

## Typy simulace

- **Podle popisu modelu:**
  - **Spojitá/diskrétní/kombinovaná**
  - Kvalitativní/kvantitativní
- **Podle simulátoru**
  - Na **analogovém/číslicovém** počítači, fyzikální
  - **Real-Time** simulace
  - **Paralelní a distribuovaná** simulace

## Analytické řešení modelů

Popis chování modelu matematickými vztahy a jeho matematické řešení. Vhodné pro jednoduché systémy nebo zjednodušený popis složitých. Dosazením korektních hodnot získáme řešení (např. model volného pádu ve vakuu - v atmosféře už by byl ale složitější).

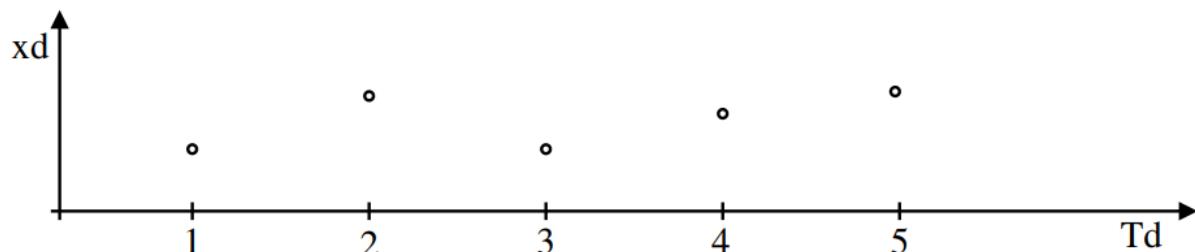
## Čas

- **Reálný** - Ve kterém probíhá skutečný děj.
- **Modelový** - Časová osa modelu (nemusí být synchronní s reálným).
- **Strojový** - Čas CPU spotřebovaný na výpočtu programu.

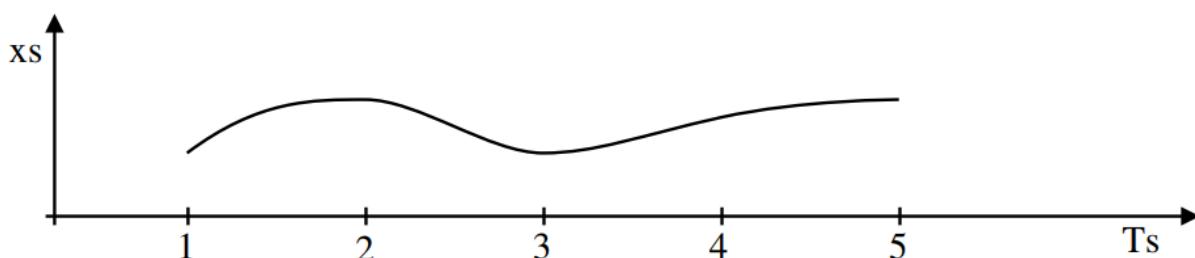
## Časová množina

Množina všech časových okamžiků, ve kterých jsou definovány hodnoty vstupních, stavových a výstupních proměnných prvků systému

- **Diskrétní** - {1,2,3,4,5}.



- **Spojitá** -  $<1.0, 5.0>$ . Tato se na počítači diskretizuje.



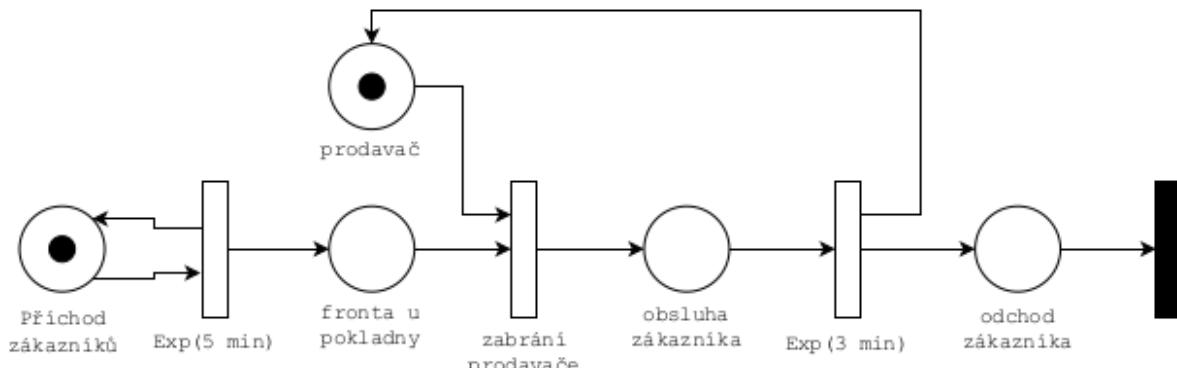
# Simulační metody

## Petriho sítě

Petriho síť je formálně definována jako pětice  $N = (P, T, I, O, M_0)$ , kde:

- $P = \{ p_1, p_2, \dots, p_m \}$  je konečná množina míst,
- $T = \{ t_1, t_2, \dots, t_n \}$  je konečná množina přechodů,  $P \cup T \neq \emptyset$  a  $P \cap T = \emptyset$ ,
- $I : T \times P \rightarrow \mathbb{N}$  je vstupní funkce, která definuje orientované křivky z míst do přechodů, kde  $\mathbb{N}$  je množina nezáporných celých čísel,
- $O : T \times P \rightarrow \mathbb{N}$  je výstupní funkce, která definuje orientované křivky z přechodů do míst,
- $M_0 : P \rightarrow \mathbb{N}$  je počáteční značení.

**Značení** znamená přiřazení žetonů do míst Petriho sítě. Orientovaná křivka směřující z místa  $p_j$  do přechodu  $t_i$  definuje  $p_j$  jako **vstupní místo přechodu  $t_i$** . Orientovaná křivka směřující z **přechodu  $t_i$**  do **místa  $p_j$**  pak definuje  $p_j$  jako **výstupní místo** přechodu  $t_i$ . Vstupní místo je místo, ze kterého je žeton při provádění přechodu **odebrán**, a naopak výstupní místo je místo, do kterého je **žeton** po provedení přechodu **přesunut**. Graficky jsou mísťa v Petriho sítích značena elipsami, obvykle je ale znázorňujeme **kružnicemi**. **Přechody** jsou značeny **obdélníky**, orientované **křivky šipkami**. **Žetony** mohou být značeny **tečkami** nebo **číslem**, které vyjadřuje počet teček – žetonů. Značení žetonů se obvykle vpisuje do značky místa. Příklad systém M/M/1 dle Kendalovy klasifikace:



## Metoda Monte Carlo

Experimentální numerická (simulační) metoda. Řeší úlohu experimentování se stochastickým modelem. Využívá vzájemného vztahu mezi hledanými veličinami a pravděpodobností, se kterými nastanou jevy. Vyžaduje generování náhodných čísel. Není příliš přesná. Vhodné, když jsou běžné numerické metody nepraktické.

Jednoduchá implementace (existuje více variant).

1. Vytvoříme stochastický model.
2. Provádíme náhodné experimenty.
3. Získanou pravděpodobnost nebo průměr použijeme pro výpočet výsledku.

Použití:

- výpočet obsahu, objemu těles (nemusíme znát obor hodnot funkce)

- výpočet integrálů, zejména vícerozměrných,
- řešení diferenciálních rovnic

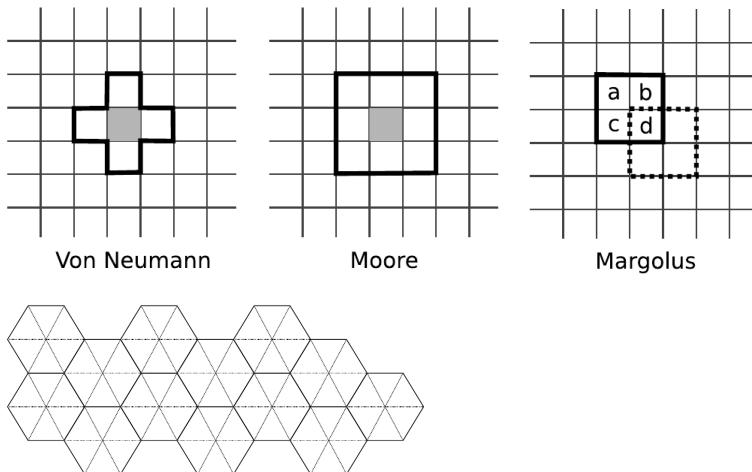
Přesnost (N je počet provedených experimentů):

$$err = \frac{1}{\sqrt{N}}$$

## Celulární automaty

**Celulární automaty** jsou diskrétní systémy. Je možné implementovat jako pole, vyhledávací tabulku (nenulové buňky). Existují také reverzibilní automaty, u kterých je možné se vracet nazpět v simulaci. Celulární automaty jsou tvořeny:

- **Buňka (cell)** - základní element, může být v jednom z **konečného počtu stavů**, např. {0, 1}.
- **Pole buněk (lattice)** - **N-rozměrné** (obvykle 1D nebo 2D). Rovnoměrné rozdělení prostoru, může být **konečné nebo nekonečné**.



- **Okolí (neighbourhood)** - Typy se liší počtem a pozicí **okolních buněk**.
- **Pravidla (Rules)** - Funkce stavu buňky a jejího okolí definující nový stav buňky v čase:  $s(t+1) = f(s(t), N_s(t))$ .

Lze je rozdělit do **4 tříd**:

- **Třída 1** - Po konečném počtu kroků dosáhnou jednoho ustáleného konkrétního stavu.
- **Třída 2** - Dosáhnou periodického opakování nebo zůstanou stabilní.
- **Třída 3** - Chaotické chování (fraktální útvary).
- **Třída 4** - Kombinace běžného a chaotického chování (např. life) - nejsou reverzibilní.

Nejznámější celulární automat je hra **Life**.

## Numerické metody

### Markovovy modely

#### Metoda snižování řádu derivace

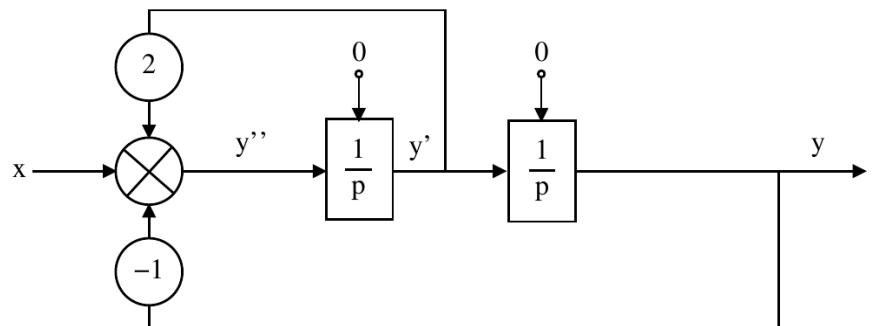
1. Osamostatnit nejvyšší řád derivace.
2. Zapojit všechny integrátory za sebe a na vstupu prvního zapojit (modifikovaný - násobení, přičítání) výsledek
3. Tato metoda funguje, pokud nejsou derivované vstupy derivace vstupů ( $x'$ ,  $x''$ , ...).

Příklad: rovnice  $y'' - 2y' + y = x$

$$y'' = 2y' - y + x$$

$$y' = \int y''$$

$$y = \int y'$$



#### Metoda postupné integrace

1. **Osamostatnit nejvyšší řád** derivace.
2. Postupná integrace rovnice a zavádění nových stavových podmínek.
3. Výpočet nových počátečních podmínek.
4. Podmínka: konstantní koeficienty.

Příklad: rovnice  $p^2y + 2py + y = p^2x + 3px + 2x$

$$p^2y = p^2x + p(3x - 2y) + (2x - y)$$

$$py = px + (3x - 2y) + \frac{1}{p}(2x - y), \text{ proměnná } w_1 = \frac{1}{p}(2x - y)$$

$$py = px + (3x - 2y) + w_1$$

$$y = x + \frac{1}{p}(3x - 2y + w_1), \text{ proměnná } w_2 = \frac{1}{p}(3x - 2y + w_1)$$

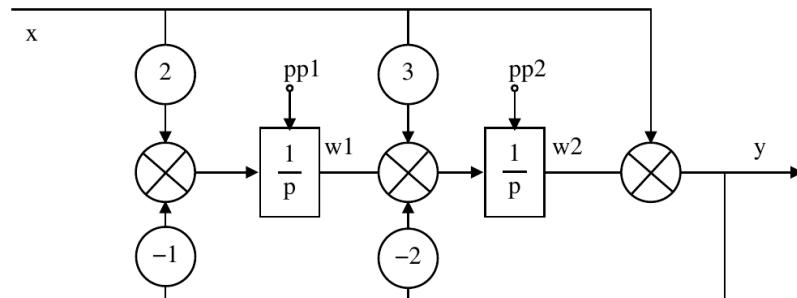
$$y = x + w_2$$

Výsledná soustava rovnic:

$$w_1 = \frac{1}{p}(2x - y), \quad w_1(0) = y'(0) - x'(0) - 3x(0) + 2y(0)$$

$$w_2 = \frac{1}{p}(3x - 2y + w_1), \quad w_2(0) = y(0) - x(0)$$

$$y = x + w_2$$



# Algoritmy řízení simulace

Různé algoritmy pro diskrétní, spojité a kombinované simulace.

## Next event (řízení diskrétní simulace)

Jedná se algoritmus řízení diskrétní simulace. Vypadá následovně:

1. **Inicializace** času, kalendáře, modelu.
2. Dokud **není kalendář prázdný**, vyjmí první záznam z kalendáře (záznamy jsou seřazeny)
3. Pokud aktivační čas události **přesáhl** čas konce simulace, ukonči simulaci.
4. **Nastav** čas na aktivační čas události.
5. Proved' **chování**, které popisuje událost a **vrat' se** na bod 2.

## Kalendář

Uspořádaná **datová struktura** uchovávající aktivační záznamy budoucích událostí.

Každá naplánovaná událost má v kalendáři záznam, který je tvořen minimálně:

- **aktivační čas**,
- **priorita**,
- **vykonávanou událost**.

Kalendář umožňuje **výběr prvního záznamu** s nejmenším aktivačním časem a **vkládání/rušení záznamu**.

## Řízení spojité simulace

Spojitá simulace je složena ze tří částí:

- metoda/funkce **Dynamic**, ve které dochází k **aktualizaci vstupů integrátorů**,
- **numerická metoda** (Euler, RK) ve které se **pro každý integrátor počítají nové stavy**,
- **hlavní cyklus**, který řídí simulaci (volá předchozí metody), **inkrementuje čas** dle **kroku** a sleduje, jestli nebylo dosaženo konce simulace a případně provádí **dokročení**.

Dokročení může být řešeno dvěma způsoby:

- pokud do konce simulace zbývá **menší časový okamžik**, než je nastavený krok, lze poslední **krok snížit** tak, aby čas po jeho provedení přesně odpovídal konci simulace. (příliš malý krok může způsobit nepřesnost)
- pokud do konce simulace **zbývá krok a kousek**, můžeme poslední **krok prodloužit** tak, aby čas po jeho provedení přesně odpovídal konci simulace.

Příklad řízení spojité simulace bez dokročení:

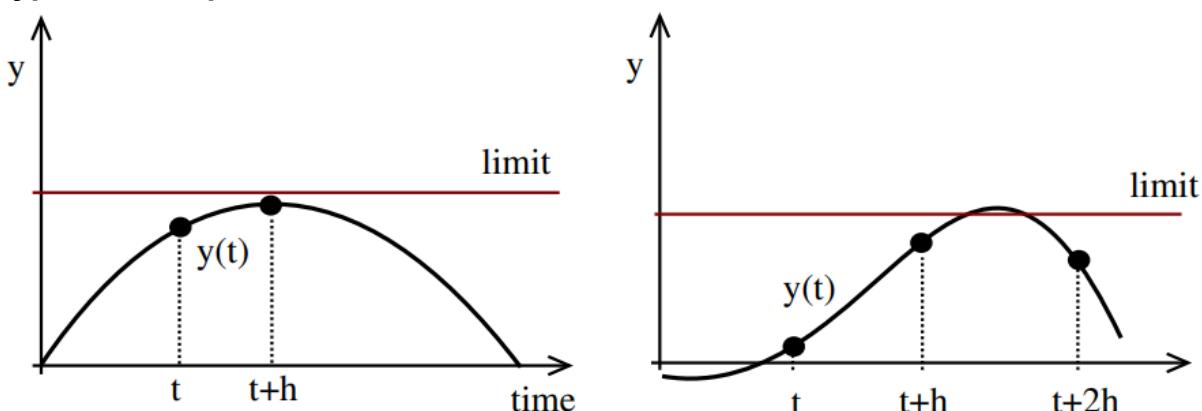
```

double yin[2], y[2] = { 0.0, 1.0 }, time = 0, h = 0.001;
void Dynamic() { // f(t,y): výpočet vstupů integrátorů
    yin[0] = y[1]; // y'
    yin[1] = -y[0]; // y'',
}
void Euler_step() { // výpočet jednoho kroku integrace
    Dynamic(); // vyhodnocení vstupů integrátorů
    for (int i = 0; i < 2; i++) // pro každý integrátor
        y[i] += h * yin[i]; // vypočteme nový stav
    time += h; // posun modelového času
}
int main() { // Experiment: kruhový test, čas 0..20
    while (time < 20) {
        printf("%10f %10f\n", time, y[0]);
        Euler_step();
    }
}

```

## Řízení kombinované simulace

U kombinované simulace je nutné řešit kombinaci **stavových událostí a numerické integrace**. Je nutné **detektovat změnu** stavových podmínek a **přesně dokročit** na čas, kdy ke změně dochází (problém příliš malého kroku). Ke **stavovým událostem** dochází při **změnách stavových podmínek**, nelze je naplánovat. Změnu stavové podmínky může být **obtížné detektovat** z důvodu **nepřesnosti numerického výpočtu** nebo **příliš dlouhého kroku**.



Algoritmus řízení kombinované simulace pracuje následovně:

1. **Inicializace** stavu, času, modelu atd.
2. **Kontrola**, že není dosažen čas **konce simulace** a její případné ukončení.
3. **Uložení** aktuálního stavu a času.
4. Provedení jednoho **kroku numerické integrace**.
5. Kontrola, jestli nedošlo ke změně stavových podmínek. Pokud ne, pokračuje se bodem 2, jinak bod 6.

6. Hledání **okamžiku** změny **stavové podmínky** (využití uložených hodnot v **kroce 3**) např. metodou půlení intervalů. Okamžik se hledá s **přesností minimálního kroku** (menší krok by vedl na nepřesnost, stejně jako na nepřesnost vede nepřesné určení času změny stavové podmínky)

Pseudokód algoritmu:

```
Inicializace stavu a podmínek
while ( čas < koncový_čas ) {
    Uložení stavu a času ***

    Krok numerické integrace a posun času
    Vyhodnocení podmínek
    if ( podminka změněna )
        if ( krok <= minimální_krok)
            Potvrzení změn podmínek
            Stavová událost ===
            krok = běžná_velikost_kroku
        else
            Obnova stavu a času ***
            krok = krok/2
            if (krok < minimální_krok)
                krok = minimální_krok
}
```

## Řízení simulace číslicových obvodů

Tato simulace je řízená událostmi a ukládání **velkého množství** událostí do kalendáře je **nepraktické/problematické**. Používá se proto princip **selektivního sledování**, kdy dochází k vyhodnocování pouze těch prvků, na které má vliv změna na vstupu. Používá se např. **pevný krok** pro změnu času) Problematické mohou být zpětné vazby v obvodech a nastavení počátečních hodnot signálů.

Princip algoritmu:

1. **Inicializace** modelu, plánování, ...
2. Dokud je naplánovaná událost, tak pokračuj s dalším bodem, jinak konec simulace.
3. Nastav hodnotu modelového času na **T** (pevným krokem, na základě právní naplánované události, ...)
4. Pro všechny události, které jsou **naplánované** na tento čas **T** proved:
  - a. **odeber** událost z plánovaných událostí (kalendáře),
  - b. **aktualizuj** hodnoty signálů,
  - c. všechny připojené prvky na tyto signály **zařaď do množiny M**.

5. Projdi všechny prvky v množině **M** a jestli změna na jeho vstupu způsobí změnu jeho výstupu, naplánuj jeho obsluhu jako novou událost.
6. Pokračuj bodem 2.

Pseudokód:

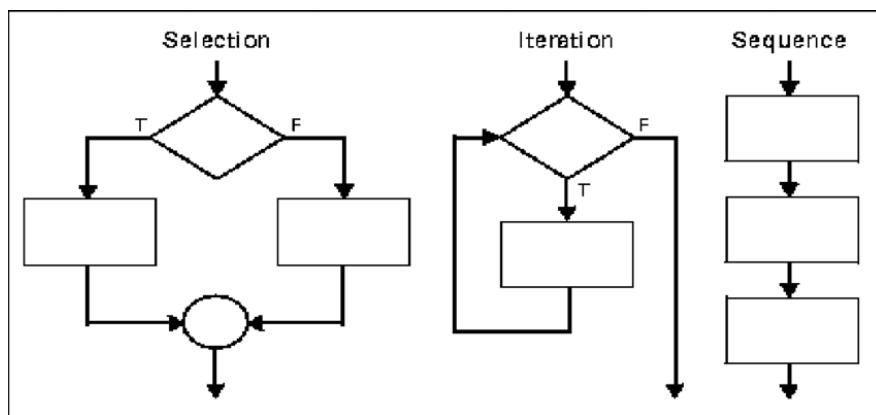
```
 inicializace, plánování události pro nový vstup
 while (je plánována událost) {
     nastavit hodnotu modelového času na T
     for (u in všechny plánované události na čas T) {
         výběr záznamu události u z kalendáře
         aktualizace hodnoty signálu
         přidat všechny připojené prvky do množiny M
     }
     for (p in množina prvků M) {
         vyhodnocení prvku p
         if (změna jeho výstupu)
             plánování nové události
     }
 }
```

# 29. Datové a řídicí struktury imperativních programovacích jazyků.

## Řídící struktury

- **Sekvence** - umožňuje provádět příkazy (**přiřazení, volání funkce, aritmetické operace, ...**) jeden po druhém. Obecně **umožňuje provádění podprogramů jeden po druhý**.
- **Selekce** - Umožňuje volbu mezi bloky kódu, které budou provedeny na základě hodnoty booleovského výrazu (**if - else if - else, switch, ...**). Obecně **umožňuje vybrat jeden ze dvou podprogramů**, který bude prováděn, na základě hodnoty booleovského výrazu.
- **Iterace** - Opakování bloku kódu na základě hodnoty booleovského výrazu (**for, while, repeat, do while, goto, ...**). Obecně **umožňuje opakování vykonávání podprogramu**.

Všechny programovací jazyky, které podporují tyto řídící struktury (a jsou vybaveny **nekonečným** sekundárním úložištěm - tento fakt je většinou ignorován) jsou **Turing Complete**. Existují i jazyky, které mají pouze jedinou instrukci a jsou Turing Complete (pomocí této instrukce je však simulována manipulace s pamětí, větvení a iterace). Tato trojice je také základem strukturovaného programování.



## Výraz/příkaz

Například **aritmetický výraz**, prostý **výskyt konstanty či proměnné, funkční volání, přiřazení, ...** Výrazový příkaz se získá **ukončením výrazu patřičným symbolem** (v C, C++, C# aj. je to středník, v Pythonu je to konec řádku). Zvláštním případem je prázdný příkaz (samostatný středník).

## Podmíněný příkaz if-else

Umožňuje rozhodnutí na základě hodnoty proměnné (**true/false**), jestli **bude** kód vykonán (true), nebo **nebude** (false). Respektive v případě nepravdy (false) se vykoná **else** větev, pokud existuje.

## Podmíněný příkaz switch

Slouží k **větvení** podle hodnoty **celočíselného výrazu**, **řetězce**, **hodnoty výčtu** atd. Řízení programu přechází do úseku kódu jehož konstantní hodnota (která jej uvádí - hodnota **case**) odpovídá hodnotě proměnné, podle které se přepíná (“switchuje” v příkazu **switch**). Switch také umožňuje provést výchozí chování (větev **default**), pokud žádný z **case** nevyhovuje.

## Cykly

Umožňují několikanásobné (**for** cyklus) opakování kódu. Nekonečné cykly (**while(true)**) je možné obvykle ukončit příkazem **break**. Přeskočení zbytku nynější iterace je možné obvykle příkazem **continue**.

- **while** - Vhodné například pro předem neznámý počet provádění. Vyhodnocuje se poprvé podmínka, tedy **tělo nemusí být vykonáno**.
- **do while (repeat until)** - Tělo je provedeno **alespoň jednou**.
- **for** - Tělo se provede **přesně stanovený počet krát**, pokud neuvažujeme modifikaci ve vyšších programovacích jazycích na podmínu.
- **goto** - v případě skoku na **návěští**, které **goto předchází**. Skákat lze pouze lokálně (tedy nelze skočit pryč z funkce - na to je např. longjump v C - to je nebezpečné narozdíl od goto).

## Složený příkaz (blok)

**Posloupnost** libovolných příkazů (**včetně dalších bloků**). Často se **syntakticky odlišuje** ({} v C, **begin end** v Pascalu, **odsazení** v Python). **Proměnné** zde definované mají často **platnost pouze v tomto bloku** - **lokální proměnné**.

## Funkce

Umožňují rozložení složitého problému na jednodušší **podprogramy**, které mají jedno vstupní místo (obvykle standardizovaným způsobem). **Deklarace** funkce se skládá z **návratového typu**, **názvu** funkce a **seznamu parametrů**. Deklaraci je často možné vynechat a funkci přímo definovat. Funkce jsou jedním z hlavních předpokladů pro **strukturované jazyky**.

## Rekurze

Metoda definování určitého objektu pomocí sebe sama.

- Umožňuje definovat **nekonečnou množinu** objektů **konečným popisem - rekurentní definice**.

- Rekurzivní struktura dat (lineární seznam)
- Rekurzivní struktura algoritmu (DFS, InOrderTraversal)
- Rekurzivní volání funkce – **funkce je volána v těle sebe samé** (obecně se tomuto případu snažíme vyhnout, **jednoduché** na implementaci, ale **nebezpečné**).

## Datové struktury

Datová struktura určuje konkrétní způsob **organizace dat v paměti počítače**.

### Datový typ

Určuje **množinu hodnot** a **množinu operací nad těmito hodnotami**. Typ proměnné se zavádí v **její deklaraci**. Je určen:

- Názvem
- Množinou hodnot, kterých může nabývat
- Množinou operací nad hodnotami

Dělení datových typů (dle několika způsobů):

- **standardní** (int, double, char, string, boolean)
- **definované uživatelem** (enum, pole, záznam, množina, soubor, ukazatel)
- **ordinální** - prvky daného typu mají jasně stanovené pořadí. Každý prvek má předchůdce a následovníka (integer, char, ...)
- **neordinální** (string, real, pointer, ...)
- **jednoduché/základní** - nemají vnitřní strukturu, s proměnnou lze pracovat jako s celkem (integer, char, real, ...)
- **strukturované** - hodnoty mají vnitřní strukturu (členění na komponenty), s proměnnou strukt. typu lze pracovat buď jako s celkem nebo s jeho jednotl. komponenty (array, struct, record, ...)

### Strukturovaný datový typ

Sestává z komponent jiného (dřív definovaného) typu = kompoziční typ. Má strukturovanou hodnotu a musí mít definované hodnoty všech komponent.

- **Homogenní** - Všechny komponenty (položky) jsou **stejného typu - pole**.
- **Heterogenní** - Komponenty (složky) jsou **rozdílného typu - struktura**.
- **Statický** - **Nemůže měnit v průběhu výpočtu** počet komponent ani způsob uspořádání.
- **Dynamický** - **Může měnit v průběhu** počet i uspořádání.

### Pole

**Homogenní** ortogonální (pravoúhlý) datový typ. Typ je **specifikovaný velikostí svých dimenzí a komponentním typem**. K prvkům se přistupuje pomocí **identifikátoru pole a indexu** prvku.

- **Vektor** - jednorozměrné pole.
- **Matice** - dvourozměrné pole.

## Řetězec

Strukturovaný homogenní datový typ. Položky mají **typ znak** (char). Má vlastnosti (a často je tak i implementován) jako **jednorozměrné pole znaků**, ale ve většině vyšších programovacích jazyků umožňuje více operací (např. porovnání, konkatenace, ...).

## Zážnam (record/struct)

Strukturovaný statický heterogenní datový typ. Komponenty mohou být libovolného datového typu. Jména a počet komponent je dán při definici typu a nemohou se měnit za běhu programu.

## Abstraktní datové typy

Při tvorbě reálných aplikací lze využít **obecného modelu datové struktury** vyjádřeného pomocí abstraktního datového typu:

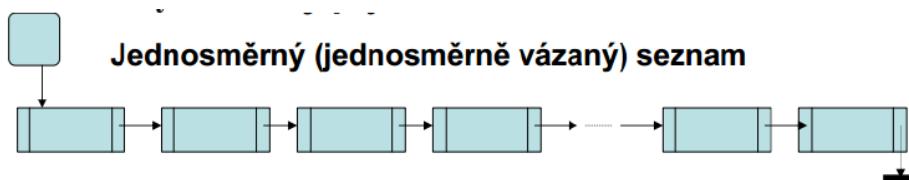
- určíme použité **datové komponenty**,
- určíme **operace** a jejich **vlastnosti**,
- **abstrahujeme** od způsobu implementace.

ADT zdůrazňuje **co** dělá, ale **potlačuje jak** to dělá. Smyslem je **zvýšit** datovou **abstrakci** a **snížit** algoritmickou **složitost** programu. ADT je definován **množinou hodnot**, kterých smí nabýt každý prvek tohoto typu, a **množinou operací** nad tímto typem

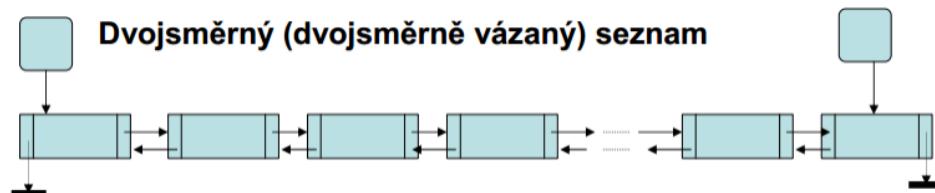
## Spojový seznam (linked list)

**Homogenní, lineární, dynamická** struktura. Prvkem seznamu může být **jakýkoliv typ** (také strukturovaný). Seznam může být prázdný.

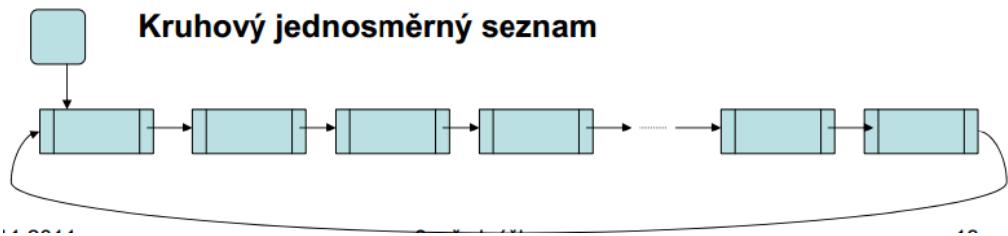
- **Ekvivalence** - Když jsou oba seznamy prázdné a nebo když se rovnají jejich první prvky a také jejich zbytky (**rekurentní definice pomocí rekurze**).
- **Jednosměrný** - Prvek ukazuje pouze na svého **následovníka**.



- **Dvojsměrný** - Prvek ukazuje na svého **následovníka** i **předchůdce**.



- **Kruhový jednosměrný** - Jednosměrný, kde **poslední prvek má za následovníka první prvek**.



- **Kruhový dvousměrný** - Dvousměrný, kde **poslední prvek má za následovníka první prvek a první prvek má za předchůdce poslední prvek**.

#### Operace nad seznamem:

- **inicializace**,
- **vložení** prvku na **začátek seznamu** (lze vložit do prázdného seznamu),
- **získání** hodnoty **prvního** prvku seznamu,
- **odstranění** prvního prvku seznamu,
- **aktivace prvního** prvku,
- **aktivace následujícího** nebo předcházejícího (u dvousměrného) prvku,
- **získání** hodnoty **aktivního** prvku,
- **změna** hodnoty **aktivního** prvku,
- **vložení** prvku **za/před aktivní** (před pouze u dvousměrného) prvek
- **test na prázdnost**,
- **test na aktivní prvek**.

## Zásobník

**Homogenní, lineární, dynamická struktura typu LIFO (Last in - First out).** Využívá se pro reverzaci pořadí, konstrukci rekurzivních programů bez rekurze - **DFS**. Implicitně používá zásobník operačního systému každý program při volání funkcí.

Lze implementovat pomocí **pole** nebo **spojového seznamu**.

Použití:

- Vyčíslování aritmetických výrazů v postfixové notaci
- Převod z infixové do postfixové notace

Operace se zásobníkem:

- **Init** - Vytvoří prázdný zásobník.
- **Push** - Vloží prvek na vrchol zásobníku.
- **Pop** - Výjme prvek z vrcholu zásobníku.
- **Top** - Přečte hodnotu z vrcholu zásobníku.
- **Empty** - True pokud je prázdný.

## Fronta

**Dynamická, homogenní a lineární struktura typu FIFO (First in - First out).** Na jedné straně se přidává (konec fronty) a na druhé se čte a odebírá – obsluhuje (začátek fronty). Využívá se např. v **systémech hromadné obsluhy**, implementaci

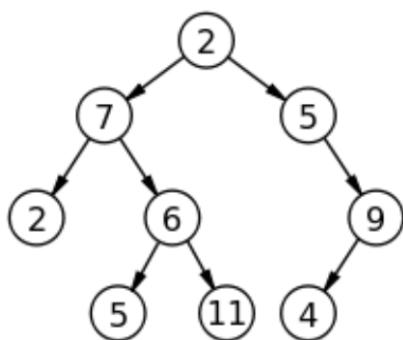
**BFS**, návrhový vzor **producent konzument**. Lze implementovat pomocí **pole** nebo **spojového seznamu**. Existují prioritní fronty, kde mohou prvky s vyšší prioritou předbíhat.

- **Init** - Vytvoří prázdnou frontu.
- **Enqueue** - Vloží prvek na konec fronty.
- **Dequeue** - Výjme prvek ze začátku.
- **Front** - Přečte hodnotu ze začátku fronty.
- **Empty** - True pokud je prázdná.

## Binární strom

**Homogenní dynamická struktura**. Rekurentní definice:

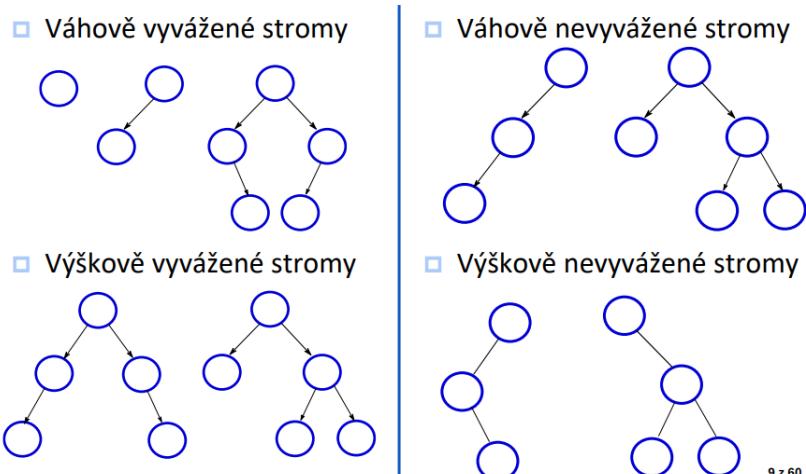
Binární strom je **bud' prázdný** nebo se sestává z **jednoho uzlu** zvaného **kořen** a **dvou podstromů** (levého a pravého) a oba podstromy mají vlastnost stromu.



- **Váhově vyvážený** - Pokud pro **všechny** jeho uzly platí, že počty uzelů jejich levého podstromu a pravého podstromu **se rovnají nebo se liší právě o 1**.
- **Výškově vyvážený** - Když pro **všechny** jeho uzly platí, že **výška levého** podstromu **se rovná výšce pravého** podstromu nebo se **liší právě o 1**.

