázi. Index lze vytvořit pro určitý sloupec tabulky, vyhledávání hodnot tohoto sloupce potom nemusí probíhat sekvenčně, ale např. pomocí stromu, což je mnohem rychlejší.

Příklady SQL dotazů:

CREATE TABLE STATION (ID INTEGER PRIMARY KEY,CITY CHAR(20),STATE CHAR(2),LAT_N REAL,LONG_W REAL);	vytvoří tabulku *STATION*
*SELECT * FROM Customers;*	vybere všechny záznamy z tabulky *Customers*
INSERT INTO STATION VALUES (13, 'Phoenix', 'AZ', 33, 112);	vloží záznam do tabulky *STATION*
SELECT ID, CITY, STATE FROM STATION WHERE LAT_N > 39.7;	selekce a projekce
SELECT MAX(TEMP_F), MIN(TEMP_F), AVG(RAIN_I), ID FROM STATS GROUP BY AREA;	agregační příkazy
*CREATE VIEW METRIC_STATS (ID, MONTH, TEMP_C, RAIN_C) AS SELECT ID, MONTH, (TEMP_F - 32) * 5 /9, RAIN_I * 0.3937 FROM STATS;*	vytvoření pohledu
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate FROM Orders INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;	select s join
SELECT * FROM Customers ORDER BY Country;	select s order by

34. Principy a struktury správy souborů a správy paměti. IOS



Soubor je základní organizační jednotka pro uchovávání dat. **Souborový systém** je systém starající se o práci se soubory a jejich uchovávání. Příkladem souborového systému je Ext4 nebo NTFS. **Virtuální souborový systém** (**VFS**) je softwarová vrstva zastřešující všechny použité souborové systémy a umožňující s nimi pracovat jednotným způsobem. Kromě samotných dat s sebou soubor nese další informace, jako jsou:

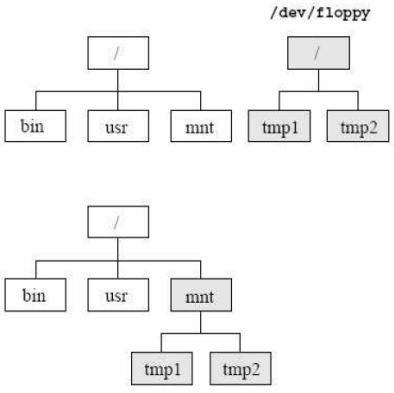
- **jméno**
- **velikost**
- **časové informace**
- **identifikace vlastníka**
- **oprávnění přístupu –** v Unixu:
  - **čtení** (**r**) – Pro běžný soubor možnost otevření pro čtení, pro adresář možnost přístupu k jeho položkám.
  - **zápis** (**w**) – Pro běžný soubor možnost otevřít pro zápis, pro adresář možnost modifikovat jeho obsah (vytvářet, přejmenovávat a rušit položky).
  - **spouštění** (**x**) – Možnost spustit soubor.
  - **sticky bit** (**t**) – Význam má jen u adresáře, kde říká, že soubor může smazat jen jeho vlastník.
  - **SUID/SGID** (**s**) – Jedná se spíše o příznak, který říká, že když bude soubor spuštěn, bude mít UID resp. GID nastaveno ne podle uživatele resp. skupiny, která jej spustila, ale podle vlastníka tohoto souboru.

  V Unixu se rozlišují práva pro **vlastníka** (definován administrátorem, identifikován pomocí UID), **skupinu** (definována administrátorem, identifikována pomocí GID, uživatel může být členem více skupin, jedna je aktuální) a **ostatní**. Efektivní hodnoty UID a GID (ty, které se použijí) se nazávají EUID a EGID. Práva se mění příkazem chmod a zapisují se typicky ve speciálním formátu (vlastník, skupina, ostatní), např.:

  - r w x - - - r - - 0704 (převod z binárního čísla) vlastník má všechna práva, skupina žádná, ostatní mají právo čtení
- **umístění**

Tato data navíc k datům souboru se nazývají **metadata**. Podle filosofie Unixu se téměř vše tváří jako soubor, proto v něm rozeznáváme tyto typy souborů:

- **obyčejný soubor** (-)
- **adresář** (d)
- **blokový speciální soubor** (b) – Je asociován se zařízením, které pracuje s daty v blocích, např. pevný disk, flash-disk, CD-ROM apod.
- **znakový speciální soubor** (c) – Je asociován se zařízením, které přenáší data po jednotlivých znacích, většinou se jedná o stream bez možnosti náhodného přístupu, např. myš, klávesnice, modem a pod.
- **pojmenovaná roura** (p) – Jedná se o perzistentní obdobu klasické roury, tento soubor umožňuje meziproesovou komunikaci. Pojmenovaná roura musí být zrušena explicitně, narozdíl od normální roury.
- **symbolický odkaz** (l) – Speciální soubor, který ve své oblasti pro data nese název jiného souboru.
- **sockety** – Umožňují meziprocesovou komunikaci pomocí síťových socketů.

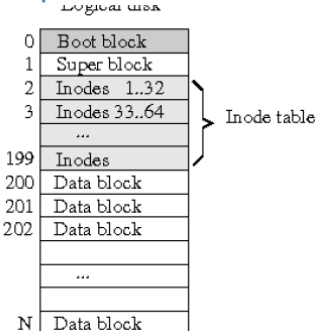


Obrázek 80: operace mount

Adresář je základní organizační jednotka pro vytváření hierarchie souborů. Obsahuje soubor dvojic jméno souboru – číslo i-uzlu (tzv. **hard-link**), soubor může mít teoreticky více jmen. Kořenem hierarchie je **kořenový adresář** (**root**, **/**), v Unixu je jen jeden. Adresář vždy obsahuje odkaz na aktuální adresář (označovaný .) a nadřazený adresář (označovaný ..). Rozlišujeme relativní cesty (začínají nekořenovým adresářem) a absolutní cesty (začínají kořenovým adresářem). Pro adresáře používáme jiné rozhraní než pro soubory, nabízí např. tyto operace:

- vyhledání souboru – Dohledání souboru na disku ze zadané cesty může být relativně pomalé (musí se otevřít všechny soubory v zadané cestě), proto se používají cache.
- vytvoření souboru – Alokují se datové struktury na disku a potom se vytvoří nová položka pro soubor v adresáři, operace musí být atomická.
- zrušení soubboru – Nejprve se zruší položky v adresáři a poté datové struktury, operace musí být opět atomická.
- přejmenování
- výpis obsahu

Možnost připojit k souborové hierarchii další zaízení umožňuje operace **mount**. Ta ztotožní kořenový adresář souborového systému daného zařízení s daným adresářem souborového systému operačního systému (tzv. přípojný bod). Odpojení se provede příkazem **unmount**.

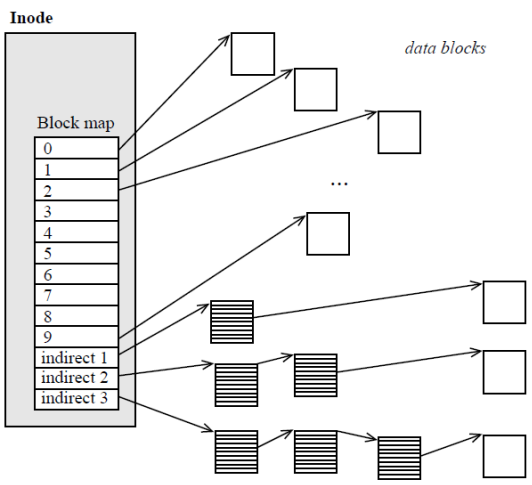


RAID (**redundant array of inexpensive discs**) je způsob, jak z více levných disků sestavit jeden, který určitým způsobem obsahuje redundanci a tedy ochranu dat. To, jakým způsobem redundanci implementujeme závisí na typu RAID, existují RAID0, RAID1, RAID2 atd.
LVM (**logical volume manager**) je metoda správy diskového prostoru, která poskytuje větší variabilitu než konvenční způsob dělení disku na logické oddíly. Je to softwarová vrstva, která vytváří logické disky z dostupných fyzických. Umožňuje za přidávat a odebírat disky, aniž by to narušilo chod systému, proto se používá např. u serverů.

Soubory se většinou ukládají na pevný disk tvořený několika plotnami (platters, záznam většinou z obou stran) skládajících se ze stop různé datové délky. Stopy jsou rozděleny na **sektory** (typicky 512 bytů). Sektory se adresují buď **CHS** (tři čísla: cylinder, head, sector) nebo **LBA** (jedno číslo: logical block address) adresováním. Operační systém pak zavádí pojem **alokační blok** (cluster, 2^n sektorů), který je nejmenší jednotkou, se kterou je uživateli umožněno pracovat.
Jelikož jsou data uložena v jedné dimenzi (tj. Jako posloupnost bytů), musíme si uvědomi, že v průběhu zacházení se soubory dochází k tzv. **fragmentaci** – nevyužívaným mezerám mezi soubory na disku. Existují dva druhy fragmentace:

- **interní** – Nevyužité místo na konci souboru.
- **externí** – Malé nevyužité místo mezi dvěma soubory.

Přístup na disk je z požadavku efektivity plánovám tzv. **plánovačem diskových operací**, který ukládá diskové požadavky do vyrovnávací paměti a různými strategiemi je optimalizuje (např. seřazením operací tak, aby se hlavy pohybovali od okraje ke středu a pak zpět). Disk může být dále dělen na **logické oblasti** (tzv. partitions), jejichž metadata jsou uložena v oblasti disku zvané **MBR** (**master boot record**). MBR obsahuje i zavaděč.
Existují různé typy souborových systémů (ufs, ext3, btrfs, …). **Virtuální souborový systém** (**VFS**) je speciální vrstva, která umožňuje pracovat se všemi souborovými systémy stejným způsobem.



Obrázek 81: i-uzel

Žurnálování (journaling) je způsob zajišťování konzistence při zápisu na disk (většinou metadat) v případě chyby nebo havárie. Vede se tzv. **žurnál**, což je záznam o prováděných akcích. Změny na disku probíhají následovně:

1. Do žurnálu je zapsáno, co a kde se bude měnit.
2. Je provedena vlastní série změn.
3. Do žurnálu je zapsáno, že operace byla úspěšně dokončena.
4. Záznam v žurnálu je zrušen.

Kompromis mezi žurnálováním a nežurnálováním představuje předřazení zápisu dat na disk před zápisem metadat – zajistí se konzistence při chybě během zápisu dat na konec souboru. Alternativou k žurnálování je např. technologie **copy-on-write** (nejprve zapisuje data na disk a potom je zpřístupní).

Informace o tom, které alokační bloky jsou volné a které ne mohou být uloženy buď jako bitový vektor (může zabírat dost místa) nebo pomocí dvou seznamů (seznam volných a obsazených bloků, každý blok je právě v jednom seznamu a není potřeba žádné místo navíc).

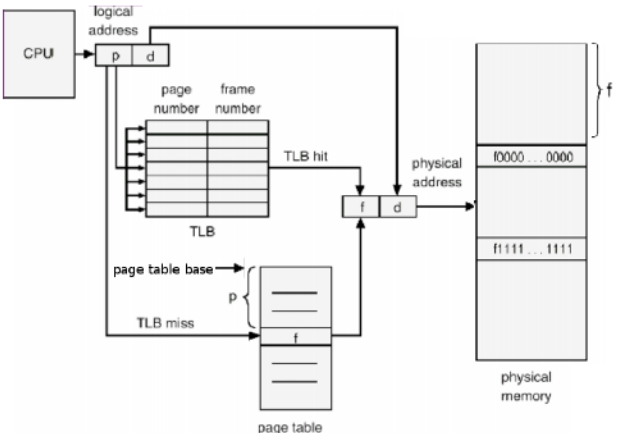
Klasický UNIXový souborový systém (FS) se skládá z těchto čístí:

- **boot block** – Slouží k zavedení systému při startu (obsahuje kód, který se provede).
- **super block** – Obsahuje informace o souborovém systému (typ, velikost, počet i-uzlů, …).
- **tabulka i-uzlů** – Jedná se o tabulku s popisy souborů.
- **datové bloky** – Obsahuje data souborů a bloky pro nepřímé odkazy.

I-uzel je základní datová struktura popisující soubor. Obsahuje metadata:

- stav (alokovaný, volný)
- typ souboru (obyčejný, adresář, zařízení, …)
- délka v bajtech
- čas poslední modifikace souboru (mtime)
- čas poslední modifikace i-uzlu
- identifikace vlastníka (číslo, UID)
- identifikace skupiny (číslo, GID)
- přístupová práva
- počet pevných odkazů (pokud klesne na 0, i-uzel a data souboru se smažou)
- tabulka odkazů na datové bloky: 10 přímých, 1 nepřímý 1. úrovně, 1 nepřímý 2. úrovně, 1 nepřímý 3. úrovně
- další informace

Teoretický limit velikosti souboru je tedy: $10 . D + N . D + N^2 . D + N^3 . D$, kde *D* je velikost bloku a *N* je počet odkazů v bloku. Jinými možnými způsoby uložení souborů jsou:



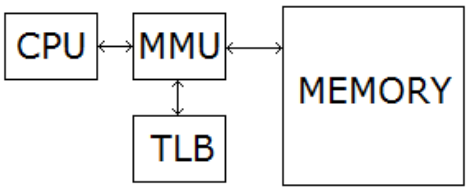
Obrázek 82: jednoúrovňová tabulka stránek s TLB

- **kontinuální uložení** – Každý soubor je uložen jako jediná spojitá posloupnost na disku, problémy nastávají při zvětšování souborů.
- **zřetězený seznam bloků** – Každý blok obsahuje kromě dat odkaz na další blok. Velikost souboru je teoreticky neomezená, ale náhodný přístup je pomalý a chyba v jednom bloku může způsobit velkou ztrátu dat.
- **FAT** (**file allocation table**) – Na začátku disku je (zdvojená) FAT tabulka, která má pro každý blok položku. Do této tabulky vedou odkazy z adresářů. Položky mohou být zřetězené a mohou mít různé příznaky, opět je problém s náhodným přístupem.
- **B+ stromy**

Extent je spojitá oblast o proměnlivé velikosti alokovaná pro soubor.

Virtuální paměť je technika umožňující provádění procesů, které nejsou zcela zavedeny do paměti. Výhodou je, že procesy mohou využívat více paměti, než je v počítači nainstalováno (díky možnosti odkládání paměti na disk, do tzv. oblasti **swap**), a také se nemusí starat o to, jestli má paměť nějakou omezenou kapacitu. Musíme rozlišovat pojmy **logický adresový prostor** (**LAP**, virtuální, se kterým pracuje kód) a **fyzický adresový prostor** (**FAP**, skutečný prostor fyzické paměti společný pro všechny). Překlad mezi logickými a fyzickými adresami provádí jednotka **MMU** (**memory management unit**). Co se týká způsobů přidělování paměti, rozlišujeme:

- **spojité přidělování –** Procesům jsou přidělovány spojité bloky paměti. Tento způsob se snadno implementuje, ale projevuje se výrazně externí fragmentace. Také je problém se zvětšováním přiděleného prostoru. Pro různé části paměti nelze určovat různá práva a odkládá-li se paměť na disk, musí se odložit celá, i když to není nutné. Nemusí způsobit interní fragmentaci.
- **segmentace** – LAP je rozdělen na tzv. segmenty, různé segmenty mohou sloužit různým účelům (procedury, data, zásobník, ...) a mohou mít různou kontrolu oprávnění. Kontrola odkládání na disk je o něco lepší než u spojitého přidělování. Logická adresa se skládá z čísla segmentu a posunu v něm. Na rozdíl od stránkování není segmentae transparentní pro programátora, využívá 2 adresy (segment + offset) a segmenty obecně nemají stejnou velikost.
- **stránkování** – LAP je rozdělen na stejně velké **stránky** (např. 4 kib), FAP na stejně velké **rámce** (které budeou odpovídat stránkám). Paměť je přidělována po rámcích, pro procesy transparentně. Tento způsob je obtížnější na implementaci, zahrnuje vyšší režii a může vznikat interní fragmentace. Výhodou je podstatná eliminace externí fragmentace, možnost jemně nastatavovat přístupová práva a odkládat na disk pouze nezbytný počet stránek. Pro mapování stránek na rámce musí existovat pro každý proces tzv. **tabulka stránek**. Ta může být buď jednoúrovňová (obashuje mapování logických stránek do FAP + příznak přítomnosti, modifikace, přístupu, práva apod.) nebo **hierarchická** (tabulka tabulek stránek, pomalejší přístup, může být sama stránkována, výrazně šetří místo). Přístup do paměti přes tabulku stránek je pomalý, protože nejdřív musíme přistoupit do tabulky a až potom k datům, proto existuje hardwarová asociativní vyrovnávací paměť **TLB** (translation look-aside buffer), která obsahuje dvojice číslo stránky:číslo rámce a díky HW implementaci se prohledává paralelně, pokud se stránka v TLB nalezne (**TLB hit**), přistoupí se do paměti okažitě, jinak (**TLB miss**) se musí přistupovat přes tabulku stránek, v tomto případě se informace o nalezené stránce uloží do TLB pro pozdější rychlé vyhledání (na základě principu lokality). Při přepnutí kontextu nebo změně tabulky stránek se musí obsah TLB invalidovat. Efektivita stránkování závisí na procentu TLB hit.



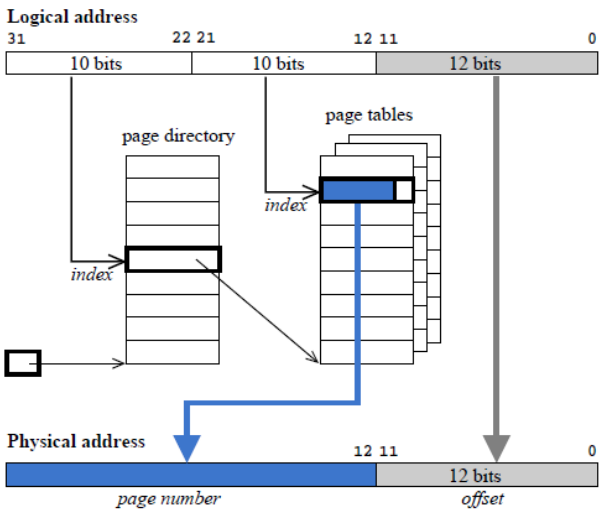
Virtualizace je prováděna buď pomocí tzv. **stránkování na žádost** (demand paging) nebo **segmentování na žádost** (demand segmenting). Stránkování na žádost znamená, že stránka je do paměti zaváděny jen tehdy, je-li zapotřebí (dojde-li k odkazu na ni). V tabulce stránek je příznak, zda je stránce přidělen rámec. Pokud stránka nemá přidělen rámec, dojde k tzv. **výpadku stránky** (**page fault**). Jedná se o přerušení od MMU informující, že nelze převést adresu. V takovém případě dochází k obsluze výpadku stránky – nejdřív se zkontroluje, zda chyba není způsobena přístupem mimo přidělený prostor, pak se alokuje rámec (buď se vezme volný nebo se odloží některý zabraný na disk, čímž se uvolní), inicializuje se stránka, upraví se tabulka stránek a opakuje se instrukce, která vyvolala výpadek. Pro výběr stránky na odložení existují např. algoritmy:

- **FIFO –** Jednoduchá implementace, může odstranit starou, ale často používanou stránku.
- **LRU** (**last recently used**) – Odkládá nejdéle nepoužitou stránku, je velmi efektivní ale obtížná na implementaci, vyžaduje vysokou HW podporu.
- **aproximace LRU algoritmem druhé šance –** Každá stránka má referenční bit, který se nastaví při přístupu, dále existuje proces, který prochází stránky kruhovým seznamem a nuluje tento bit. Pokud narazí na stránku, jejíž bit je 0, odloží ji.

Odkládání stránek může být:

- **lokální –** V rámci procesu**.**
- **globální –** Bez ohledu na procesy.

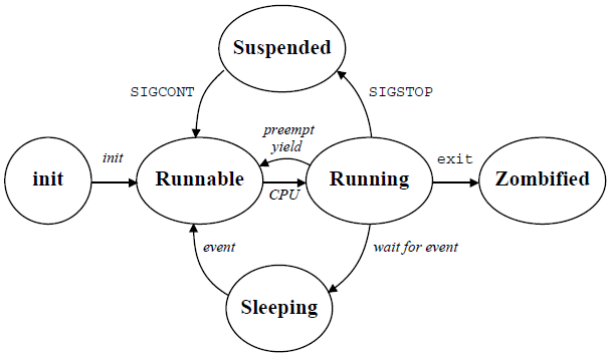
Stránky lze někdy uzamykat (zajistí, že nedojde k jejich odložení) nebo sdílet mezi procesy. Termín **copy-on-write** označuje texhniku, kdy se zkopírováním procesu pomocí fork vytvoří pouze kopie tabulky stránek, které se označí jako copy-on-write. Kopie stránky pro tento konkrétní proces se vytvoří až tehdy, chce-li jeden z procesů zapisovat. Stránkování provádí **page daemon**.



Obrázek 83: dvouúrovňová tabulka stránek

35. Plánování a synchronizace procesů, transakce.

IOS, IDS



Obrázek 84: stavy procesu v Unixu

Proces je konkrétní běžící instance programu. Obecně je proces definován:

- identifikátorem (**PID**)
- stavem plánování
- programem, kterým je řízen
- obsahem registrů
- daty a zásobníkem
- vazbou na další zdroje OS (otevřené soubory, signály apod.)

Tyto informace jsou uloženy ve struktuře nazývané **process control block** (**PCB**, je uložen v paměti nepřístupné uživateli). Proces zabírá v Unixu tyto části paměti:

- uživatelský adresový prostor (user address space) – Kód, data, zásobník, data knihoven. Jsou přístupné uživateli.
- uživatelská oblast (user area) – Uložena pro každý proces v user address space. Je přístupná pouze jádru a obsahuje informace důležité zejména pro běh (PID, obsah registrů, deskriptory souborů, ...).
- záznam v tabulce procesů – Trvale v jádru. Obsahuje informace důležité i tehdy, pokud proces neběží (PID, stav plánování, událost, na kterou se čeká, ...).
- tabulka paměťových regionů – Informace o virtuální paměti procesu.
- zásobník jádra – Využívaný za běhu služeb jádra pro daný proces.

Kontext procesu je jeho stav. Rozlišujeme:

- uživatelský – kód, data, zásobník, sdílená data
- registrový – obsah registrů
- systémový – uživatelská oblast, položka tabulky procesů, ...

Plánování procesů obstarává **plánovač procesů** (**scheduler**) a jedná se o proces přidělování CPU procesům na určitou dobu. Operační systém Unix rozlišuje následující stavy každého procesu v rámci plánování:

vytvořený (init)	Je ještě neinicializovaný.
připravený (runnable)	Mohl by běžet, ale nemá CPU.
běžící (running)	Používá CPU.
mátoha (zombified)	Po *exit*, rodičovský proces ještě nepřevzal návratový kód.
čekající (sleeping)	Čeká na událost (např. dokončení *read*).
odložený (suspended)	Zmrazený signálem *sigstop*.