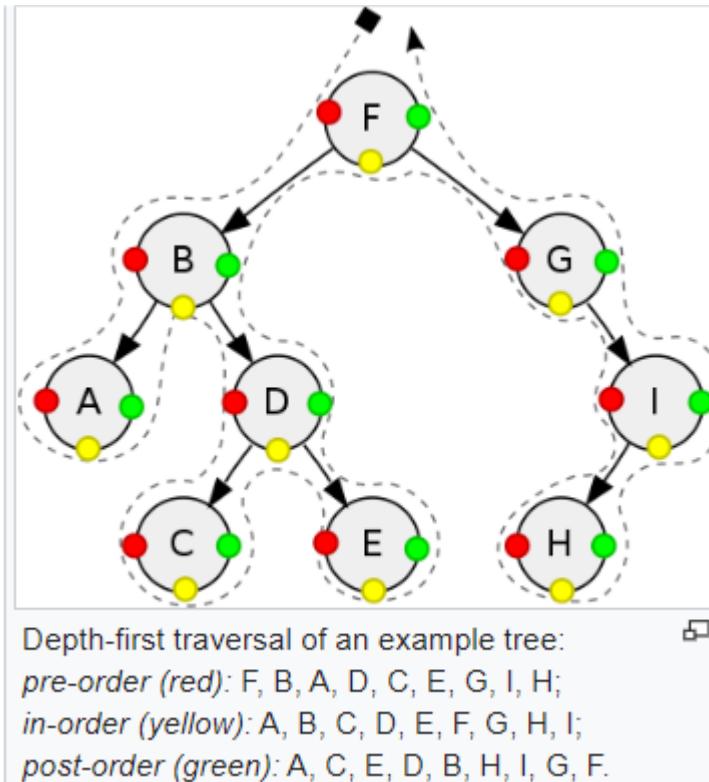
Při zajištění vyváženosti **nemůže dojít k degradaci** stromu na seznam.

Samovyvažující se stromy: **red-black tree** (červený uzel nemá červeného následníka a na každé cestě z libovolného uzlu k listu je stejný počet černých uzelů), **AVL tree** (uzly mají váhu - 0: **zcela vyvážený** uzel, -1: **výška levého** podstromu je o jedna **větší**, 1: **výška pravého** podstromu je o jedna **větší**, pokud dojde ke změně váhy na **-2/2**, je nutné situaci napravit - **rotace**).



## Průchody stromem

- **Preorder** - aktuální uzel, levý podstrom, pravý podstrom.
- **Inorder** - levý podstrom, aktuální uzel, pravý podstrom.
- **Postorder** - levý podstrom, pravý podstrom, aktuální uzel.
- **InvPreOrder** - aktuální uzel, pravý podstrom, levý podstrom.
- **InvInOrder** - pravý podstrom, aktuální uzel, levý podstrom.
- **InvPostOrder** - pravý podstrom, levý podstrom, aktuální uzel.



## Vyhledávací tabulka (Look-up table, hash table)

Homogenní, obecně dynamická struktura, ve které má každá položka zvláštní složku - **klíč**. Klíč by měl být v tabulce unikátní (s jedinečnou hodnotou), aby bylo podle něj možné provádět (ostré) vyhledávání. V ideálním případě je vyhledávání s konstantní časovou složitostí.

### Tabulka s přímým přístupem

Implementuje se pomocí pole, ve kterém jsou klíče mapovány na indexy pole. To vyžaduje vzájemně jednoznačné zobrazení (**bijekce**) mapující každý prvek množiny klíčů **K** do množiny indexů pole **H**. Jedná se o ideální strukturu pro vyhledávání, která je ale v praxi prakticky nepoužitelná (stejně velké pole jako počet klíčů je nereálné). Proto se používá mapovací (hashovací) funkce, která obvykle mapuje větší počet klíčů na menší počet hodnot. Vznikají tak **kolize** - dva různé klíče jsou namapovány do stejného místa (na stejný index). **Synonyma** jsou poté – dva nebo více klíčů, které jsou namapovány do téhož místa. Tabulky je vhodné navrhovat tak,

aby bylo jejich očekávané naplnění **70%-75% (load factor)**. Problém kolizí a synonym lze řešit:

- **Implicitní zřetězení**: adresa následníka se získá pomocí funkce z adresy předchůdce (otevřená adresace). V praxi se **umístí prvek** s klíčem mapovaným na již obsazenou pozici na **první volnou pozici**, která následuje za získaným indexem. Vyhledávání se poté provádí **sekvenčně** (případně s nějakým **krokem** - jednotkový má tendenci tvořit shluky, ten se může zvětšovat, velikost tabulky by poté měla odpovídat **prvočíslu**) od této pozice, dokud se nenarazí na požadovaný prvek nebo na prázdné místo (musí se uvažovat přechod z konce na začátek, realizováno pomocí modulo).
- **Explicitní zřetězení**: adresa následníka je obsažena v jeho předchůdci (zřetězení záznamů). Lze řešit **spojovým seznamem** na každém indexu pole, položky s klíči, které se mapují na stejný index jsou poté ukládány do těchto seznamů. Při použití nekvalitní mapovací funkce může degradovat na seznam s lineární složitostí. To lze řešit použitím **stromu místo seznamu** na každém indexu. Pak bude v nejhorším případě složitost vyhledávání logaritmická (při vyváženém stromu).

V obou případech lze tabulku v případě nutnosti zvětšit a záznamy přemapovat. Jedná se ale o složitou operaci.

### Mapovací (hashovací) funkce

Kvalitní mapovací funkce by měla splňovat tyto požadavky:

- Determinismus: Pro daný klíč vrátí vždy stejnou hodnotu.
- Rovnoměrné (uniformní) rozložení: Na každé místo se mapuje přibližně stejně velké množství klíčů.
- Využití celých vstupních dat: Zohlednění každého bitu, viz následující bod.
- Vyhnutí se kolizím podobných klíčů: V praxi bývá řada klíčů velice podobných.  
Rychlý výpočet.

Výsledek mapovací funkce musí být v rozsahu pole, do kterého jsou prvky ukládány (lze zajistit operací modulo délky pole)

### Odkazy:

- <http://dudka.cz/studyIAL>

# 30. Vyhledávání a řazení

## Řazení

- **Třídění (sorting)** položek neuspořádané množiny je **uspořádání do tříd podle** hodnoty daného **atributu** – klíče položky. **Mezi třídami nemusí být definovaná relace uspořádání** (jablka, hrušky, švestky lze třídit). Často se používá **řazení (sorting) pro třídění**.
- **Řazení (ordering, sequencing)** je **uspořádání** položek podle **relace lineárního uspořádání** nad klíči. Termín se nepoužívá často.
- **Setřídění (merging)** je vytváření souboru seřazených položek **sjednocením** několika souborů položek téhož typu, které **jsou již seřazeny**. Příkladem je algoritmus Merge sort.

### Vlastnosti řadících algoritmů

Slouží pro výběr vhodného algoritmu pro kníkrétní implementaci.

#### Přirozenost

Algoritmus se chová přirozeně pokud:

- doba potřebná k seřazení **náhodně uspořádaného pole je větší, než k seřazení již uspořádaného pole**,
- doba potřebná k seřazení **opačně seřazeného pole je větší, než doba k seřazení náhodně uspořádaného pole**.

Jinak říkáme, že se algoritmus nechová přirozeně.

#### Stabilita

Stabilita vyjadřuje, zda mechanismus algoritmu zachovává **relativní pořadí klíčů se stejnou hodnotou**. Sekvence 7,5',3,1,5'',9,2,5''',8,4,6 pro následující příklad.

- **stabilní** algoritmus: 1,2,3,4,**5',5'',5'''**,6,7,8,9.
- **nestabilní** algoritmus: 1,2,3,4,**5'',5',5'''**,6,7,8,9 (nebo jinak).

## Algoritmy řazení

Podle principu řazení je dělíme na:

- Princip **výběru (selection)** – přesouvají **maximum/minimum** do výstupní posloupnosti. Např.: **Selection sort**, **Bubble sort** a jeho modifikace (**Ripple sort**, **Shaker sort**, **Shuttle sort**)
- Princip **vkládání (insertion)** – vkládají postupně prvky **do seřazené** výstupní posloupnosti. Např.: **Insertion sort**, ...

- Princip **rozdělování (partition)** – rozdělují postupně množinu prvků na dvě podmnožiny tak, že prvky jedné jsou menší než prvky druhé. Např.: **Quick sort**, **Shell sort**, ...
- Princip **slučování (merging)** – setřídíují se postupně dvě seřazené posloupnosti do jedné. Např.: **Merge sort**, ...
- **Jiné principy.** Např.: **řazení tříděním**, ...

Podle typu procesoru:

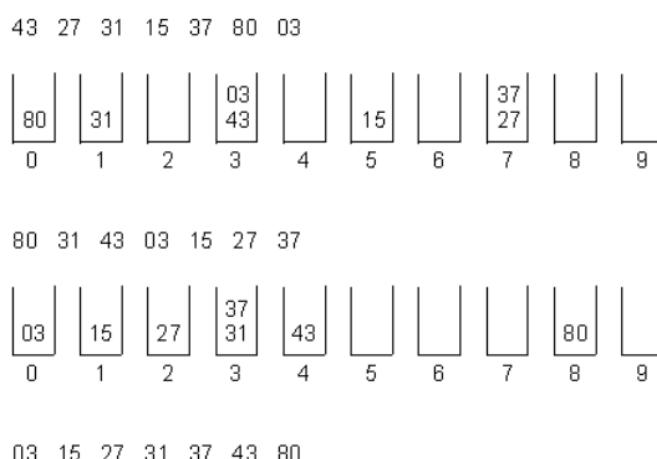
- **sériové** (jeden procesor) – jedna operace v daném okamžiku,
- **paralelní** (více procesorů) – více souběžných operací

Podle přístupu k paměti:

- **přímý (náhodný)** přístup – metody vnitřního řazení (řazení polí)
- **sekvenční** přístup – metody vnějšího řazení (řazení souborů a seznamů)

## Řazení tříděním - Radix sort

Řazení tříděním musí probíhat od **nejnižší po nejvyšší** prioritu (**od LSB po MSB**, od jednotek přes desítky, stovky, tisíce, ... tj. podle základu). Jedná se o **třídění pomocí příhrádek** (u desítkového základu jich je 10 a třídění provádíme pro každou číslici klíče). Využití pro řazení děrných štítků.



Příklad **správného** (vlevo) a **špatného** (vpravo) postupu řazení tříděním na množině {342, 835, 942, 178, 256, 493, 884, 635, 728}:

342	728	178
942	835	256
493	635	342
884	342	493
835	942	635
635	256	728
256	178	835
178	884	884
728	493	942



178	728	342
256	635	942
342	835	493
493	342	884
635	942	635
728	256	835
835	178	256
884	884	728
942	493	178



Vlastnosti:

- **stabilní**,
- **nechová se přirozeně**,
- **nepracuje in situ**,
- **linearitmická  $O(n \cdot \log n)$**  časová složitost.

## Selection sort

Princip metody je postaven na **nalezení extrémního prvku** v zadaném segmentu pole a jeho **výměna na konec (začátek) seřazené části pole**. Příklad řazení od největšího k nejmenšímu, viz <https://www.algoritmy.net/article/4/Selection-sort>:

1. (3 2 8 7 6) - zadané pole,
2. (3 2 8 7 6) - nejvyšší číslo je 8, prohoďme ho tedy s číslem 3 na indexu **0**,
3. (8 2 3 7 6) - nejvyšší číslo je 7, prohoďme ho tedy s číslem 2 na indexu **1**,
4. (8 7 3 2 6) - nejvyšší číslo je 6, prohoďme ho tedy s číslem 3 na indexu **2**,
5. (8 7 6 2 3) - nejvyšší číslo je 3, prohoďme ho tedy s číslem 2 na indexu **3**,
6. (8 7 6 3 2) - seřazeno

Vlastnosti:

- **nestabilní** - z důvodu výměny prvků,
- měla by být **přirozená**,
- složitost: **kvadratická  $O(n^2)$** ,
- pracuje **in situ**.

## Bubble sort

Stejně jako selection sort pracuje metoda na principu **nalezení extrémního prvku** a jeho **umístění na konec (začátek) již seřazené části**. Liší se ale v principu **nalezení extrému výměny** prvků, viz <https://www.algoritmy.net/article/3/Bubble-sort>.

Postup:

- Porovnává se každá dvojice a v případě **obráceného uspořádání** (záleží, jestli řadíme od největšího nebo od nejmenšího) **se přehodí**.
- Při pohybu **zleva doprava a řazení od nejmenšího k největšímu** se tak **maximum dostane na poslední pozici. Minimum se posune o jedno místo směrem ke své konečné pozici**.

Vlastnosti:

- **stabilní** - na rozdíl od selection sort,
- **přirozená** - nejrychlejší metoda pro již seřazené pole,
- složitost: **kvadratická  $O(n^2)$** ,
- pracuje **in situ**.

Další **odvozené metody** od Bubble sort (Ripple sort, Shaker sort, Shuttle sort) jsou obecně rychlejší ale né z pohledu **časové složitosti**, ta zůstává  **$O(n^2)$** .

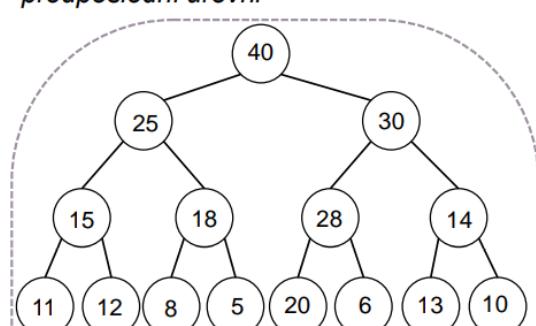
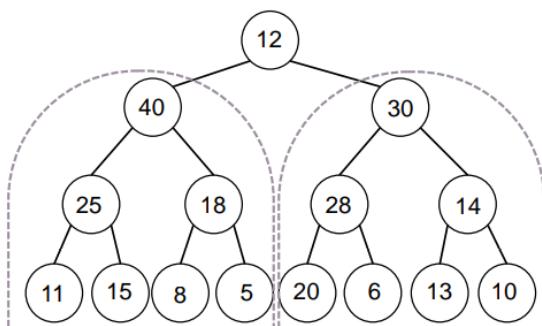
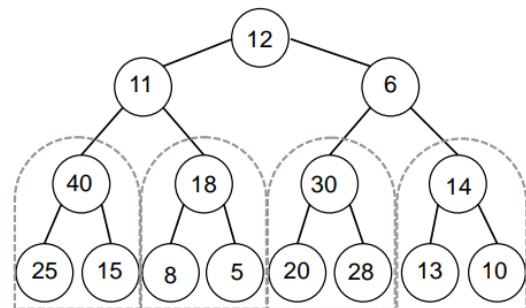
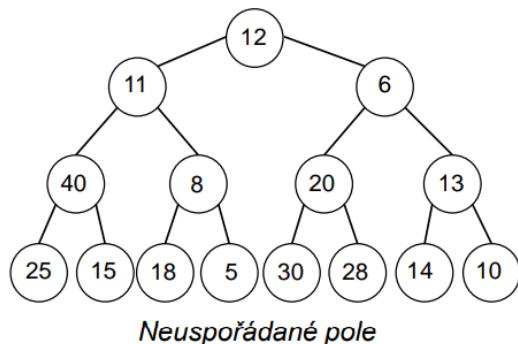
## Heap sort

Princip algoritmu je založený na **hromadě**. Hromada je struktura **stromového typu**, pro niž platí, že mezi **otcovským** uzlem a **všemi jeho synovskými** (to platí i v **libovolných podstromech**) uzly platí **stejná relace uspořádání** (buď je menší, nebo větší). Nejčastěji se používá **binární hromada**, která je založena na **binárním stromu**, který je konstruován tak, aby:

- všechny hladiny kromě poslední **byly plně obsazene**,
- **poslední hladina je zaplněna zleva**.

Binární hromadu lze také použít např. k implementaci prioritní fronty (pokud neexistují prvky se stejnou prioritou a stačí nám přístup k nejprioritnějšímu prvku).

### Vytvoření hromady



46 z

### Rekonstrukce hromady

Rekonstrukce **hromady** se realizuje pomocí **prosetí/zatřesení** (sift), která **znovuustanoví** hromadu **porušenou pouze v kořeni** Heap sort in 4 minutes :

- na místo kořenového uzlu přesuneme **nejnižší a nejpravější uzel**,
- zatřeseme s hromadou, což způsobí **propadnutí se prvku v kořeni** na místo, kam patří a **vypropagování největšího prvku** do kořene. Děje se tak **porovnáním s levým a pravým synem** a výměnou za toho **většího/menšího** (podle toho, jak řadíme).

## Princip algoritmu

1. **Vytvoření hromady** (lze uchovávat v poli),
2. **Odebrání kořene** a umístění jej na konec pole (a zmenšení hromady, uspořádanou část pole již ignorujeme).
3. Pokud není hromada prázdná, **prosetí/zatřesení** s hromadou a pokračování s bodem 2, jinak máme seřazeno.

Vlastnosti:

- **nestabilní**,
- **nechová se přirozeně**,
- **in situ**,
- **linearitmická složitost  $O(n \cdot \log n)$** .

## Insertion sort

Pracuje na **principu řazení karet** v ruce, vyberu neseřazenou a vložím ji na místo, kam patří. Princip algoritmu:

1. pole dělíme na **seřazenou část** (levou) a **neseřazenou část** (pravou), na začátku tvoří seřazenou část jeden prvek.
2. Z neseřazené části **vyber první prvek**.
3. V seřazené části **najdi jeho umístění** (místo, kde jeho pravý prvek je větší/menší a jeho levý prvek je menší/větší, případně jsou stejné)
4. **posuň prvky** na pravo od nalezené pozice k pozici umísťovaného prvku **o 1**.
5. Umísti tento prvek.
6. Pokračuj, dokud není neseřazená posloupnost prázdná.

Vlastnosti:

- **stabilní**,
- **přirozený**,
- **in situ**,
- **kvadratická složitost  $O(n^2)$** .

## Bubble-insert sort

Modifikace, u které je v průběhu porovnávání prováděn přesun položek doprava výměnou za porovnávaný prvek.

## Binary-insert sort

Modifikace, u které je vyhledávání prováděno binárním způsobem (půlení intervalů). Pro zajištění stability metody musí binární vyhledávání při více stejných hodnotách vybrat místo za nejpravějším výskytem (Dijkstrova metoda).

<https://www.algoritmy.net/article/8/Insertion-sort>

## Quick sort

Algoritmus fungující na principu **rozděl a panuj**. Jedná se o jeden z **nejrychlejších** algoritmů pro řazení (jeho rychlosť je ale nedeterministická). Princip algoritmu:

1. rozděl množinu prvků na dvě podmnožiny tak, aby:
  - a. první obsahovala prvky **menší/větší** nebo rovny **mediánu**,
  - b. druhá obsahovala prvky **větší/menší** nebo rovny **mediánu**.
2. Takto získané podmnožiny opět rozděl na další podmnožiny a obdobně uspořádek prvky.
3. Skonči, pokud množina obsahuje pouze 1 prvek.

Přesun prvků podle mediánu je realizován následovně:

- Procházíme pole současně zleva a zprava.
- Zleva hledáme prvek větší nebo roven mediánu, zprava prvek menší nebo roven mediánu.
- Nalezené prvky vyměníme a hledáme další prvky pro výměnu.
- Proces ukončíme až se dvojice indexů překříží

**Medián** je prvek, který dělí množinu na dvě části tak, že platí:

- nejméně 50 % hodnot je menších nebo rovných mediánu,
- nejméně 50 % hodnot je větších nebo rovných mediánu.

Samotné vyhledání mediánu je časově náročné, takže se používá **pseudomedián**, což může být **náhodná hodnota** z množiny nebo hodnota, která je ve **středu** množiny na **indexu ( $\text{left}+\text{right})/2$** , případně lze vybrat medián ze tří prvků (na začátku, konci a uprostřed, či jinak).

Vlastnosti:

- **nestabilní**,
- **nepracuje přirozeně**,
- **linearitická** časová složitost - v **nejhorším** případě je složitost **kvadratická**,
- pracuje **in situ**, ALE **potřebuje zásobník** pro ukládání hranice ještě nezpracovaných částí. Důležitý trik pro zmenšení zásobníku je **na zásobník uložit (pouze) tu větší část** a dále, tedy dřív, **zpracovávat vždy menší** z rozdelených částí. To zaručí, že stačí zásobník velikosti  **$\log_2(n)$** , tj. například pro miliardu prvků stačí hloubka 30 položek. Bez této optimalizace by byla paměť potřebná pro zásobník **až lineární**, takhle je logaritmická  **$O(\log(n))$** .

<https://www.algoritmy.net/article/10/Quicksort>

## Shell sort

Algoritmus fungující na principu **rozděl a panuj**. Vytváří sekvence prvků, které seřazuje bublinovým průchodem. Princip:

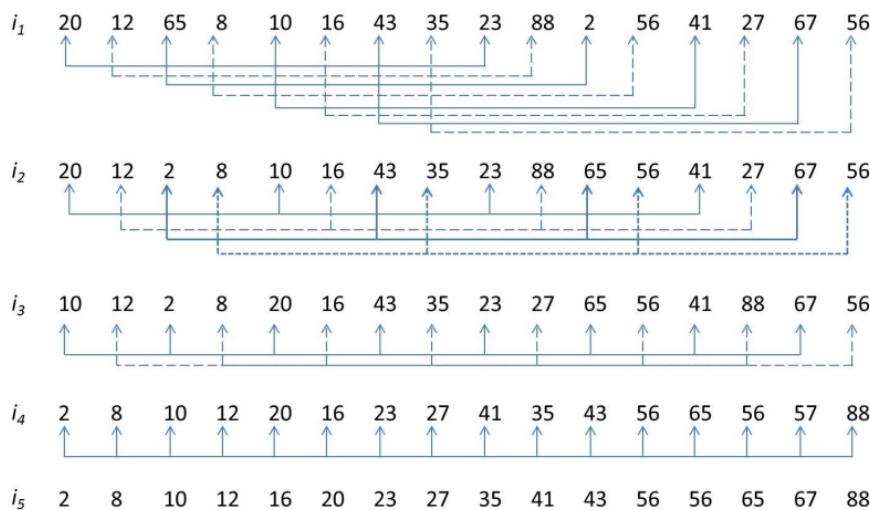
1. Rozděl vstupní pole na posloupnosti o **délce 2** (prvky musí být co nejdále od sebe),
2. Každou posloupnost seřaď pomocí pomoci bubble sort,
3. Rozděl vstupní pole na posloupnosti o **délce 4** (prvky musí být co nejdále od sebe),
4. ...
5. Rozděl vstupní pole na posloupnost o **délce pole** a seřaď je pomocí bubble sort.

nejlepší délky posloupností - 1, 4, 10, 23, 57, 132, 301, 701, 1750.

Vlastnosti:

- **nestabilní**,
- měla by být **přirozená**,
- pracuje **in situ**,
- časová složitost je v nejhorším případě **kvadratická**, ale vybráním vhodné řady lze snížit na  $O(n^{3/2})$  nebo  $O(n^*(\log n)^2)$ .

<https://www.algoritmy.net/article/154/Shell-sort>



## Merge sort

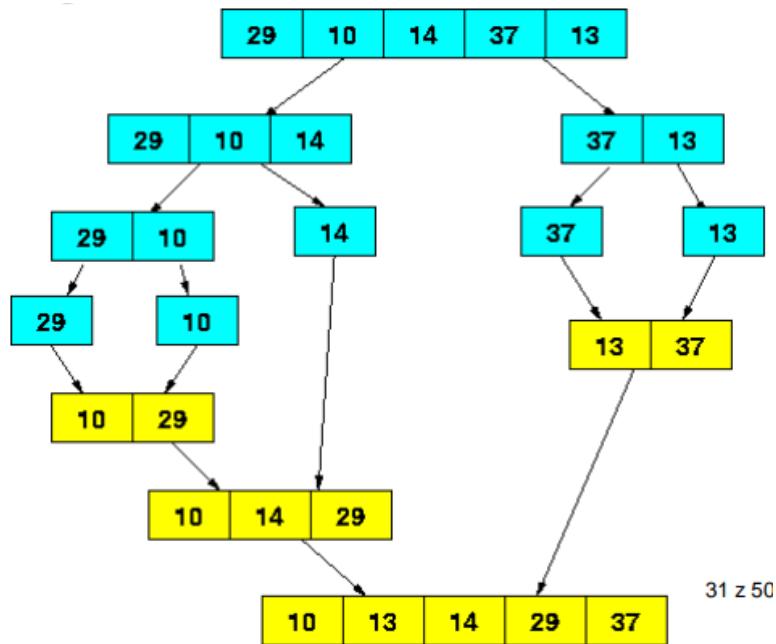
Další z algoritmů typu **rozděl a panuj**. Princip:

1. Postupně **půlíme** pole (v případě lichého počtu má jedna část o prvek navíc) dokud není o velikosti 1.
2. Poté **spojujeme** vždy dvě sousední pole tak, aby **vzniklo jedno seřazené** pole. Toto seřazování je jednoduché, protože již **spojujeme seřazená pole**, tudíž máme dva ukazatele do těchto polí a vždy vybíráme prvek, který je mezi polí **nejmenší/největší**.

Vlastnosti:

- **stabilní**,
- spíš není přirozená, vždy se musí dělit a poté setříďovat,
- **in situ**, ale prostorová složitost je **logaritmická**  $O(\log(n))$  pro uchovávání hranic polí po dělení - lze použít zásobník, ale jednodušší je použití **rekurze**,
- **linearitmická**  $O(n^*\log n)$  časová složitost

<https://www.algoritmy.net/article/13/Merge-sort>



## Shrnutí

Název		Časová složitost			Dodatečná paměť	Stabilní	Přirozená	Metoda
Název	Znám jako	Minimum	Průměrné	Maximum				
bublinkové řazení	bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	záměna
řazení haldou	heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	ne	ne	halda, záměna
řazení vkládáním	insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	vkládání
řazení slučováním	merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	ano	ano	slučování
rychlé řazení	quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	ne	ne	záměna
řazení výběrem	selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	zprav. ne	ne	výběr
Shellovo řazení	shell sort	$O(n^{1 + \frac{c}{\sqrt{m}}})$ [10]		$O(n \log^2 n)$ [10]	$O(1)$	ne	ano	vkládání
comb sort	(hřebenové řazení)	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(1)$	ne	ano	záměna
introspektivní třídění	Introsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	ne	?	záměna, výběr

## Řazení dle více klíčů

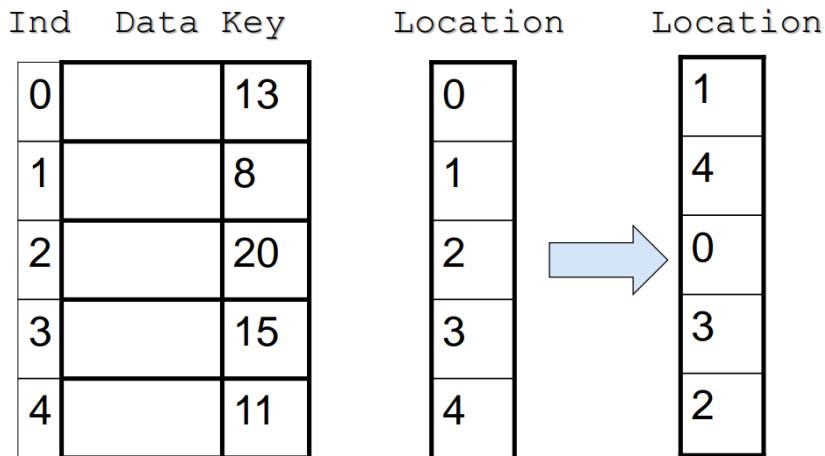
Lze řešit opakováním řazením, které se podobá se radixovému (přihrádkovému) řazení např. pro den, měsíc a rok. Musí platit:

- řadíme **od klíče s nejmenší prioritou po klíč s největší prioritou**,
- nutné použít **stabilní řadící metodu**.

Druhou možností je provádět řazené **pouze jednou** ale současně **porovnávat všechny klíče** počínaje od toho s největší prioritou. Z čehož plyne třetí možnost, a to sloučit klíče do jednoho (aglomerovaný klíč). U příkladu se dny, měsíci a roky se jedná o rodné číslo bez posledních 4 znaků - RRMMDD (pozor ženy mají MM zvýšené o 50)

## Řazení bez přesunu položek

Vhodné v případech, kdy řadíme **velké objekty** a jejich **přesuny v paměti** jsou **drahé**. K tomu, abychom položky museli přesunovat, používáme **pomocné pole - pořadník**. Po dokončení řazení **pořadník udává**, v jakém **pořadí** by měly být seřazeny **položky původního pole**. Tj. na první pozici pořadníku je index prvního prvku seřazeného pole atd.



Pole seřazené pomocí **pořadníku** lze také **zřetězit**, v jednotlivých prvcích musí být **vyhrazená paměť** pro **index následujícího prvku**. Zřetězené prvky pak lze **procházet sekvenčně principem spojových seznamů**, nebo je lze **prevést do seřazeného pole** (to je možné i bez zřetězení podle pořadníku - MacLarenův algoritmus)

## Vyhledávání

Vyhledávání lze realizovat jako:

- sekvenční vyhledávání v **neseřazeném poli**,
- sekvenční vyhledávání v **neseřazeném poli se zarázkou**,
- sekvenční vyhledávání v **seřazeném poli**,
- sekvenční vyhledávání v poli **seřazeném podle pravděpodobnosti vyhledání** klíče (nejpravděpodobnější klíče jsou na začátku pole),
- sekvenční vyhledávání v poli s **adaptivním uspořádáním podle četnosti vyhledání** (prvky se s četností vyhledávání přesouvají k začátku pole),
- **binární vyhledávání v seřazeném poli**,
- **binární vyhledávání pomocí stromů**,
- vyhledávání pomocí **stromů s více klíči ve vrcholech**,
- vyhledávání pomocí tabulky s rozptýlenými položkami (hash table).

Při vyhledávání sledujeme doby vyhledání (**kritéria**) při **úspěšném/neúspěšném** vyhledávání.

- **minimální**,
- **maximální**,
- **průměrná/střední**.

## Sekvenční vyhledávání

Prvky pole se prochází jeden po druhém.

### V neseřazeném poli

Nejrychleji jsou nalezeny položky na počátku vyhledávání. Jednoduchá implementace (**procházení polem a test na hodnotu**).

- **minimální čas úspěšného vyhledání = 1**
- **maximální čas úspěšného vyhledání = N**
- **průměrný čas úspěšného vyhledání = N/2**
- **Čas neúspěšného vyhledání = N**

### V neseřazeném poli se zarážkou

Na konci struktury je "**zarážka**", což je **hledaný klíč**, tedy hodnota je vždy nalezena. Toto přidává **urychljení** v tom, že není potřeba po každém porovnání klíčů **testovat konec struktury**, pouze se otestuje na konci vyhledávání jestli se **nalezla zarážka nebo hledaná položka**.

### V seřazeném poli

Jakmile se narazí na hodnotu klíče, která je **větší než hledaný klíč**, vyhledávání se ukončuje jako **neúspěšné**. Tato metoda se **urychlí pouze pro neúspěšné hledání**. Vyhledávání v seřazeném poli lze ale urychlit např. **půlením intervalů**, viz dále. Funguje pouze pro klíče, u kterých **existuje relace uspořádání**. Problematické jsou operace vkládání a odebírání prvků, je nutné pole **znovu seřadit**, respektive **posunout segment pole v paměti**. Odebírání lze řešit **zaslepěním** (označením indexu za nevyužitý, respektive nastavení jeho hodnoty na hodnotu, která nebude hledána).

### Podle četnosti přístupu

Nejčastěji hledané položky **jsou na začátku struktury - pole/seznamu** (používá se k tomu **počítadlo vyhledávání** a jednou za čas je pole podle počítadla seřazeno). Druhou variantou je, že po nalezení položky se **nalezená položka prohodí se svým levým sousedem** (není potřeba počítadlo, nejčastěji hledané položky se tak přesouvají k začátku struktury) - **adaptivní rekonfigurace podle četnosti vyhledávání**. Postupy lze využít i u prvků, které mezi sebou nemají definovanou relaci uspořádání.

### Podle pravděpodobnosti

Jedná se o podobný případ viz sekce výše, kde pravděpodobnosti vyhledávání jsou známé předem a podle toho je pole seřazené (poté se již uspořádání nemění).

## Nesekvenční vyhledávání

Přístup k hledaným položkám může být náhodný.

### Binární vyhledávání v poli (seřazeném)

Provádí se nad **seřazenou množinou klíčů** s náhodným přístupem (pole). Metoda připomíná metodu **půlení intervalů**. Složitost je při nejhorším **logaritmická O(log n)**. Pole **půlíme** a podle **prostřední hodnoty** prohledáváme **levou nebo pravou část**. Opět je zde stejný problém s **vkládáním a odebíráním prvků** z pole.

### Dijkstrova varianta binárního vyhledávání

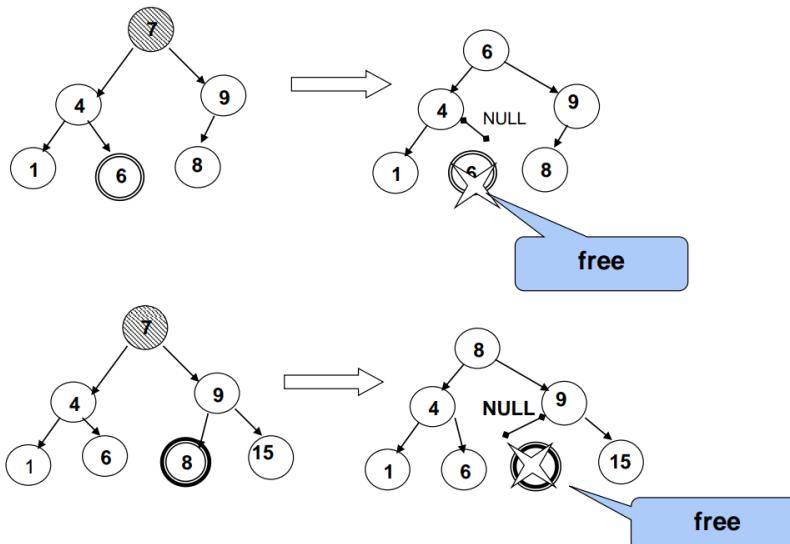
Vychází z předpokladu, že v poli může být **více položek se shodným klíčem**. Hledá se tedy **nejpravější nebo nejlevější klíč** (ostatní pak lze získat jednoduše sekvenčním průchodem od nalezeného indexu). Příklady pro vyhledávání nepravějšího výskytu:

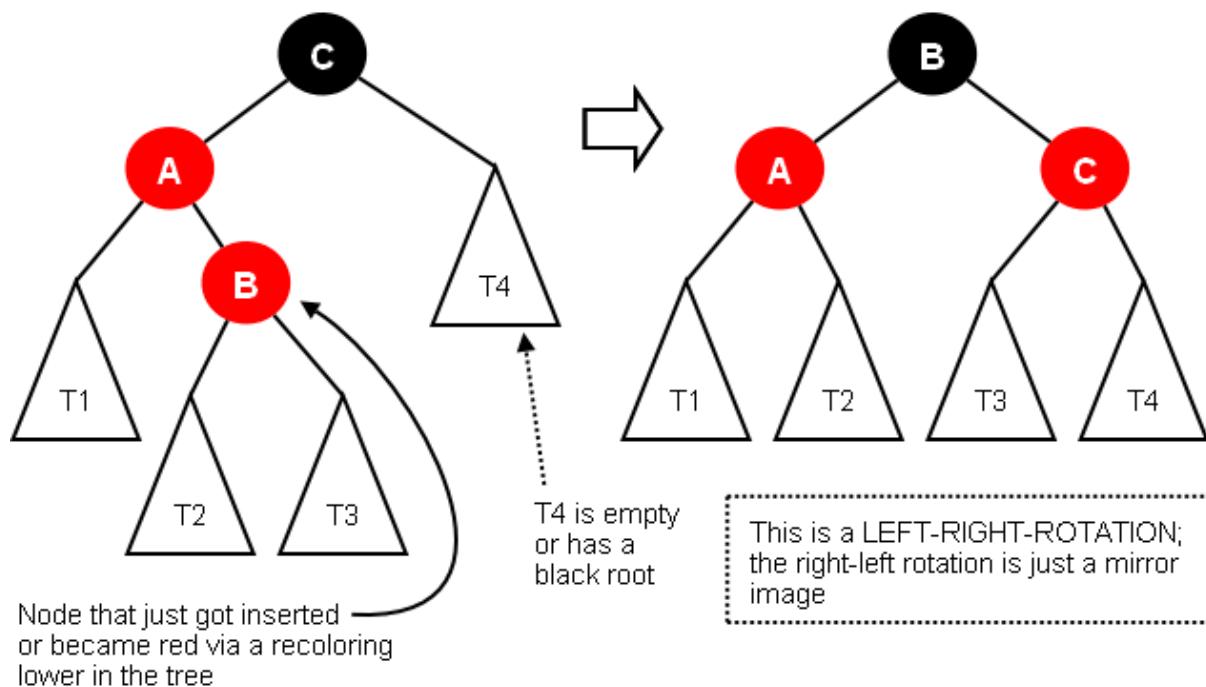
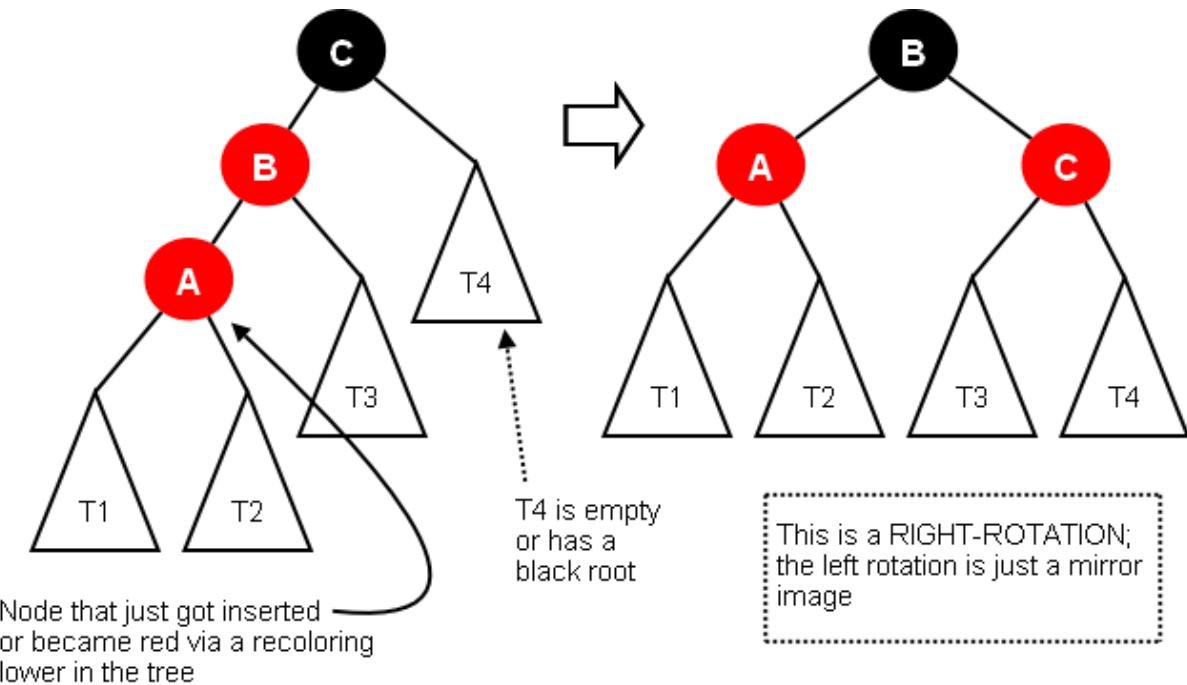
- V poli: 1,2,3,4,5,5,6,6,6,8,9,13 najde algoritmus klíč **K=6** na pozici **8** (počítáno od 0).
- V poli: 1,1,1,1,1,1,1,1,1,1,2 najde algoritmus klíč **K=1** na pozici **9**.