Operační systém musí při přepínání procesů uchovávat jejich kontext, neboli jeho stav (kód, data, zásobník, …), aby jej mohl při příštím přidělení obnovit. Samotné přepínání kontextu řeší na základě plánovače tzv. **dispečer**. Přepnutí trvá stovky až tisíce instrukcí.
Procesy tvoří v systému Unix hierarchii. Rodičem všech uživatelských procesů je proces **init** (s PID = 1). Existují procesy jádra, které nejsou potomkem procesu init, např. swapper. Pokud procesu skončí předek, stane se jeho rodičem automaticky init. Proces nemůže být definitivně ukončen, dokud jeho rodič nepřevezme jeho návratovou hodnotu – do té doby proces čeká jako tzv. **zombie** proces. Nové procesy se vytvářejí systémovým voláním **fork**, které duplikuje daný proces a ten se stane potomkem duplikovaného procesu (v programu se rodič a potomek rozliší podle návratové hodnoty fork). Vytvořený proces dědí všechno od svého rodiče až na výjimky (např. PID, stav zámků, ...). Volání **exec** nahradí aktuální proces jiným. Výpis stromu procesů lze provést příkazem **pstree**.
V Unixu rozlišujeme u procesů např. tyto identifikátory:

- **PID** (identifikátor procesu)
- **PPID** (identifikátor předka)
- UID (reálný uživatel) a GID (skupina reálného uživatele)
- EUID (efektivní uživatel) a EGID (skupina efektivního uživatele)
- ...

Rozlišujeme dva typy plánování:

- **preemptivní** – Procesu může být odebrán procesor, aniž by se ho sám vzdal. Děje se tak často na základě přerušení od časovače po vypršení tzv. **časového kvanta**.
- **nepreemptivní** – Procesy se samy musí vzdávat procesoru, nemůže jim být odebrán jiným způsobem.

Plánování může být ovlivněno systémem swapování, který může říkat, jaké procesy mají paměť a mohou běžet, nebo systémem spouštění nových procesů, který může běh odkládat na vhodnou dobu (mluvíme o tzv. střednědobém a dlouhodobém plánování). Plánovače používají většinou varianty některého z klasických plánovacích algoritmů:

- **FCFS** (**first come, first served**) – Procesy čekají na přidělení CPU ve FIFO frontě. Algoritmus je nepreemptivní.
- **Round-Robin** – Preemptivní obdoba FCFS. Každý proces má přidělené **časové kvantum**, po kterém mu je CPU odebráno.
- **SJF** (**Shortest Job First**) – CPU se přiděluje procesu, který požaduje nejkratší dobu na provedení dalších instrukcí – algoritmus je nepreemtivní a nepřerušuje proces. Zvyšuje propustnost systému, ale hrozí **stárnutí** (či **hladovění** – **starvation**) – čekající proces nemá záruku, že na něj přijde řada.
- **SRT** (**Shortest Remaining Time**) – CPU se přiděluje procesu, jemuž zbývá nejméně času k dokončení své aktuální výpočetní fáze.
- **Víceúrovňové plánování** – Procesy jsou rozděleny do různých skupin, typicky podle priority. V každé skupině může být použit jiný plánovací algoritmus. Navíc musí být ještě použit algoritmus, který rozhodne, ze které skupiny se bude proces vybírat. Může hrozit hladovění procesů.
- **Víceúrovňové plánování se zpětnou vazbou** – Nově vzniklý proces je zařazen do fronty s nejvyšší prioritou a postupně klesá. Používají se varianty s dynamickou prioritou, která může klesat nebo stoupat v závisloti na chování procesu. Cílem je zajistit rychlou reakci interaktivních procesů.

V Linuxu se dříve používal **O(1) plánovač** (s konstantní časovou složitostí), novější verze používá **CFS** (completely fair scheduler). V plánovačích s prioritami může nastat **problém inverze priorit**. Jedná se o tuto situaci, kdy procesy s nižší prioritou legálně předbíhají v alokaci zdrojů procesy s vyšší prioritou. Tento problém může, ale také nemusí vadit. **Úloha** je skupina paralelně spuštěných procesů propojených do kolony (pipeline).

Meziprocesová komunikace (IPC) může probíhat pomocí těchto prostředků:

- **signály** – Čísla zasílaná přes speciální rozhraní, jsou generovány při chybách, externích událostech nebo na žádost procesu. Často vznikají asynchronně k činnosti programu. Příklady signálů jsou např. SIGKILL (tvrdé ukončení) nebo SIGCHILD (zpráva o ukončení nebo pozastavení potomka). Reakci na signál lze naprogramovat (s výjimkou např. SIGKILL), existuje implicitní obsluha určitých signálů. Některé signály lze rovněž blokovat. Signály se posílají voláním **kill**.
- **roury** – Propojují množinu procesů jejich standardními vstupy/výstupy tak, že výstup jednoho procesu jde přímo na vstup jiného.
- **sdílená paměť**
- **sockety**
- ...

Moderní systémy umožňují procesům běžet ve více paralelních **vláknech**. Vlákna běží z hlediska uživatele současně, stejně jako procesy, ale jsou součástí jednoho procesu a sdílí stejnou paměť, prostředky (otevřené soubory, ...) a kód (pozice v něm se však liší). Vlákno je definováno:

- identifikátorem (thread ID)

- aktuální pozicí v programu
- obsahem registrů
- zásobníkem

Výhodou vlákem je rychlost odezvy např. GUI aplikací (složitý výpočet běžící v odděleném vlákně neblokuje GUI), rychlejší přepínání a spouštění než u procesů, implicitní sdílení paměti mezi vlákny, úspornost alokace zdrojů, možnost paralelního programování. Nevýhodou může být složitější programování vícevláknových aplikací a potenciální problémy paralelního programování (race condition).

Synchronizace procesů (potažmo i vláken) se zabývá řešením problémů souřasně běžících programů, které sdílí určité prostředky. Typicky se setkáváme s tzv. **race condition**, neboli časově závislými chybami, což je nepředvídatelné chování systému v závisloti na načasování událostí. Příkladem race condition může být kód:

```
if (x != 0) y = z / x;
```

kde *x* je sdílená proměnná. Tento řádek kódu bude přeložen jako sekvence více instrukcí pro procesor a nebude proto atomický. **Atomický** příkaz je takový, který se buď provede celý bez přerušení, nebo se neprovede vůbec. V tomto případě je možné, že po ověření platnosti podmínky (*x* != 0) se přepne kontext, řízení dostane jiný proces, který změní hodnotu proměnné *x* na 0, a po přepnutí kontextu zpět a návratu k příkazu *y* = *z* / *x* dojde k dělení nulou.

Takovéto situace vyžadují tzv. **výlučný přístup** ke zdrojům, tzn. v určitém úseku kódu musíme zajistit, že k našim zdrojům (proměnná *x*) nebude v daném čase kromě nás mít nikdo jiný přístup. Tento úsek kódu se nazývá **kritická sekce**. Existují různé mechanismy, jak zajistit výlučný přístup ke kritické sekci. Patří mezi ně:

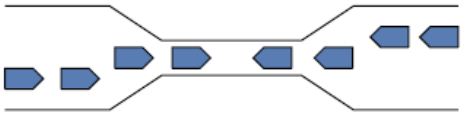
- **semafory** – Při vstupu do kritické sekce se volá určitý příkaz, který do ní proces buď pustí, nebo jej donutí pasivně čekat, až bude přístupná. Při opouštění kritické sekce se volá příkaz pro uvolnění semaforu. Jedná se typicky o celočíselnou proměnnou *S* (na začátku rovna 1) přístupnou dvěma atomickými operacemi:
  - **lock** (**P**, **down**) – Dekrementuje *S*. Je-li $S < 0$, zařadí volající proces do čekající fronty.
  - **unlock** (**V**, **up**) – Inkrementuje *S*. Je-li $S \leq 0$, obnoví běh prvního procesu z fronty čekajících.

  Speciálním případem semaforu je **mutex**, který umožňuje semafor odemnknout pouze procesem, který jej zamkl (umožňuje optimalizovanou implementaci).
- **monitory** – Vysokoúrovňový synchronizační prostředek, který zapouzdřuje data a definuje nad nimi operace. Jedná se v podstatě o objekt, jehož metody mají implicitně zajištěn výlučný přístup. Umožňuje také čekat procesům do doby, než bude platit určitá podmínka. Implementace je možná pomocí semaforů.

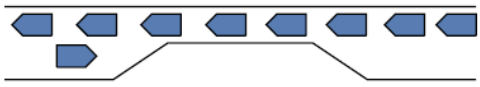
Synchronizace procesů není triviální a mohou nastat problémy. Jsou jimi především:

- **uváznutí** (**deadlock**) – Máme skupinu procesů *A*, které jsou všechny pozastaveny a čekají na podmínku, která by ovšem mohla být splněna pouze v případě, že by běžel některý z procesů skupiny *A*. **Coffmanovy nutné a postačující podmínky** uváznutí jsou (jejich zrušením předejdeme uváznutí):
  - vzájemné vyloučení při používání prostředků
  - vlastnictví alespoň jednoho zdroje, pozastavení a čekání na další
  - prostředky vrací proces, který je vlastní, po jejich využití
  - cyklická závislost na sebe čekajících procesů

  Jiným možným řešením je **detekce a zotavení se z uváznutí** (ukončením některých procesů ze skupiny *A*).
- **stárnutí/hladovění** (**starvation**) – Procesy čekají na podmínku, u které není zaručeno, že nastane.



Obrázek 85: deadlock

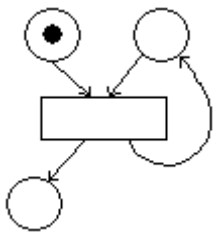


Obrázek 86: starvation

Transakce je skupina příkazů vykonávaná jako celek. **Transakční systém** je systém podporující vykonávání transakcí a zajišťující jejich určité požadované vlastnosti. Tyto vlastnosti se nazývají **ACID** a jsou to:

- **atomicity –** Transakce se buď provede celá, nebo se neprovede vůbec. Je nutné zajistit, že když se neprovede celá, stav systému se vrátí do stavu před začátkem jejího vykonávání.
- **consistency** – Každá transakce změní stav systému z validního na jiný validní stav.
- **isolation** – Zajišťuje, že transakce, které běží současně, změní takovým způsobem, jako by běžely sekvenčně jedna po druhé.
- **durability** – Zajišťuje, že jakmile je transakce vykonaná, systém zachová změny, které provedla. Musí se počítat s možnými haváriemi, data musí být ukládána redundantně v nevolatilních pamětech.

Transakční model může být buď **plochý** (flat, transakce nelze zanořovat, např. SQL) nebo **strukturovaný** (transakce mohou být zanořené). V jazyku SQL se transakce provede zadáním příkazů, tvořících tělo transakce, a následným příkazem COMMIT (potvrzení transakce) nebo ROLLBACK (zrušení transakce). Existuje i tzv. částečný ROLLBACK, který umožňuje navracení se k tzv. SAVEPOINTům. Pro zajištění souběžného provádění více transakcí slouží tzv. **schéma řízení**. To určuje tzv. **plán**, který definuje pořadí provádění instrukcí souběžných transakcí.

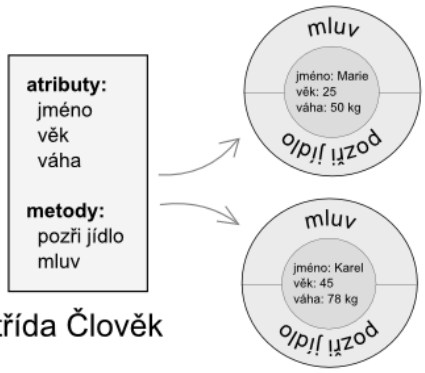


Obrázek 87: deadlock v Petriho síti

36. Objektová orientace (základní koncepty, třídně a prototypově orientované jazyky, OO přístup k tvorbě SW). IPP, IUS

Objektová orientace je přístup k abstrakci reálného světa, který reprezentujeme jako množinu vzájemně komunikujících objektů. Tento přístup můžeme využít jak při programování, tak i při návrhu nebo třeba testování.