### Binární vyhledávání v binárním vyhledávacím stromu (binary search tree)

Podobné binárnímu vyhledávání v seřazeném poli. Je-li **vyhledaný klíč roven kořeni**, vyhledávání končí **úspěšně**. Je-li **klíč menší než klíč kořene**, pokračuje vyhledávání v **levém podstromu**, je-li **větší**, pokračuje v **pravém podstromu**. Vyhledávání končí **neúspěšně**, je-li **prohledávaný (pod)strom prázdný**. Výhodou uspořádání do BVS je **snazší vkládání a mazání prvků**. Při vkládání uzlu je uzel vložen na listovou úroveň (pokud se jedná o samovyvažující se strom může být nutné **provést rotace**, aby **bylo dodrženo vyvážení**, viz obrázek). U mazání je nutné dodržet uspořádání stromu a pokud je mazán vnitřní uzel, musí být nahrazen některým uzel z listové úrovně takto:

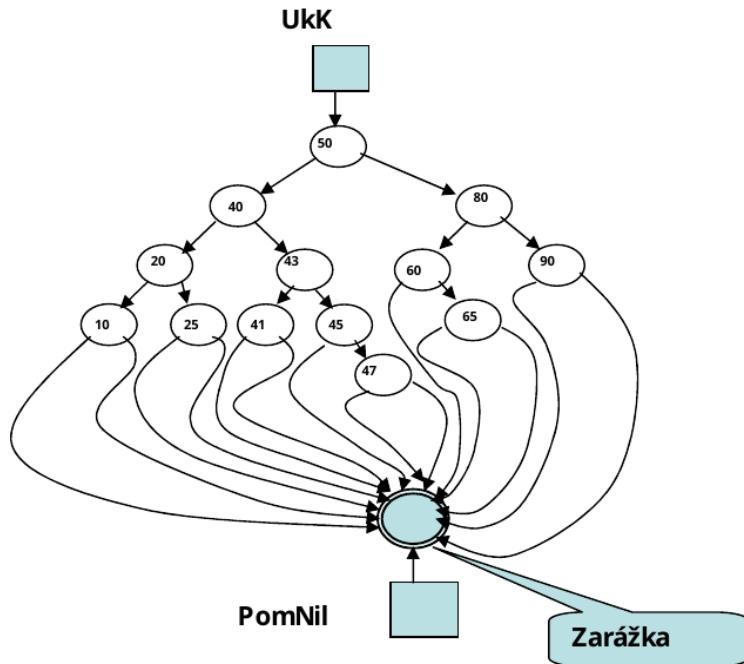
- **nejpravější uzel levého podstromu** rušeného uzlu (**maximum v levém podstromu**),
- **nejlevější uzel pravého podstromu** rušeného uzlu (**minimum v pravém podstromu**).





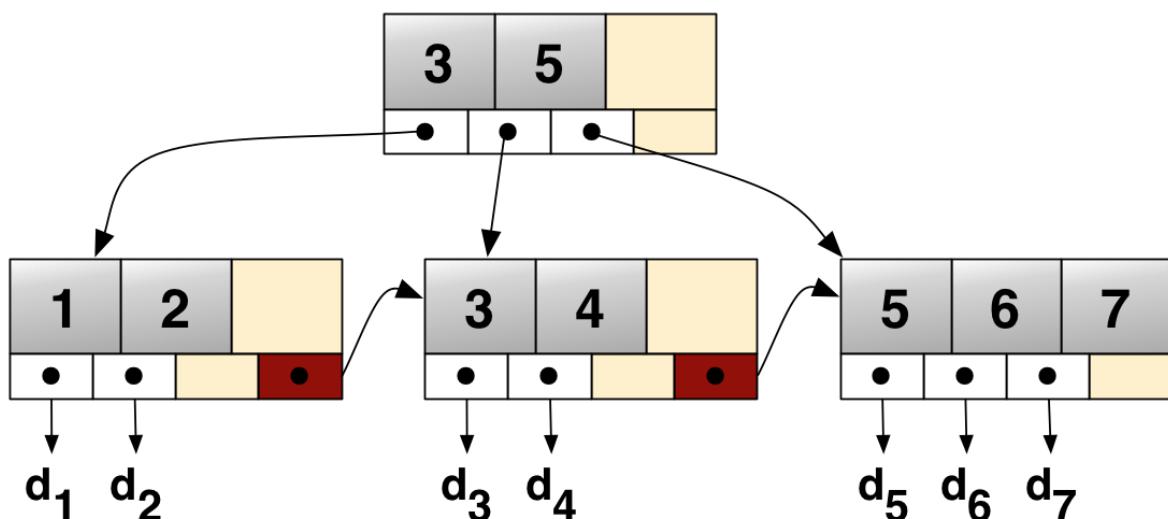
## BVS se zarážkou

Funguje podobně jako u pole, všechny listové uzly ukazují na zarážku. Na konci vyhledávání musí být provedený test, jestli se vyhledala hledaná hodnota nebo zarážka.



### Vyhledávání pomocí stromů s více klíči ve vrcholech

Vnitřní (interní) vrcholy obsahují libovolný nenulový počet klíčů. Pokud ve vrcholu (uzlu) leží  $k$  klíčů, pak má  $k+1$  synů ( $s_0, \dots, s_k$ ). Hledání cesty v uzlu se provádí obvykle sekvenčně/binárně (už nelze rozlišit pouze pravý/levý podstrom). **Snižuje se hloubka** stromu ale **zvyšuje se náročnost hledání cesty**. Vnitřní uzly nemusí nést data a slouží pouze pro vyhledávání, data jsou uložena až na listové úrovni (**B+ stromy**, které se využívají v DB, file systémech, ...). Počet klíčů ve vrcholech se většinou může pohybovat od do určitého počtu - např. **(a,b) stromy** (kořen má **2** až **b** synů, ostatní vnitřní vrcholy **a** až **b** synů,  $a \geq 2$ ,  $b \geq 2a-1$ ). Při **vkládání** se uzly **štěpí** (je-li potřeba), při **mazání** se **slučují** (je-li potřeba). Slovník lze například implementovat pomocí **písmenkového stromu** (trie), kde se každý uzel větví dle počtu písmen v abecedě, což umožňuje sdílet části stromů mezi více slov a vyhledání je velmi rychlé.



## Vyhledávání v tabulkách s rozptýlenými položkami (TRP - hash table)

Základem je princip **tabulky s přímým přístupem**. Využívá se zde mapovací funkce, která **rovnoměrně mapuje klíče na indexy** v paměti (polička tabulky). Což v ideálním případě bez kolizí znamená **konstantní  $O(1)$**  časovou složitost vyhledávání. Problém je vznik kolizí (klíče, které jsou mapovány na stejný index), poté musí být vyhledávání dokončeno sekvenčně. V **extrémním případě** tak může být rychlosť vyhledávání **lineární  $O(n)$** . Vyhledávání v TRP má **index-sekvenční** charakter.

- **explicitní** řetězení synonym: spojový seznam, binární vyhledávací strom,
- **implicitní** řetězení synonym: ukládání synonym na první/další volné místo (dle kroku) v tabulce (poli).

## Vyhledávání v textu

Hledaným klíčem je řetězec (několik znaků) textu.

### Naivní (Brute-force) algoritmus

Algoritmus **porovnává** symboly textu a vzorku **zleva doprava**, při **neshodě** symbolů **posune** vzorek o jednu pozici doprava.

- složitost:  $O(m \cdot n)$ , kde **m** je počet znaků textu a **n** je počet znaků vzorku, jedná se ale o **nejhorší případ** a většinou je porovnání daleko rychlejší.

### Knuth-Morris-Prattův algoritmus

Algoritmus využívá princip **konečného automatu**. Porovnávání textu stále probíhá **zleva doprava**. Při neshodě se nevrací v textu zpět (posun o 1 v předchozím algoritmu), ale snaží se **posunout vzorek** tak, aby symbol textu, na kterém došlo k neshodě, porovnal s **jiným symbolem vzorku**, který předchází symbolu s neshodou (pokud žádný takový neexistuje, posouvá vzorek na začátek).

**Text:** clanekokokosu

**Aktuální přiložení vzorku:** kokos

**Další možné přiložení:** kokos

- složitost  $O(m)$  pro konstrukci automatu,  $O(n)$  pro porovnání, celkově  $O(m+n)$ .

### Boyer-Mooreův algoritmus

Přikládá vzorek k textu **zleva doprava**, ALE porovnává **zprava doleva**, což umožňuje provádět větší skoky (některé **symboly textu nemusí být vůbec porovnány** se symboly vzorku). Pokud se při porovnávání narazí na **symbol**, který se ve vzorku **nenachází**, lze vzorek **posunout o jeho délku**, **jinak** je nutné vzorek **posunout** tak, aby byl **přiložen** na další **nejpravější výskyt** daného symbolu. Dále lze přidat **posun na základě podřetězce**.

## Rabin-Karpův algoritmus

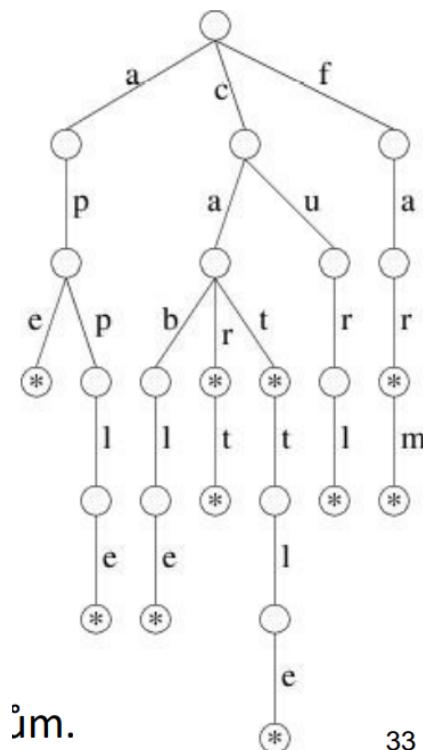
Vyhledávání vzorku založené na hashování. Postup:

- Posouváme okénko délky **m** (délky vzorku) po textu a **počítáme hash** pro danou část textu.
- Je-li **hash shodný s hashem vzorku**, porovnáme danou část textu se vzorkem **znak po znaku**.

Aby tato metoda byla efektivní, musíme umět **rychle počítat hash** (v konstantním čase). Lze zařídit tak, že který **opouštíme** od hashe určitým způsobem **odečteme** a znak, na který **najízdíme** určitým způsobem **přičteme** (Ordinální hodnota asi nebude úplně efektivní ale bude taky fungovat, většinou jsou ale znaky násobeny nějakým **polynomem**).

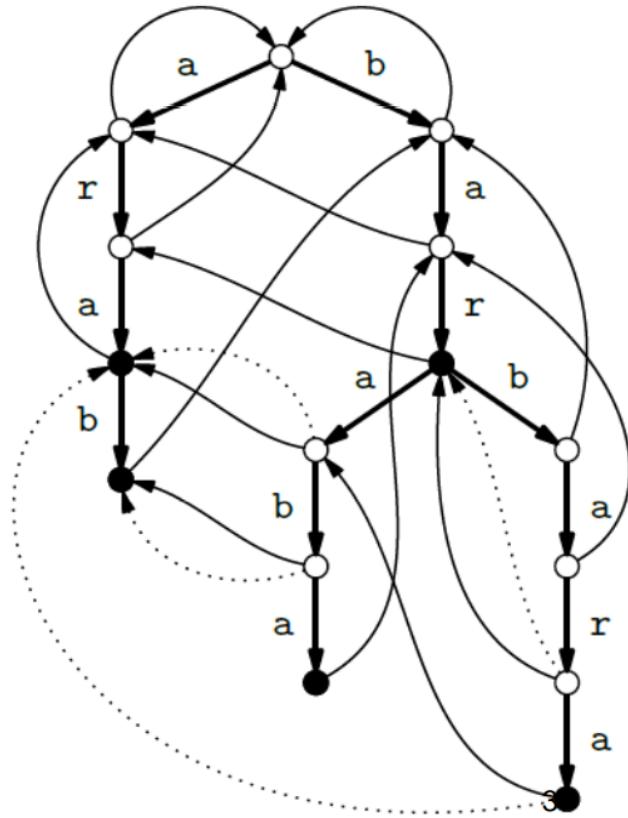
## Písmenkové stromy - hledání více vzorků

Umožňují současné vyhledávání více vzorků v textu. Na jedné cestě od kořene se může nacházet více vzorků (na obrázku jsou konce slov označeny hvězdičkou).



## Algoritmus Aho-Corasicková

- Postupujeme automatem po dopředných hranách, pokud můžeme.
- Nelze-li použít žádnou dopřednou hranu, vracíme se po zpětných hranách.
- Pokud se dostaneme zpět až do kořene a ani zde nelze jít s daným symbolem žádnou dopřednou hranou, symbol je zahozen.
- V každém stavu zkонтrolujeme, zda neodpovídá konci slova. Pokud ano, ohlášíme výskyt. Z každého stavu pomocí zkratek nalezneme také všechny sufixy, které jsou také slovem a ohlášíme.



### odkazy

- Animace řazení:  
<https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

# 31. Pravděpodobnost a statistika (základní pojmy, náhodná veličina a vektor, rozdělení pravděpodobnosti, generování pseudonáhodných čísel, bodové a intervalové odhady parametrů, testování hypotéz, regresní a korelační analýza).

## Základní pojmy

$\Omega$  - množina všech hodnot kterých může **náhodná veličina X** nabývat - **základní prostor**. Základní prostor pro dva po sobě následující hody vypadá následovně  $\Omega = \{ (\text{rub}, \text{rub}), (\text{líc}, \text{líc}), (\text{rub}, \text{líc}), (\text{líc}, \text{rub}) \}$

## Náhodný jev

libovolná podmnožina  $\Omega$

- jev nemožný:  $\emptyset$  za daných podmínek jev **nemůže nastat** (na dvou tradičních (šestibokých) hracích kostkách nám padne součet 30),
- jev jistý:  $\Omega$  - za daných podmínek jev **nastane vždy** (při hodu mincí padne rub nebo líc).

## Průnik jevů

Průnik jevů **A** a **B** je jev, který nastane právě tehdy, když **nastanou** jevy **A** a **B současně**. Značíme jej  $\mathbf{A} \cap \mathbf{B}$ . Pokud je  $\mathbf{A} \cap \mathbf{B} = \emptyset$ , mluvíme o jevech **disjunktních (neslučitelných)**

## Sjednocení jevů

Sjednocení jevů **A** a **B** je jev, který nastane právě tehdy, když **nastane alespoň jeden** z jevů A a B. Značíme jej  $\mathbf{A} \cup \mathbf{B}$ .

## Opačný jev

Opačný jev (doplňek) k jevu **A** je jev, který nastane právě tehdy, když **nenastane jev A**. Značíme **A'** a platí  $\mathbf{A}' = \Omega \setminus \mathbf{A}$ .

## Úplný systém jevů

Jevy  $\mathbf{A1}, \mathbf{A2}, \dots$  tvoří úplný systém jevů, jestliže  $\mathbf{A1} \cup \mathbf{A2} \cup \dots = \Omega$ . Pokud navíc platí  $A_i \cap A_j = \emptyset, \forall i \neq j$ , jedná se o **úplný systém neslučitelných jevů**.

## Axiomatická definice pravděpodobnosti

Nechť  $(\Omega, \mathcal{A})$  je jevové pole a  $P$  je množinová funkce definovaná na  $\mathcal{A}$  s vlastnostmi:

- 1)  $P(\Omega) = 1$ ,
- 2)  $P(A) \geq 0 \forall A \in \mathcal{A}$ ,
- 3) jestliže  $A_k \in \mathcal{A}, k = 1, 2, \dots$ , jsou navzájem disjunktní jevy ( $A_i \cap A_j = \emptyset$  pro  $i \neq j$ ), pak

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i).$$

Funkci  $P$  nazveme **pravděpodobností** a trojici  $(\Omega, \mathcal{A}, P)$  nazveme **pravděpodobnostním prostorem**.

## Klasická pravděpodobnost

Klasická pravděpodobnost je definována jako **podíl počtu příznivých jevů ku počtu všech možných jevů**.  $P(A) = |A|/|\Omega|$ , kde:

- $|A|$  značí počet prvků množiny příznivých jevů,
- $|\Omega|$  značí počet prvků všech možných jevů - velikost základního prostoru.

## Vlastnosti pravděpodobnosti

Nechť  $(\Omega, \mathcal{A}, P)$  je pravděpodobnostní prostor. Pak pravděpodobnost  $P$  má následující vlastnosti:

- (1)  $P(\emptyset) = 0$ ,
  - (2)  $A, B \in \mathcal{A}, A \cap B = \emptyset \Rightarrow P(A \cup B) = P(A) + P(B)$
  - (3)  $A, B \in \mathcal{A}, A \subseteq B \Rightarrow P(B - A) = P(B) - P(A)$
  - (4)  $A, B \in \mathcal{A}, A \subseteq B \Rightarrow P(A) \leq P(B)$
  - (5)  $A \in \mathcal{A} \Rightarrow 0 \leq P(A) \leq 1$
  - (6)  $A \in \mathcal{A} \Rightarrow P(\bar{A}) = 1 - P(A)$
  - (7)  $A, B \in \mathcal{A} \Rightarrow P(A \cup B) = P(A) + P(B) - P(A \cap B)$
  - (8)  $A_1, \dots, A_n \in \mathcal{A} \Rightarrow P(\bigcup_{i=1}^n A_i) \leq \sum_{i=1}^n P(A_i)$
- 
- (9)  $A_1, \dots, A_n \in \mathcal{A} \Rightarrow P(\bigcup_{i=1}^n A_i) = \sum_{i=1}^n P(A_i) - \sum_{i=1}^{n-1} \sum_{j=i+1}^n P(A_i \cap A_j) + \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^n P(A_i \cap A_j \cap A_k) + \dots + (-1)^{n-1} P(A_1 \cap \dots \cap A_n)$

## Podmíněná pravděpodobnost

O proběhlém pokusu máme doplňující informace a může tak lépe určit jeho pravděpodobnost.

Nechť  $(\Omega, \mathcal{A}, P)$  je pravděpodobnostní prostor,  $B \in \mathcal{A}$ ,  $P(B) > 0$ . Pak číslo

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

nazveme **podmíněnou pravděpodobností** jevu  $A$  za podmínky, že nastal jev  $B$ .

## Bayesův vzorec

Mějme dva náhodné jevy  $A$  a  $B$ , přičemž  $P(B) \neq 0$ . Potom platí:

$$P(A | B) = \frac{P(B | A) P(A)}{P(B)}, \text{ kde}$$

- $P(A | B)$  je podmíněná pravděpodobnost jevu  $A$  za předpokladu, že nastal jev  $B$
- $P(B | A)$  je podmíněná pravděpodobnost jevu  $B$  podmíněná výskytem jevu  $A$
- $P(A)$  a  $P(B)$  jsou pravděpodobnosti jevů  $A$  a  $B$

Lze jednoduše vyjádřit z podmíněné pravděpodobnosti dosazením za průnik.

Sčítání pravděpodobností

$$P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$$

Pro **nezávislé** jevy jsou průniky **nulové**.

## Náhodná veličina a vektor

Náhodná veličina je číselné ohodnocení náhodného pokusu (který dokážeme číselně ohodnotit). **Náhodnou veličinu** značíme **velkými písmeny X, Y, Z**.

**Pravděpodobnost** se značí **malými x, y, z, p**.

Pravděpodobnost, že náhodná veličina  $X$  nabývá hodnoty  $x$  zapíšeme jako

$$P(X = x).$$

Podobně lze interpretovat  $P(X < x)$ ,  $P(X \geq x)$ , atd.

Rozlišujeme dva základní typy náhodných veličin:

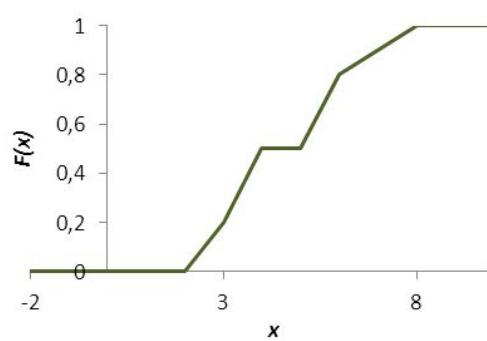
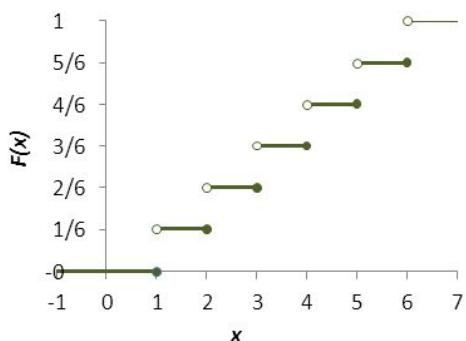
- **diskrétní**,
- **spojitá**

## Distribuční funkce

Popisuje **rozdelení (pravděpodobnostního chování)** náhodných veličin. Je **neklesající a zprava spojitá**. Distribuční funkce  $F(x)$  náhodné veličiny  $X$  přiřazuje každému reálnému číslu  $x$  pravděpodobnost, že náhodná veličina  $X$  nabude hodnoty **menší nebo rovné číslu x**.

$$F(x) = P(X \leq x)$$

U **diskrétní** náhodné veličiny je distribuční funkce **schodovitá** (vlevo).

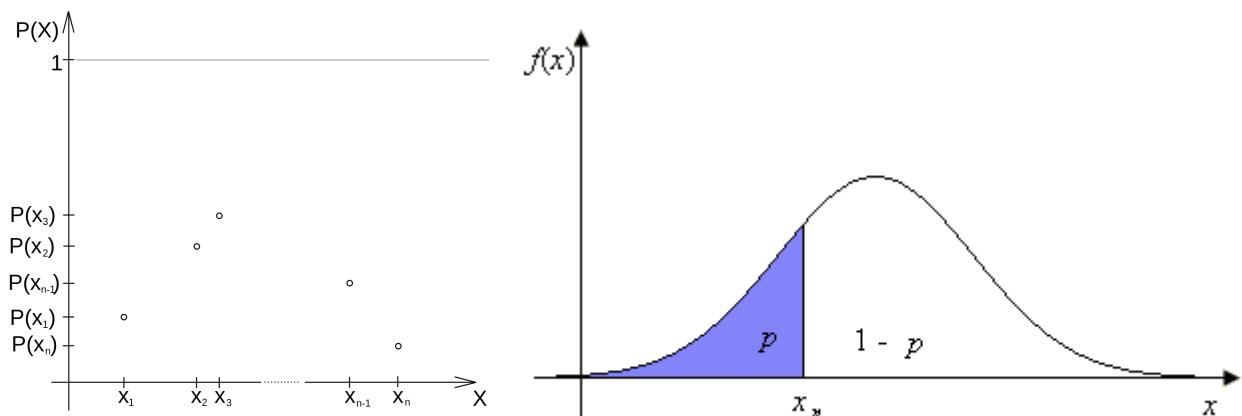


Vlastnosti:

- $0 \leq F(x) \leq 1$  pro  $\forall x \in \mathbb{R}$
- $F(x)$  je neklesající a zprava spojitá funkce.
- $\lim_{x \rightarrow -\infty} F(x) = 0, \lim_{x \rightarrow \infty} F(x) = 1$
- $P(a < X \leq b) = F(b) - F(a)$  pro každé  $a, b \in \mathbb{R}, a < b$
- $P(X = x) = F(x) - \lim_{t \rightarrow x^-} F(t)$
- $F$  má nejvýše spočetně mnoho bodů nespojitosti

Funkce hustoty pravděpodobnosti (Pravděpodobnostní funkce u diskrétní)

Určuje **rozdělení pravděpodobnosti náhodné veličiny** (u **diskrétní** náhodné veličiny lze vyjádřit tak, že se určí pravděpodobnost  $P(x)$  pro všechna  $x$  definičního oboru veličiny  $X$ ). Značíme jí  $f(x)$  a získá se první derivací distribuční funkce  $F'(x) = f(x)$ .



Vlastnosti:

- $f(x) \geq 0$
- $f(x) = \frac{dF(x)}{dx}$
- $\int_{-\infty}^{\infty} f(x) dx = 1$
- $P(a \leq X \leq b) = \int_a^b f(x) dx$

## Charakteristiky náhodné veličiny

### Střední hodnota $E(X)$

Střední hodnota **diskrétní** náhodné veličiny (1. obrázek) je **pravděpodobnostně vážený průměr všech jejích možných hodnot**. Pro **spojitou** náhodnou proměnnou je součet nahrazen **integrálem**.

$x$	0	100	200	400
$p(x)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

$$E(X) = \sum_x x \cdot p(x) = 100$$

$$E(X) = \int_M x f(x) dx$$

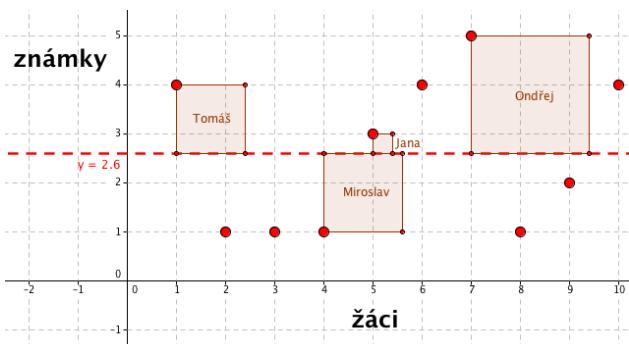
$$E(a) = a$$

$$E(aX + b) = aE(X) + b$$

$$E(X \pm Y) = E(X) \pm E(Y)$$

### Rozptyl $D(X)$

Také **střední kvadratická odchylka**. Jedná se o **charakteristiku variability** rozdělení pravděpodobnosti **náhodné veličiny**, která vyjadřuje variabilitu rozdělení **souboru náhodných hodnot kolem její střední hodnoty**. Jedná se o **součet obsahů čtverců** jednotlivých hodnot dle vzdálenosti od střední hodnoty.



$$D(X) = E(X^2) - [E(X)]^2,$$

$$E(X^2) = \sum_{x \in M} x^2 \cdot p(x).$$

$$\sigma^2 = \int_{-\infty}^{\infty} [x - E(X)]^2 p(x) dx = \int_{-\infty}^{\infty} x^2 p(x) dx - [E(X)]^2$$

$$D(X) \geq 0$$

$$D(aX + b) = a^2 D(X)$$

$$D(X \pm Y) = D(X) + D(Y) \text{ pro nezávislé náhodné veličiny } X, Y$$

### Směrodatná odchylka $\sigma(X)$

Směrodatná odchylka vypovídá o tom, **nakolik se od sebe navzájem typicky liší jednotlivé případy** v souboru zkoumaných hodnot. Vypočítá se jako odmocnina z rozptylu

$$\sigma(X) = \sqrt{D(X)}$$

## Náhodný vektor

Často je výsledkem pokusu **n-tice** reálných čísel - **vektor**. Poté můžeme zjednodušeně říct, že se jedná o **vektor více náhodných veličin**, které jsou definované na pravděpodobnostním prostoru.

Nechť  $X, Y$  jsou náhodné veličiny definované na stejném pravděpodobnostním prostoru  $(\Omega, \mathcal{A}, P)$ . Pak  $(X, Y)'$  nazveme **náhodným vektorem**.

### Sdružená distribuční funkce

Obdobná distribuční funkci pro jednu náhodnou veličinu s tím rozdílem, že je **n rozměrná** (n je počet veličin náhodného vektoru). Platí také obdobná pravidla.

$$\lim_{x \rightarrow -\infty} F(x, y) = \lim_{y \rightarrow -\infty} F(x, y) = 0$$

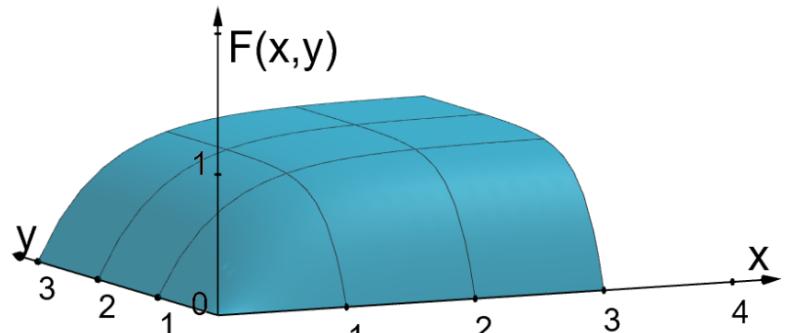
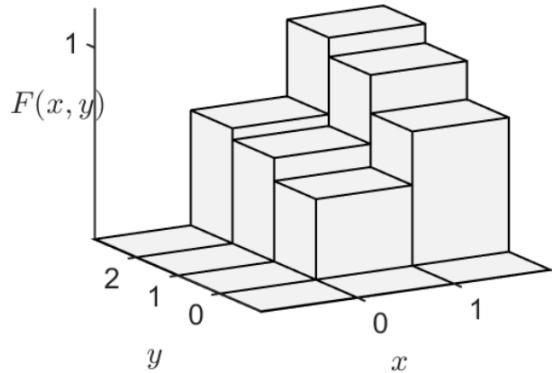
$$\lim_{(x,y) \rightarrow (\infty, \infty)} F(x, y) = 1$$

$$\sum_x \sum_y p(x, y) = 1.$$

$$F(x, y) = \sum_{u \leq x} \sum_{v \leq y} p(u, v).$$

$$F(x, y) = \int_{-\infty}^x \int_{-\infty}^y f(u, v) dv du.$$

Příklad **diskrétní vlevo** a **spojité vpravo**:



### Marginální distribuční funkce

Jedná se o distribuční funkci **jedné z náhodných proměnných vektoru**, pokud jsou náhodné jevy popsané **zbylými proměnnými jisté** (100%).

Je-li  $F(x, y)$  sdružená distribuční funkce veličin  $X$  a  $Y$ , pak **marginální distribuční funkce** veličiny  $X$  a veličiny  $Y$  je

$$F_X(x) = P(X \leq x) = \lim_{y \rightarrow \infty} F(x, y), \quad x \in \mathbb{R},$$

$$F_Y(y) = P(Y \leq y) = \lim_{x \rightarrow \infty} F(x, y), \quad y \in \mathbb{R}.$$

Sdružená pravděpodobnostní funkce (sdružená hustota pst.)

$$p(x, y) = P(X = x, Y = y)$$

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \, dx \, dy = 1$$

Marginální pravděpodobnostní funkce

Všechny ostatní náhodné veličiny, až na jednu (ta, u které zjišťujeme marginální pravděpodobnostní funkci), jsou jisté (100%)

$$p_X(x) = P(X = x) = \sum_y p(x, y), \quad x \in \mathbb{R},$$

$$p_Y(y) = P(Y = y) = \sum_x p(x, y), \quad y \in \mathbb{R}.$$

$x \setminus y$	0	1	2
0	0,42	0,12	0,06
1	0,28	0,08	0,04

$$p_X(x) = \sum_y p(x, y)$$

$x$	0	1
$p_X(x)$	0,6	0,4

$$p_Y(y) = \sum_x p(x, y)$$

$y$	0	1	2
$p_Y(y)$	0,7	0,2	0,1

$$f_X(x) = \int_{-\infty}^{\infty} f(x, y) \, dy = \int_0^{\infty} e^{-x-y} \, dy = e^{-x}, \quad x > 0$$

$$f_Y(y) = \int_{-\infty}^{\infty} f(x, y) \, dx = \int_0^{\infty} e^{-x-y} \, dx = e^{-y}, \quad y > 0$$

Podmíněné rozdělení

Mějme diskrétní náhodný vektor  $(X, Y)'$  se sdruženou pravděpodobnostní funkcí  $p(x, y)$ . Funkci

$$p(x|y) = P(X = x | Y = y) = \frac{p(x, y)}{p_Y(y)}, \quad p_Y(y) > 0,$$

nazveme **podmíněnou pravděpodobnostní funkcí** veličiny  $X$  za podmínky, že veličina  $Y$  nabyla hodnoty  $y$ .

Podobně

$$p(y|x) = P(Y = y | X = x) = \frac{p(x, y)}{p_X(x)}, \quad p_X(x) > 0,$$

nazveme podmíněnou pravděpodobnostní funkcí veličiny  $Y$  za podmínky, že veličina  $X$  nabyla hodnoty  $x$ .

Mějme spojitý náhodný vektor  $(X, Y)'$  se sdruženou hustotou pravděpodobnosti  $f(x, y)$ . Funkci

$$f(x|y) = \frac{f(x, y)}{f_Y(y)}, \quad f_Y(y) > 0,$$

nazveme **podmíněnou hustotou pravděpodobnosti** veličiny  $X$  za podmínky, že veličina  $Y$  nabyla hodnoty  $y$ .

Podobně

$$f(y|x) = \frac{f(x, y)}{f_X(x)}, \quad f_X(x) > 0,$$

nazveme podmíněnou hustotou pravděpodobnosti veličiny  $Y$  za podmínky, že veličina  $X$  nabyla hodnoty  $x$ .

## Nezávislost

Mějme (diskrétní nebo spojitý) náhodný vektor  $(X, Y)'$ . Pak veličiny  $X, Y$  jsou **nezávislé**, právě když

$$F(x, y) = F_X(x) \cdot F_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

$$F(x, y) = f_X(x) \cdot f_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

Mějme diskrétní vektor  $(X, Y)'$ . Pak veličiny  $X, Y$  jsou **nezávislé**, právě když

$$p(x, y) = p_X(x) \cdot p_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

Mějme spojitý náhodný vektor  $(X, Y)'$ . Pak veličiny  $X, Y$  jsou **nezávislé**, právě když

$$f(x, y) = f_X(x) \cdot f_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

$x \setminus y$	3	4	5	$p_X(x)$
1	0,3	0,24	0,06	0,6
2	0,2	0,16	0,04	0,4
$p_Y(y)$	0,5	0,4	0,1	1

$$p(x, y) = p_X(x) \cdot p_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2$$

Náhodné veličiny  $X$  a  $Y$  jsou nezávislé.

## Střední hodnota a rozptyl

$$E(X) = \sum_x x \cdot p_X(x), \quad E(Y) = \sum_y y \cdot p_Y(y),$$

$$E(X) = \int_{-\infty}^{\infty} x \cdot f_X(x) dx, \quad E(Y) = \int_{-\infty}^{\infty} y \cdot f_Y(y) dy,$$

$$E(XY) = \sum_x \sum_y xy \cdot p(x, y),$$

$$E(XY) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} xy \cdot f(x, y) dx dy$$

$$D(X) = E(X^2) - [E(X)]^2, \quad D(Y) = E(Y^2) - [E(Y)]^2,$$

$$D(X) = E(X^2) - [E(X)]^2, \quad D(Y) = E(Y^2) - [E(Y)]^2,$$

kde

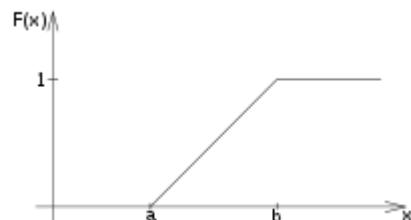
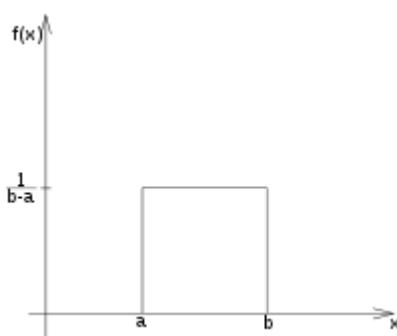
$$E(X^2) = \sum_x x^2 \cdot p_X(x), \quad E(Y^2) = \sum_y y^2 \cdot p_Y(y).$$

kde

$$E(X^2) = \int_{-\infty}^{\infty} x^2 \cdot f_X(x) dx, \quad E(Y^2) = \int_{-\infty}^{\infty} y^2 \cdot f_Y(y) dy.$$

## Rozdělení pravděpodobnosti

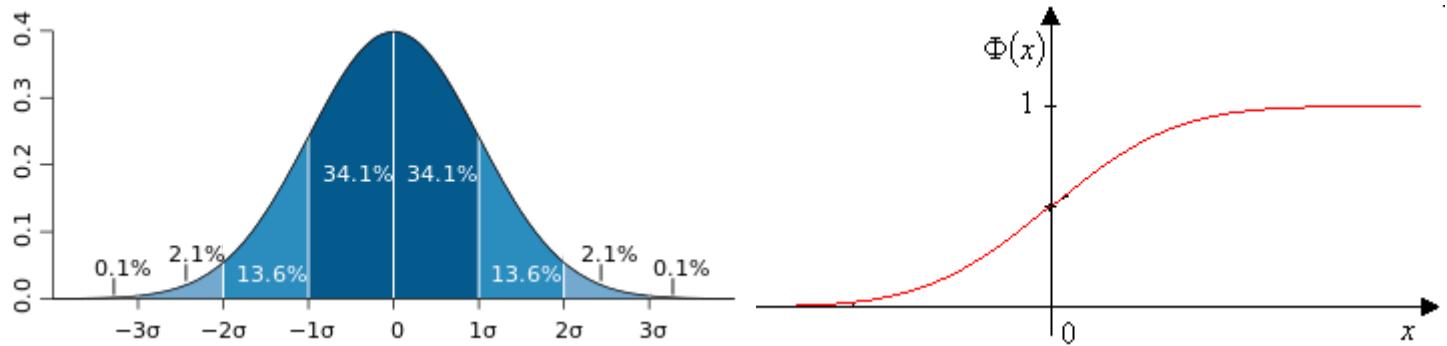
### Rovnoměrné rozložení



$$f(x) = \begin{cases} \frac{1}{b-a} & \text{pro } x \in (a, b) \\ 0 & \text{pro } x \notin (a, b) \end{cases}$$

$$F(x) = \begin{cases} 0 & \text{pro } x \leq a \\ \frac{x-a}{b-a} & \text{pro } a < x < b \\ 1 & \text{pro } x \geq b \end{cases}$$

## Normální rozdělení



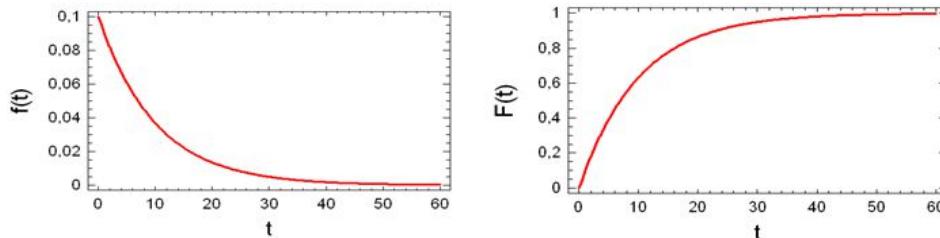
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$F(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

Jeho důležitost ukazuje **centrální limitní věta** (CLV), jež zhruba řečeno tvrdí, že součet či aritmetický průměr **velkého počtu libovolných** vzájemně nezávislých a nepříliš „divokých“ náhodných veličin se vždy **podobá normálně rozdělené** náhodné veličině.

## Exponenciální rozdělení

Určuje **dobu mezi dvěma následnými výskytůmi dané náhodné události**.

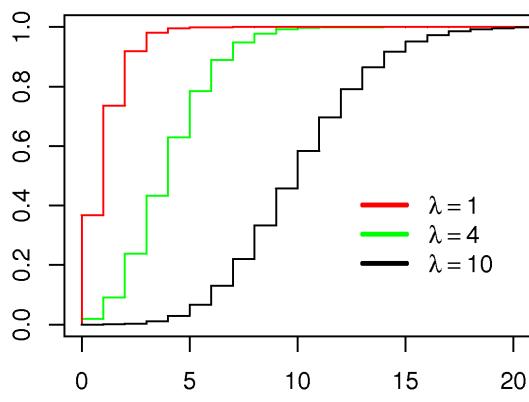
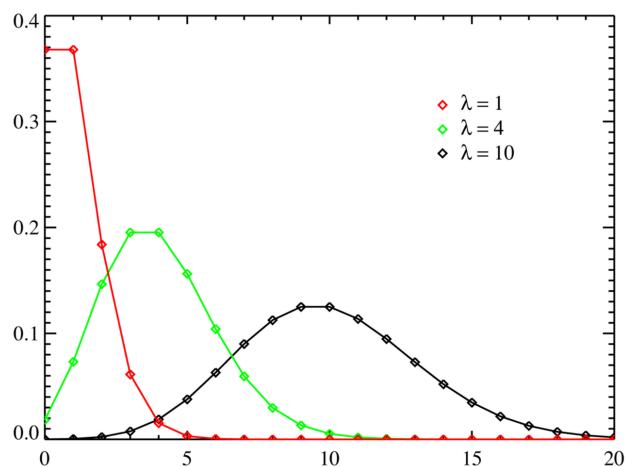


$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & ; x > 0, \\ 0 & ; x \leq 0. \end{cases}$$

$$F(x) = \begin{cases} 1 - e^{-\lambda x} & ; x > 0, \\ 0 & ; x \leq 0. \end{cases}$$

## Poissonovo rozdělení

Jedná se o **diskrétní rozdělení**. Udává **počet výskytů dané náhodné události za jednotku času**.

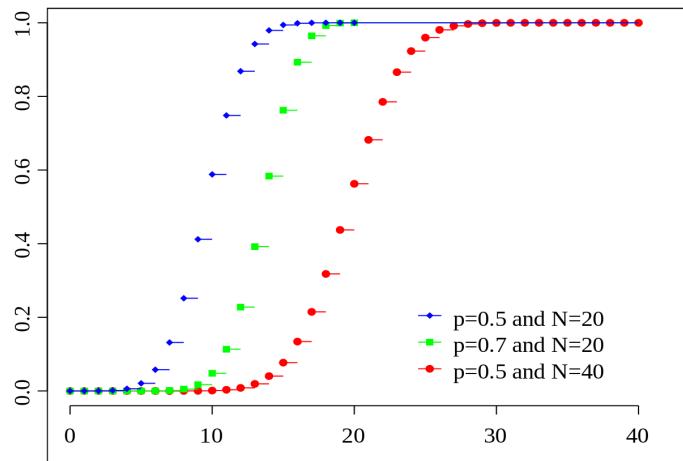
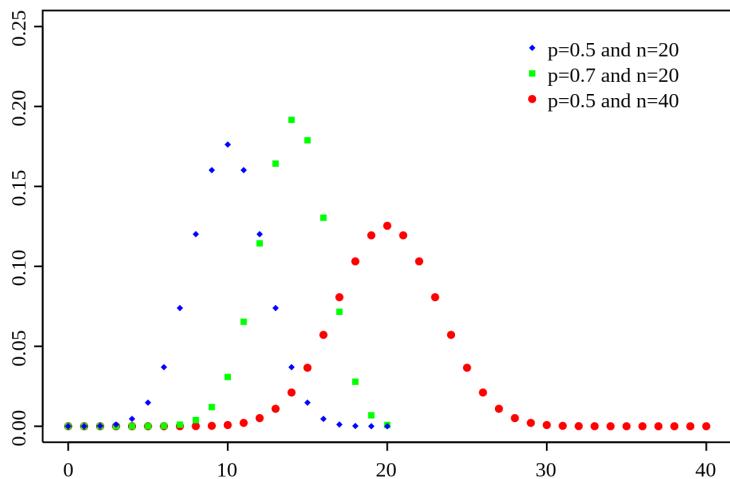


$$P(X = x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

$$F(x) = \sum_{x_i \leq x} \frac{\lambda^{x_i}}{x_i!} e^{-\lambda}$$

## Binomické rozdělení

Opět se jedná o **diskrétní rozdělení**.



$$P[X = x] = \binom{n}{x} p^x (1 - p)^{n-x}$$

## Generování pseudonáhodných čísel

- **fyzikální zdroje náhodnosti**: využívají senzory zařízení (teplota, akcelerace, poloha, ...), vygenerovaná čísla jsou opravdu náhodná (**nedeterministické** generování). Problémem je rychlosť, generují jen málo bitů za sekundu.
- **algoritmické generátory**: pseudonáhodné (**deterministické**), generují řádově miliardy bitů za sekundu.

### Kongruentní generátor

$$X_{n+1} = (ax_n + b) \bmod m$$

konstanty  $a$ ,  $b$ ,  $m$  musí být vhodně zvolené, jinak budou generovaná čísla nekvalitní (závislost mezi čísly).

- generují **rovnoramenné rozložení**,
- generují **konečnou posloupnost** čísel - perioda generátoru.

```
static uint32_t ix = SEED; // počáteční hodnota, 32b
double Random(void) {
    ix = ix * 69069u + 1u; // mod 2^32 je implicitní
    return ix / ((double)UINT32_MAX + 1.0);
}
```

Požadavky na generátor:

- **Rovnoměrnost rozložení**,
- Statistická **nezávislost generované posloupnosti**,

- Co **nejdelší perioda**,
- Rychlosť.

Další algoritmy generování:

- **Mersenne twister** (perioda  $2^{19937} - 1$ ),
- **Xorshift**.

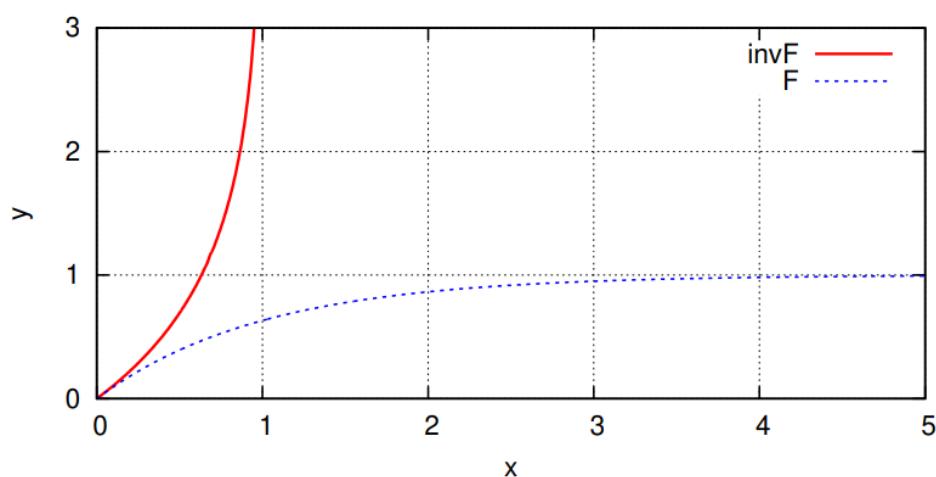
## Transformace na jiná rozložení

Náhodné a pseudonáhodné generátory generují ideálně čísla dle **rovnoramenného rozložení**. Při výpočtech ale často potřebujeme generovat čísla dle jiného rozložení.

### Metoda inverzní transformace

Ideální metoda pro generování rozložení, rozložení je určeno **analyticky** pomocí **inverzní distribuční funkce cílového rozložení**. Často ale distribuční funkci nemusíme znát nebo **nelze vyjádřit její inverzní funkce elementárními funkcemi**.

**Příklad:** Exponenciální rozložení  $F(x) = 1 - e^{-\frac{x-x_0}{A}}$   
 $y = x_0 - A * \ln(1 - x)$  viz. obrázek pro  $x_0 = 0, A = 1$



### Vylučovací metoda

Sérií pokusů hledáme číslo, které vyhovuje funkci hustoty cílového rozložení. Metoda je **nevzhodná** pro **neomezená rozložení** a rozložení, kde je **velká část** hustoty pravděpodobnosti koncentrovaná **do malého intervalu** (**špičaté** normální rozložení). Abychom byli schopni touto metodou generovat rozložení, musíme znát jeho ohrazení (osa x - **definiční obor** a osa y - **obor hodnot**). Funguje na principu, že se vygenerují 2 pseudonáhodná čísla x a y, x se dosadí **do funkce hustoty pravděpodobnosti f(x)** a její hodnota se **porovná s y**. Pokud je menší, je x vráceno jako **hodnota náhodné veličiny**, jinak se proces **opakuje**. Problém může být mnoho iterací při nesprávném rozložení.

## Kompoziční metoda

**Složitou funkci hustotu** (případně distribuční) **rozdělíme** na několik jednodušších po **intervalech**. Na každém intervalu můžeme použít **jinou metodu** pro generování rozložení.

## Bodové a intervalové odhady parametrů

- **Základní soubor** (populace) - obsahuje **všechny** vymezené jednotky.
- **Výběrový soubor** (výběr) - obsahuje pouze **některé** jednotky.

**Vlastnosti výběrového souboru** se snažíme **zobecnit pro celý základní soubor**. Např. při volebním průzkumu se snažíme zobecnit **1000 dotázaných** voličů **na 8 milionu** voličů. Musíme použít **reprezentativní** výběr - **náhodný**.

### Bodový odhad

**Neznámý parametr** (např. průměr) **základního souboru** odhadujeme pomocí **jediného čísla, bodu** na základě **výběrového souboru**. Bodovým odhadem parametru základního souboru jsou **popisné charakteristiky výběrového souboru**. pro odhady používáme **výběrový průměr** a **výběrový rozptyl**. Příklad:

- Pokud je **neznámým parametrem základního souboru** 10 000 nových baterek **průměrné napětí**, může jich náhodně vybrat **200 - výběrový soubor**. U těchto 200 baterek určíme **průměrné napětí**, např. 1.49 V a tuto hodnotu prohlásíme za odhad neznámého parametru (v tomto případě průměrného napětí) základního souboru. Takto bychom bodový odhad mohli opakovat např. i pro hmotnost.

Bodový odhad nebude téměř nikdy **přesnou** hodnotou neznámého parametru. Za **lepší** považujeme ten, jehož **rozdělení je více koncentrované okolo neznámé hodnoty** parametru → má **menší rozptyl**, respektive **směrodatnou odchylku**.

- **NEstranný odhad**: Odhad je nestranný, pokud skutečnou hodnotu parametru **nepodhodnocuje ani nenadhodnocuje**. Nezaručí dobrý odhad, pouze **vyloučí systematickou chybu**. **Výběrový průměr a výběrový rozptyl** jsou nestranné odhady, pouhý **rozptyl není nestranný**.

**Konzistentní** (dobrý) odhad se **blíží** k skutečné hodnotě odhadovaného neznámého parametru, **čím větší počet pozorování provádíme** (čím větší je výběrový soubor).

### Střední kvadratická chyba

Umožňuje měřit přesnost bodového odhadu, kombinuje **vychýlení a rozptyl**. Jsme poté schopni například říct, že odhadem je **1.49 V** se střední kvadratickou chybou **0.008V**.

## Výběrový průměr

$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^n X_i$$

Jedná se o **aritmetický průměr**.

## Výběrový rozptyl

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2,$$

## Intervalové odhady

Protože bodové odhady jsou poměrně nepřesné, používají se v praxi spíše intervalové odhady, u kterých můžeme říct, že s **určitou pravděpodobností** (vysokou **95-99% - interval spolehlivosti**) se zde **neznámý parametr základního souboru bude vyskytovat**.

- Spolehlivost odhadu je dána zvolenou **pravděpodobností** intervalu spolehlivosti, **čím** je pravděpodobnost **větší**, tím je daný odhad **spolehlivější**. **Čím** je ale **odhad spolehlivější**, tím se **zvětšuje** i příslušný **interval spolehlivosti**. Tj. **čím širší** interval spolehlivosti, tím je odhad **spolehlivější**, ale méně přesný, což je **nepraktické**. Mezi spolehlivostí a přesnosti je **nepřímá úměrnost**.

Pokud budu mnohokrát po sobě generovat náhodný výběr  $X_1, \dots, X_n$  a pro každý výběr sestrojím 95% interval spolehlivosti pro parametr  $\theta$  (resp. parametrickou funkci  $\tau(\theta)$ ), pak přibližně 95 % z nich bude obsahovat skutečnou hodnotu parametru  $\theta$  (resp. parametrickou funkci  $\tau(\theta)$ ).

Např. pokud budu pro různé náhodné výběry 100x určovat 95% intervaly spolehlivosti pro parametr  $\theta$ , pak v asi 5 případech dostanu interval, který neobsahuje skutečnou (správnou) hodnotu parametru  $\theta$ .

V některých případech nás může zajímat pouze **horní hranice - pravostranný interval spolehlivosti**, nebo jenom **dolní hranice - levostranný interval spolehlivosti** pro odhad parametru. Poté používáme **jednostranné intervaly spolehlivosti**.

**Rozdělení používaná při intervalových odhadech:**

- **Studentovo t** rozdělení,
- **Chí kvadrát** rozdělení,
- **Asymptotické normální** rozdělení.

## Odhad relativní četnosti

Odhadovaná veličina je rozdělena **alternativně** - může **nabývat 2 stavů** (kluk - holka, rub - líc, ...). Na základě **výběrového souboru** poté **odhadujeme relativní četnost** výskytu (pravděpodobnost výskytu znaku) **v základním souboru** (např. kolik se za rok narodí holek a kolik kluků podle dat z 1 měsíce).

## Testování hypotéz

Používá se v situacích, kdy **potřebujeme rozhodnout o správnosti nějakého tvrzení**. Např. jestli vede nová technologie výroby k zlepšení kvality vyráběných výrobků.

### Statistická hypotéza (tvrzení)

Jedná se o **tvrzení o parametrech** rozdělení, z něhož pochází **náhodný výběr (střední hodnota, rozptyl, směrodatná odchylka, ...)** a o **typu tohoto rozdělení** (normální, exponenciální, rovnoměrné, ...). Dva druhy hypotéz:

- **Nulová hypotéza  $H_0$  - předpoklad**, který vyslovíme o určitém parametru či tvaru rozdělení pravděpodobnosti sledované náhodné veličiny.
  - Pacient je nemocný
- **Alternativní hypotéza  $H_1$**  - popisuje, jaká situace nastává, **když nulová hypotéza neplatí**.
  - Pacient je zdravý

**Testování statistických hypotéz** je postup, kterým na základě hodnot náhodného výběru ověřujeme **platnost nulové hypotézy**. Testování může mít **dva závěry**:

- **$H_0$  zamítáme** a tím pádem **platí alternativní hypotéza  $H_1$**  (s určitou vysokou pravděpodobností).
- **$H_0$  nezamítáme**, to znamená, že  $H_0$  buď platí, nebo nemáme dostatek informací k tomu, abychom mohli  $H_0$  zamítнуть. Říkáme, že hypotéza  **$H_1$  se neprokázala**.

### Testovací statistika

Jedná se o **testovací kritérium**. Obor hodnot testovací statistiky rozdělíme na dvě části:

- **kritický obor  $W$** : obor **zamítnutí** hypotézy,  $T \in W \rightarrow \text{zamítáme } H_0$ ,
- **obor přijetí  $W'$** : obor, ve kterém **nezamítáme** hypotézu.  $T \notin W \rightarrow \text{nezamítáme } H_0$ .

Platnost  $H_0$  posuzujeme na základě **náhodného výběru**  $\Rightarrow$  můžeme se dopustit chybných závěrů. Chyby:

- **Chyba 1. druhu  $\alpha$** : zamítáme hypotézu, která platí.  $\alpha = \text{hladina významnosti}$  testu (obvykle **0,05**  $\rightarrow$  zamítáme  $H_0$  a s pravděpodobností aspoň **0,95** ( $1-\alpha$ ) platí  $H_1$ ).

- **Chyba 2. druhu  $\beta$ :** nezamítáme hypotézu, která neplatí.  $1 - \beta = \text{síla testu}$  - pravděpodobnost, že zamítneme hypotézu, která neplatí.

Chyby 1. a 2. druhu **jdou proti sobě** – nelze minimalizovat obě. (nepřímá úměrnost).

## Testy

- **Studentův t-test:**
  - test střední hodnoty rozdělení se **známým rozptylem**,
  - test zda dvě normální rozdělení mající **stejný** (byť neznámý) **rozptyl**, z nichž pocházejí dva **nezávislé náhodné výběry**, mají **stejné střední hodnoty**.
- **F-test** je jakýkoliv statistický test, ve kterém má testová statistika rozdělení F.

## Regresní a korelační analýza

- **regrese:** Hledáme **funkční závislost** náhodné **veličiny na jiné veličině** (jednostranná závislost). **Umožňuje předpovědi**, např. odhad výšky dcery na základě výšky matky.
- **korelace:** Hledáme **sílu** (těsnost) **závislosti dvou náhodných veličin** (vzájemná závislost). **Neslouží** k předpovědím. Např. určujeme, jak těsně spolu souvisí výška matky a výška dcery. Nebo investoři zkoumají, jak jsou nějaké akcie korelované a investují např. do těch které jsou korelované minimálně.

## Regresní analýza

Umožňuje **vyjádřit vztah** mezi proměnnou, kterou chceme popisovat (**vysvětlovaná proměnná**), a množinou **vysvětlujících proměnných**.

- Hledáme vztah mezi množstvím zkonzumované zmrzliny (vysvětlovaná proměnná) a teplotou vzduchu (vysvětlující proměnná).
- Hledáme vztah mezi počtem bodů u zkoušky (vysvětlovaná proměnná) a počtem hodin, které student strávil přípravou na zkoušku (vysvětlující proměnná).

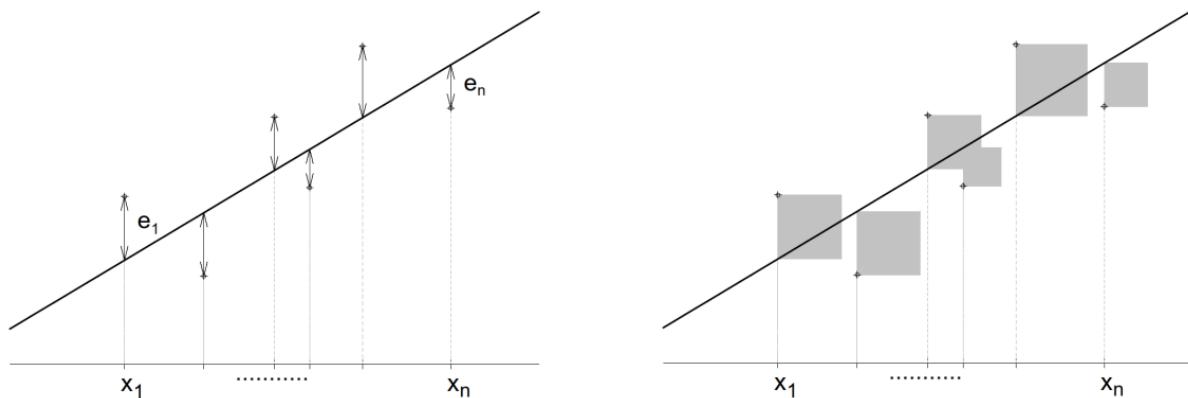
Při regresní analýze odhadujeme:

- **regresní přímku**,
- **regresní parabolu**,
- **regresní rovinu**.

## Metoda nejmenších čtverců

Slouží pro odhad **regresní přímky**, **regresní paraboly** i **regresní roviny**.

$$Y_i = \beta_0 + \beta_1 x_i + e_i \quad \Rightarrow \quad e_i = Y_i - (\beta_0 + \beta_1 x_i)$$



## Lineární regrese

Závislost mezi veličinami je popsána funkcí, která je **lineární v parametrech** (koeficientech).

## Nelineární regrese

Závislost mezi veličinami je popsána funkcí **nelineární v parametrech**.

## Korelační analýza

Úkolem je stanovit sílu závislosti mezi sledovanými kvantitativními znaky. Hledáme závislost mezi veličinami, které **spolu (logicky) mohou souviseť**. Např. přidávání olova do benzingu a jeho zvýšený výskyt v ovzduší a snižující se IQ a zvyšující se kriminalita. Naopak ale **korelace neimplikuje kauzalitu** - pokud existuje korelace mezi dvěma proměnnými, nelze z toho ještě vyvozovat, že jedna je příčinou a druhá důsledkem (Např. výdaje na podporu rozvoje vědy a počet sebevražd - obě v poslední době rostou, ale podpora vědy nezpůsobuje sebevraždy).

# 32. Hodnocení složitosti algoritmů (paměťová a časová složitost, asymptotická časová složitost, určování časové složitosti).

Složitost algoritmů definuje kritéria, podle kterých jednotlivé algoritmy hodnotíme. Nejčastějšími kritérii jsou **časové a paměťové nároky algoritmu**. Reálné doby běhu programů se liší na různých strojích a s různými vstupy. Proto zavádíme **časovou a prostorovou složitost algoritmů**, které umožňují:

- vyjádření vlastností algoritmu **nezávisle** na technických podrobnostech (počítači, na kterém běží)
- popis chování algoritmu pomocí **jednoduchých matematických funkcí**.

## Časová složitost

Časová složitost je odvozena od počtu elementárních operací (sčítání, násobení, porovnání, ...). Udává **řádový počet provedených elementárních operací v závislosti na velikosti vstupu** (počet prvků řazeného pole, větší z čísel, u kterého hledáme společného dělitele). Příkladem mohou být 2 různé (nejedná se o stejnou funkcionality) algoritmy součtu:

### Algoritmus Součet1:

```
1. for i ← (1, n) do
2.   for j ← (1, n) do
3.     s ← s + j
4.   end for
5. end for
6. return s
```

Počet elementárních kroků:  $c_1 \cdot n^2 + c_2$

Zjednodušeně:  $n^2$

### Algoritmus Součet2:

```
1. while n >= 1 do
2.   s ← s + n
3.   n ← n / 2
4. end while
5. return s
```

Počet elementárních kroků:  $c_3 \cdot \log_2 n + c_2$

Zjednodušeně:  $\log_2 n$

- **Součet1:** U prvního algoritmu lze jednoduše **detekovat cyklus** podle **n** a **vnořený cyklus** podle **n** (obecně nemusí být jednoduché vnořené cykly detekovat). To znamená, že pro **každé n** se provede **n iterací**, celkově tedy **n\*n** iterací, neboli **n^2** iterací, a v každé iteraci je **provedeno sčítání**.
- **Součet2:** Zde je pouze **jeden cyklus** (while), tudíž hned na první pohled by se dalo **chybně** říct, že bude provedeno n operací. To ale není pravda, protože je nutné si v těle cyklu všimnout **dělení n** v každé iteraci, to znamená, že bude provedeno **log2(n)** operací.

## Postup určení časové složitosti

1. Zhodnotíme, co je vstupem  $n$  a jak **měřit jeho velikost**.
2. Určíme maximální možný počet elementárních kroků algoritmu (elementárních operací jako je sčítání) provedených pro vstup o **velikosti  $n$** .
3. Ve výsledné formuli **ponecháme pouze nejrychleji rostoucí člen**, ostatní zanedbáme.
4. Seškrťáme multiplikativní konstanty (**odebereme násobení konstantou**)

## Průměrná časová složitost

Určujeme, pokud pro **různé vstupy stejné velikosti** vykoná algoritmus **různý počet kroků** (např. řazení seřazeného pole, náhodně uspořádaného pole a opačné seřazeného pole). Za časovou složitost poté považujeme **aritmetický průměr** časových nároků **přes všechny vstupy** (docela nereálné) dané velikosti. Proto u příkladu se řazením používáme jinou vlastnost, a to přirozenost algoritmu.

## Asymptotická časová složitost

Jedná se o nejčastěji používané hodnotící kritérium, vychází z toho, že pro **malá  $n$**  se **nejrychleji rostoucí člen** nemusí výrazně projevovat. Popisujeme tedy chování algoritmu pro  **$n$  blížící se k nekonečnu**. Používají se tři různé asymptotické složitosti:

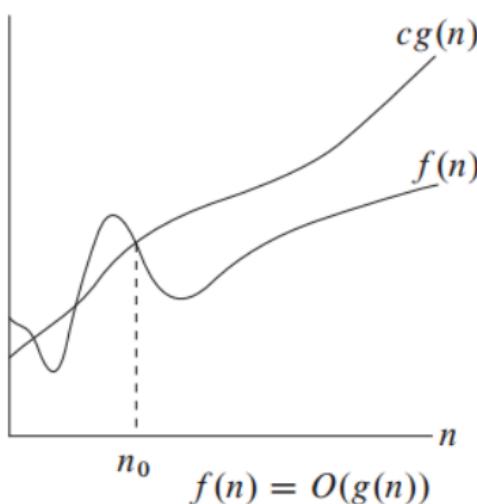
- $O$  – Omikron (velké  $O$ ,  $\mathcal{O}$ , big  $O$ ) – horní hranice chování (**nejčastější**),
- $\Omega$  – Omega – dolní hranice chování,
- $\Theta$  – Théta – třída chování.

U definování složitostí se snažíme, aby co nejvíce odpovídaly složitosti algoritmu (tj. hledáme **nejmenší horní hranici** a **největší dolní hranici**). Není podstatná konkrétní přesná časová závislost, stačí **charakterizovat třídu**.

## Složitost Omikron - $\mathcal{O}$

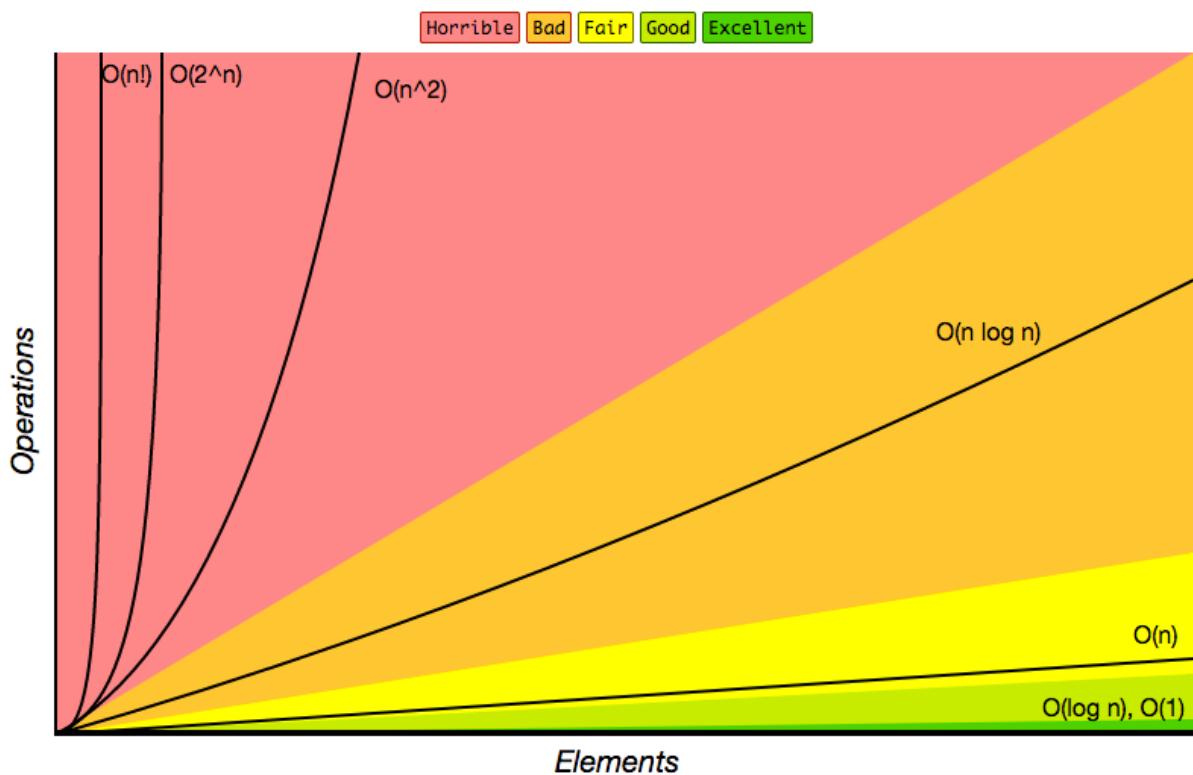
Vyjadřuje **horní hranici** časového chování algoritmu (omezuje chování algoritmu shora, tj. algoritmus **nebude nikdy pomalejší**, než dané omezení). Značíme jako **Omkron( $g(n)$ )** nebo ' $O(g(n))$ ' nebo jen  $O(g(n))$ .

- Zápis  $f(n) \in O(g(n))$ , označuje, že funkce  $f(n)$  je **rostoucí maximálně tak rychle** jako funkce  $g(n)$ . Dostatečně velký **násobek funkce  $g(n)$  shora omezuje funkci  $f(n)$**  pro **dostatečně velké  $n$** .



## Nejčastěji používané složitosti:

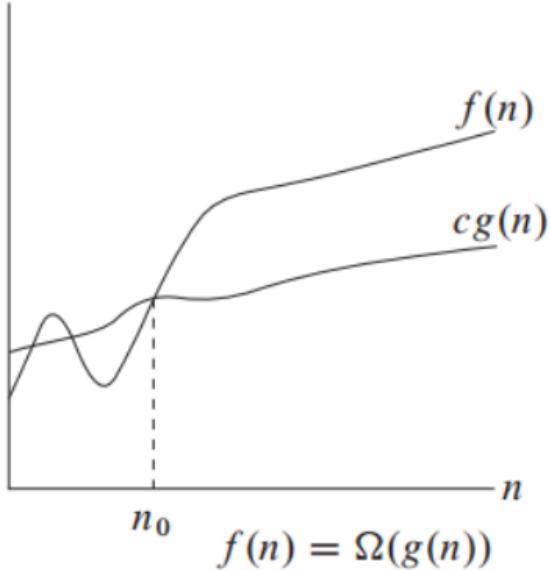
Asymptotická složitost	Vyjádření	Popis a nejčastější případ
konstantní	$O(1)$	Počet operací je pro libovolně velká vstupní data zhruba stejný. Typicky se jedná o nějaký jednoduchý matematický výpočet, jeden přístup k paměti, apod.
logaritmická	$O(\log n)$	Typicky ideální případ vyhledávání ve stromě s $n$ prvky.
lineární	$O(n)$	Náročnost algoritmu se zvyšuje podobně jako velikost vstupních dat. Typicky jeden průchod polem. Některé algoritmy s touto složitostí mohou být implementovány i proudově.
lineárně logaritmická	$O(n \cdot \log(n))$	Typická složitost rozumného řadícího algoritmu.
kvadratická	$O(n^2)$	Náročnost algoritmu roste jako druhá mocnina velikosti vstupních dat. Typicky průchod všech dvojic v poli.
polynomiální	$O(n^k), k \in R$	
exponenciální	$O(k^n), k \in R$	
faktoriálová	$O(n!)$	Typicky vyhodnocování všech možných permutací $n$ prvků, například v brute-force algoritmech.



## Složitost Omega - $\Omega$

Složitost Omega vyjadřuje **dolní hranici** časového chování algoritmu (omezuje chování algoritmu zdola, tj. algoritmus **nebude nikdy rychlejší**, než dané omezení). V praxi nemá moc využití. Značíme jako  $\text{Omega}(g(n))$ ,  $\Omega(g(n))$  nebo jen  $\Omega(g(n))$ .