Objekt je entita představující část systému, která zapouzdřuje své stavové informace a vyznačuje se svým chováním. Objekt by měl připomínat jednotku podobnou černé skříňce, která je specializovaná na jediný úkol. **Zpráva** je komunikační jednotka, kterou může objekt poslat jinému objektu. Zpráva může mít své parametry. Pokud objekt obdrží zprávu, nějak na ni reaguje. Reakcí je většinou zavolání funkce patřící tomuto objektu, která se nazývá **metoda**. **Rozhraní** nebo **protokol** objektu je množina zpráv, jimž objekt rozumí.
Abstrakce je míra zjednodušení, ignorování nepodstatných atributů. Při vyšší abstrakci jsme schopni vidět problém z větší dálky, máme na něj širší pohled, ale nevnímáme detaily. Míru abstrakce můžeme během programování měnit dle potřeby.



Mazi programovacími jazyky nalezneme dva základní přístupy k **OOP** (**objektově orientovanémů programování**):

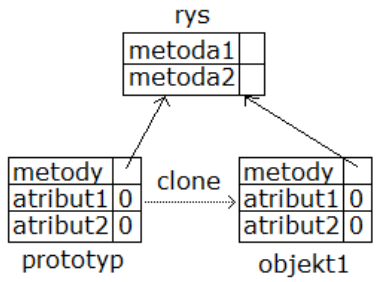
- **třídní** - Mezi základní pojmy patří třída, která slouží jako předpis pro objekty – definuje jejich vlastnosti (stavové proměnné) a metody (kód představující chování objektu a sloužící jako jeho rozhraní). Objekty jsou konkrétní **intance** tříd. Patří sem např. C++, Java, Python, …
- **beztřídní** (**prototypově orientované**, classless) – Neznají pojem třídy, nové objekty vytvářejí na základě objektů již existujících (tzv. prototypů) tzv. **klonováním**.

Delegace je situace, kdy objekt, který nerozumí dané zprávě, přepošle tuto zprávu objektu, na který má odkaz a který jí bude rozumět.

Objektové jazyky dále můžeme dělit na:

- **čistě objektové** – Vše je objekt. Existuje základní třída, od níž všechny ostatní dědí.
- **hybridní** – Kombinuje OOP s jiným přístupem, existují základní datové typy, které nejsou objekty, a můžou existovat např. funkce, které nejsou metodami (nepatří žádnému objektu).

V třídních jazycích se těsně po vytvoření objektu volá jeho **konstruktor**, což je speciální inicializační metoda. Třídní jazyky mohou také zavádět tzv. **statické** atributy a metody, které nepatří objektům této třídy, ale samotné třídě. Tzn. statický atribut je globální proměnnou, na niž se odkazujeme pomocí jména třídy a statická metoda je volána pomocí jména třídy, nikoli objektu (pro její volání nemusí žádný objekt existovat).



Obrázek 88: klonování

OOP se vyznačuje především následujícími vlastnostmi:

- **zapouzdření** – Objekty skrývají svou vnitřní implementaci a stavová data, vystupují pouze prostřednictvím svého rozhraní (metod). Jde o základ přehlednosti.
- **polymorfismus** (mnohotvarost) – Využívá výhody zasílání zpráv místo přímého volání funkcí. Je možné vytvořit dva různé objekty, které na stejnou zprávu reagují odlišně (zavoláním jiné metody). Díky tomu se volající objekt nemusí starat, komu zprávu zasílá, jen si ověří, zda jí příjemce bude rozumět. V programovacích jazycích je polymorfismus realizován pomocí tzv. **virtuálních metod**, což jsou metody, jejichž definice se neurčuje na základě typu ukazatele na objekt, ale na základě typu objektu – která definice virtuální metody se použije se tedy určuje až za běhu programu, ne při překladu (tzv. **pozní vazba,** provádí se pomocí tabulky virtuálních metod **VMT**, opakem je **časná vazba**, tedy určení kódu metody již při překladu). Každá třída má jednu VMT obsahující ukazatele na metody, objekty mají za běhu ukazatel na VMT, který se nastaví na VMT konkrétní třídy.
- **dědičnost** – Možnost vytvářet objekty na základě jiných. Chceme-li vytvořit objekt, který se podobá jinému, ale má mírně odlišné vlastnosti, můžeme to udělat pomocí dědičnosti. Dědičnost se vyskytuje v třídních jazycích. Provádíme ji s třídami a vzniká nám díky ní hierarchie tříd (matematicky jde o reflexivní, tranzitivní a antisymetrickou relaci na množině tříd). Dědičnost může být:
    - **jednoduchá** – Třída může mít pouze jednoho předka. Nedostatky tohoto způsobu dědičnosti většinou nahrazuje mechanismus **rozhraní**. Příkladem jazyka s jednoduchou dědičností je např. Java.
    - **vícenásobná –** Třída může mít i více předků. Vznikají zde problémy např. s tím, které implementace metod se zdědí v případě, že mají oba rodiče stejnou metodu s různou implementací. Musí být určena pravidla, která tyto situace řeší (např. zdědí se metoda dříve uvedené třídy). Příkladem jazyka s vícenásobnou dědičností je např. C++.

Mechanismus rozhraní umožňuje definovat tzv. **rozhraní** (interfaces), což je soubor deklarací metod (nikoli definic). Různé objekty potom mohou tzv. **implementovat** libovolný počet rozhraní, což znamená, že definují kód všech metod rozhraní. Tím lze získat jistotu, že objekt implementující dané rozhraní bude rozumět všem zprávám definovaným rozhraním. Lze mít rozhraní např. *talking*, které zaručí, že každý, kdo jej implementuje, bude implementovat metodu *talk()*. Tento mechanismus se používá v jazycích bez vícenásobné dědičnosti, aby doplnil jednoduchou dědičnost o její nedostatky.

Mechanismus výjimek je mechanismus pro efektivní ošetřování výjimečných chybových stavů (např. dělení nulou, přístup mimo hranice pole apod.). Umožnuje oddělit výkonný kód aplikace od kódu ošetřování chyb tak, že zavádí bloky *try* a *catch*. Vždy se nejdříve uvede blok *try* a v něm výkonný kód, potom následuje blok *catch*, který odchytává výjimky a ošetřuje je. Nastane-li v bloku *try* výjimka, přesune se vykonávání kódu do bloku *catch* (a výjimka se mu předá). Mechanismus výjimek není použitelný výhradně v OOP, ale dobře se s ním kombinuje, jelikož výjimky mohou být objekty. Programátor tak může definovat výjimky vlastní, využívat výhody dědičnosti apod.

Modifikátory přístupu jsou klíčová slova v objektově orientovaných jazycích, která umožňují manipulovat se zapouzdřeností. Většinou můžeme každý atribut čí metodu označit těmito modifikátory:

- **public** (veřejný) – K atributu/metodě má přístup kdokoli, kdo má přístup k objektu.
- **private** (soukromý) – K atributu/metodě má přístup pouze tato třída. Třídy od této třídy zděděné k tomuto atributu/metodě přístup nemají.
- **protected** (chráněný) – K atributu/metodě má přístup pouze tato třída a třídy od této třídy zděděné (oslabený private modifikátor).

V OOP je nutné rozlišovat pojmy:

- **redefinice metody** (**overriding**) – Nová definice kódu pro metodu. Takto můžeme vytvořit metodu, která bude mít za různých situací různý kód (konkrétní kód se vybere pomocí polymorfismu). Metoda má vždy stejnou signaturu (jméno, počet a typ parametrů).
- **přetěžování metody** (**overloading**) – Definice metody se stejným jménem, ale jinými parametry. Takto můžeme vytvořit metodu, která má určité jméno, ale podle toho, jaké jí předáme parametry, může mít různou implementaci. To, která definice metody se použije, se určí již při překladu.

Porovnáváme-li dva objekty, rozlišujeme dva pojmy:

- **identita** – Objekty jsou identické, jedná-li se o tentýž objekt (objekt porovnaný sám se sebou).
- **shodnot** – Objekty jsou shodné, jsou-li stejné třídy a mají-li stejný obsah (stav).

Vytváříme-li kopii objektu, můžeme to provést dvěma způsoby:

- **hluboká kopie** (**deep copy**) – Zkopíruje se objekt, i objekty, na nějž má tento objekt reference. Obvykle je nutné uvést, do jaké hloubky se hluboká kopie provede.
- **mělká kopie** (**shallow copy**) – Hluboká kopie do hlobky nula. Zkopíruje se objekt, ale objekty, na které odkazuje, nikoli.

Rušení nadále nepotřebných objektů může probíhat dvěma způsoby:

- **automaticky** – Je většinou možné pouze v případě, že kód běží ve virtuálním stroji. Rušení provádí tzv. **garbage collector**. Ten jednou za určitou dobu ruší objekty, na které neexistuje programově dostupný odkaz.
- **manuálně** – Objekty ruší programátor speciálním příkazem. Před zrušením objektu se automaticky volá jeho **destruktor** (speciální metoda pro tyto účely).

Výhodami a nevýhodami OOP jsou:

výhody	nevýhody
vysoká míra abstrakce	složitější koncept (těžší na naučení)
jednoduchá dekompozice problémů	někdy neexistuje analogie s reálným světem a je složité provést dekompozici
korespondence s reálným světem	výsledný kód může být pomalejší kvůli vyšší režii
přehlednost, udržovatelnost a rozšiřitelnost	pro jednoduché aplikace může být zbytečně složité
modularita a znovupoužitelnost	

V souvislosti s OOP se často zmiňují tzv. **návrhové vzory** (**design patterns**). Jde o pojmenovaná řešení často se vyskytujících problémů. Příkladem je návrhový vzor **jedináček** (**singleton**) umožňující vytvořit pouze jedinou instanci objektu určité třídy tak, že její konstruktor uvedeme jako soukromý, a k vytváření instancí definujeme statickou metodu, která kontroluje, zda jde o první vytváření objektu nebo ne.

Virtuální stroj je softwarová vrstva, jejímž primárním účelem je odstínit hardwarovou specifikaci počítače, na němž běží. Zajišťuje tak platformovou nezávislost. Může pracovat způsobem:

- přímé interpretace kódu
- kompilace do mezikódu a jeho následná interpretace
- kompilace do mezikódu a jeho následná kompilace do nativního kódu

Skládá-li se objekt z jiných objektů, rolišujeme tyto možnosti:

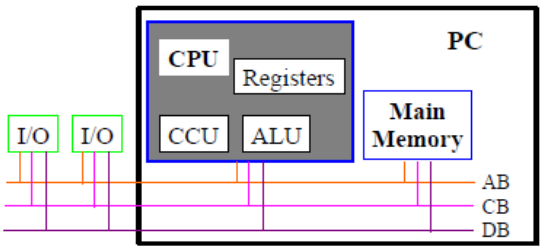
- **agregace** – Části jsou pro objekt volitelné (např. přívěs pro auto).
- **kompozice** – Části jsou nepostradatelné pro existenci objektu (např. křídla pro letadlo).

Formálně jsou OO jazyky popsány pomocí σ (sigma) **kalkulu** (podobně jako jsou funkcionální jazyky popsané pomocí λ kalkulu).