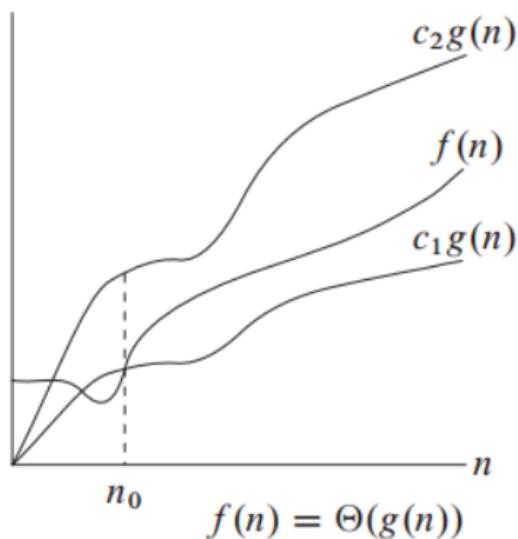
- Zápis  $f(n) \in \text{Omega}(g(n))$ , označuje, že funkce  $f(n)$  je **rostoucí minimálně tak rychle** jako funkce  $g(n)$ . Funkce  $g(n)$  je tak **dolní hranicí množiny všech funkcí**, určených zápisem  $\text{Omega}(g(n))$ .



### Složitost Théta - $\Theta$

Složitost **Theta** vyjadřuje třídu chování algoritmu – ohraničuje funkci  $f(n)$  z obou stran (slouží jako **dolní i horní hranice**, tedy že funkce nebude **nikdy rychlejší** než  $c_1 \cdot g(n)$  a nebude **nikdy pomalejší** než  $c_2 \cdot g(n)$ ). Značíme jako **Theta( $g(n)$ )**, nebo  **$\Theta(g(n))$** .

- Zápis  $f(n) \in \Theta(g(n))$  označuje, že funkce  $f(n)$  roste **stejně tak rychle** jako funkce  $g(n)$ .



### Třídy složitosti

- **P** - Pokud existuje Turingův stroj, který úlohu vyřeší v polynomiálním čase.
- **NP** - Pokud existuje nedeterministický turingův stroj, který rozhodne úlohu v polynomiálním čase

## Orienteční rychlosť výpočtu

Orientečná typické hodnoty velikosti vstupu  $n$ , pro které algoritmus s danou časovou složitostí ještě většinou zvládne na běžném PC spočítat výsledek **ve zlomku sekundy nebo maximálně v řádu sekund**.

$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
1 000 000 – 100 000 000	100 000 – 1 000 000	1000 – 10 000	100 – 1000

$2^{O(n)}$	$O(n!)$
20 – 30	10 – 15

## Paměťová (prostorová) složitost

Udává **spotřebu paměti**, případně **diskového prostoru** v závislosti na **vstupních datech**. Prostorovou složitost **nebývá kritická** a často ji neřešíme (paměť si narozdíl od času můžeme kupit). Nicméně může mít **záasadní** vliv na realizovatelnost algoritmu. Obvykle je u algoritmů ale **prostorová složitost menší**, než časová. Např. u většiny řadících algoritmů je prostorová složitost **konstantní  $O(1)$** . U Quick sort je **logaritmická  $O(\log(n))$**  - nutné si uchovat hranice ještě nezpracovaných částí. Stejně tak je nutné si uchovávat hranice polí u Merge sort - **rekurze znamená paměť**. U algoritmu pro prohledávání stavového prostoru **BFS** je např. prostorová složitost stejná jako časová, a to exponenciální  **$O(b^d)$**  (b je faktor větvení, d hloubka). U DFS je časová složitost stále exponenciální, ale prostorová pouze  **$O(b^d)$** . U backtracking je pak složitost pouze **lineární  $O(d)$** .

## Asymptotická paměťová složitost

Obdobně jako u časové složitosti se používají **tři** různé asymptotické složitosti:

- $\mathcal{O}$  – Omikron (velké O,  $\mathcal{O}$ , big O) – horní hranice chování (**nejčastější**),
- $\Omega$  – Omega – dolní hranice chování,
- $\Theta$  – Théta – třída chování.

Mají i stejný význam.

## Určování složitosti

### Určování časové složitosti

Hlavní dva jevy, které řešíme při určování časové složitosti jsou **cykly** (zejména **vnořené**) a **rekurze** závisející na **vstupních datech**.

- **Vnoření cyklů** většinou znamená násobení složitosti → vede na polynomiální časovou složitost: dva vnořené cykly -  $O(n^2)$ , tři vnořené cykly -  $O(n^3)$ .

Nutno sledovat úpravy řídící proměnné cyklu, např. její dělení vede na logaritmickou složitost  $O(\log(n))$ .

- Rekurze většinou znamená **exponenciální složitost**. Záleží, na počtu **rekurzivních volání funkce** v jejím těle. Pro dvě volání je složitost  $O(2^n)$ , pro tři volání  $O(3^n)$ , ... Počet volání můžeme označovat jako **faktor větvení** (zejména se používá u stromů).

Při více cyklech za sebou uvažujeme ten **nejhorší - nejrychleji rostoucí člen**.

## Určování paměťové složitosti

Musíme identifikovat, jaké používáme proměnné

- **lokální statické proměnné**,
- **dynamické proměnné**,
- **REKURZE** - má **velký vliv** na paměťovou složitost programu, při každém rekurzivním volání funkce se **ukládají všechny její lokální proměnné**.

Globální statické proměnné můžeme ignorovat (považovat za **konstantní** paměťovou náročnost), lokální statické proměnné můžeme ignorovat jen v případě, že se funkce **nevolá** rekurzivně. U **dynamických** proměnných **musíme sledovat alokaci** v cyklech **nebo při rekurzi** a jak **velká paměť** je alokována (pokud cyklem projdeme **n krát**, a **n krát** alokujeme **n prvků**, je **prostorová složitost kvadratická** (časová je lineární)).

## Další zdroje:

- <https://docplayer.cz/108481191-Prostorova-pametova-slozitost-algoritmu.html>
- [Asymptotická složitost](#)
- [Třídy složitosti a Turingovy stroje](#)

# 33. Životní cyklus softwaru (charakteristika etap a základních modelů).

## Životní cyklus softwaru

Mechanismus uplatňovaný při návrhu softwaru. Využívá se při vývoji velkých a složitých projektů. Je tvořena z **pěti etap**.

1. **Analýza a specifikace požadavků** (asi 8%),
2. **Architektonický a podrobný návrh** (asi 7%),
3. **Implementace a testování částí** (asi 12%),
4. **Integrace a testování** (asi 6%),
5. **Provoz a údržba** (asi 67%).

Bodům 1 a 2 je zejména nutné věnovat velkou pozornost, protože pozdější oprava chyb je **velmi dražá** (50x až 200x dražší, než její oprava při analýze a návrhu).

Při vývoji SW jsou spolupracují 3 strany:

- **Zákazník** - Objednal SW (může být i samotným uživatelem, většinou to tak ale není, SW objednává management, pracují s ním běžní zaměstnanci).
- **Dodavatel** - Vytváří SW (analytici, programátoři, testeři, management)
- **Uživatel** - Bude SW používat

## Analýza a specifikace požadavků

Snažíme se **přesně specifikovat** co zákazník chce (ale neřešíme jak toho dosáhnout). Zákazník často neví, co přesně chce, nebo chce něco, co vlastně nepotřebuje. Úkolem analytiků je se zákazníkem požadavky probrat (**interview, dotazník, studium dokumentů, pozorování při práci, analýza existujícího SW**) a co nejvíce se přiblížit tomu, co opravdu potřebuji. Výstupem je samotná **specifikace**, dále výstupem může být **analýza rizik** nebo **studie vhodnosti**, ale především **akceptační testy**, které vycházejí z jeho požadavků a které zákazník provede při převzetí. Výstupem také může být diagram případů užití, který slouží jako most mezi programátory a zákazníkem. Pokud jsou splněny, je SW v pořádku (**je verifikován**, otázkou je, jestli software po implementaci opravdu plní funkce, které zákazník potřeboval - validace. Následné změny jsou ale poté již nad rámec projektu a zákazník je musí zaplatit).

Kategorie požadavků:

- **Funkcionální** - co má projekt dělat.
- **Požadavky na provoz systému** - za jakých podmínek bude systém pracovat (např. bude jej současně obsluhovat 200 lidí).
- **Požadavky na výsledný systém** - podmínky pro vývoj a nasazení.

- **Požadavky na vývojový proces** - požadavky zákazníka na dodržování norem (např. použití nějakého standardizované postupu autentizace).
- **Požadavky na rozhraní** - komunikace se systémem (např. webová aplikace).
- **Externí požadavky** - požadavky dle charakteru aplikace (např. legislativní požadavky).

## Architektonický a podrobný návrh

Architektonický návrh se spíše zaměřuje na to, co budeme dělat, podrobný návrh na to, jak to budeme dělat.

### Architektonický návrh

Řeší se návrh aplikace od **fyzické úrovni** (např. že se bude jednat o třívrstvý informační systém s fyzicky odděleným DB serverem a klientem jako webový prohlížeč, i když část z toho může už být součástí specifikace - zákazník požaduje webovou aplikaci). Dále se může řešit jaké budeme používat nástroje pro vývoj - **programovací jazyky**. Následuje logické členění aplikace, např. výběr vhodného návrhového (architektonického) vzoru - **MVC**, **MVVM**, **MVP**. Následuje rozdělení celého projektu na podproblémy nejlépe nějak **logicky ucelené a rozhraní mezi nimi**. Můžeme je nazývat **moduly** (v případě MVC se může jednat o kontrolery). Příkladem může být modul pro správu uživatelů, ten bude skoro ve všech aplikacích. Členění do modulů může být výhodné také v tom, že na každém může pracovat jiný programátor - skupina programátorů. Nakonec by měly být produktem architektonického návrhu návrh **integračních testů a testů celého systému a plán nasazení do provozu**.

### Podrobný návrh

U informačního systému bude podrobný návrh spíše obnášet **návrh uložení dat v databázi** (např. pomocí ER diagramu) a **návrh uživatelského rozhraní**. U vědecké aplikace to bude spíše **návrh algoritmů a datových struktur**. Dále je součástí podrobného návrhu také návrh způsobu ošetřování neočekávaných a chybových stavů. Výstupem by měl být také **podrobný odhad ceny a nároky na lidské zdroje**. Výstupem také bude návrh **testů jednotlivých modulů**.

### Implementace a testování částí

Nejedná se pouze o psaní kódu. V rámci implementace se také řeší **dokumentace** kódu, tvorba **uživatelské příručky/manuálu**. Chystáte se **testovací prostředí**, které bude simulovat prostředí zákazníka a na kterém se bude aplikace testovat. Testovací prostředí je spojené s **tvorbou skriptů pro automatickou publikaci CI/CD**. **Testování jednotlivých modulů** a další...

## Integrace a testování

Integrace je spojená se **začleňováním modulů** do jednoho celku - systému. Testování obnáší **integrační testy a testy systému jako celku**. Často se při provádění testů vracíme k předešlým dvoum bodům a **opravujeme vzniklé chyby**. Jakmile jsme přesvědčení, že SW odpovídá specifikaci, předáváme jej zákazníkovi k akceptační testování. To opět může odhalit nějaké chyby, jakmile je ale akceptační testování provedeno a SW je tímto předán zákazníkovi, jsou následující změny již prováděny v rámci provozu a údržby.

## Provoz a údržba

V prvé řadě se jedná o **opravu chyb**, které nebyly identifikovány ani akceptačním testováním. Dále jde o **aktualizaci použitých nástrojů** (např. z důvodu bezpečnostních rizik) a správu serveru, na kterém aplikace běží (někdy může provádět sám zákazník). Obecné řešení problémů při provozu, např. přesun na jiný server atd. Dále může jít o **změny** nebo o **přidání rozšiřujících funkcí** (postupně se přichází na to, že SW není validní - uživatelé doopravdy potřebují něco jiného, nebo změny mohou být způsobené např. legislativou). Nakonec je potřeba pravidelně provádět **zálohy dat, kontrolovat stav úložiště, kontrolovat vytížení serveru, ...**

## Pojmy

Přehled pojmu, které by se asi úplně nejsou součástí okruhu, ale mohou se hodit.

## Adůležité vlastnosti u vývoje SW

- Splnění požadavků
- Cena
- Čas

## Typy SW produktu

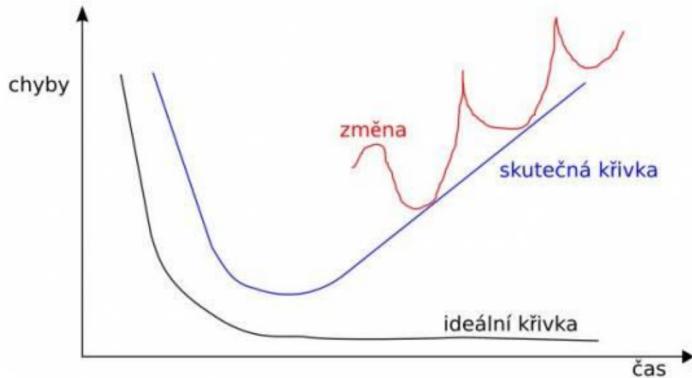
- **Generický software** - SW pro širokou veřejnost, tzv. krabicový SW. Musí být velice dobře otestovaný (Ubuntu, Photoshop, Microsoft Office...).
- **Zákaznický software** - Využíjen pro určitého zákazníka. Většinou pro danou specializovanou oblast neexistuje generický SW. Cena je mnohem větší (menší poptávka). Firma si ho přímo objedná a specifikuje požadavky (IS VUT, FIT KIT).

## Vlastnosti software

- **Použití** - Správnost, spolehlivost, efektivnost, použitelnost, bezpečnost.
- **Přenos** - Přenositelnost, znovupoužitelnost, interoperabilita (spolupráce s jinými systémy).
- **Změny** - Udržovatelnost, modifikovatelnost, testovatelnost, dokumentovanost.

## Problémy při vývoji software

- **Nevyhnutelné** - Složitost, přizpůsobivost (na změny zadání), nestálost, neviditelnost (nevíme přesně jak jsme blízko konci).
- **Ostatní** - Specifikace požadavků (nejasnosti v zadání), náchylnost k chybám (chyby se těžko odhalují), práce v týmu, nízká znovupoužitelnost, dokumentování, stárnutí softwaru (opravou chyb se zanesou nové chyby).



## Modely životního cyklu SW

U každého projektu se etapy mohou lišit, společné rysy je však možné popsat modely životního cyklu SW. Definují etapy a jak po sobě jdou (ale ne však délku ani rozsah). Modely životního cyklu dělíme na **Heavyweight** a **Agilní**.

	Heavyweight	Agilní
přístup	<b>prediktivní</b>	<b>adaptivní</b>
velikost projektu	velká	malá
velikost týmu	velká	malá (kreativní)
styl řízení	centralizovaný příkaz-kontrola	decentralizovaný vedení-spolupráce
dokumentace	<b>velký objem</b>	<b>malý objem</b>
zdůraznění (důraz na)	<b>process-oriented</b>	<b>people-oriented</b>
fixní kritéria	<i>funkcionalita</i>	<i>čas a zdroje</i>
proměnná kritéria	<i>čas a zdroje</i>	<i>funkcionalita</i>

### Heavyweight modely

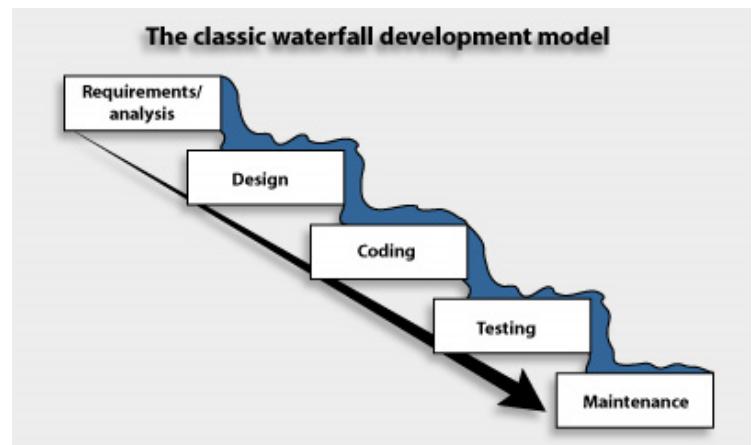
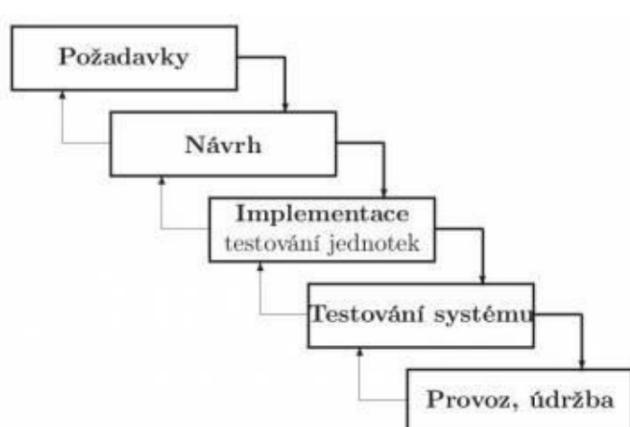
Využívají se většinou u **velkých projektů** s **velkým počtem programátorů**, s **centralizovaným řízením** a **prediktivním přístupem** (velké množství detailního plánování pro dlouhý časový úsek). **Malá flexibilita**, špatná reakce na změny uprostřed vývoje - vyžaduje **stabilní požadavky**. Funkcionalita je daná potřebný čas a peníze je možné měnit. Malé nebo střední zapojení zákazníka při vývoji. Důraz je

kladen na **procesy**, ty musí **běžet neustále** i při výměně zaměstnanců. Zaměstnanci jsou pouze zdroje dostupné v několika rolích (programátor, analytik, tester, ...) a lze je nahradit - **podstatná je role**. Předávání informací je založené především na dokumentaci (aby bylo možné obměňovat zaměstnance, u velkých projektů přes několik let to ale jinak nemusí jít) - **velká míra byrokracie**. Např. projekty NASA.

## Vodopádový model

Etapy jsou řazeny za sebou **sekvenčně**, po skončení jedné začíná další. V případě nalezení chyb je potřeba se vrátit a projít znova. Neodpovídá reálnému vývoji, dnes už se jedná spíše o teoretický model.

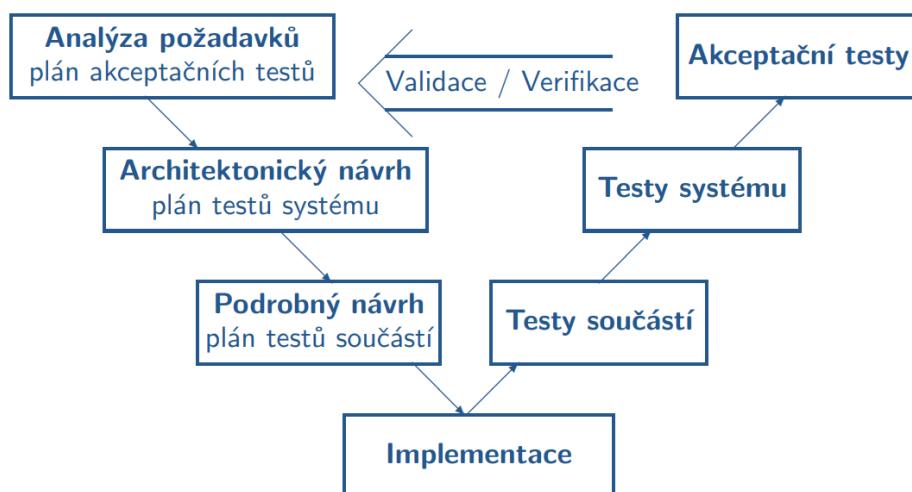
- **výhody:** jednoduchý na řízení - srozumitelný, dobrá výsledná struktura - vše se dělá na poprvé a většinou se nepředělává/nevylepšuje.
- **nevýhody:** k zákazníkovi se SW dostává až na úplném konci vývoje, **problém s validací** a detekováním chyb. Následné opravy mohou být velmi drahé.



## V-model

Vychází z vodopádového modelu, má stejné základní vlastnosti a výhody (zachovává jednoduchost a srozumitelnost vodopádového modelu), snaží se řešit nevýhody, zejména **validaci**. Písmeno V symbolizuje grafické uspořádání etap, ale také zdůrazňuje **validaci a verifikaci**. Dělí se levou a pravou část

- **levá část:** vývojové aktivity a plánování testů,
- **pravá část:** testovací aktivity - provádění testů.



## W-model

Vychází z V-modelu. Druhé V je spojeno s ověřováním jednotlivých aktivit (analýza, architektonický návrh, ...) a s testováním. Opět je model poměrně jednoduchý, ale k zákazníkovi se výsledný SW poprvé dostává až na konci vývoje, problém s validací přetrvává.

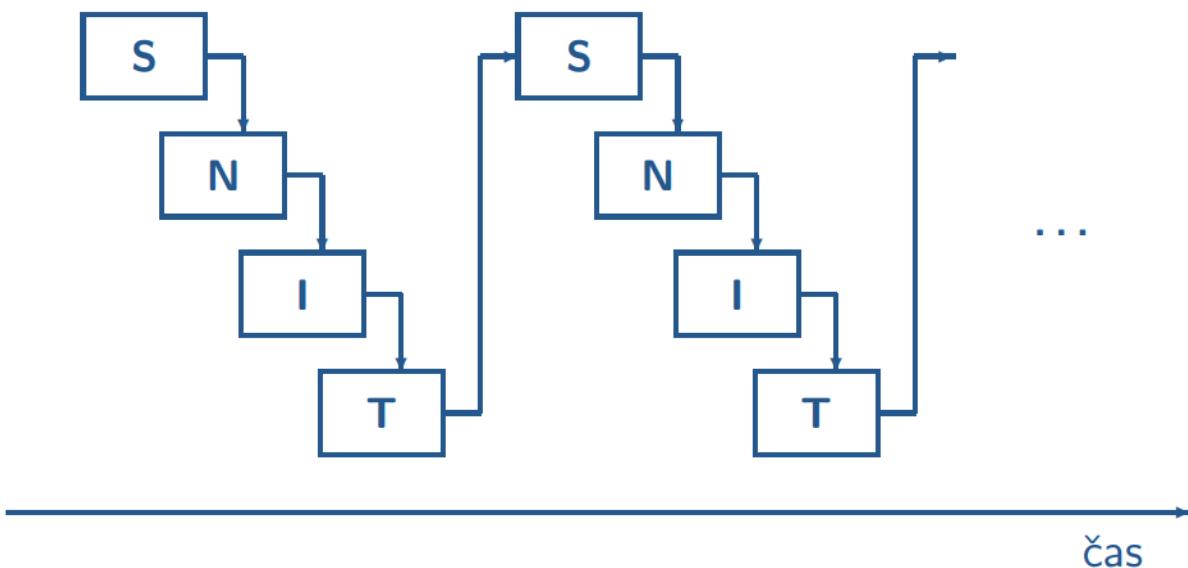
- **levá strana:**
  - **V1:** analýza, specifikace, architektonický návrh, podrobný návrh, implementace,
  - **V2:** ověřování výstupů etap z V1, plánování a návrh testů.
- **pravá strana:**
  - **V1:** provádění navržených testů,
  - **V2:** ladění, oprava chyb, změny v kódu, regresní testování.

## Iterativní model

Proces je rozdělen do **iterací**, kde každá je **instance vodopádového modelu**.

Výsledkem každé iterace je reálný výsledek (rozšíření systému/aplikace o novou funkci, oprava chyb z předešlé iterace).

- **Výhody:** tvorba reálných výsledků v každé iteraci umožňuje zákazníkem výsledek validovat a lze rychleji odhalit chyby ve specifikaci, které mohou být napraveny v další iteraci.
- **Nevýhody:**
  - náročnější na řízení - vyžaduje účast zákazníka,
  - vede na hroší strukturu kódu, z důvodu postupného přidávání funkcí a vylepšení. Lze však eliminovat **refaktORIZACÍ** (změna kódu bez změny jeho funkce - otázka proč to ale platí, když to nic nedělá...)



## Inkrementální model

Jedná se o kombinaci sekvenčních a iteračních metodik softwarového vývoje. Na základě specifikace se stanový **ucelené části systému**, např. **moduly**, které jsou zákazníkovi postupně předávány (možnost validace a oprav). Následně se pracuje jedním ze dvou způsobů:

- pro **každou** (malou) část systému je prováděna **série vodopádů** (iterační přístup) po jejím dokončení je **část systému předvedena zákazníkovi** a až poté se **začíná s další**.
- počáteční analýza, specifikace a návrh jsou provedeny **jedním vodopádem**. Vývoj (implementace) probíhá **iterativně** kombinovaný s **prototypováním**, v každé iteraci je systém rozšířen (vytvořen nový prototyp) a ten je prezentován zákazníkovi, nakonec je poslední prototyp považován za konečný systém.

Zhodnocení:

- **Výhody:** Inkrementální způsob umožňuje postupnou validaci a omezuje projektová rizika.
- **Nevýhody:** Vyžaduje zvýšenou pozornost při návrhu a implementaci rozhraní mezi moduly. Vývoj po částech může vést ke ztrátě vnímání logiky celého celku.

## Spirálový model

Zavádí do vývoje prototypování a klade **důraz na analýzu rizik**. Jednotlivé etapy se opakují vždy na **vyšším stupni zvládnuté problematiky** (analýza, specifikace, arch. návrh, ...), výsledkem je **prototyp**. Spirála je rozdělena do 4 kvadrantů:

1. **Stanovení cílů:** určení **funkcionálních a výkonnostních požadavků**, určení omezujících podmínek (čas a cena), návrh možných alternativ.
2. **Vyhodnocení** stanovených cílů: ověření jejich splnitelnosti, **analýza rizik, prototypování a simulace**,
3. **Realizace** stanovených cílů a jejich testování,
4. **Plánování** následujícího cyklu.

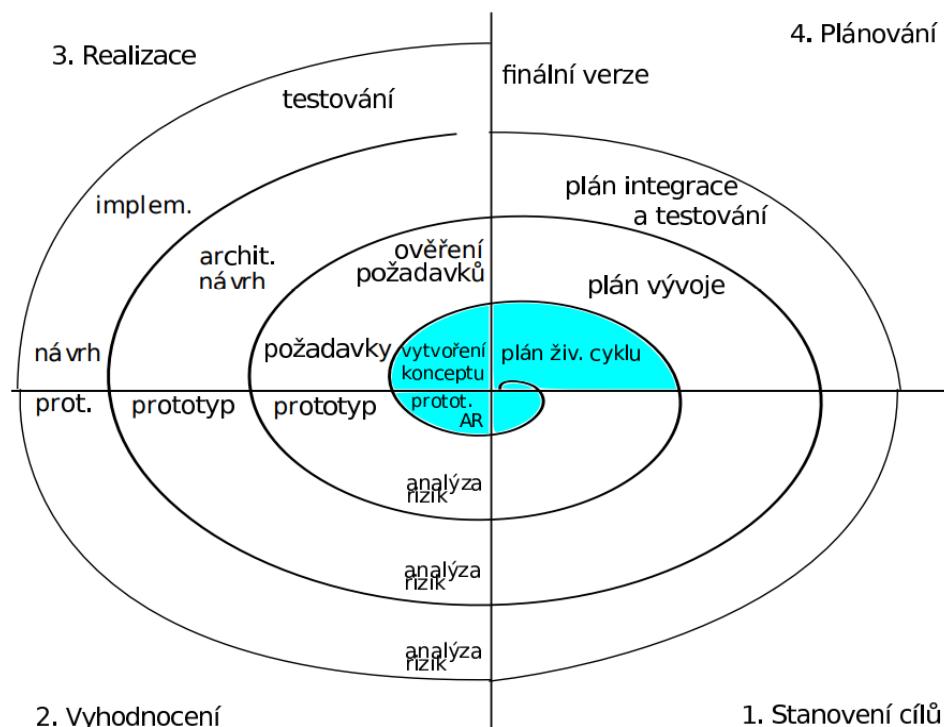
Význam jednotlivých cyklů (jejich počet není pevně stanovený), po každém cyklu je dokončený **Milestone**.

1. globální rizika, základní koncept vývoje, volba metod a nástrojů,
2. tvorba a ověřování specifikace požadavků,
  - a. vyhodnocení záměrů a cílů projektu, rozhodnutí o dalším pokračování. Jsou identifikovány všechny požadavky a jejich chápání je shodné mezi zákazníkem a dodavatelem.
  - b. Je vytyčena cena, plán priority aj.
  - c. Jsou identifikovány rizika a postupy pro jejich zmírnění.
3. vytvoření a ověření návrhu,
  - a. Vyhodnocení výběru architektury, požadavky a architektura jsou stabilní.
4. implementace, testování a integrace (vodopádový přístup)

- a. Je vytvořena stabilní verze schopná testování u zákazníka, pokud nejsou odhaleny problémy, je systém je připravený na distribuci pro uživatele.
- b. Porovnání plánovaných a skutečných výdajů.

Zhodnocení:

- Výhody:
  - Vhodný pro složité a velké projekty,
  - chyby a nevhovující postupy jsou odhaleny dříve pomocí analýzy rizik.
- Nevýhody:
  - Analýza rizik musí být na vysoké úrovni, jinak postup může selhat
  - výsledný SW je k dispozici až po posledním cyklu (problém s validací)



## Analýza rizik

Zjištění možných ohrožení průběhu projektu a připravení reakce na tato ohrožení, lze včas vyloučit nevhodná řešení.

- **projektová:** odchod lidí, snížení rozpočtu,
- **technická:** neznámé technologie, selhání HW,
- **obchodní:** špatný odhad zájmů,

## Rational Unified Process (RUP)

Nejedná se o model životního cyklu SW ani o konkrétní metodiku vývoje SW, jde o **rozšiřitelný framework**, který je možné **uzpůsobit organizaci a příslušnému projektu**. Jedná se o **komerční produkt** (firma Rational Software), který je dodávaný společně s patřičnými nástroji. Je založený na osvědčených praktikách:

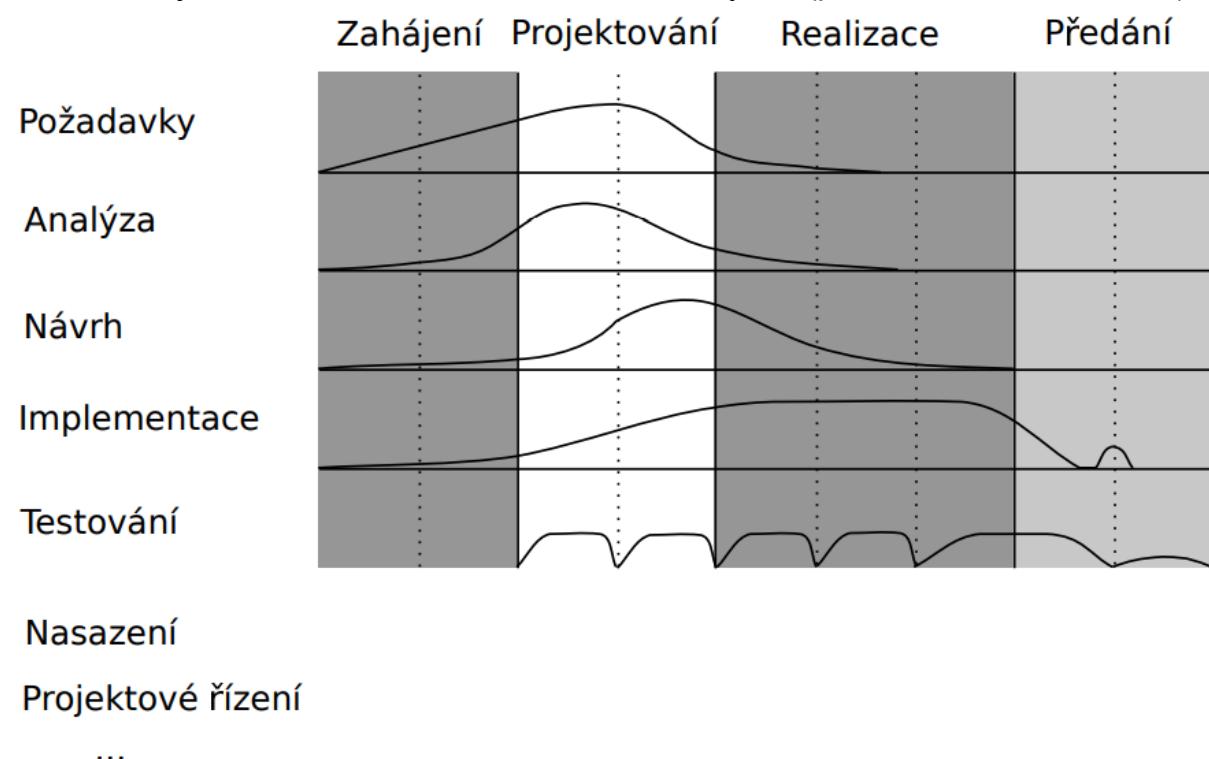
- **iterativní vývoj**, délka iterace je 2 až 6 týdnů,

- **vizualizovaný návrh systému** - využití UML a jiných nástrojů,
- **průběžná kontrola kvality**,
- **řízení změn**,
- **využití existujících komponent**.

Vývoj je rozdělen do několika cyklů. První nejdůležitější je **Initial Development Cycle** jehož výsledkem je funkční SW. Následný další vývoj (vylepšení, opravy) probíhá formou **Evolution Cycles**. Základní cyklus (Initial Development Cycle) je rozdělen na 4 fáze:

1. **Zahájení** (asi 10%) 1 až 2 iterace,
2. **projektování** (asi 30%) 2 až 4 iterace,
3. **realizace** (asi 50%) 2 až 4 iterace,
4. **předání** (asi 10%) aspoň 2 iterace (beta verze, plná verze).

Každá fáze je rozdělena na iterace o délce 2 až 6 týdnů (počet iterací se může lišit).



Ze spirálového modelu přebírá mezníky/milníky - **Milestones**.

Zhodnocení:

- **Výhody:**
  - robustní, lze upravit pro potřeby projektu,
  - iterativní přístup, včasné odhalení rizik,
  - grafický návrh - vazba na UML,
  - detailní propracovanost
- **Nevýhody:**
  - u malých projektů může představovat příliš velkou zátěž - neefektivní vývoj.
  - komerční produkt

## Agilní metodiky

Využívají se většinou u **malých projektů s malým počtem programátorů**, s **decentralizovaným řízením** (vedení založené na spolupráci) a **adaptivním přístupem** (menší množství plánování, úpravy dle reakcí zákazníka) - **velká flexibilita**. Čas a peníze jsou obvykle dané a funkcionality se implementuje na základě toho (co se stihne, co bude zaplaceno). Vyžadují **intenzivní zapojení zákazníka** do procesu vývoje. Agilní metodiky kladou důraz na lidi a jejich **individualitu** (people oriented), člověk je v dané oblasti profesionál (analytik, programátor, tester) a je schopen rozhodovat technické otázky práce, velmi důležitá je **komunikace v rámci týmu**. Omezuje se **byrokracie** a formální požadavky. Ověření správnosti je prováděno **zpětnou vazbou** od zákazníka. Odchod schopných zaměstnanců může být kritický. Agilní metodiky dobře reagují na změny v průběhu vývoje, což je běžné.

### Rapid Application Development

Metodika založená na **rychlém iterativním vývoji prototypů**. Brzská dispozice funkčních verzí, vyžaduje **intenzivní zapojení zákazníka** do vývojového procesu - **zpětná vazba** na poskytnuté verze. Zaměřuje se zejména na **splnění potřeb a požadavků zákazníka** (business potřeb SW). Metodika je určena pro menší a středně velké týmy. Zhodnocení:

- **Výhody:** flexibilita a schopnost rychlé reakce na změny požadavků od zákazníka. Vyšší kvalita zpracování business potřeb (základník dostává doopravdy to, co chce). Více projektů splňuje termíny a ceny.
- **Nevýhody:** nižší kvalita návrhu a výsledného kódu způsobená změnami, což vede na problémy s udržovatelností.

### Extrémní programování (XP)

Populární agilní metoda, která je založena na **komunikaci** (v týmu a se zákazníkem, zákazník se prakticky stává členem týmu) a **iterativním vývoji**. SW není dodán zákazníkovi celý v určitý termín v budoucnosti, ale je dodáván způsobem, který odráží aktuální potřebu uživatelů - **do systému vložíme to, co potřebujeme, když to potřebujeme**. Dva významní členové týmu:

- **kouč:** zajišťuje komunikaci v týmu, pomáhá programátorům s technickými dovednostmi, komunikuje s velkým šéfem a vyšším managementem.
- **velký šéf:** provádí zásadní rozhodnutí, komunikuje s vyšším managementem.

Extrémní programování klade velký důraz na **spolupráci v týmu**, manažeři, vývojáři a zákazník jsou si rovní. Je založeno na 5 základních principů:

1. **komunikace:** programátoři, zákazníci a manažeři spolu musí komunikovat - využití technik, které komunikaci vyžadují (pair programming, code review, ...),
2. **jednoduchost:** jednoduché věci se realizují a upravují s menším počtem chyb (v případě potřeby je lze rozšířit). Přírůstkové malé změny, uvolňování

malých verzí systému (nejpodstatnější požadavky, které jsou postupně vylepšované a doplňované)

3. **zpětná vazba a testování:** vše musí být otetováno, ke každé funkci píšeme testy, klidně i před tím, než začneme programovat (Test Driven Development). Jednotkové i integrační testy, zpětná vazba zákazníka.
4. **odvaha:** nebát se zahodit naprogramovaný kód, nebát se zkusit neznámé. Pokud je potřeba, nebát se provést zásadní změny.
5. **respekt:** Každý malý úspěch prohlubuje respekt k jedinečným příspěvkům každého člena týmu.

Zhodnocení:

- **Výhody:**
  - iterativní inkrementální proces,
  - ladění na základě zpětné vazby - zapojení uživatelů,
  - založený na testování,
  - průběžná integrace
- **Nevýhody:**
  - vyžaduje dodržování základních principů (neustálé psaní testů, párové programování, atd)
  - nedefinuje přesný postup.

## Collective-Code-Ownership

Každý člen týmu má **možnost a povinnost ovlivňovat kód** (přidání nových funkcí, odstranění chyb, refaktorizace), což snižuje riziko, že výpadek člena týmu výrazně zbrzdí práci a podporuje pocit odpovědnosti vývojáře za kvalitu kódu (kdokoliv to po něm bude čist). Vyžaduje **jednotný styl programování**.

## Průběžná integrace

Automatizované sestavování, automatizované spouštění napsaných testů, automatizovaná publikace na testovací prostředí testování.

## Scrum

Agilní metoda odvozená ze hry rugby, lze kombinovat s XP. Dělí se na tři základní fáze:

- **pre-game:** plánování a **architektonický návrh**, využívá se Product Backlog.
- **game:** vývoj, který probíhá v iteracích, iterace se nazývá **Sprint** a tvá okolo **30 dní**, výsledkem každé iterace je funkční inkrement. Na začátku každého sprintu je setkání mezi vývojáři, zákazníkem, uživateli atd. a definují se cíle sprintu - **Sprint Backlog** (seznam úloh nutných pro dosažení cíle). Každý den v průběhu sprintu se provádí **Scrum Meeting** - 15 min setkání členů vývojového týmu. **Scrum Master** je vedoucí týmu. Na konci sprintu se provádí **Sprint Review** - vyhodnocení proběhlého sprintu.
- **post-game:** integrace výsledků jednotlivých sprintů, testování (integrační, celého systému, akceptační), dokumentace, školení uživatelů.

Zhodnocení:

- **Výhody:**
  - iterativní inkrementální proces,
  - časté uvolňování verzí,
  - architektura je navržena před procesem vývoje,
  - jednoduchý proces,
  - zapojení uživatelů.
- **Nevýhody:**
  - nedefinuje přesný postup,
  - integrace až po vytvoření všech inkrementů.

## Crystal

Jedná se o rodinu metodologií, základní přístupy a techniky sdílí s **XP**. Více lidí je ale schopno tento proces akceptovat. Rozděluje projekty do **kategorií dle kritičnosti a důležitosti**:

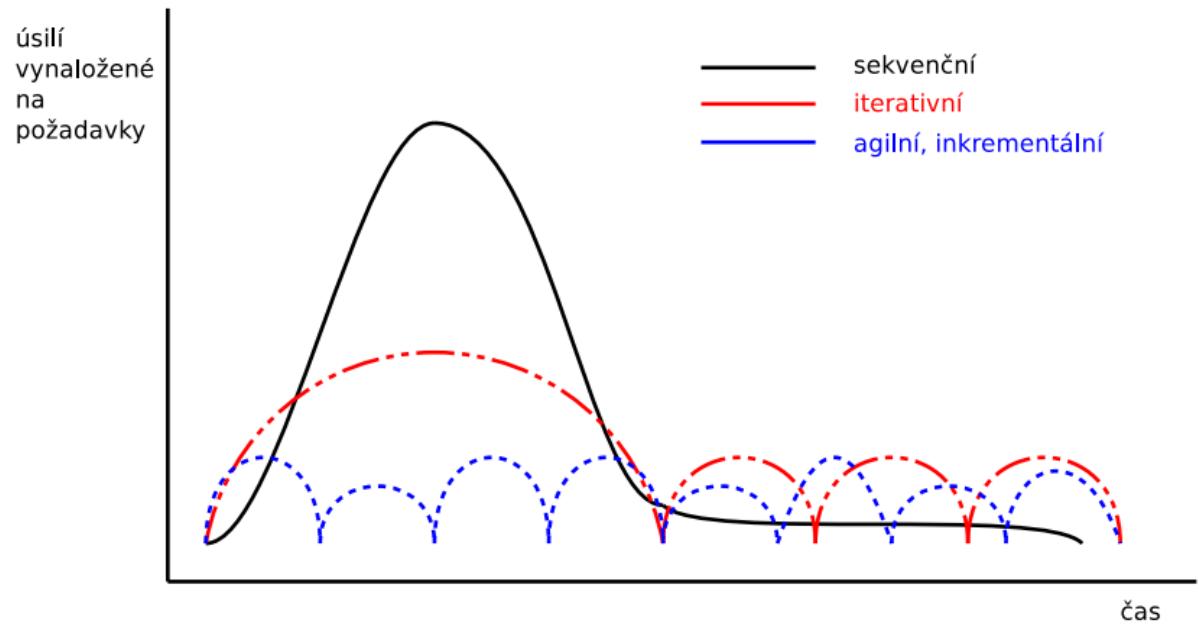
- Comfort **C**,
- Discretionary Money (volné uvážení) **D**,
- Essential Money **E**,
- Life **L**.

kategorie metodik, které se používají se označují barvou, liší se dle velikosti týmu a délce projektu



Zhodnocení:

- **Výhody:**
  - iterativní inkrementální proces - časté uvolňování verzí,
  - průběžná integrace,
  - zapojení uživatelů - požadavky se ladí během celého vývoje na základě zpětné vazby
- **Nevýhody:**
  - nedefinuje jasný společný proces,
  - není vhodný pro vysoce kritické projekty,
  - velká závislost na přímé komunikaci.



# 34. Jazyk UML

Unified Modeling Language je v softwarovém inženýrství **standardizovaný grafický jazyk** pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. UML nabízí standardní způsob zápisu a podporuje **objektově orientovaný přístup k analýze**. Základním cílem je **vizualizace, specifikace struktury a chování navrhovaného systému**. Napomáhá k dekompozici systému. Umožňuje modelovat **business procesy**, konkrétní **příkazy programovacího jazyka**, vztahy mezi třídami v OOP. UML diagramy dělíme do dvou skupin (respektive do 3, **ERD do UML NEPATŘÍ**):

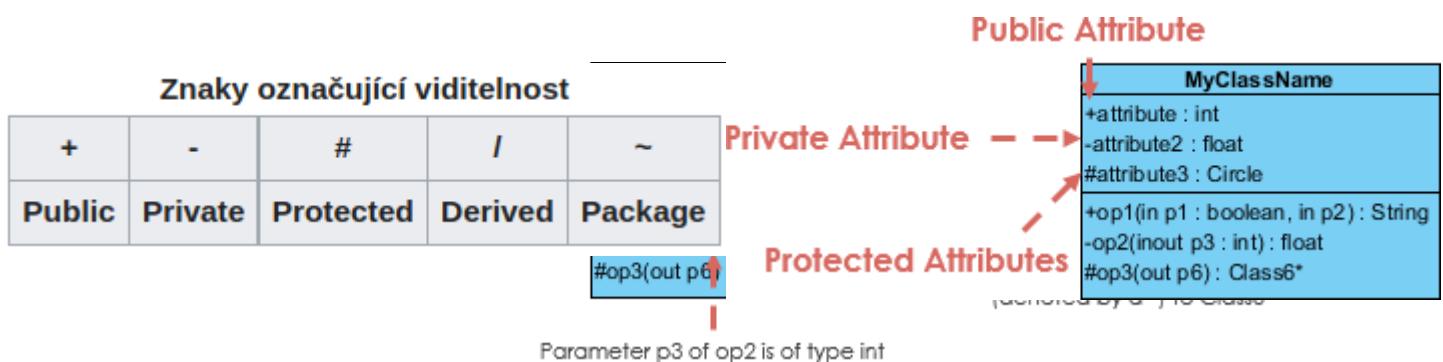
- **Structure diagrams** (Diagramy struktury)
- **Behaviour Diagram** (Diagramy chování)
- **Interaction diagrams** (Diagramy interakce)

## Diagramy struktury (Structure diagrams)

Popisují strukturu systému, tedy **z čeho je systém složený**. Představují **statické aspekty** systému. Zdůrazní části/komponenty, které musí být přítomny v modelovaném systému. Jsou široce používány při **dokumentaci architektury softwarových systémů**. **Diagram komponent** například popisuje, jak je softwarový systém rozdělen na **komponenty** a ukazuje **závislosti** mezi těmito komponentami.

### Diagram tříd

Zobrazuje **třídy a statické vztahy** mezi nimi. Popisuje **statickou strukturu systému** a znázorňuje datové struktury. Každá třída je tvořena **jménem, seznamem atributů a seznamem operací**. Atributy a operace (metody, funkce) mají stanovenou **viditelnost**. Atributy mají specifikovaný **typ** a operace mají **vstupní parametry a výstupní typy**. Diagramem tříd se je popisován **doménový model**.



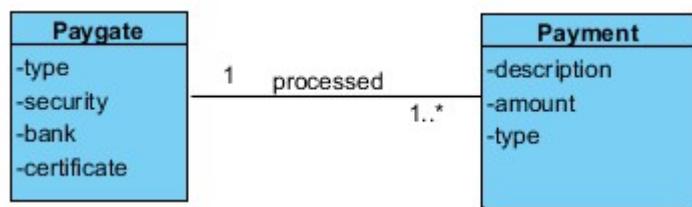
### Mohutnost (Multiplicity)

Mohutnost vztahu udává, kolik instancí jedné třídy může být svázáno s instancí třídy druhé

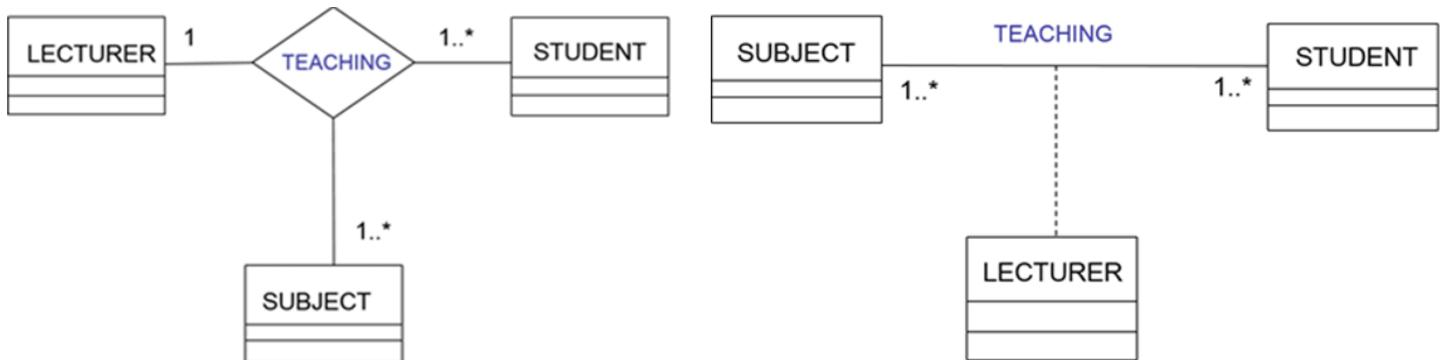
0	0..1	1	0..*	1..*
Žádná instance (zcela výjimečně)	Žádná nebo právě jedna	Právě jedna instance	Žádná nebo více instancí	Jedna nebo více instancí

## Asociace

Asociace určuje **základní vztah** mezi dvěma entitami. Ty mohou existovat nezávisle na sobě. Zakreslujeme ji jednoduchou plnou čáru. Mohou mít název (sloveso, podstatné jméno). Asociace mohou být **binární** (unární je také brána jako binární ale **reflexivní**), **ternární** a **n-ární**. Jiné než binární asociace není moc doporučené používat (nepřehledné, obvykle lze převést na binární nebo na asociační třídu).

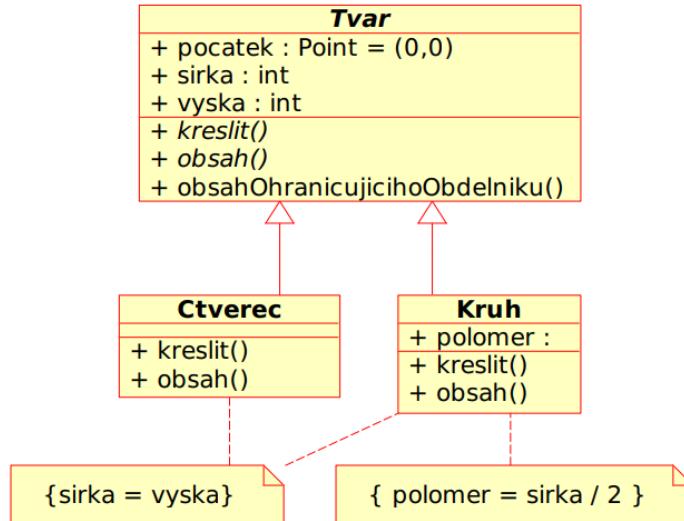


Převod ternární asociace na binární s asociační třídou:



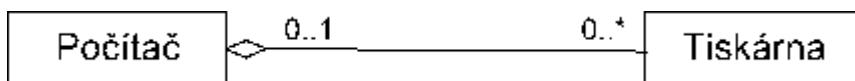
## Dědičnost (Generalization)

Vyjadřuje vztah generalizace/specializace mezi třídami. Odvozená třída sdílí **atributy** (dle viditelnosti) **chování** (operace), **vztahy** a **omezení** obecnější třídy. Odvozená třída může navíc přidávat a **modifikovat** atributy a chování (operace).



## Agregace (Aggregation)

Agregace představuje **volnou vazbu** mezi celkem a součástí, kdy jeden objekt (**celek**) využívá služby dalších objektů (**součástí**). Například vztah mezi počítačem a tiskárnou je vztah typu **agregace**, kdy počítač s tiskárnou tvoří jeden celek, ale **tiskárna může existovat i bez počítače**. Dokonce může být **součást** (tiskárna) využita i jiným **celkem** (např. mobil) - může být součástí více kolekcí. Agregace je **formou asociace** a v grafické podobě se odlišuje **prázdným kosočtvercem** na straně celku.



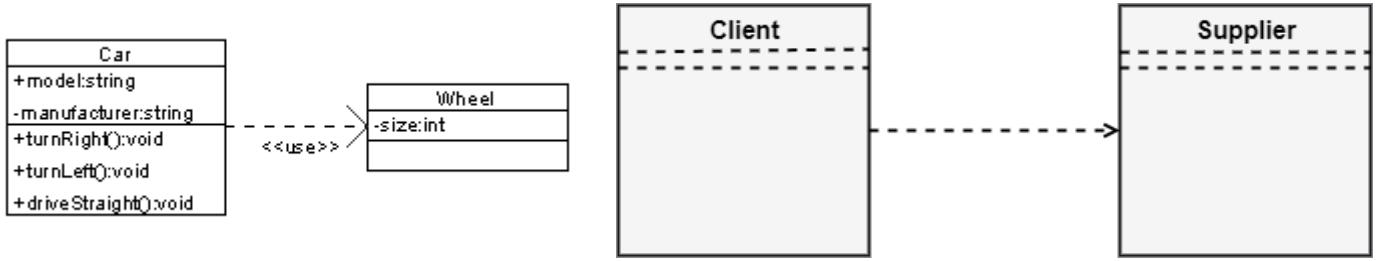
## Kompozice (Composition)

Kompozice je podobná agregaci, avšak reprezentuje **silnější vztah**. Entita části (součást) **nemá bez celku smysl**, nemůže bez celku existovat. Pokud zanikne celek, zanikají automaticky i jeho části. Příkladem kompozice je vztah mezi fakturou (**celek**) a jejími položkami (**součásti**). Každá položka **musí být součástí právě jedné faktury**, faktura má jednu a více položek. Jestliže fakturu zahodíme, nezbudou nám po ní ani žádné položky. **Kompozice je formou asociace** a v grafické podobě se odlišuje **plným kosočtvercem** na straně celku.



## Závislost (Dependency)

Závislost je slabší forma vztahu, která naznačuje, že **jedna třída závisí na jiné**.



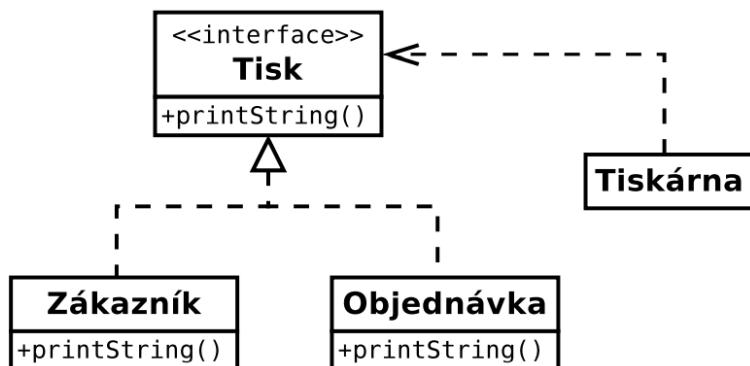
Závislost bývá doplněna o **stereotyp**, který ji blíže specifikuje. Nejčastěji se používá stereotyp **<<use>>**. Pokud není uveden, jedná se automaticky o stereotyp **<<use>>**.

Další stereotypy na příkladu klient/dodavatel:

- **<<initiate>>** / **<<create>>**: klient vytváří instance dodavatele,
- **<<send>>**: operace klienta zasílá signál příjemci (dodavateli),
- **<<call>>**: klientská třída volá operaci dodavatele,
- **<<trace>>**: klient realizuje dodavatele,
- **<<refine>>**: klientská třída poskytuje detailnější informace než dodavatel.

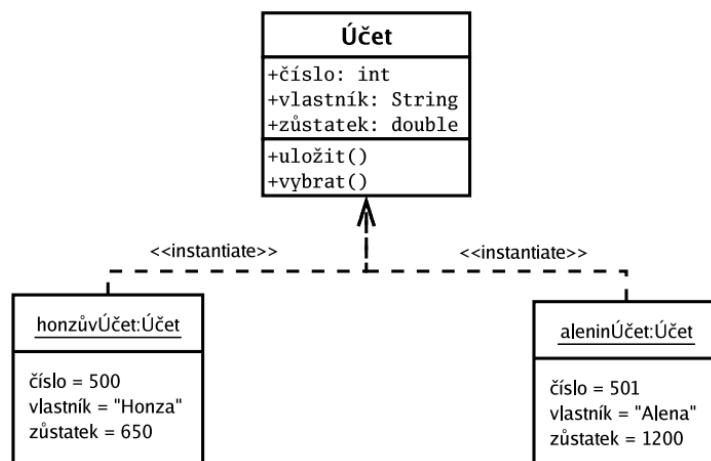
## Realizace

Udává vztah mezi **třídou a rozhraním** (interface). Označuje fakt, že třída **implementuje všechny operace z daného rozhraní** (respektive implementuje rozhraní). Objekt používající rozhraní pak umí používat i tuto (implementační) třídu.

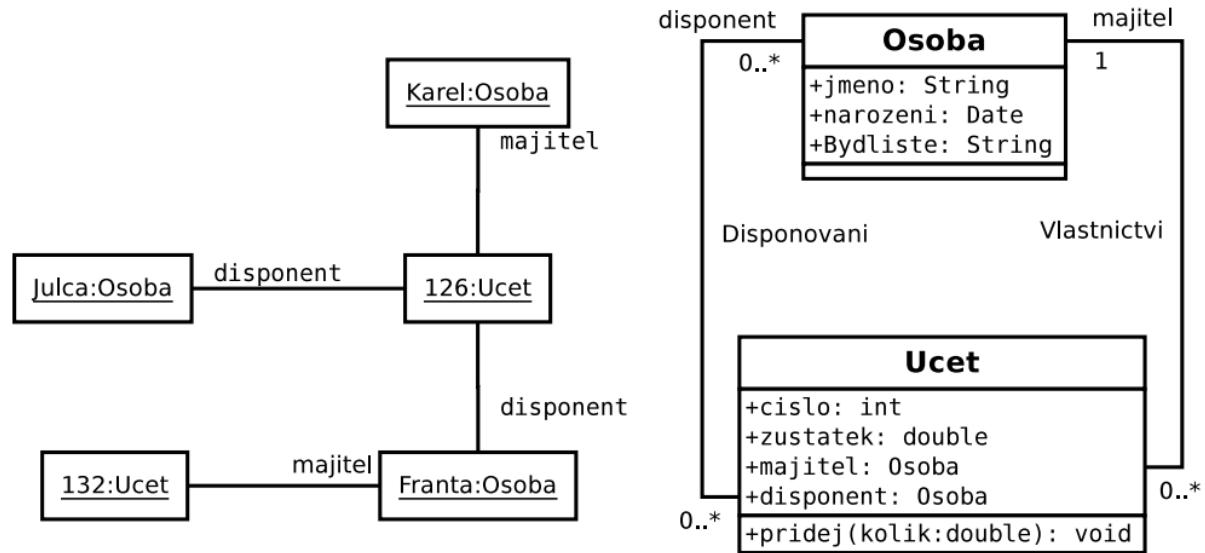


## Diagram objektů (Object Diagram)

Je úzce svázán s diagramem tříd, **znázorňuje objekty a jejich relace** (vztahy) v určitém čase. Relace jsou **dynamické** (nemusí trvat po celou dobu existence objektů).

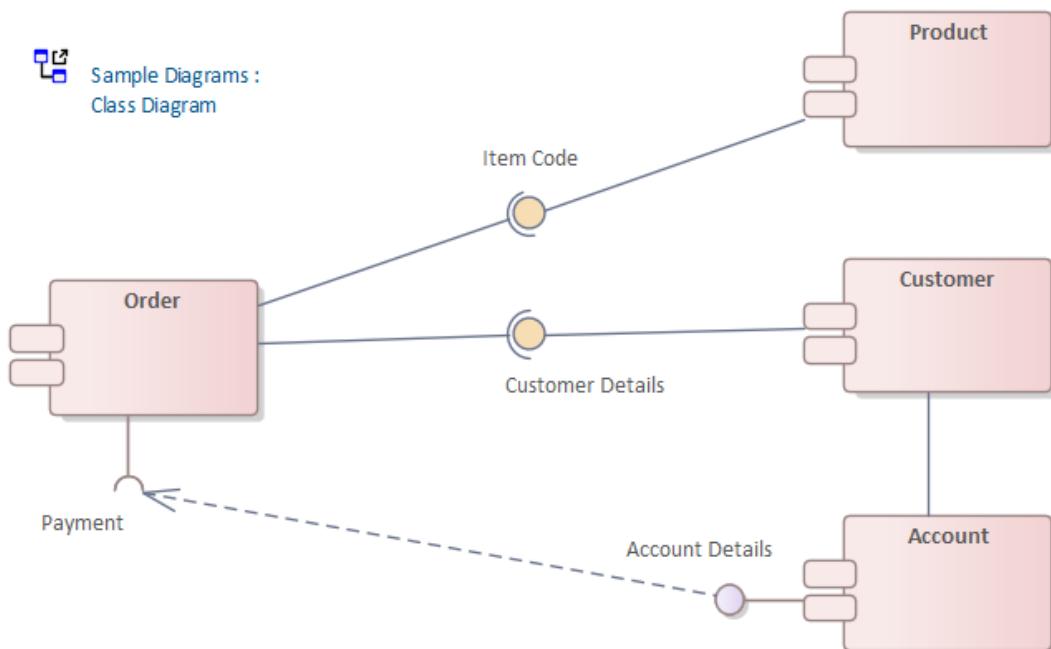


Při vytváření objektů dochází k vytváření **spojení** mezi nimi, spojení jsou **instance asociací**.



## Diagram komponent

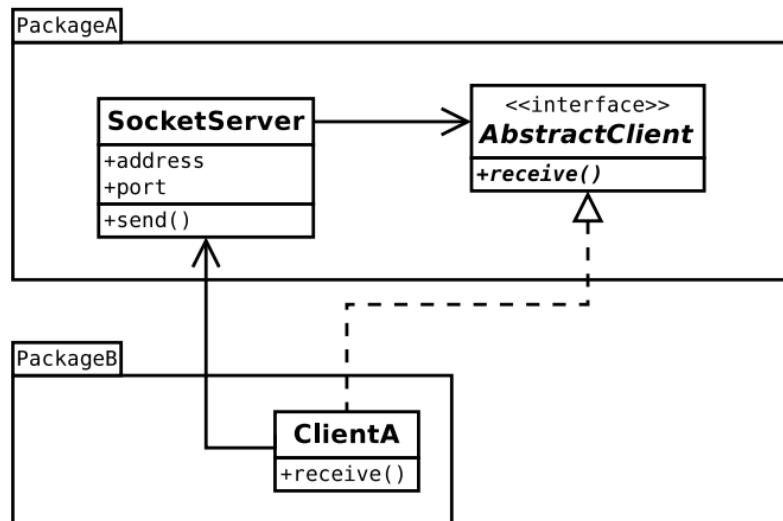
Diagram komponent znázorňuje **komponenty** použité v systému, tím mohou být **logické komponenty** (např. business komponenty, procesní komponenty) nebo také **fyzické komponenty**. Popisuje, jak je softwarový systém rozdělen na komponenty a ukazuje **závislosti** mezi těmito komponentami.



## Diagram seskupení

Umožňuje **seskupit sémanticky související elementy**, umožňuje **zapouzdřit prostor jmen**. Definuje sémantické hranice modelu a umožňuje souběžnou práci v

etapě návrhu. Např. třídy a rozhraní řešící komunikaci dáme do jednoho balíčku a třídy obsluhující klienta do druhého.



## Diagramy chování (Behaviour Diagram)

Popisují **chování systému**, tedy jak systém funguje. Diagramy chování představují **dynamický aspekt** systému. Zdůrazňují, co se může stát v modelovaném systému. Jsou široce používány k popisu **funkčnosti softwarových systémů**. Diagram případu užití například zobrazuje uživatele v nějaké roli a operace, které může provádět.

### Diagram případů užití

Diagram případů užití zachycuje vnější pohled na modelovaný systém, specifikujeme pomocí něj **účastníky** a způsoby, jak budou modelovaný **systém používat** - případy užití. Používá se v Use-Case driven přístupech (metodika RUP). Prvky diagramů užití:

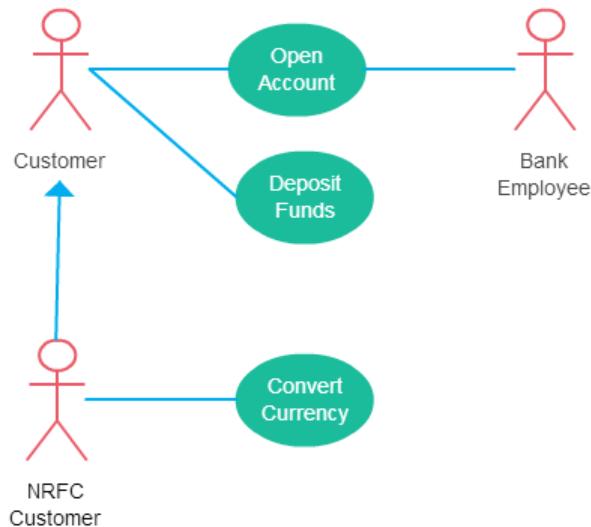
- **hranice systému**,
- **účastník (aktér/actor)**: subjekt (většinou uživatel, může mít i speciální podobu, např. čas, jiný systém), který se systémem pracuje,
- **případ užití**: funkce, kterou systém vykonává jménem (akcí) jednotlivých účastníků nebo v jejich prospěch,
- **interakce**: ukazuje účast autora (účastníka) na provádění případu užití.

### Asociace mezi účastníkem a případem užití

Účastník musí být pomocí asociace spojen aspoň s jedním případem užití. Více účastníků může být asociováno s jedním případem užití (viz obrázek na konci kapitoly).

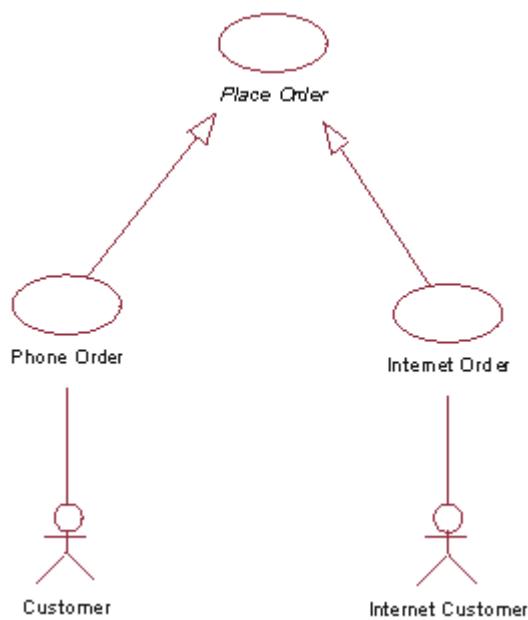
## Generalizace účastníka (dědičnost)

Zobecnění účastníka znamená, že jeden účastník může **zdědit roli druhého**. Potomek **zdědí všechny případy použití předka**. Potomek má **navíc** jeden nebo více případů použití, které jsou pro něj specifické (pro danou roli) a předek ji nemá.



## Generalizace případu užití

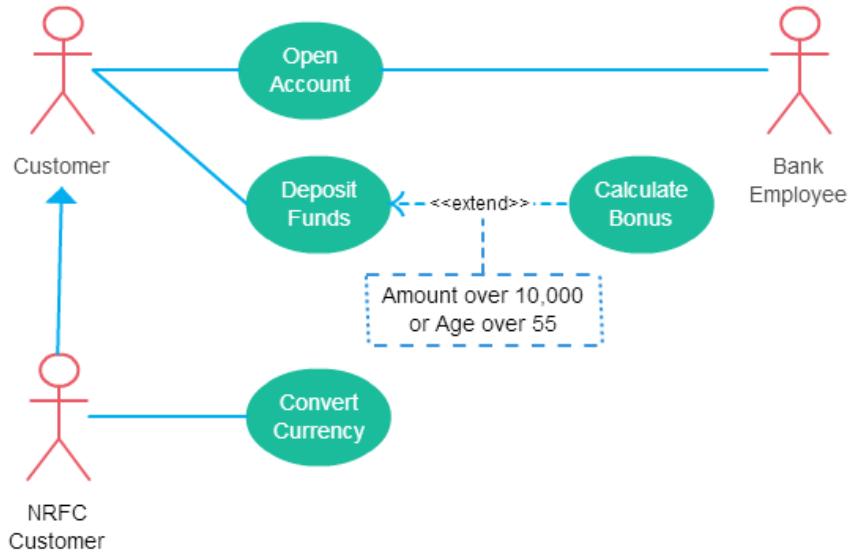
Je to podobné jako zobecnění účastníka. Chování předka dědí potomek. Používá se, když existuje **společné chování mezi dvěma případy užití** a také **specializované chování specifické pro každý případ použití**. Viz příklad na obrázku.



## Extend mezi dvěma případy užití

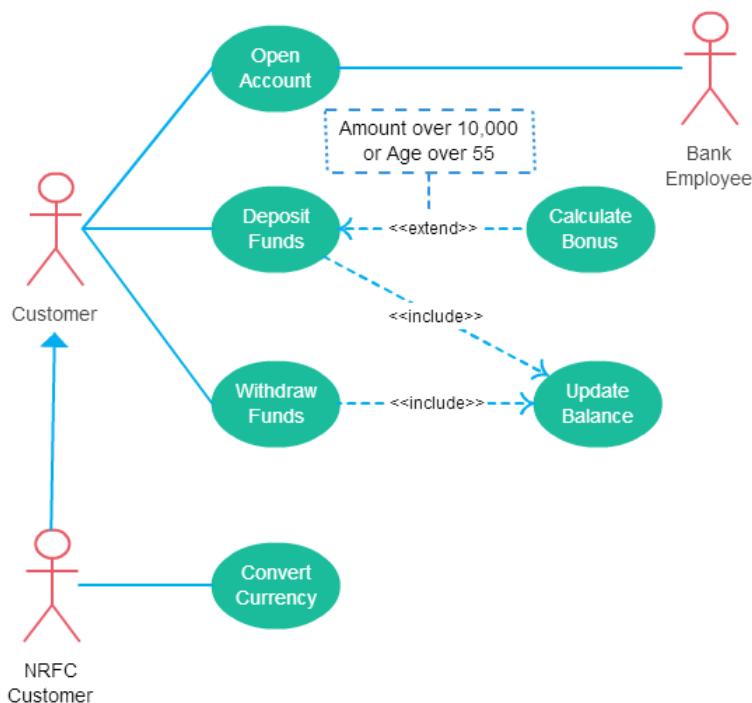
Vazba extend rozšiřuje funkcionality daného případu užití (přidává do systému další funkcionality). Případ užití vázaný pomocí extend (rozšiřující případ užití) se **může provést ale nemusí** (záleží na okolnostech, případně se může provádět pouze podmíněně). Rozšiřující případ užití je **závislý** na rozšiřovaném případu užití

(sám o sobě nedává smysl). Naopak **rozšířovaný případ užití** musí dát sám o sobě **smysl**.



### Include mezi dvěma případy užití

Zahrnutý/přiložený (included) případ použití **je povinný a není volitelný**. Základní případ použití je **bez přiloženého případu použití neúplný**. Používá se v případě, že je nějaká funkcionálnita **důležitá natolik**, že nemůže být součástí nějakého případu užití. Chceme jí tímto zdůraznit.

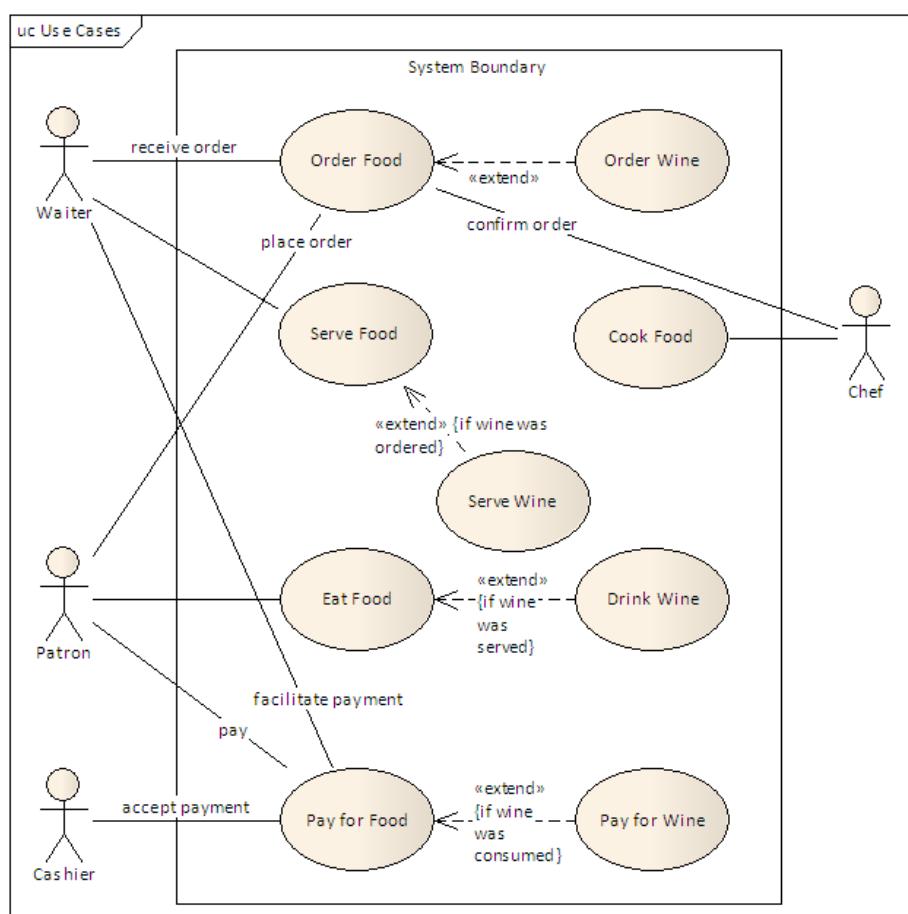


### Detailly případů užití

Slouží pro konkretizaci use case. Často se používá **tabulka** - má vstupní podmínky a tok událostí.

název identifikátor	<b>Případ užití: Platit daň z přidané hodnoty</b>	
účastníci	<b>ID: UC1</b>	
stav před	<b>Účastníci:</b> Čas finanční úřad	
kroky	<b>Vstupní podmínky:</b> 1. Je konec fiskálního čtvrtletí? <b>Tok událostí:</b> 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	<b>Následné podmínky:</b> 1. Finanční úřad přijímá daň z přidané hodnoty.	