37. Programování v jazyku symbolických instrukcí (činnost počítače, strojový jazyk, symbolický jazyk, asembler). IAS, INP

Jazyk symbolických instrukcí (**JSI**, často nesprávně asembler) je nízkoúrovňový jazyk, který zjednodušuje psaní kódu v binárním strojovém jazyce tím, že jim přiřazuje symbolická pojmenování. Podoba jazyka je dána konkrétním procesorem a jeho instrukční sadou, program je tedy vázán na konkrétní procesor (nebo rodinu procesorů). Program, který překládá jazyk symbolických instrukcí do strojového kódu se nazývá **asembler**. Narozdíl od vysokoúrovňových jazyků není JSI strukturovaný, takže je obecně složitější v něm psát programy, ale kód v něm běží rychleji a jeho princip je velmi blízko hardwaru. Počítač se obecně skládá z následujících částí:

- **aritmeticko-logická jednotka** (**arithmetic and logic unit** , **ALU**) – Jednotka shopná vykonávat instrukce programu, který mám být počítačem vykonán.
- **řadič** (**control unit**, **CU**) – Vydává pokyny jednotkám tak, aby správně spolupracovaly.
- zdroj **taktovacích impulsů** (**clock**) – Zdroj sinchronizačních signálů pro jednotky.
- **registry** (**registers**) – Speciální paměťová úložiště s malou kapacitou a rychlou přístupovou dobou.
- **zdroj hodin**
- **paměť** (**memory**) – Úložiště dat a instrukcí o větší kapacitě ale mírně vyšší přístupové době.
- **vstupní** a **výstupní** zařízení (**input** and **output** devices) – Zařízení pro komunikaci počítače s okolím.

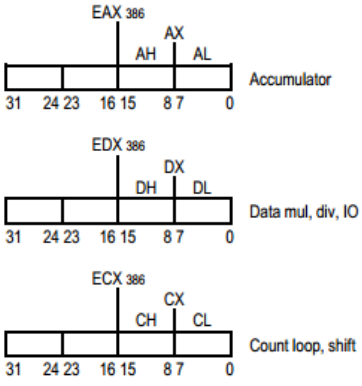


Obrázek 89: zjednodušené schéma počítače

První čtyři jednotky tvoří **procesor** neboli **CPU** (**central processing unit**). Všechny podsystémy procesoru a počítače spolu komunikují prostřednictvím **adresové**, **datové** a **řídící sběrnice** (**address bus**, **data bus** and **control bus**). K významným registrům (ve střadačové architektuře) CPU patří **střadač** (**accumulator**, označovaný **A**), registr ukazatele instrukcí (**instruction pointer** register, **IPR**) a instrukční registr (**instruction register**, **IR**, udržuje aktuální instrukci). Nejmenší adresovatelnou jednotkou hlavní paměti je obvykle jedna **slabika** (**byte**) skládající se obvykle z 8 bitů, dalšími jednotkami jsou slovo (**word**, proměnlivá velikost dle architektury, často 32 nebo 64 bitů), dvouslovo (**double word**, dvě slova) apod. Činnost počítače spočívá v transformaci vstupních údajů na údaje výstupní vykonáváním programu skládajícího se z tzv. **instrukcí**. Soubor všech instrukcí se nazývá **strojový kód**. Data i instrukce jsou uloženy v paměti počítače. Za instrukci jsou považována data, na něž ukazuje registr ukazatele instrukcí. Činnost počítače probíhá v režii řadiče následujícím způsobem:

Do instrukčního registru (IR) se uloží instrukce, tj. obsah paměťového místa, na které ukazuje IPR.

1. Nastaví se nový obsah registru IPR (adresa následující instrukce).
2. Dekóduje se obsah IR, tzn. určí se **operační kód** (číslo identifikující instrukci) a adresy operandů (mohou být v instrukci, registrech nebo paměti).
3. ALU provede dekódovanou instrukci.
4. Pokud instrukce neukončila program, pokračuje se znovu od bodu 1.



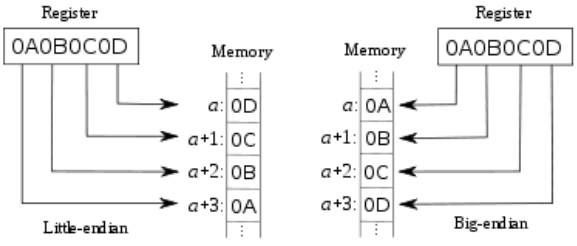
Procesory pentium mají různé režimy činnosti:

- **reálný režim** – Implicitní, program má přístup k celé paměti. Není zde podpora ochrany paměti ani multitaskingu.
- **chráněný režim** – Musí se do něj přepnout z reálného režimu. Umožňuje kontrolu paměti, virtuální paměť, stránkování, bezpečný multitasking a kontrolu přístupu do paměti. Tento mód umožňuje operačnímu systému kontrolovat aplikace, které pod ním běží.

Dále lze programovat 16 nebo 32 bitové aplikace. Tyto módy se liší velikostmi registrů. Základními typy operandů jsou **byte** (8 bitů), **word** (2 byty), **doubleword** (2 wordy) a **quadword** (4 wordy). Hodnoty v paměti můžou být uspořádány buď způsobem **little-indian** (nejméně významná slabika má nejnižší adresu) nebo **big-endian** (nejméně významná slabika má nejvyšší adresu). Intel používá little-endian. Jendotky paměti mohou dále být:

kilobyte (kB)	1000 B
kibibyte (kilobyte, KB, KiB)	1024 B
megabyte (MB)	1000^2 B
mebibyte (megabyte, MB, MiB)	1024^2 B
gigabyte (GB)	1000^3 B
gibibyte (gigabyte, GB, GiB)	1024^3 B
atd.	

Pro výpočty v plovoucí řádové čárce se používají samostatné jednotky. Dříve to byl **matematický koprocesor** (**FPU**, **floating-point unit**), dnes je však nahrazován jednotkami MMX a SSE. MMX a SSE jsou specializovány na práci s multimediálními daty a umí zpracovávat více dat najednou (SIMD, single instruction – multiple data) pomocí speciálních instrukcí a registrů.
příklady instrukcí procesoru intel jsou:



Intel

Obrázek 91: little-endian a big-endian

```
MOV kam,co  - přesouvá data z co (registr, paměť) do kam (registr, paměť)
NOP         - nedělá nic
CMP co,scim - porovná co (registr) s scim (registr) a nastaví příznaky
JMP adresa  - nepodmíněný skok na adresu
JLE adresa  - podmíněný skok, skočí, pokud je výsledek porovnání menší nebo roven
PUSH co     - uloží hodnotu v co (registr) na zásobník
RET kolik   - navrácení z procedury, uvolní kolik míst ze zásobníku
LOOP kam    - dekrementuje obsah ECX a pokud je ECX = 0, pak provede skok
```

Velmi důležitou částí paměti je **zásobník**, díky němuž je možné volat podprogramy a zanořovat se v kódu. Zásobník roste směrem k nižší adrese. Když se volá procedura, uloží se na zásobník návratová adresa (aktuální instrukce) a parametry procedury (buď zleva doprava nebo naopak, v závislosti na konvenci jazyka). Procedura si na zásobník dále může ukládat lokální proměnné. Tyto tři věci dohromady tvoří tzv. **zásobníkový rámec** (**stack frame**). Na začátku procedury obvykle provádíme dvojici operací:

```
PUSH ebp
MOV ebp,esp
```

Tímto uchováme obsah registru ebp a poté jej použijeme jako ukazatel na začátek rámce. To děláme proto, že registr esp (ukazatel na zásobník) můžeme v proceduře používat, ale ebp se měnit nebude a bude stále ukazovat na stejné místo, proto díky němu můžeme jednoduše přistupovat k parametrům procedury. Na konci procedury musíme samozřejmě ebp obnovit instrukcí POP. Je také nutné "uklidit" parametry ze zásobníku, což může dělat buď procedura sama, nebo to nechá na tom, kdo ji zavolal (opět podle konvence).

Máme-li program napsaný v jazyku symbolických instrukcí, můžeme jej zkompilovat pomocí assembleru. Výstupem je tzv. **objektový kód**, což je kód ve strojovém jazyku, který ale ještě potenciálně obsahuje relativní adresy (např. volání procedury z nějaké knihovny). Spustitelný kód dostaneme pomocí tzv. **linkeru**, což je program, který dostane jako vstup množinu objektových souborů a spojí je do jediného spustitelného souboru, v němž všechny instrukce už obsahují absolutní adresy.
Registry procesorů intel jsou:

- obecné: EAX, EBX, ECX, EDX
- segmentové (slouží k adresování): CS, DS, ES, FS, GS, SS
- indexové: ESI, EDI, EBP, EIP (instruction pointer), ESP (stack pointer)
- příznakový: EFLAGS

Příznak carry značí bezznaménkové přetečení (např. 200 + 75 na 8 bit), příznak overflow znaménkové (např. 100 + 50 na 8 bit).
Adresování operandů dělíme na:

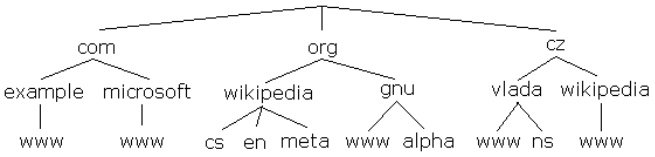
- **immediate** – hodnota je zakódovaná přímo v instrukci
- **hodnota v registru** – instrukce odkazuje na hodnotu v registru
- **hodnota v paměti** – hodnota je v paměti, adresování může být přímé (adresa přímo v instrukci) nebo nepřímé (adresa v registru)

Při použití segmentace se lineární adresa vypočítá jako: segment * 16 + offset mod 20.

38. Služby aplikační vrstvy (email, DNS, IP telefonie, správa SNMP, Netflow). IPK, ISA

Aplikační vrstva je nejvyšší (sedmou) vrstvou referenčního modelu ISO/OSI. Mezi služby definované na této vrstvě patří:

- **e-mail** – E-mail nebo také elektronická pošta je systém pro odesílání a přijímání elektronických zpráv přes síť. Uživatelé mají své adresy ve formátu *uživatel@server*. Systém využívá protokol **SMTP** (**simple mail transfer ptotocol**, port 25) pro odesílání pošty na servery. Pro příjem pošty ze serverů používají uživatelé různé protokoly, nejčastěji **POP3** nebo **IMAP**. Formát e-mailu se skládá z hlavičky (odesílatel, předmět, datum, ...) a těla. Pokud je SMTP server nedostupný, odešle se zpráva na záložní poštovní server podle priorit v MX záznamu. MIME (multipurpose internet mail extension) je rozšíření e-mailu o možnost jiných kódování než jen ASCII a možnost posílat binární přílohy.
- **DNS** (**domain name system**) – Jedná se o hierarchický systém doménových jmen, realizovaný distribuovanou sítí DNS serverů a DNS protokolem (TCP/53 a UDP/53). Hlavním účelem systému je vzájemný překlad doménových jmen a IP adres uzlů sítě, nabízí však i další možnosti, jako např. ukládání dodatečných dat k adresám. Záznamy DNS mohou být následujících typů:
  - **A** (address record) – Doménovému jménu přiřazuje IPv4 adresu. Např. *moje IN A 1.2.3.4*
  - **AAAA** (IPv6 record) – Doménovému jménu přiřazuje IPv6 adresu. Např. *moje IN AAAA 2001:718:1c01:1:02e0:7dff:fe96:daa8*
  - **CNAME** (canonical name) – Alias pro již existující jméno. Např. *www IN CNAME web*
  - **MX** (mail exchange record) – Adresa a priorita (nižší číclo = vyšší) serveru pro příjem elektronické pošty pro danou doménu. Např.: stud.fit.vutbr.cz *IN MX 10 eva.fit.vutbr.cz. # vyšší priorita*
    IN MX 20 kazi.fit.vutbr.cz.
  - **NS** (name server record) – Určuje autoritativní server pro danou zónu, slouží k budování hierarchické struktury DNS. (Pokud server nemá konkrétní záznam, pak je pomocí NS záznamu schopný zjistit, kdo jej má.)
  - **PTR** (pointer record) – Mapuje IP adresu na doménové jméno (reverzní mapování). Tyto záznamy jsou uloženy v podstromu doménové hierarchie **arpa** – např. 12.8.229.147.in-addr.arpa.
  - **SOA** (start of authority record) – Každá zóna má právě jeden SOA záznam. Ten obsahuje název primárního serveru a email správce. Dále obsahuje informace např. o sériovém čísle (identifikující změnu záznamů), dobu platnosti dat pro sekundární server apod.
  - **TXT –** Dodatečná textová data o doméně, serveru apod. daného DNS uzlu.
  - **SRV –** Obecný záznam pro jiné služby, používá se pro nové služby místo toho, aby se vytvářely stále nové typy záznamů (jako např. MX pro e-mail).