## Příklad diagramu užití



## Diagram aktivit (Activity diagram)

Reprezentuje objektově orientovaný **vývojový diagram**. Umožňuje modelování:

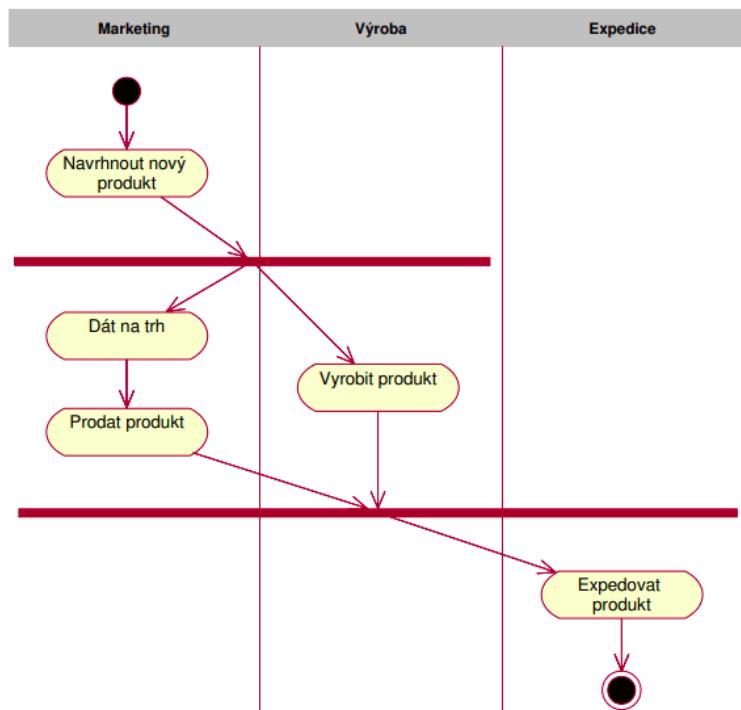
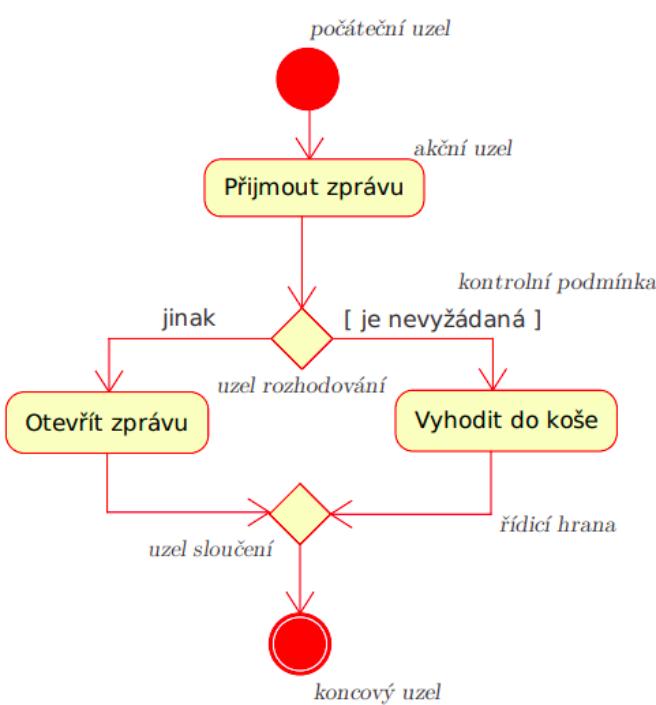
- scénářů případů užití,
- detailů **algoritmů** a operací (funkcí),
- modelování **obchodních procesů**.

Prvky diagramu:

- **uzly**:
  - **akční uzly**: modelují aktivitu,
  - **řídící uzly**: modelují rozhodování, např. počáteční uzel, koncový uzel,
  - **objektové uzly**: modelují objekty podílející se na aktivitách.
- **hrany**:
  - **řídící hrany**: modelují přechody mezi uzly,
  - **objektové hrany**: modelují cesty objektů mezi uzly.

### Možnosti modelování

- tok událostí a dat,
- rozhodování,
- větvení a slučování,
- iterace,
- paralelní toky.



### Stavový diagram

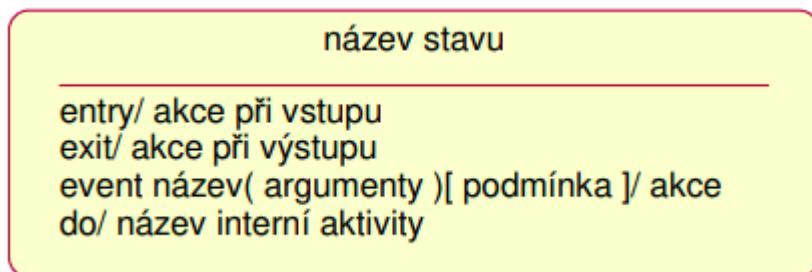
Umožňují modelování životního cyklu jednoho reaktivního (měnící se) objektu. Je to zvláštní případ stavového automatu. Mohou také modelovat dynamické chování těchto objektů:

- třídy, respektive instance tříd (objekty),
- případy užití,
- podsystémy,

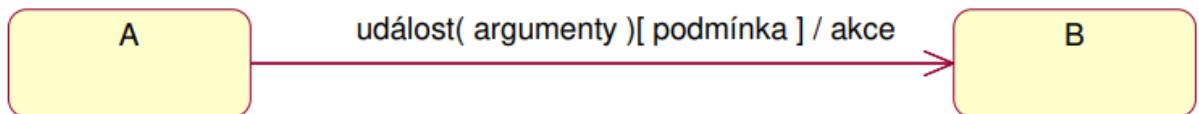
- systémy.

Je tvořen:

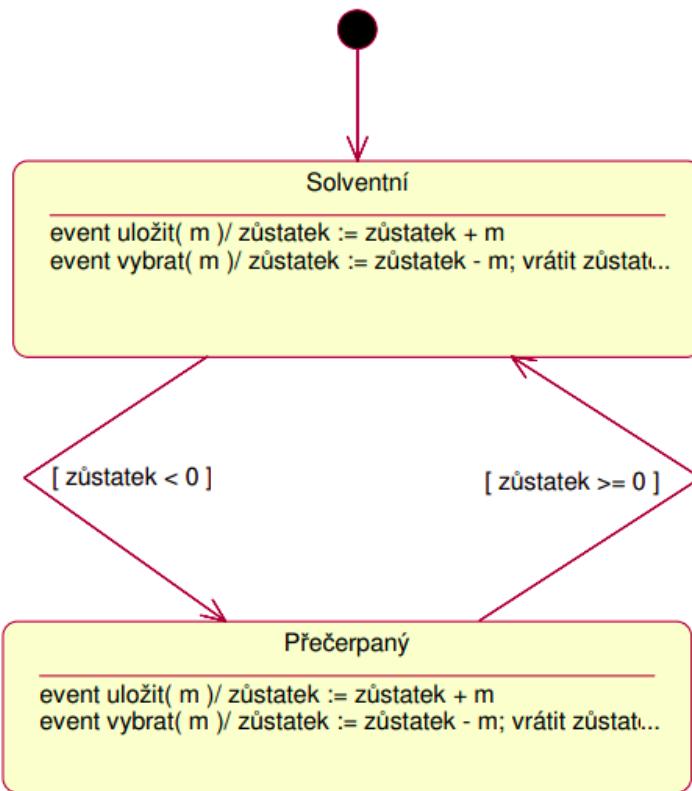
- **stavy**,



- **přechody**,



- **události**: udávají, kdy dojde k přechodu z jednoho stavu do druhého. Události mohou být i ve stavech.



## Reaktivní objekt

- reaguje na vnější události,
- životní cyklus je modelován jako řada stavů, přechodů a událostí,
- chování je důsledkem předchozího chování, následující stav závisí na předchozím stavu.

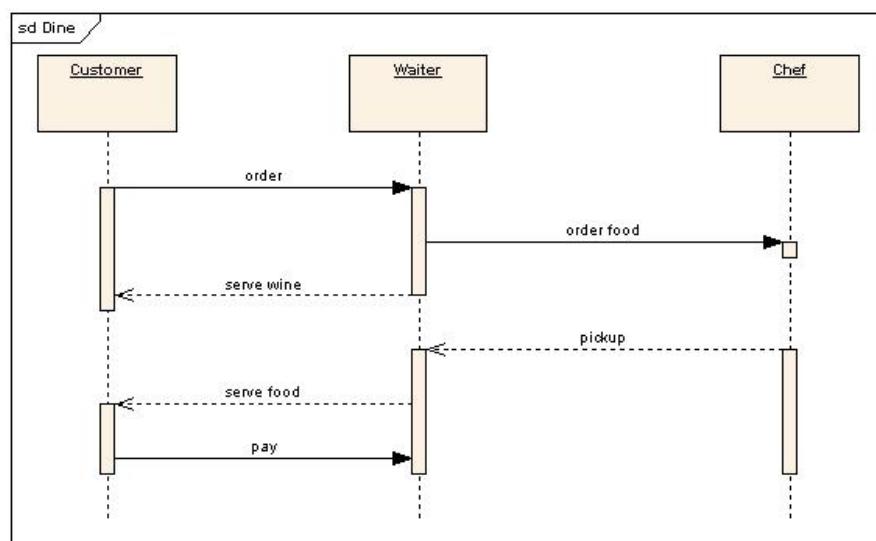
## Diagramy interakce (Interaction diagrams)

Jedná se o **podskupinu** diagramů chování. Popisují **interakci mezi jednotlivými částmi systému** (včetně uživatele). Zdůrazňují **tok řízení** a dat mezi částmi v modelovaném systému. Sekvenční diagram například ukazuje, jak spolu objekty komunikují pomocí **sekvence zpráv**.

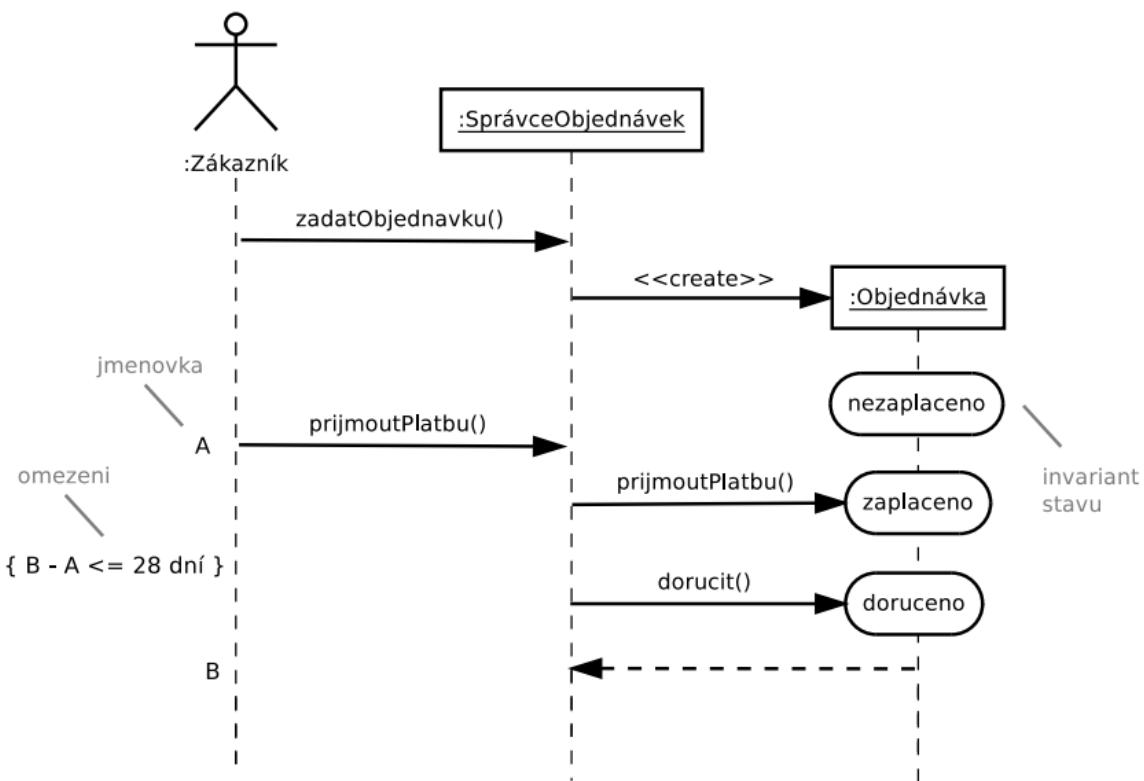
### Sekvenční diagram

Zdůrazňuje časově orientovanou posloupnost předávání zpráv mezi objekty - chronologické zaslání zpráv. Sekvenční diagramy jsou většinou přehlednější a srozumitelnější než diagramy komunikace. Jsou tvořeny:

- **čarami života**: zobrazuje časovou osu určitého objektu,
- **vodorovné šipky**: reprezentují zprávy posílané mezi objekty, existuje více typů.



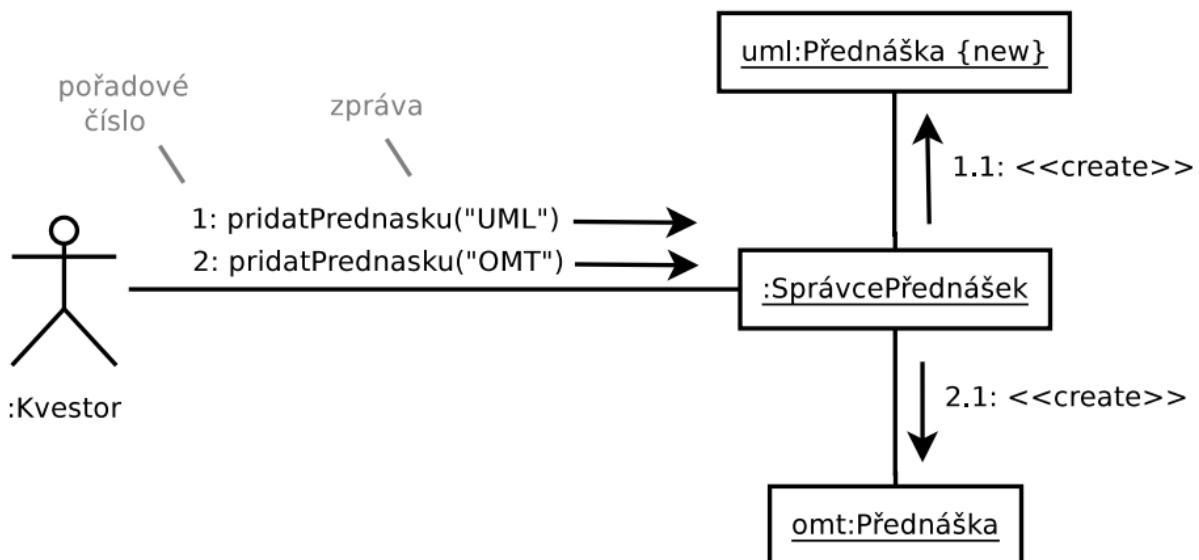
Sekvenční diagram může být například rozšířen pomocí **omezení**, **zobrazení stavů** atd.



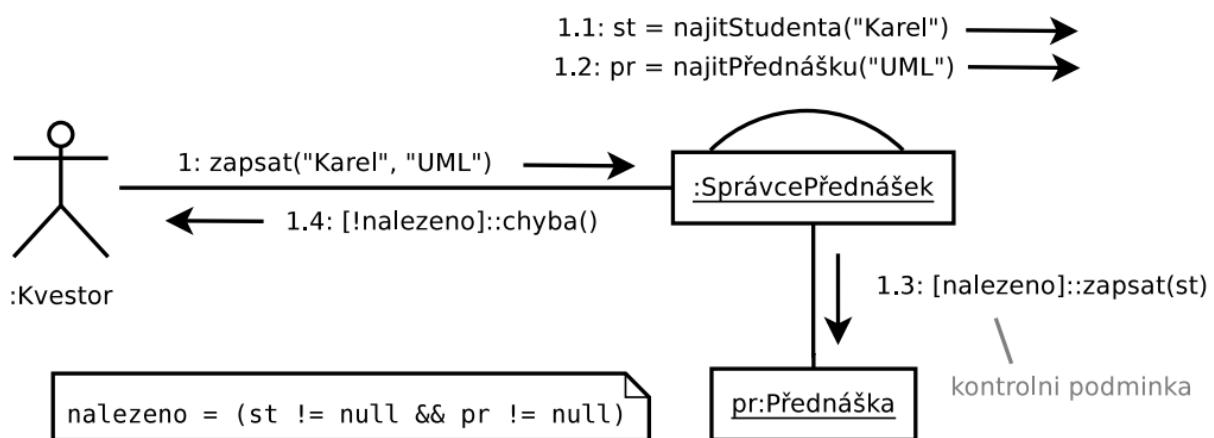
### Diagram komunikace

Zdůrazňují strukturální vztahy mezi objekty, jsou vhodné spíše pro rychlé zobrazení komunikace mezi objekty. Jsou méně přehledné než sekvenční diagramy.

- objektu jsou **spojeny linkami**,
- zprávy jsou řazeny podle **hierarchického číslování**.



Diagramy komunikace mohou být doplněny o **větvení, kontrolní podmínky atd.**



## Základní pohledy

Projekce systému na jeden z jeho klíčových aspektů.

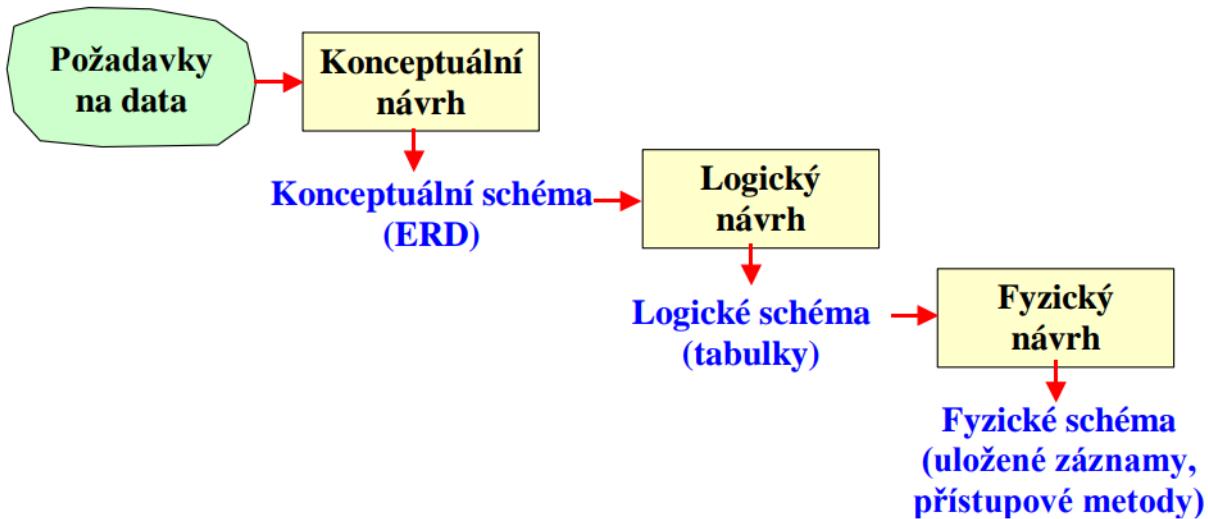
- **Strukturální** - Popisuje vrstvu mezi objekty a třídami, jejich asociace a možné komunikační kanály.
- **Datový** - Popisuje stavy systémových komponent a jejich vazby.
- **Pohled na chování** - Popisuje jak systémové komponenty interagují a charakterizuje reakce na vnější systémové operace.
- **Pohled na rozhraní** - Je zaměřeno na zapouzdření systémových částí a jejich potenciální použití okolím systému.

# 35. Konceptuální modelování a návrh relační databáze.

## Konceptuální modelování

Konceptuální modelování je první krok při **návrhu uložení dat v databázi** (návrhu databáze). Patří do etapy **analýzy požadavků**. Jeho cílem je analyzovat požadavky **na data**, která budou **uložena v databázi**. Zabývá se **modelováním reality** (objektů a jejich vlastností, které potřebujeme ukládat). Snaží se **nebýt** ovlivněno budoucími prostředky řešení. Umožňuje **sjednotit chápání** aplikace mezi uživateli (zákazníky), analytiky a programátory. Výsledek je **použit při logickém návrhu databáze** (jednotlivých tabulek) a slouží také jako **dokumentace**. Obvykle se provádí graficky pomocí:

- **ER modelů** popsaných ER diagramy: jedná se o **strukturovaný** přístup. Při strukturovaném návrhu lze však také použít OOP při implementaci IS.
- **diagramu tříd**: jedná se o objektově orientovaný přístup.



### Logický návrh

Cílem je navrhnut **strukturu databáze** (strukturu jednotlivých tabulek). Popisuje, jak jsou **data uložena** v databázi, **vztahy** mezi nimi a **integritní omezení**, tak aby neexistovala redundancy a struktura dat v databázi byla co nejjednodušší.

### Fyzický návrh

Jde o **fyzické uložení dat** (sekvenční soubor, indexy, clustery, ...). Musí se navrhnut vzhledem k **SŘBD** pro efektivní přístup.

# Entity Relation model/Entity Relation diagram

ER model je založen na chápání světa jako množiny základních objektů - **entit** (Entity) a vztahů (Relations) mezi nimi. Popisuje data **staticky** ("v klidu"), **neukazuje**, jaké **operace** s daty budou probíhat. Někdy se označuje také jako ERA – třetím základním prvkem modelu jsou **atributy** (Attributes) jednotlivých entit nebo vztahů.

## Entita

**Objekt** reálného světa rozlišitelný od jiných objektů, o níž chceme mít informace v DB. Jedná se o **konkrétní objekt**, např. klient s ID 25.

## Entitní množina (typ entity)

Definuje typ/množinu **entit**, které **sdílí tytéž vlastnosti** neboli atributy. Jedná se skupiny stejných objektů, např. klient, bankovní účet, ...

## Atribut

**Vlastnost** entity nebo vztahu, která nás v kontextu daného problému **zajímá a jejíž hodnotu chceme mít v DB** uloženu. Atributy mohou být:

- **jednoduché** (PSČ) a **složené** (celá adresa),
- **jednohodnotové** (popis - objekt má obvykle pouze jeden popis) a **vícehodnotové** (telefon - můžeme požadovat uložení více tel. čísel),
- **povolující prázdnou hodnotu** - NULL (hodnota neexistuje nebo může existovat, ale mi jí zatím neznáme, např. popis) a **nepovolující prázdnou hodnotu** (objekt nemůže existovat bez této hodnoty - např. název),
- **odvozené** (věk je odvozený od data narození) v **OLTP** ne, v **OLAP** ano.

## Doména atributu

Obor hodnot atributu.

## Vztah

Asociace (**spojení**) mezi dvěma nebo více entitami. Např. klient s číslem klienta K999 vlastní účet s číslem účtu U100.

## Primární klíč

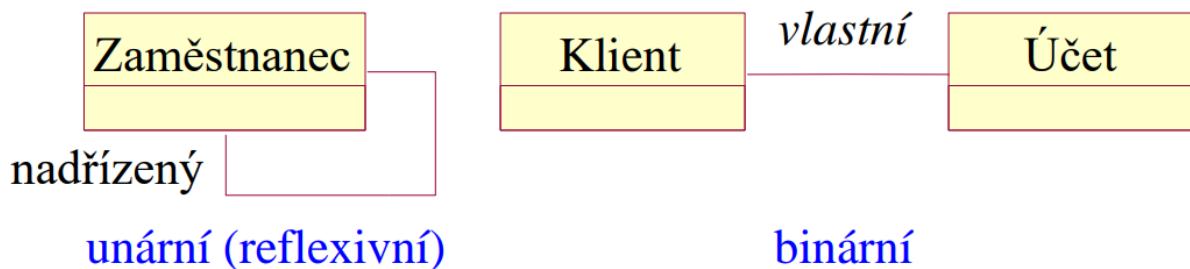
Jedná se o jeden (můžeme vybrat) z **kandidátních klíčů**, což je atribut nebo množina atributů, který/která je v dané množině entit unikátní. Kandidátní klíč musí být minimální (nemůže existovat kandidátní klíč, který má méně atributů).

## Vztahy

Spojují **entitní množiny** a vyjadřují vztahy mezi nimi.

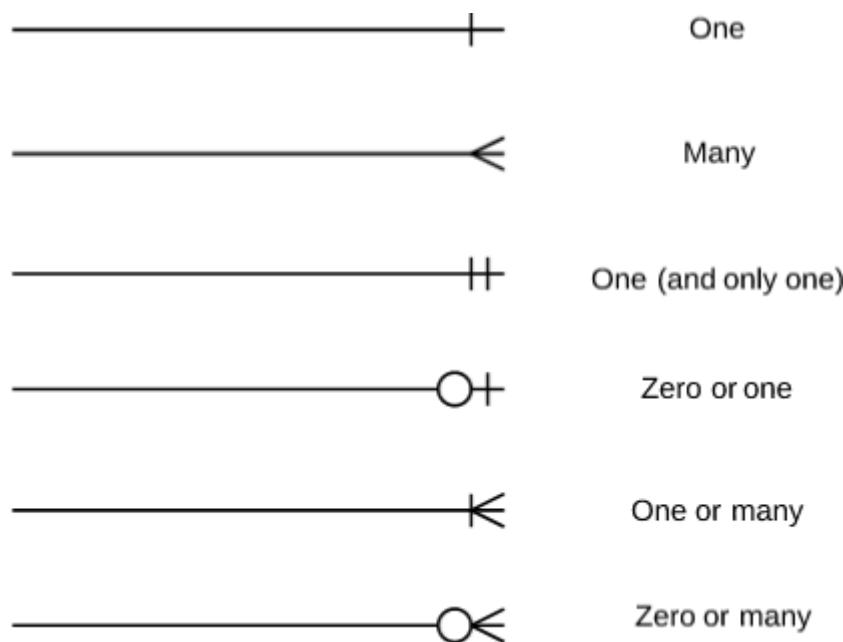
## Stupeň

Určuje kolik je vztahem **propojeno entitních množin**.



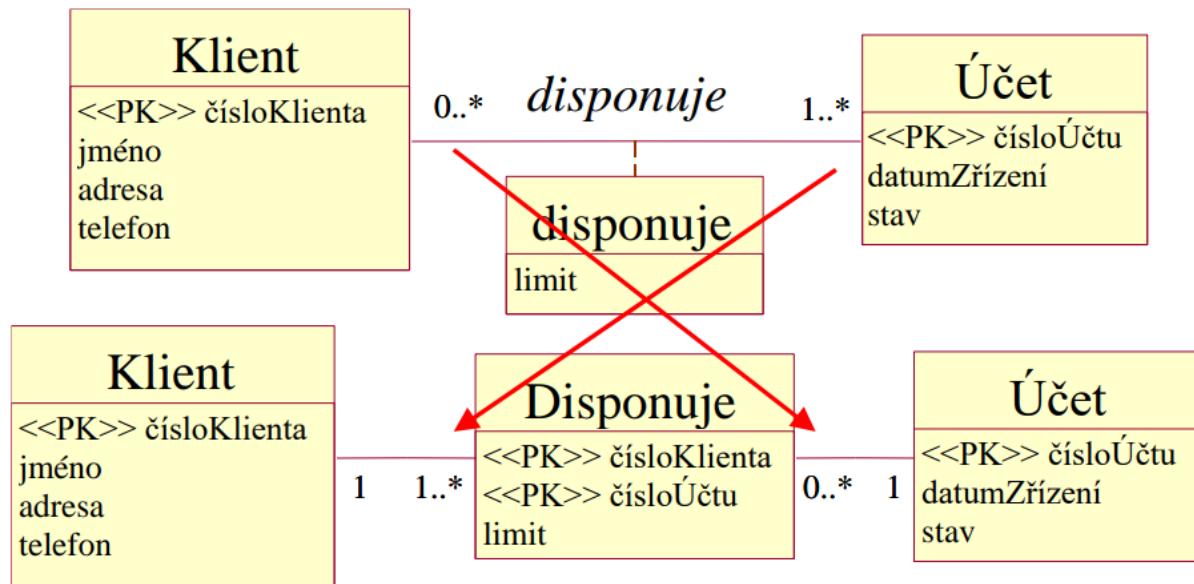
## Kardinalita

Udává **maximální počet vztahů** daného typu, ve kterých může **participovat jedna entita** (1, M, případně přesněji) z entitní množiny. Jedná se o vlastnost **každého "konce"** vztahu. Pro návrh schématu databáze je rozhodující správné určení **maximální kardinality**. Kardinalitu značíme znaky (0, 1, \*) nebo pomocí symbolů na obrázku.



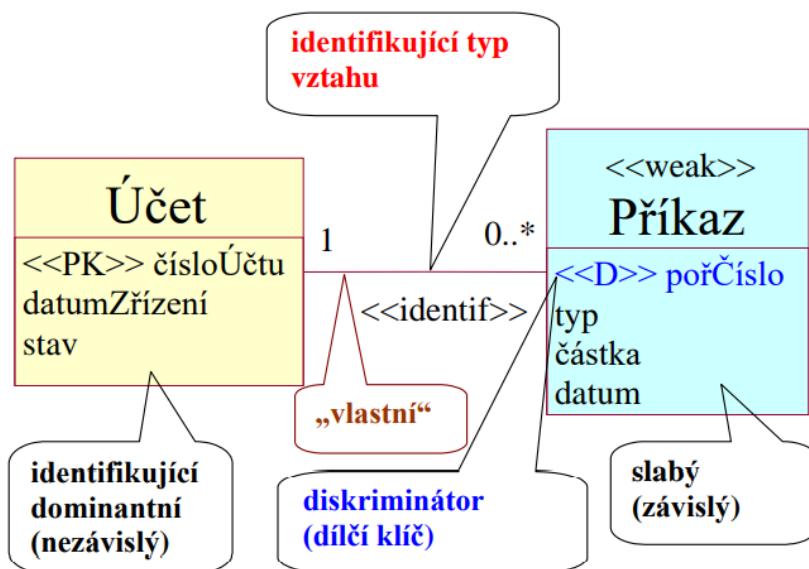
## Atributy vztahu

Rozvíjí vztah, do databáze ale musí být uloženy ve zvláštní (vazební) tabulce.  
Převádíme je na tuto tabulku viz druhý obrázek.



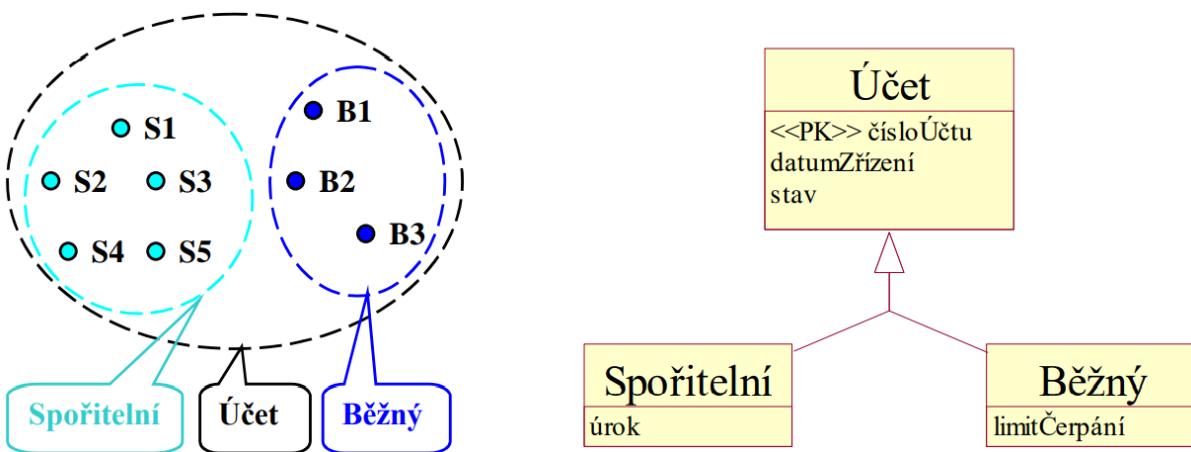
## Slabé a silné entity

- **silná entita** může existovat nezávisle na ostatních,
- **slabá entita** je závislá na jiném **jednom dominantním** typu entity. Slabé entity jsou identifikovány **primárním klíčem silné entity a diskriminátorem - dílčím klíčem**. Primární klíč silné entity slouží jako **cizí klíč**. Nemohou existovat sami o sobě a při zániku dominantní entity také zanikají.



## Generalizace a specializace

Vychází z principů OOP. Umožňuje rozširovat entity o **další atributy** (specializované entity) a současně **sdílet jiné** (obecná entita). Existovat sama o sobě může i obecná entita. Primární klíč odvozené (specializované) entity je stejný jako ten obecné.



## Jména/Názvy

- **srozumitelná**, musí vyjadřovat význam typů entit a vztahů,
- **typ entit**: podstatná jména,
- **typ vztahů**: slovesa, předložky,
- je-li jméno typu vztahu jasné ze jmen typů entit, není nutné ho uvádět,
- při několika **různých typech vztahů** mezi **stejnými** entitními množinami je **nutné** použít jméno vztahu.

## Rozdíl ER diagramu a diagramu tříd

ER diagram	Diagram tříd
<b>Typ entity</b>	Třída
<b>Entita</b>	Objekt
<b>Atribut</b>	Proměnná/atribut
–	Operace/metoda
<b>Typ vztahu</b>	Asociace
<b>Vztah</b>	Vazba (link)
<b>Kardinalita</b>	Násobnost
<b>Typ slabé entity</b>	Kvalifikovaná asociace
<b>Generalizace/specializace</b>	Generalizace/specializace

## Návrh relační databáze

Schéma relační databáze lze získat jedním z následujících dvou způsobů:

- vytvořením **koncepтуálního modelu** a jeho transformací (transformací ER diagramu),
- použitím **normalizace** s tím, že na počátku předpokládáme, že **všechny informace** budou uloženy v **jedné tabulce** a tu normalizujeme.

## Transformace ER diagramu na tabulky relační databáze

Jedná se o další krok při návrhu databáze. Jde o převod z **konceptuálního návrhu** na **návrh logický**. Tento způsob se používá častěji než normalizace, ale současně dbáme na to, aby data byla v požadované normální formě.

- **Vztahy s kardinalitou 1:1** ujistíme se, že je opravdu vhodné vytvářet tento vztah a není lepší tabulky **spojit do jedné** (neplatí při vazbě 0..1). Jinak rozšíříme jednu z tabulek o **cizí klíč** do druhé a přidáme do ní **atributy vztahu**.
- **Vztahy s kardinalitou 1:M** záznamy tabulky, které jsou ve vztahu **pouze s jedním záznamem** (vystupují s kardinalitou 1) druhé tabulky musí obsahovat **cizí klíč** do této tabulky. Tyto záznamy musí být také **rozšířeny o atributy vazby**, protože na libovolný záznam druhé tabulky může takto **odkazovat M záznamů první tabulky**.
- **Vztahy s kardinalitou M:M** reprezentujeme přidáním do databáze (vazební) tabulky. Primární klíč se obvykle volí jako **složený klíč z primárních klíčů původních tabulek**. Do vazební tabulky se také přidávají atributy vazby.
- **Vztah vyššího stupně** z naprosté většiny vyžaduje tvorbu vazební tabulky, která jako.
- **Vztah slabé entitní množiny**: tato vazba je **vždy 1:M** a provádí se stejně s tím, že tabulka reprezentující slabou entitní množinu má složený primární klíč z cizího klíče do **dominantní tabulky a diskriminátora**.
- **Generalizace a specializace** lze transformací do tabulek databáze řešit třemi způsoby
  - a. Vytvoření obecné tabulky se sdílenými atributy a tabulek specializovaných, které obsahují pouze specifické atributy. Mají **stejný primární klíč**, který je **současně cizím klíčem** do obecné tabulky.
    - **výhody**: Tato varianta je vhodná pro **disjunktní i překrývající se** specializace a pro **specializaci úplnou i částečnou**. Vhodné také při očekávání **nových specializací**.
    - **nevýhody**: nutnost provádět operaci **spojovalní**.
  - b. Vytvoření dvou nezávislých tabulek, které budou obsahovat atributy obecné entitní množiny a poté každá atributy své specializované entitní množiny. Tento způsob **neumožňuje vytváření pouze obecných entit** (musely by se ukládat do jedné ze specializovaných tabulek, což není vhodné).
    - **výhody**: Není potřeba provádět operace spojování - join.
    - **nevýhody**: specializace musí být **úplná** (nemůže existovat obecný záznam) a **disjunktní**, nelze provádět **společné** operace nad obecnými daty.
  - c. Vytvoří se pouze **jedna tabulka**, která obsahuje **obecné atributy i atributy všech specializací**. Jednotlivé specializace pak můžeme rozlišovat na základě **testu prázdné hodnoty** ve specializovaných sloupcích (pokud je to možné a některý ze sloupců specializace

nemůže být prázdný) nebo přidáním speciálního sloupce - **diskriminátoru**.