Obrázek 92: hierarchie doménových jmen

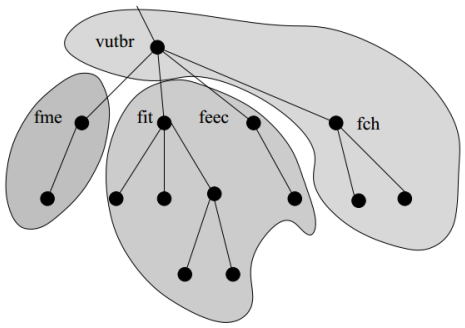
Doména je jednoznačné jméno identifikující počítač nebo síť v internetu. Domény tvoří hierarchický prostor. Jednotlivé úrovně v doménovém jméně jsou odděleny tečkou a jsou seřazeny od nekonkrétnější (např. www) po nejobecnější (např. cz, také doména 1. úrovně). Kořenem stromu je uzel "" (prázdný řetězec) – proto by správně měla všechna doménová jména končit tečkou, i když ta se běžně vynechává.

DNS servery dělíme na:

- **primární** – Pro každou doménu existuje právě jeden. Poskytuje autoritativní záznamy o doménách, které spravuje.
- **sekundární** – Udržuje kopie dat z primárního serveru, poskytuje autoritativní záznamy.
- **záložní** (caching-only) – Jenom si pamatuje dotazy a odpovědi, poskytuje neautoritativní (neúplné a neaktuální) záznamy.

Rezoluci (vyhledání odpovědi v DNS) provádí resolver, což je klientský program pro tyto účely. Rezoluce může být:

- **rekurzívní** – Když server nezná odpověď, sám si ji zjistí od jiného serveru.
- **iterativní** – Když server nezná odpověd, sdělí klientovi, kde ji najít, a ten si ji tam dohledá sám.



Obrázek 93: DNS zóny

DNS záznam se skládá z následujících polí:

- **jméno** – doménové jméno
- **typ** – typ záznamu, např. A, NS, PTR, ...
- **třída** – dnes vždy IN (internet)
- **TTL** (**time to live**) – Počet sekund, po který zůstává záznam platný.
- **další data** (+ jejich délka) – jiná data, např. ip adresa pro A záznam nebo priority a jména serverů pro MX záznam

Zóna je fyzická část DNS pod jednotnou správou a obsahuje část prostoru doménových jmen. Každý DNS tedy obsahuje jen část prostoru doménových jmen – zónu. Díky zónám může být správa decentralizovaná. O registraci a správu domén se strará organizace ICANN.

Změny záznamů lze provádět jen na primárním serveru. Na sekundární server se potom změny automaticky zkopírují takto:

1. Sekundární server pošle žádost o aktualizaci primárnímu a zároveň mu pošle svůj SOA se sériovým číslem.
2. Primární server podle sériového čísla pozná, k jakým mezitím došlo změnám a z databáze změn pošle informace o tom, co se změnilo (**přírůstkový přenos zón**).

U DNS je nutné hlídat jednat integritu dat a jednak se chránit před útoky, např. podvrhování záznamů. Jako reakce na bezpečnostní rizika vzniklo rozšíření DNS **DNSSEC**, které používá **asymetrickou kryptografii** (elektronické podpisy) k ochraně dat. Zavádí nové záznamy: **DNSKEY** (veřejný klíč pro ověřování podpisů), **RRSIG** (podpis daného záznamu), **DS** (záznam pro ověření záznamu DNSKEY, uložen v nadřazené doméně), ... Každá zóna má **ZSK** (zone signature key) a **KSK** (key signature key). ZSK je možné použít k ověření pravosti záznamů dané domény, ale sám ZSK může být podvržen. ZSK je proto podepsán pomocí KSK, který je spravován nadřazenou doménou. Tím vzniká tzv. **řetězec důvery –** aby bylo možné nějaký záznam ověřit, musíme důvěřovat serveru na vrcholu hierarchie, tzn. jeho klíči. Pokud zaručíme platnost tohoto klíče, zaručíme platnost záznamů všech domén níže v hierarchii. Kvůli DNSSEC znatelně narůstá velikost DNS paketu.

- **IP telefonie** – Jedná se o implementaci telefonní sítě nad protokolem IP. Tato síť je propojena s klasickou telefonní sítí (PSTN) prostřednictvím tzv. **bran** (**gateways**). Základní operací je převod hlasu na IP datagramy, což se děje pomocí tzv. **DSP** (digital signal processor), které převádí analogový signál na digitální a zpět. DSP dále zkomprimuje a zakóduje signálhlas určitým **kodekem** (kódovacím algoritmem). V IP telefonii se dále využívá řada protokolů k zajištění jejího fungování, jako např.:
  - signalizační: **H.323** (standard zahrnující více protokolů), SIP, …
  - přenosový: RTP/RTCP
- **správa SNMP (Simple Network Management Protocol)** – SNMP je protokol sloužící ke správě sítě. Existuje v několika verzích. Zařízení se zde dělí na:
  - **manager (NMS)** – Shromažďuje data od agentů, jichž se může dotazovat. Na data se dívá jako na proměnné, které si může vyžádat, popř. i nastavit. Může běžet na stejném nebo i jiném počítači jako agent.
  - **agent** – Běží na určitém uzlu sítě, shromažďuje si data do **MIB** (**Management Information Bases**) hierarchické databáze a při dotazu je posílá managerovi. Managera může sám kontaktovat jen pomocí tzv. příkazu trap. MIB se skládá z objektů identifikovaných pomocí OID. Obsahuje informace např. o systému, který na zařízení běží, dobu jeho běhu, statistiky apod.

  Tato zařízení spolu komunikují SNMP aplikačním protokolem.
- **Netflow** (síťové toky) – **Síťový tok** je posloupnost paketů mající společnou vlastnost a procházející bodem pozorování za určitý časový interval. Tok může být buď aktivní nebo neaktivní v případě, pokud se v něm po určitou dobu nevyskytl žádný paket. Netflow identifikuje tok podle:
  - zdrojové a cílové IP adresy
  - zdrojového a cílového portu
  - názvu logického rozhraní
  - protokolu L3 (ICMP, TCP, UDP, …)
  - hodnoty ToS (typ služby)

  Netflow byl standardizován firmou Cisco. Existují zde dva typy zařízení: **exportér** a **kolektor**.
  - **exportér –** Zařízení (HW i SW) v síti, které vytváří záznamy o tocích a ukládá je do paměti NetFlow cache. Exportér sám posílá data, jež nashromáždil, a to v těchto případech:
    - detekce konce toku (např. u TCP příznak RST ci FIN)
    - neaktivita toku (tzv. neaktivní timeout)
    - příliš dlouhý tok (tzv. aktivní timeout)
    - zaplnění paměti NetFlow cache
  - **kolektor –** Pouze SW zařízení, přijímá data z exportérů, zpracovává je a ukládá. Narozdíl od SNMP se sám nedotazuje exportérů a čeká, až jej kontaktují oni.

Jak exportér, tak i kolektor mohou využívat **vzorkování**, aby snížili nárok na HW. Vzorkování znamená, že se vybírají pouze některé pakety toku. Rozlišujeme vzorkování **deterministické** (analyzujeme každý *n*tý paket) a **náhodné** (analyzujeme vždy jeden náhodně vybraný paket z *n*). Exportér a kolektor komunikují protokolem NetFlow. NetFlow lze použít k monitorování vytížení sítě, shromažďování statistik, detekci útoků, odchylek od běžného provozu apod.

39. TCP/IP komunikace (model klient-server, protokoly TCP, UDP a IP, řízení a správa toku TCP). IPK, ISA

TCP (**transmission control protocol**) a **IP** (**internet protocol**) jsou standardní protokoly používané pro komunikaci v Internetu. TCP je protokol **transportní vrstvy** (společně s protokolem UDP, který je potenciálně rychlejší ale nezaručuje spolehlivý přenos) a stará se o spolehlivý přenos dat ve správném pořadí. IP je protokol **síťové vrstvy** a umožnuje adresovat síťová rozhraní v Internetu. Protokol IP existuje ve verzích **IPv4** a **IPv6**.

Model **klient-server** je způsob síťové komunikace, kdy jeden proces (**klient**) zasílá požadavky druhému (**serveru**), který na ně pouze čeká a odpovídá. Zpracování požadavku probíhá na straně serveru. Komunikaci většinou zahajuje klient, popř. klienti, jichž může být více. Server může být tzv. konkurentní, což znamená, že je schopný obsluhovat více požadavků současně. Klient se serverem komunikuje pomocí určitého protokolu.

Protokol je soubor syntaktických a sémantických pravidel určujících způsob výměny informací mezi dvěma entitami.

Jako **TCP/IP** bývá označován model sítě, který je používán v internetu podle prvních dvou nejdůležitějších protokolů, které využívá. Zahrnuje následující síťové vrstvy (tabulka nabízí porovnání s obecným referenčním **ISO/OSI modelem**):

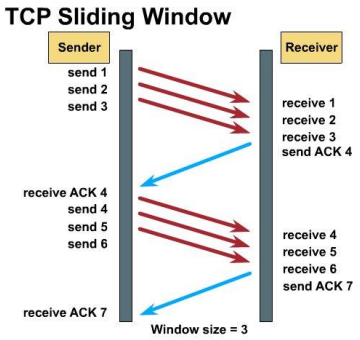
č.	TCP/IP	ISO/OSI	popis	příklady protokolů
5	**aplikační** (**application**)	**aplikační** (**application**)	Poskytuje aplikacím přístup ke komunikačnímu systému.	FTP, DNS, POP3, SSH, DHCP, SMTP, Telnet, ...
		prezentační (**presentation**)	Transformuje data do tvaru, kterému rozumí aplikace.	Samba, ...
		relační (**session**)	Umožňuje udržovat realce a řídit dialog mezi systémy.	SSL, NetBIOS, ...
4	**transportní** (**transport**)	**transportní** (**transport**)	Zajišťuje přenos dat mezi koncovými uzly.	TCP, UDP, ...
3	**síťová** (**network**)	**síťová** (**network**)	Stará se o směrování a adresování.	IP, ...
2	**linková** (**data-link**)	**linková** (**data-link**)	Poskytuje spoj mezi dvěma sousedními systémy.	Ethernet, ...
1	**fyzická** (**physical**)	**fyzická** (**physical**)	Stará se o fyzický přenos dat přenosovým médiem.	FireWire, fyz. vrstva USB, Bluetooth, ...

Port je číslo jednoznačně identifikující socket na běžícím počítači. Mechanismus portů umožňuje adresování na transportní vrstvě (podobně jako IP adresa umožňuje adresování na síťové vrstvě nebo e-mailová adresa na aplikační).

Na transportní vrstvě se používají typicky dva protokoly:

- **TCP** – Zajišťuje spolehlivé doručení zprávy, ale kvůli režii k tomuto potřebné je mírně pomalejší než UDP. Hodí se pro přenos souborů a dat, v nichž nesmí nastat chyba. První fází komunikace je zahájena serverem, který začne poslouchat na určitém portu. Klient se potom pokusí se serverem na tomto portu spojit, server mu odpoví potvrzením a klient potvrdí ustanovení spojení – tato série akcí se nazává **three way-handshake** (SYN, SYN-ACK, ACK)**.** Potom probíhá samotný přenos dat, při němž se uplatňují následující mechanismy:
  - Spolehlivost doručení doručení je zajištěna sekvenčními čísly paketů. Ty zajistí, že budou data přijata ve správném pořadí, popř. že se znovu pošlou data, která se ztratila (příjemce čeká na paket s určitým sekvenčním číslem a dokud jej nedostane, nepošle zpět potvrzení – pokud odesilatel nedostane do určité doby potvrzení, pošle data znovu).
  - K eliminaci chyb slouží kontrolní součet v TCP hlavičce paketu. Nesedí-li ověření, paket se posílá znovu (protože se nepošle potvrzení).
  - Používá se správa toku pomocí mechanismu **sliding window** (existují různé varianty, např. stop an wait s velikostí okna = 1 nebo go back n, kdy se přijímá jen následující očekávaný packet). Data se odesílají po *N* paketech, odesílatel pak vždy posune začátek okna na poslední paket v řadě potvrzených paketů. Tento mechanismus automaticky přizpůsobuje rychlost přenosu jak příjemci, tak rychlosti sítě, neboť odesílatel může poslat nová data až tehdy, když dostane potvrzení na poslední paket z okna.
  - Používá se kontrola zalcení – pokud se ztratí paket, odesílatel sníží rychlost vysílání dat, protože předpokládá, že příčinou ztrát je zahlcení bufferu příjemce. **Zahlcení** je situace, kdy se posílá více dat, než linka stíhá přenášet.

  Spojení je nakonec ukončeno podobným způsobem, jako byl na začátku three-way handshake.
- **UDP** – Umožňuje rychlejší přenos, ale nezaručuje spolehlivé doručení ani doručení ve správném pořadí (nicméně poskytuje určitou kontrolu správnosti dat pomocí kontrolního součtu, ta však může být ignorována). UDP je vhodné pro hry, streaming a jiné aplikace, kde určité množství ztráty dat nevadí tolik jako ztráta rychlosti přenosu. UDP samo o sobě neposkytuje kontrolu zahlcení. Také se nevytváří ani neukončuje žádné spojení, jako tomu je u TCP. Zprávám posílaným pomocí UDP se říká **datagramy**.



Obrázek 94: sliding window

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Obrázek 96: hlavička TCP

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Obrázek 95: hlavička UDP

Protokol **IP** zavádí adresy koncových zařízení, tzv. **IP adresy**. V IPv4 jsou tyto adresy 32bitové a zapisují se ve formátu *A.B.C.D*, kde *A*,*B*,*C* a *D* jsou čísla 0 – 255. Rozmezí těchto adres však nestačilo pokrýt všechna zařízení v internetu, a proto verze IPv6 zavedla 128bitové adresy. Zápis těchto adres je složitější, v základu má ale tvar osmi čtyřmístných hexadecimálních čísel oddělených dvojtečkou. Mezi rozdíl IPv6 oproti IPv4 patří:

- Větší adresový prostor.
- Nemá v hlavičce kontrolní součet.
- Neprovádí se **fragmentace** (rozdělování paketů, aby prošly některými routery).

V počátcích IPv4 byly adresy rozděleny na tzv. třídy:

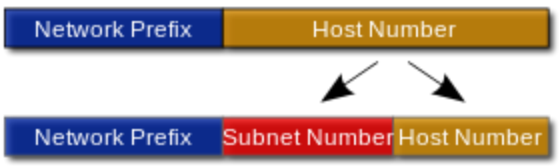
třída	počáteční bity	celkem bitů na síť	sítí	adres v síti	první adresa	poslední adresa
A	0	8	128 (2^7)	16777216	0.0.0.0	127.255.255.255
B	10	16	16384 (2^{14})	65536	128.0.0.0	191.255.255.255
C	110	24	2097152 (2^{21})	256	192.0.0.0	223.255.255.255
D (multicast)	1110	nedefinováno	nedefinováno	nedefinováno	224.0.0.0	239.255.255.255
E (rezervováno)	1111	nedefinováno	nedefinováno	nedefinováno	240.0.0.0	255.255.255.255

Tento systém vedl k plýtvání adresami (někomu nestačil počet adres v síti třídy C, tak si rezervoval blok B, ten ale většinou jej z daleka nevyužil) a byl nahrazen systémem **CIDR** (**classless internet domain routing**). CIDR zapisuje IP adresu jako adresu a prefix sítě ve formátu a.b.c.d/prefix, kde prefix je počet bitů zleva značících síť. Např.:

192.168.100.5/24 – maska podsítě je 255.255.255.0 (24 jedniček zleva), adresa v této síti tedy musí mít tvar 192.168.100.*X*, podporuje tedy 256 adres.

Některé IP adresy jsou rezervované, např. 255.255.255.255 je **broadcast** (vysílání všem v dané síti), 127.0.0.0/8 je **loopback** (samotné zařízení). Některé adresy jsou tzv. **privátní**, tzn. že nemohou být použity v internetu, ale pouze v soukromých sítích. Jsou to 10.0.0.0/8, 172.16.0.0/12 a 192.168.0.0/16. Adresa končící na 0 značí celou síť, 255 broadcast, proto nemohou být použity pro konkrétní zařízení.

Rozdělování sítě na více sítí se nazývá **subnetting**.

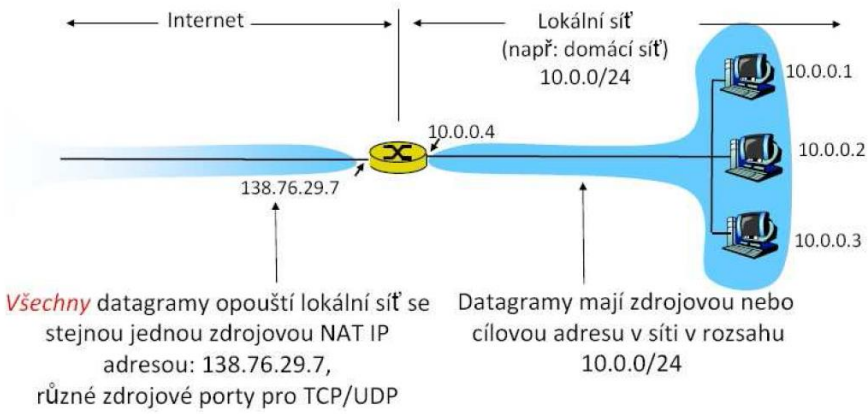


Překlad adres (**NAT**, **network address translation**) je technika umožňující, aby síť počítačů vystupovala v internetu pod jednou adresou. Pokud chce počítač z vnitřní sítě vyslat zprávu ven, vyšle jej na rozhraní pro vnější síť. To si tento požadavek zapamatuje a vyšle jej pod určitým portem, tzn. změní hlavičku požadavku. Pokud přijde odpověď na tento port, rozhraní ví, že je určeno konkrétnímu počítači a odpověď mu přepošle. NAT má tyto vlastnosti:

- Z internetu nelze zahájit spojení s konkrétním počítačem v NATu.
- Je bezpečný (útočník se nedostane ke konkrétnímu počítači).
- Šetří IP adresy.
- Ve vnitřní síti lze snadno měnit IP adresy.
- Za NATem nemůže klasicky být server, protože by nebyl adresovatelný. Toto lze vyřešit tzv. **forwardingem**, což je situace, kdy nastavíme rozhraní do internetu, aby žádosti z internetu automaticky přeposílalo na konkrétní počítač.

Síťové programování probíhá za pomoci knihoven, např. **BSD schránek** (**BSD sockets**), které implementují rozhraní síťových soketů. **Soket** je koncový bod identifikovaný IP adresou a portem a je možné přes něj posílat a přijímat data.

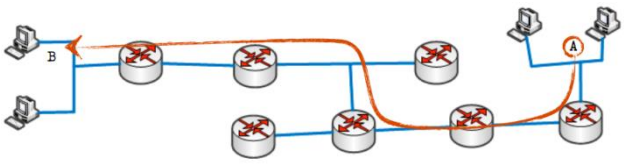
Obrázek 98: subnetting



Obrázek 97: NAT

40. Směrování a filtrování dat v Internetu (algoritmy Link-state a Distance-vector, RIP, OSPF, klasifikace paketů a filtrování, firewally, kvalita služeb). IPK, ISA

Směrování (**routing**) je určování cest paketů za pomoci směrovačů v počítačové síti. O směrování se stará **síťová vrstva**. Ta také zajišťuje šíření směrovacích informací pomocí směrovacích protokolů. **Přepínání** (**forwarding**) je lokální akce, při níž směrovač přesune data z jednoho svého vstupu na jedno ze svých výstupních rozhraní.



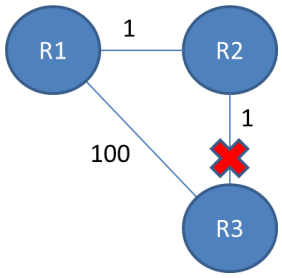
Obrázek 99: znázornění směrování

Pro směrování se používají tzv. **směrovací tabulky**, které si směrovače uchovávají. Tabulka rozhoduje o tom, na které výstupní rozhraní se daný paket pošle (přepne), a to na základě jeho adresy. V tabulce není z kapacitních důvodů možné uchovávat všechny možné existující IP adresy, proto se uchovávají pouze prefixy různých délek a adresa paketu se vždy testuje na **nejdelší shodu** s prefixem (**longest prefix match**) např. takto:

201.10.6.17

4.0.0.0/8	Serial0/0
4.83.128.0/17	Serial0/0
201.10.0.0/21	Serial1/0
201.10.6.0/23	Serial0/1
126.255.103.0/24	Serial0/0

Pro hledání nejdelšího shodného prefixu se používají tzv. **prefixové stromy** (**Patricia trees**). Směrovací tabulky mohou být nastavovány buď **staticky** (manuálně, nevýhodné z hlediska změn sítě) nebo **dynamicky** (automaticky na základě algoritmů). **Brána** (**gateway**) je směrovač, který propojuje lokální síť s vnější sítí.



Obrázek 100: problém počítání do nekonečna

Pro účely směrování existují **směrovací protokoly**, jimiž si směrovače vyměňují informace o síti. Doba potřebná k rozšíření informace ke všem směrovačům se nazývá doba **konvergence**.

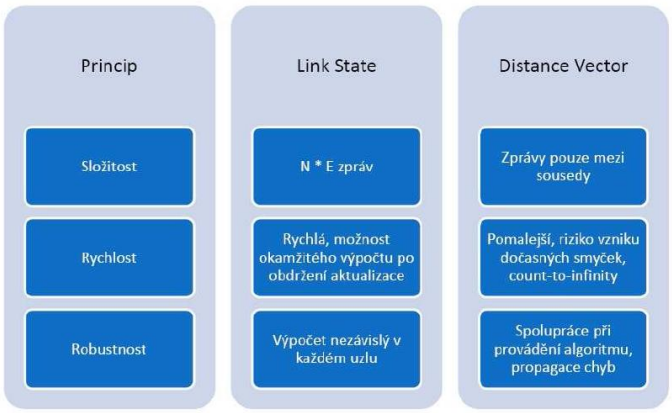
Směrovací algoritmy dělíme na:

- **link-state** – Každý uzel sítě má kompletní informaci o topologii celé sítě. Každý uzel, který zjistí změnu v síti (pravidelně vysílá HELLO pakety sousedům), musí tuto informaci poslat všem ostatním uzlům sítě (tzv. LSA flooding). Jediná informace, která se přenáší, je ta o propojení uzlů (nikoli celé směrovací tabulky). Jsou vhodnější pro větší sítě. Po *k* iteracích jsou známy cesty do *k* cílů.
- **distance-vector** – Název odráží fakt, že si uzly pamatují vektor vzdáleností ke všem ostatním uzlům sítě. Uzly musí periodicky informovat své sousedy o změnách v síti. Uzly si vyměňují části svých směrovacích tabulek. Jsou méně výpočetně náročné než link-state, ale více zatěžují síť. Vyskytuje se zde **problém počítání do nekonečna**. Tuto situaci ukazuje obr. – když se cesta mezi R2 a R3 stane nedostupnou, ale R2 o tom zatím neinformuje R1, bude se paket poslaný do R3 posílat donekonečna mezi R1 a R2 (protože R1 si myslí, že nejlepší cesta do R3 vede přes R2 a R2 posílá paket zpátky na R1, protože cesta přes něj je nedostupná). V praxi se počítá do maximálního omezeného počtu přeposlání paketu.

RIP (**routing information protocol**) je distance-vector směrovací protokol. Jako metriku používá počet skoků a zamezuje směrovacím smyčkám omezením tohoto počtu na maximální hodnotu, která je 15 (uzel vzdálený 16 je považován za nedostupný, v nekonečnu). Periodicky se aktualizuje každých 30 sekund, uzel, který se 180 sekund neohlásí, je považován za neplatný. RIP zprávy jsou přenášeny pomocí UDP. Existují různé verze RIP.

OSPF (**open shortest path first**) je link-state směrovací protokol. Pro výpočet cesty používá **Dijkstrův algoritmus**. Informace se posílají v případě změny okamžitě, jinak alespoň jednou za 30 minut. Spojení se sousedy se kontroluje pravidelnými ECHO zprávami.

Autonomní systém je množina IP sítí pod společnou technickou správou, která reprezentuje vůči Internetu společnou routovací politiku. Pro routování uvnitř autonomního systému se používá některý z tzv. Interior Gateway protokolů (např. OSPF), pro routování mezi autonomními systémy se používají tzv. Border Gateway Protocol. Internet by teoreticky mohl tvořit jeden velký autonomní systém, ale směrovací tabulky by byly příliš velké. Autonomní systém je identifikován jednoznačným číslem.



Obrázek 101: link-state vs distance-vector

Firewall je systém (software/hardware), který kontroluje příchozí a odchozí síťový provoz. Je definován sadou pravidel, která rozhodují, zda daný paket projde nebo ne. Filtrování může být:

- **filtrování paketů** – Kontroluje jednotlivé pakety izolovaně.
- **stavové** – Pamatuje si stav TCP spojení, provoz je považován za dvousměrnou výměnu paketů v rámci relace, pravidla se generují dynamicky.
- **aplikační** – Pracuje na aplikační úrovni, rozumí aplikačním protokolům (např. FTP).

Filtrování může buď propouštět vše, co není zakázáno (**exkluzivní přístup**), nebo propouštět jenom to, co je v splní pravidla (**inkluzivní přístup**). **Bezpečnostní zóny** definují zvláštní politiku pro různé části sítě (např. LAN, DNS, e-mail a internet). Pokročilá filtrovací pravidla mohou být časově nebo kontextově závislá. Kromě firewallů existují další systémy pro ochranu síťového provozu, např.:

- **IDS** (**intrusion detection system**) – Snaží se detekovat útok kontrolou provozu, monitorováním dat a detekcí neočekávaného chování.
- **IPS** (**intrusion prevention system**) – Snaží se odklonit nebo odfiltrovat již probíhající útok.

Firewally musí provádět tzv. **klasifikaci paketů**, což je algoritmický problém snažící se rychle roztřídit pakety podle sady pravidel. Snažíme se pro paket najít první shodu (záleží na pořadí pravidel) s konkrétním pravidlem, které definuje, jaká akce se provede, výsledkem je tedy číslo pravidla. Paket obsahuje *K* hlaviček (např. IP adresa, zdroj. port atd.), přičemž každé pravidlo obsahuje na každou hlavičku podmínku. Podmínka může vyžadovat jedno ze tří typů porovnání: **přesné** (např. na protokol), **prefixové** (většinou na IP adresu) nebo **intervalové** (port, IP adresa apod.). Množina pravidel se nazývá **klasifikátor**. Mezi konkrétní způsoby vyhledávání pravidel patří:

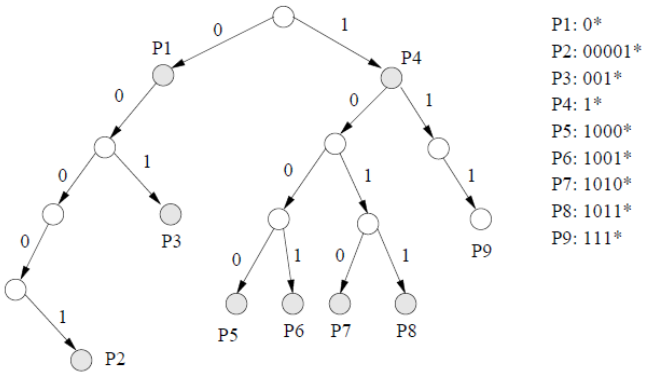
	Cíl. IP	Zdroj. IP	Cíl. port	Zdroj. port	Příznaky	Poznámka
1	147.229.*.*	*	25	*	*	povol příchozí emaily
2	147.229.*.*	*	53	*	UDP	povol dotazy DNS
3	147.229.*.*	*	23	*	*	povol telnet
4	153.13.2.5	147.229.5.1	123	123	UDP	povol NTP info
5	*	117.16.*.*	*	*	*	povol odchozí pakety
6	117.16.*.*	*	*	*	TCP ACK	povol zprávy ACK
7	*	*	*	*	*	blokuj vše ostatní

Obrázek 102: příklad pravidel pro LAN 147.229.0.0/16

- **lineární vyhledávání** – Jednoduchý způsob, projdou se postupně všechna pravidla jedno za druhým a porovnávají se všechny dimenze, časová složitost je lineární, hodí se pro malý počet pravidel.
- **stromové vyhledávání** – Využívá binární stromovou strukturu trie pro vyhledávání podle jedné dimenze (IP adresy). Procházení je rychlejší než u lineárního prohledávání.
- **rozšíření trie do více dimenzí** – Využívá strom stromů trie, aby bylo možné vyhledávat podle více dimenzí, prostorová složitost roste exponenciálně.

- **grid of tries bez duplicit, se zpětným vyhledáváním** – Optimalizovaný strom stromů trie, v němž jsou odstraněny duplicity a jsou přidány zpětné ukazatele. Prostorová složitost je nižší, ale časová o něco vyšší.
- **lineární vyhledávání ve více dimenzích** – Kombinace stromového vyhledávání v jedné dimenzi a lineárního dohledání v dalších.
- **lineární vyhledávání ve více dimenzích bitovým vektorem** – Bitové vektory obsahují výskyty prefixů v pravidlech (bitový vektor má tolik bitů, kolik je pravidel), nejlepší shoda se zjistí logickou operací AND.
- **vektorový součin** – Množina kombinací všech hodnot všech dimenzí, časově náročné. Pro každou dimenzi se sestaví bitový vektor, kde každý bit odpovídá jednomu pravidlu (1 = odpovídá, 0 = neodpovídá).

Multicast je situace, kdy jeden počítač v Internetu posílá stejná data více příjemcům. Teoreticky by mohl posílat data každému zvlášť (tzv **unicast**), takovou zátěž však není nutné podstupovat a je možné, aby každý datový paket poslal jen jednou a routery je potom doručily více příjemcům. Multicast je realizován protokolem **IGMP** (**internet group management protocol**). Příjemci dat se pomocí něj mohou připojit k tzv. multicastovým skupinám (které mají IP adresu třídy D), jimž jsou data zasílána.

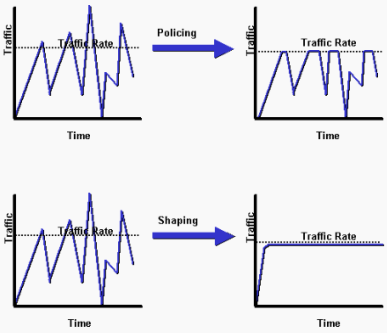


Obrázek 103: příklad trie

Dijkstrův algoritmus, který vypočítá ceny cest z uzlu *A* je následující:

```
Dijkstrův algoritmus (uzel A):

pro všechny uzly v:
  pokud v je sousedem A:
    cena(v) <- ohodnocení hrany z A do v
  jinak
    cena(v) <- nekonečno

N = { A }

cyklus:
  najdi uzel w, který není v N a cena cena(w) je minimum
  N += w

  pro všechny uzly v, které nejsou v N a sousedí s w:
    cena(v) <- minimum(cena(v),cena(w) + ohodnocení hrany z w do v)
dokud nejsou všechny uzly v N
```

SLA (**service level agreement**) je smlouva, v níž se definuje **kvalita služeb** (**QOS**, **quality of service**) zajištěná provozovatelem pro zákazníka (např. dostupnost, šířka pásma, ...). Smlouva může rozlišovat různé typy dat (rozeznají se díky značení paketů).

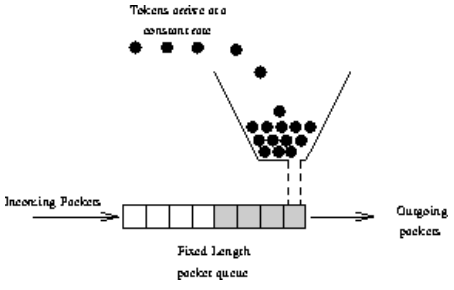


K zajištění QOS se používá:

- **traffic shaping** (**rozložení provozu**) – Zpožďuje některé pakety, aby se předešlo vytížení.
- **traffic policing** (**omezení provozu**) – Zahazuje některé pakety, aby se předešlo vytížení, nepotřebuje fronty (narozdíl od shapingu).
- **značení paketů** – Používá se např. pole TOS (type of service) v IP hlavičce.

Routery používají různé typy front:

- FIFO – Jednoduché, ale neporadí si s prioritami.
- Prioritní fronty – Existuje více front, každá pro jednu prioritu. Vybírá se nejdříve z nejprioritnější fronty a až se vyprázdní, přejde se k nižší prioritě. Hrozí hladovění.
- Cyklické prioritní fronty – Prioritní fronty s tím, že se pakety vybírají postupně z každé fronty pořád dokola, řeší problém hladovění.
- Váhové fronty – Fronty vznikají a zanikají dynamicky, každý tok má vlastní frontu.
- **Token Bucket** – Vhodné pro rozložení provozu. Existuje vědro o určité velikosti, do něhož se periodicky generují tokeny až po jeho zaplnění. Paket na začátku fronty může projít v případě, že vědro obsahuje alespoň 1 token, v takovém případě se token odstraní.
- **Leaky bucket** – Token bucket s velikostí vědra 1 a bez fronty, hodí se pro omezení provozu.



Obrázek 104: token bucket

Modely zajištění QOS jsou:

- **integrované služby** – Princip je založen na vytváření virtuálních okruhů, na jejichž udržování se musí podílet každý směrovač na cestě. Z tohoto důvodu se musí na všech směrovačích rezervovat prostředky (např. pásmo), což se dělá pomocí protokolu RSUP (reliable state update protocol). Definují se tyto třídy paketů:
  - **garantovaná** – Jsou garantovány určité parametry, např. maximální zpoždění apod.
  - **kontrolovaná zátěž** – Pro tato data se systém vždy snaží zajistit podmínky podobné nezatížené síti, avšak nic není garantováno.
  - **best effort** – Pro tato data se snaží zajistit co nejlepší doručení s ohledem na výše zmíněné třídy.
- **diferenciované služby** – Principem je značení paketů a snaha o jejich nejlepší doručení (best effort). Rozlišují se zde jednak **hraniční prvky**, které označují pakety vstupující do sítě a **páteřní prvky**, které se snaží na základě značení o jejich efektivní doručování.

RED (**random early detection**) je technika, při které se náhodně zahazují pakety ve frontách, přičemž pravděpodobnost zahození paketu roste s délkou fronty. Eliminuje se tak rapidní zpomalení, které nastane bez použití RED ve chvíli, kdy se zaplní buffery, začnou se zahazovat pakety a všechny TCP toky začnou ihned zpomalovat svou rychlost.

WRED (**weighted RED**) je vylepšení RED, v němž pravděpodobnost zahození paketu záleží taky na jeho třídě (prioritní pakety se zahazují méně nebo vůbec).