- **výhody**: není potřeba provádět spojování,
- **nevýhody**: může vést na velké tabulky a mnoho prázdných hodnot, zejména při větším počtu disjunktních specializací. V tomto případě je také vhodné přidat další sloupec identifikující jednotlivé specializace, aby se nemusel provádět složitý test na prázdné hodnoty.

d. Vytvoření tabulky pro **obecnou entitní množinu a jedné tabulky pro všechny specializace**, v této tabulce lze opět zavést **diskriminátor**.

Primární klíč v obou tabulkách bude stejný a slouží i jako cizí klíč.

- **výhody**: Lze rychle pracovat s daty obecné entitní množiny.
- **nevýhody**: spojování, prázdné hodnoty, test na prázdnost sloupců.

## Normalizace

Normalizace je druhý způsob, jak můžeme postupovat při návrhu relační databáze pro uložení požadovaných dat. Tento způsob (pokud provedený správně) **zajíšťuje**, že databáze nebude trpět nedostatky špatného návrhu. Při použití normalizace se vychází z **jedné tabulky**, ve které jsou uložena všechna data. Pokud není tabulka navržena správně, je **nutné ji rozdělit** na dvě nebo více tabulek jednodušších. To, jestli je tabulka navržena správně nebo není určuje normální forma, podle které tabulku konstruujeme. Pro OLTP systémy požadujeme alespoň **3. NF**.

### 1. Normální forma

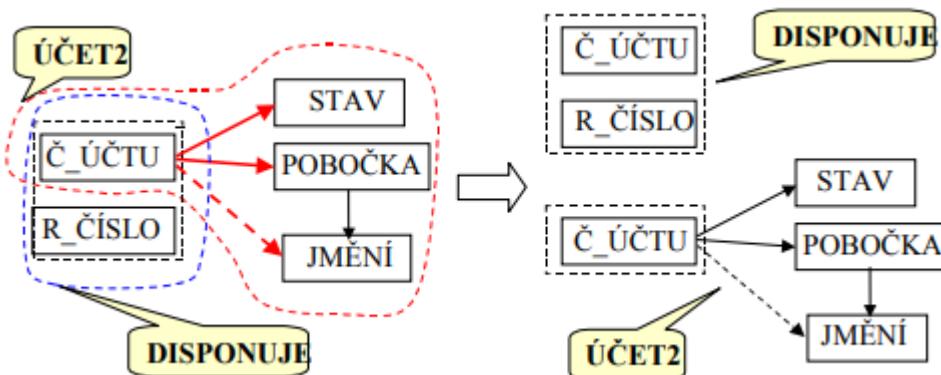
Schéma relace je v 1. normální formě, pokud její atributy obsahují pouze **atomické** (skalární) hodnoty, které nelze dále dělit. Typickým příkladem převodu tabulky do 1. normální formy je **převod atributu adresa** na několik atributů jako: **země**, **město**, **PSČ**, **ulice**, **číslo popisné**, ... Na obrázku u příkladu s telefony je také možné řešení **vytvořit vazbu 1:M**.

Osoba			
Jméno	Příjmení	Adresa	Telefony
Jan	Novák	Havlíčkova 2 Praha 3	125789654;601258987;789456123
Petr	Kovář	Svatoplukova 15 Brno	369852147;357951456;963852741
Pavel	Pavel	Papalášova 25 Kocourkov	546789123;123456789;987456123

Osoba					
Jméno	Příjmení	Adresa	Telefon mobilní	Telefon domů	Telefon do práce
Jan	Novák	Havlíčkova 2 Praha 3	601258987	789456123	125789654
Petr	Kovář	Svatoplukova 15 Brno	357951456	963852741	369852147
Pavel	Pavel	Papalašova 25 Kocourkov	123456789	987456123	546789123

## 2. Normální forma

Schéma relace (tabulka) je v druhé normální formě, pokud je v **1. NF** a každý její **neklíčový atribut**, je **plně funkčně závislý** na každém kandidátním klíči relace (musí být závislý na všech attributech složeného klíče). Převod do 2. NF zajišťujeme rozdelením tabulky na více tabulek a propojením pomocí cizích klíčů. Na obrázku tímto **cizím klíčem** bude **Č\_ÚČTU**.



## 3. Normální forma

Schéma relace je ve třetí normální formě, právě když je ve 2NF a neexistuje žádný **neklíčový atribut**, který je **tranzitivně závislý** na některém **kandidátním klíči**. Tranzitivní závislost je závislost mezi minimálně dvěma atributy a klíčem, kde **jeden atribut je funkčně závislý na klíči a druhý atribut je funkčně závislý na prvním atributu**. Řešíme tak, že vytvoříme novou tabulkou s těmito atributy, kde první atribut je klíčem a v původní tabulce zůstane pouze první atribut jako cizí klíč do nové tabulky.

Zaměstnanec									
Os. č	Jméno	Příjmení	Ulice	C. pop.	C. or.	Město	PSC	Funkce	Plat
1	Jack	Smith	Studentská	1151	12	Jihlava	58601	CEO	150000
2	Franta	Vomáčka	Prášilova	1235	25	Praha10	10000	Senior Software Architect	80000
3	Pepa	František	E. Beneše	12	57a	Plzeň	12345	Senior Software Architect	80000
4	Pavel	Novák	Mánesova	157	14	Kocourkov	99999	Junior Developer	30000
5	Petr	Koukal	Kluzká	1855	28	Praha10	10000	Database Designer	75000
6	Honza	Novák	U Klausů	28	15a	Plzeň	12345	Junior Developer	30000

transformujeme na:

### Zaměstnanec

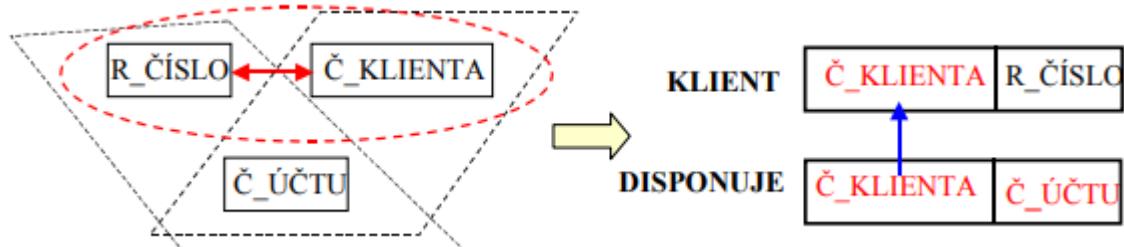
Os. č	Jméno	Příjmení	Ulice	C. pop.	C. or.	Město	PSC	Funkce
1	Jack	Smith	Studentská	1151	12	Jihlava	58601	1
2	Franta	Vomáčka	Prášilova	1235	25	Praha10	10000	2
3	Pepa	František	E. Beneše	12	57a	Plzeň	12345	2
4	Pavel	Novák	Mánesova	157	14	Kocourkov	99999	3
5	Petr	Koukal	Kluzká	1855	28	Praha10	10000	4
6	Honza	Novák	U Klausů	28	15a	Plzeň	12345	3

### Zaměstnanec

ID_Fun	Funkce	Plat
1	CEO	150000
2	Senior Software Architect	80000
3	Junior Developer	30000
4	Database Designer	75000

### Boyce-Coddova normální forma

Schéma relace je v Boyce-Coddově normální formě, pokud je ve 3. normální formě a v relaci existuje **pouze jeden** kandidátní klíč, nebo jich existuje více, ale jsou **disjunktní** (nemají společný atribut). Jinak řečeno, pokud v relaci **existují dva** (nebo více) **složené** kandidátní klíče, které **sdílí nějaký atribut**, není relace v **BCNF**. Opět řešíme tak, že relaci rozdělíme na dvě (2 tabulky) a sdílený atribut použijeme jako cizí klíč.



### Funkční závislost

Pokud je **Y** funkčně závislé na **X** ( $X \rightarrow Y$ ), pak se **nemůže stát** aby **2 řádky** mající stejnou hodnotu **X** měly různou hodnotu **Y**. (nepřesný příklad: **Datum narození** je funkčně závislé na **rodném čísle** - nemůže se stát že u 2 záznamů **se stejným rodným číslem** bude rozdílné datum narození. Nepřesný protože nelze mít 2 záznamy se stejným primárním klíčem).

# 36. Reprezentace a uložení strukturovaných dat, serializace a deserializace, relační datový model, jazyk SQL, transakce (DB a business).

## Reprezentace a uložení strukturovaných dat

Nestrukturovaná data (celočíselná, reálná, znaky, řetězce, datum - záleží jak uložené, čas, výčtové typy) mají často význam jen pokud je ukládáme strukturovaně. Existují základní dva způsoby, jak strukturované datové typy vytvářet. Jedná se o **strukturu a kolekci**.

### Struktura (prostá struktura)

Struktura je tvořena **pevným počtem pojmenovaných dílčích hodnot** (dvojice jméno a hodnota) obecně **různých datových typů**. Jedná se o **uspořádanou n-tici**, jejíž prvky jsou prvky kartézského součinu více množin. Často **strukturu** nazýváme také jako **záznam**. Při ukládání dat v (operační) paměti pomocí struktury se názvy jednotlivých dílčích hodnot neukládají (pracuje se s offsetem), názvy se připojují až při serializaci.

### Kolekce

Kolekce je tvořena **předem neomezeným** (tj. proměnným) počtem hodnot **stejného datového typu**. Matematicky se jedná o **množinu**, respektive **multimnožinu**, protože obvykle chceme umožnit vícenásobné uložení stejného prvku. **Kolekci** také často nazýváme jako **seznam, posloupnost** nebo **řetězec**. Do kolekce můžeme **vkládat** prvky, **získávat** jejich hodnoty a **mazat** je. K tomu používáme **iterátor** (ukazovátko do kolekce). Dále nad kolekcí můžeme provádět souhrnné operace jako získání počtu prvků, průměrné hodnoty, největší, nejmenší, ... případně můžeme iterovat přes všechny prvky. Nad kolekcí může existovat jedno nebo více definovaných uspořádání podle klíčů jejich prvků.

### Objekt

Objekt je struktura, kterou lze jednoznačně identifikovat - je mu přiřazena jednoznačná identifikace (**OID** - object identification). Tuto skutečnost využíváme při ukládání do databází, OID většinou **generuje SŘBD**. Díky OID může objekt vystupovat ve vztazích.

## Ukládání strukturovaných dat

Strukturovaná data jsou obvykle tvořena různě **zanořenými strukturami a kolekcemi**. Mohli bychom je do souboru ukládat binárně, tak jak jsou uložené v operační paměti a informace, co a jak (tj. **metadata** o datech) je v tomto souboru uložené nějak jinak např. do dalšího souboru. Tento způsob je ale dost nepraktický, proto používáme **standardizované formáty** uložení (JSON, XML, YAML, ...), ve kterých jsou uloženy i (některá) metadata o ukládaných datech (jejich **názvy**, jiná metadata jsou ukládána např. pomocí **Document Type Definition** (DTD) a **XML Schema Definition** (XSD) v případě XML a **JSON-LD** v případě JSON). A formát těchto souborů (**meta2data**) jsou právě popsána standardem a již se nemusí s daty předávat.

## Serializace

Serializace (marshalling) je proces **konvertování** datových **struktur** nebo **stavů objektů** do formátu, který je čitelný člověkem i strojově a může být **uložen na disk** nebo **přenášen po síti**.

## Deserializace

Při deserializaci provádíme **rekonstrukci** serializované hodnoty na **tentýž nebo i jiný formát** (např. serializovaný objekt s určitým pojmenováním atributů můžeme deserializovat do jiného objektu, který má ale stejně pojmenované atributy, ale může mít nějaké další nebo nějaké mohou chybět).

## Relační datový model

Relační databáze jsou založené na definicích **množin**, **kartézského součinu** a **relací**, jak je známe z matematiky. **Relace je podmnožinou kartézského součinu** a v případě relačních databází ji chápeme jako **tabulku**. Konkrétně tabulku tvoří **schéma relace** (záhlaví tabulky - názvy sloupců) a **tělo relace** (představuje uložená data v tabulce po řádcích). Počet atributů relace označujeme jako **stupeň** (řád) relace, kardinalita těla relace (počet řádků) označujeme jako **kardinalitu relace**. Postup odvození tabulky z matematických definicí:

$$D_{LOGIN} = \{xnovak00, xnovak01, xcerny00, xzelen05, xmodry02, \dots\}$$

$$D_{JMÉNO} = \{\text{Eva, Jan, Pavel, Petr, Zdeněk, ...}\}$$

$$D_{PŘÍJMENÍ} = \{\text{Adam, Černý, Novák, Modrý, Zelená, ...}\}$$

$$D_{ADRESA} = \{\text{Cejl 9 Brno, Purkyňova 99 Brno, Brněnská 15 Vyškov, ...}\}$$

Relace  $R_{STUDENT}$  by potom mohla vypadat například následovně:

$$R_{STUDENT} \subseteq D_{LOGIN} \times D_{JMÉNO} \times D_{PŘÍJMENÍ} \times D_{ADRESA}$$

$$R_{STUDENT} = \{(xcerny00, Petr, Černý, Brněnská 15 Vyškov), \\ (xnovak00, Jan, Novák, Cejl 9 Brno), \\ (xnovak01, Pavel, Novák, Cejl 9 Brno)\}$$

$R_{STUDENT} : \{ ($

<b>xcerny00,</b>	<b>Petr,</b>	<b>Černý,</b>	<b>Brněnská 15 Vyškov</b>
<b>xnovak00,</b>	<b>Jan,</b>	<b>Novák,</b>	<b>Cejl 9 Brno</b>
<b>xnovak01,</b>	<b>Pavel,</b>	<b>Novák,</b>	<b>Cejl 9 Brno</b>

$),$   
 $),$   
 $\} ).$



$R_{STUDENT} :$

<b>xcerny00</b>	<b>Petr</b>	<b>Černý</b>	<b>Brněnská 15 Vyškov</b>
<b>xnovak00</b>	<b>Jan</b>	<b>Novák</b>	<b>Cejl 9 Brno</b>
<b>xnovak01</b>	<b>Pavel</b>	<b>Novák</b>	<b>Cejl 9 Brno</b>

### Přidání schématu (záhlaví) relace

$R_{STUDENT} :$

<b>login:</b> $D_{LOGIN}$	<b>jméno:</b> $D_{JMÉNO}$	<b>příjmení:</b> $D_{PŘÍJMENÍ}$	<b>adresa:</b> $D_{ADRESA}$
<b>xcerny00</b>	<b>Petr</b>	<b>Černý</b>	<b>Brněnská 15 Vyškov</b>
<b>xnovak00</b>	<b>Jan</b>	<b>Novák</b>	<b>Cejl 9 Brno</b>
<b>xnovak01</b>	<b>Pavel</b>	<b>Novák</b>	<b>Cejl 9 Brno</b>

Atribut

Doména

Schéma  
relace

Tělo  
relace

N-tice

- **Doména** - pojmenovaná množina skalárních hodnot téhož typu.  
Př) Doména křestních jmen, doména příjmení
- **Skalární hodnota** - nejmenší sémantická jednotka dat, atomická (vnitřně nestrukturovaná).  
Př) Josef – z domény křestních jmen; Novák – z domény příjmení
- **Složená doména** – doména složená z několika jednoduchých domén.  
Př) složení domén křestních jmen a příjmení, hodnoty jsou dvojice, např. (Josef, Novák)
  - Atribut **relace** považujeme za sloupec tabulky,
  - N-tici **relace** považujeme za řádek tabulky.
  - **Doména** - Množina hodnot, kterých může atribut nabývat. Hodnoty mohou být pouze **skalární**.

Název **relační model** a **relační databáze** je odvozen od faktu, že relace je základním abstraktním pojmem modelu a jedinou strukturou databáze na logické úrovni.

### Schéma relační databáze

Schématem relační databáze nazýváme **dvojici** ( $R, I$ ), kde:

- $R = \{R_1, R_2, \dots, R_n\}$  je množina schémat relací,
- $I = \{I_1, I_2, \dots, I_m\}$  je množina integritních omezení.

### Integritní omezení

Omezení ukládaných dat do databáze plynoucí z reality.

- **specifická**: pro konkrétní aplikaci (pole může nabývat určitých hodnot, může nabývat pouze určitého počtu znaků, nesmí být NULL, ...)
- **obecná**: platí v **každé databázi (primární a cizí klíč)**.

### Kandidátní klíč

**Atribut** nebo **složený atribut** pro který platí:

- je jednoznačný,
- je minimální (nelze už redukovat).

Každá relace v teorii relačního modelu má alespoň jeden kandidátní klíč.

### Primární klíč

Primární klíč je jeden z kandidátních klíčů. Primární klíč je základním prostředkem pro adresování n-tice v relačním modelu (řádek tabulky je jednoznačně identifikován primárním klíčem). **Žádné komponenta primárního klíče nesmí být prázdná NULL**.

### Cizí klíč

**Atribut** nebo **složený atribut** relace **R2**, pro který platí:

- každá jeho hodnota je buď **zadaná**, nebo je každá **prázdná**,
- V FK je uložena hodnota, která je **shodná** s nějakou hodnotou primárního klíče **jiné relace R1**.

## Relační algebra

Relační algebra je dvojice  $RA = (R, O)$ , kde:

- **R** je množina relací,
- **O** je množina operací (s relacemi)

Množina operací zahrnuje:

- **tradiční množinové operace**: sjednocení, průnik, rozdíl, kartézský součin,
- **speciální relační operace**: projekce, selekce (restrikce), spojení a dělení.

Tradiční množinové operace mají stejné výsledky jako u množin, lze ale provádět (až na kartézský součin) s relacemi, které mají stejné schéma (stejně záhlaví tabulky).

R1		
A	B	C
0	a	d
1	a	e
2	b	f

R2		
A	B	C
2	c	d
0	a	d
0	a	e

R1 union R2		
A	B	C
0	a	d
1	a	e
2	b	f
2	c	d
0	a	e

R1 intersect R2		
A	B	C
0	a	D

R1 minus R2		
A	B	C
1	a	e
2	b	f

R1 times R2

AR1	BR1	CR1	AR2	BR2	CR2
0	a	d	2	c	d
0	a	d	0	a	d
0	a	d	0	a	e
1	a	e	2	c	d
...	...	...	...	...	...

## Projekce

Projekce je operace při které **vybíráme jen některé sloupce** z původní tabulky. Z relace **R** tak vytváříme relaci **R[X, Y, ..., Z]** se schématem **(X, Y, ..., Z)** a tělem obsahující patřičné (redukované) **n-tice** odpovídající novému schématu z původní relace **R**. V SQL tomu odpovídá zápis **SELECT name, surname FROM users**, kde users je tabulka obsahující např. (name, surname, title, email, phone, ...). Součástí operace projekce je i **odebrání duplicitních řádků**, které projekcí vzniknou (při SELECT tomu tak ale není).

R				
A	B	C	D	E
0	a	x	3	1
1	b	y	6	2
0	b	y	4	2

R [B,C,E]		
B	C	E
a	x	1
b	y	2

## Selekce (restrikce)

Restrikce je operace, při které je **zachováno původní schéma** realce (záhlaví tabulky), ale jsou vybrány pouze **některé n-tice** relace (řádky tabulky), které odpovídají určité podmínce. V SQL tomu odpovídá zápis **WHERE id = 42...** (s různými operátory pro porovnání).

**R**

A	B	C	D	E
0	a	x	3	1
1	b	y	6	2
0	b	y	4	2

**R where A=0**

A	B	C	D	E
0	a	x	3	1
0	b	y	4	2

## Spojení

Spojení je operace, při které je dochází ke **sloučení dvou schémat** relace (dvou záhlaví tabulek) tak, že schéma výsledné relace obsahuje **všechny atributy původních relací** a minimálně jeden atribut je mezi původními relacemi sdílen.

Podle tohoto **sdíleného atributu** je prováděno spojení, tak že jsou spojeny n-tice původních relací, které mají stejnou hodnotu tohoto sdíleného atributu. V SQL zapisujeme:

- **INNER JOIN companies ON names.id = companies.id...** Vrací záznamy z levé tabulky které mají **odpovídající** záznam v pravé tabulce (tj. vynechává řádky, které na sebe nelze navázat).
- **LEFT JOIN companies ON names.id = companies.id...** Vrací **všechny** záznamy z **levé tabulky**. K nim připojí odpovídající záznamy z pravé tabulky a pokud žádný odpovídající záznam neexistuje jsou sloupce odpovídající pravé tabulce **prázdné** (ve výsledku nemusí být všechny řádky pravé tabulky).
- **RIGHT JOIN companies ON names.id = companies.id...** Vrací **všechny** záznamy z **pravé tabulky** a připojí k nim odpovídající záznamy z levé tabulky a pokud žádný odpovídající záznam neexistuje jsou sloupce odpovídající levé tabulce **prázdné** (ve výsledku nemusí být všechny řádky levé tabulky).
- **OUTER JOIN companies ON names.id = companies.id...** Vrací všechny záznamy obou tabulek, pokud mezi nimi neexistuje spojení, jsou atributy patřičné tabulky ve výsledné prázdné.

**R1**

A	B	C
0	a	d
1	a	e
2	b	f

**R2**

C	D	E
e	1	0
d	1	1
d	0	1

**R1 join R2**

A	B	C	D	E
0	a	d	1	1
0	a	d	0	1
1	a	e	1	0

# Jazyk SQL

Structured Query Language je jazyk pro dotazování **relačních databází**. SQL je standardizovaný, nicméně se některé příkazy mohou lišit napříč výrobcí (jedná se ale o nestandardní části). Jazyk je **case insensitive**.

## Definice dat

Tvorba databázových objektů, **tabulky, pohledy, indexy**.

### CREATE TABLE

```
CREATE TABLE Persons (
    PersonID int GENERATED AS IDENTITY PRIMARY KEY,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);

CREATE TABLE new_table_name AS
    SELECT column1, column2, ...
    FROM existing_table_name
    WHERE ....;
```

### CREATE INDEX

```
CREATE (UNIQUE) INDEX index_name
ON table_name (column1, column2, ...);

CREATE INDEX idx_lastname
ON Persons (LastName);
```

### CREATE VIEW

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

### ALTER

```
ALTER TABLE table_name
ADD column_name datatype;

ALTER TABLE table_name
DROP COLUMN column_name;

ALTER TABLE table_name
ALTER COLUMN column_name datatype;
```

```
ALTER TABLE table_name  
MODIFY COLUMN column_name datatype;
```

## DROP

```
DROP TABLE table_name;
```

## Manipulace s daty

Při manipulaci s daty jsou operandem bázové tabulky nebo pohledy, výsledkem tabulka.

## SELECT

```
SELECT [ALL|DISTINCT] položka [[AS] alias_s1], ...  
      FROM tabulkový_výraz [[AS] [alias_tab]], ...  
      [WHERE podmínka]  
      [GROUP BY jm_sloupce_z_FROM|číslo, ...]  
      [HAVING podmínka]  
      [ORDER BY jm_sloupce_z_SELECT|číslo [ASC|DESC], ...]
```

```
SELECT DISTINCT K.*  
FROM Klient K, Ucet U  
WHERE K.r_cislo=U.r_cislo  
      AND U.pobocka='Jánská'  
  
SELECT jmeno, r_cislo, COUNT(*) pocet, SUM(stav) celkem  
FROM Klient NATURAL JOIN Ucet  
GROUP BY r_cislo, jmeno  
  
SELECT jmeno, r_cislo, SUM(stav) celkem  
FROM Klient NATURAL JOIN Ucet  
GROUP BY r_cislo, jmeno  
HAVING SUM(stav)>100000  
  
SELECT K.jmeno, K.r_cislo, SUM(stav) celkem  
FROM Klient K, Ucet U  
WHERE K.r_cislo=U.r_cislo AND K.mesto<>'Brno'  
GROUP BY K.jmeno, K.r_cislo  
HAVING SUM(stav) > ALL  
      (SELECT SUM(stav)  
       FROM Klient K, Ucet U  
       WHERE K.r_cislo=U.r_cislo AND K.mesto='Brno'  
       GROUP BY K.r_cislo)
```

## **INSERT**

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

INSERT INTO table_name
VALUES (value1, value2, value3, ...);

INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger',
'4006', 'Norway');
```

## **UPDATE**

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;

UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

## **DELETE**

```
DELETE FROM table_name
WHERE condition;

DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste';
```

## Pohled (View)

Virtuální DB struktura (v pohledech nejsou uložena data), může zprostředkovávat data z nula a více tabulek. Pracuje se s nimi jako s tabulkami. Mohou z tabulek vybírat jen určité řádky nebo sloupce, mohou obsahovat výrazy, mohou spojovat data z více tabulek, mohou odkazovat na další pohledy. Používá se pro zjednodušení práce (opakované dotazování). Vytváří se pomocí dotazu SELECT.

## Kurzor (Cursor)

Prostředek, který umožňuje sekvenčně zpracovávat data - číst i modifikovat. Umožňuje např. upravovat jednotlivé řádky tabulky imperativním způsobem na místo deklarativního způsobu jako u dotazu UPDATE.

## Uložená procedura (Stored procedure)

Uložená procedura (rutina) je **sada příkazů SQL**, které jsou uložené na databázovém serveru a vykonává se tak, že je zavolána prostřednictvím dotazu názvem, který jím byl přiřazen (je to určitá obdoba funkce).

## Trigger

Trigger je uložený program, který se spustí automaticky jako reakce na určitou akci s danou tabulkou (před nebo po ní). Triggery nastavujeme na dotazy UPDATE, INSERT, DELETE.

## DB transakce

Databázová transakce poskytuje mechanismus, který zajišťuje, že při manipulací s daty v databázi, bude databáze setrvávat v **konzistentním stavu**. Databázová transakce splňuje vlastnosti, které jsou známé pod akronymem **ACID**. Jedno příkazové operace nemusí být obaleny transakcí, databázové systémy k nim přistupují jako ke transakcím.

### Atomicita (Atomicity)

Databázová transakce je již dále nedělitelná. Tzn. buď se provedou všechny její části (příkazy), nebo se neprovede žádný. Např. při vkládání dat do DB, která mají být uložena do více tabulek se musí vždy uložit všechny, nebo operace selže tak, že se neuloží žádná.

### Konzistence (Consistency)

Transakce musí **zachovat konzistenci** databáze, tzn. **nesmí být porušeno** žádné ze specifických ani obecných **integritních omezení**.

### Izolovanost (Isolation)

Operace prováděné v rámci nedokončené transakce **nemohou ovlivnit jiné probíhající transakce** (např. pokud dojde k rollback - jiná transakce nemůže použít data, která byla upravena touto transakcí, protože úpravy mohou být anulovány. Případně lze dovolit použití těchto dat, ale anulovat všechny transakce, které je používaly). Musí být zajištěno, že **transakce probíhají jedna po druhé, pokud manipulují se stejnými daty**.

### Trvalost (Durability)

Poté co je transakce dokončena (commit), je zajištěno, že provedené změny budou mít **trvalý charakter**, a to i při **výpadku systému**.

## Business transakce

Jako business transakce chápeme běžné operace, které provádí uživatelé v informačním systému. Např. **vytvoření faktury, nákup zboží v eshopu, vystavení dokladu, příjem zboží** atd. Vytvoření faktury může znamenat zadání nákupčího, data splatnosti, položek faktury a její odeslání emailem nebo vytisknutí. Tato business transakce je tvořena několika databázovými transakcemi, které v průběhu realizace business transakce zajišťují konzistenci v databázi. Např. **přidání každé položky faktury může představovat novou DB transakci** (stav faktury s 1 nebo 2 položkami je konzistentní, nekonzistentní by bylo, kdyby se nějaká položka uložila pouze částečně). Systémy, které se zabývají transakčním zpracováním se označují jako **On-Line Transaction Processing (OLTP)** systémy.

# 37. Webová uživatelská a aplikační rozhraní, správa sezení a autentizace.

## Webová uživatelská rozhraní

Webová uživatelská rozhraní jsou založená na technologiích HTML, CSS, a JavaScript, se kterými dokáží pracovat webové prohlížeče.

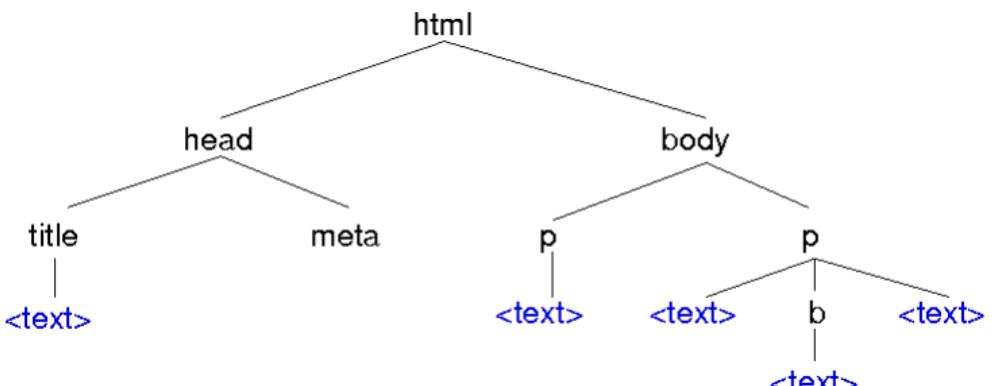
### HTML

**Hypertext Markup Language** je značkovací jazyk používaný pro tvorbu webových stránek, které jsou propojeny **hypertextovými odkazy**.

- MIME: **text/html**,
- koncovka: **.html, .htm**

Základem HTML jsou **značky**, které vymezují **úseky** dokumentu. Obvykle jsou značky **párové** (`<div></div>`, `<p></p>`), ale existují i **nepárové** (prázdné - void, `<br>`, `<hr>`), tyto značky mohou mít pouze atributy, ale nemají obsah. Dále HTML dokument tvoří **komentáře/poznámky**, které nezpracovává prohlížeč (`<!-- Libovolný text -->`). Pro vkládání **speciálních znaků** (`<`, `>`, `&`, ...) používáme escape sekvence (`&lt;`, `&gt;`, `&amp;`, ...). Používáním HTML značek vytváříme **stromovou strukturu** dokumentu.

```
<!DOCTYPE html>
<html lang="cs">
  <head>
    ... hlavička ...
  </head>
  <body>
    ... tělo dokumentu ...
  </body>
</html>
```



Samotný html obsah bez stylování pomocí CSS není uživatelsky přívětivý (má dle mého názoru škaredý vzhled).

### Základní ovládací prvky HTML

- **<input>**: slouží pro zadání vstupu uživatelem, který poté lze zaslat na server pomocí formuláře metodou POST nebo GET. Má různé typy (text, submit, phone, hidden, button, email, radio, ...)

- **<textarea></textarea>**: slouží pro zadání rozsáhlého textu, lze nastavit kolik má mít řádků a sloupců a může být zvětšovací.
- **<select><option></option></select>**: umožňuje uživateli výběr jedné z hodnot.
- **<button></button>**: tlačítko například pro odeslání formuláře.

## Základní HTML prvky pro zobrazování obsahu

- **<nav></nav>**: element, který by měl obsahovat navigaci na stránce pro SEO,
- **<ol><li></li></ol>**: očíslovaný seznam,
- **<ul><li></li></ul>**: neočíslovaný seznam,
- **<table><tr><td></td></tr></table>**: tabulka (zvýraznění buněk pomocí <th>), lze nastavit spojování buněk,
- **<a href></a>**: odkaz na jinou stránku,
- **<p></p>**: odstavec,
- **<div></div>**: obecný blokový element,
- **<span></span>**: obecný řádkový element.

## Kaskádové styly CSS

Mechanismus (jazyk), který umožňuje návrhářům oddělit vzhled dokumentu od jeho struktury a obsahu. HTML elementům lze pomocí atributu **class** přidat různé způsoby zobrazení, element lze také stylovat pomocí elementu **style**, to ale není doporučeno. Pomocí kaskádových stylů lze:

- **stylovat text**: barvu, velikost, font, podtržení, kurzívou, tučný text, aj.
- **nastavovat odsazení elementů**,
- **nastavovat pozici elementů**,
- **specifikovat barvu**: pozadí, okraje, stínu,
- **vytvářet animace**,
- **specifikovat různé zobrazení pro různě velké obrazovky**,
- **specifikovat zobrazení pro tisk**.

## JavaScript JS

JS je skriptovací objektově orientovaný jazyk, který dokáže interpretovat (snad) všechny webové prohlížeče a umožňuje tak vytvářet dynamické webové stránky. Lze pomocí něj manipulovat s **DOM** (Document Object Model), což je objektově orientovaná reprezentace HTML (a XML). Lze tak měnit vzhled elementů a přidávat nebo odebírat je bez znovunačtení stránky. Současně lze pomocí JS **asynchronně** (tj. bez toho aby se zaseklo uživatelské rozhraní) dotazovat server pomocí **AJAX** (Asynchronous JavaScript And XML) a umožnit tak například zapsání změn bez nutnosti načtení stránky, jinak řečeno umožňuje tvořit jedno stránkové aplikace. JS na HTML elementy napojujeme pomocí **event listeners**.

## Aplikační rozhraní

Aplikační rozhraní nabízí způsob, jak spolu mohou **komunikovat** dvě (a více) aplikací, respektive jedna aplikace se může **dotazovat** na informace, které poskytuje ta druhá. Typickým příkladem využití webového API je když vytvářím aplikaci kde musí uživatel vyplnit adresu (např. nějaká dovážková služba), tak použiji API, které za prvé validuje adresu a umí na základě ulice vyhledat město, zemi, PSČ atd. Uživateli lze nabídnout tento výsledek dotazu na API a on se nemusí namáhat s vyplňováním a neudělá omylem chybu. Existuje několik způsobů, jak lze API programovat.

### SOAP – Simple Object Access Protocol

Standardizovaný protokol, který specifikuje, strukturu dotazu i odpovědi na dotaz. Je nezávislý na HTML, lze jej použít i např. s SMTP protokolem. Každá zpráva je zabalena do **obálky (envelope)** a je tvořena **hlavičkou (header)**, která však není povinná a ve zprávě může chybět, a **tělem (body)** zprávy, které obsahuje dotaz, respektive odpověď na dotaz. Pro serializaci dat používá formát XML.

- **výhody:** jedná se o standardizovaný protokol (vhodnější pro veřejná api, např. vládní)
- **nevýhody:** objemnost přenášených dat, která je dána použitím serializačního formátu XML. S tím také souvisí náročné zpracování a validace příchozích dat.

### REST – Representational State Transfer

REST není protokol, jedná se pouze o návrh architektury, který by měly splňovat systémy, které chtějí implementovat tak zvané RESTful aplikační rozhraní. Je závislý na HTTP protokolu a využívá jeho metody a stavové kódy. Nad každým zdrojem jsou definovány CRUD operace a využití metod je následující:

- **C** zastává operaci **Create** – vytvoření objektu/objektů daného typu, operace se váže na HTTP dotaz typu **POST**,
- **R** zastává operaci **Read** – čtení objektu/objektů, operace se váže na HTTP dotaz typu **GET**,
- **U** zastává operaci **Update** – aktualizaci objektu/objektů, operace se váže na HTTP dotaz typu **PUT** nebo **PATCH**,
- **D** zastává operaci **Delete** – odstranění objektu/objektů, operace se váže na HTTP dotaz typu **DELETE**.

Jako serializační formát REST používá nejčastěji JSON, ale lze použít i XML či jiný.

Zhodnocení REST:

- **výhody:** jednoduchost implementace a široká podpora různými knihovnami a frameworky. Integrace s protokolem HTTP a z toho plynoucí jednodušší zpracování dotazů a také flexibilita při výběru formátu přenášených dat.

- **nevýhody:** nejedná o standard. To způsobuje odlišnosti u jednotlivých implementací, ať už jde o použitý formát dat, nebo i způsob sestavování URL koncových bodů.

## GraphQL – Graph Query Language

GraphQL také není protokol, jedná se o silně typovaný jazyk definující syntax, pomocí něhož lze vytvořit dotaz žádající přesně konkrétní data. Také není závislý na protokolu a kromě HTTP lze využít i s např. MQTT. Pro serializaci dat používá GQL výhradně formát JSON.

- **výhody:** Hlavní výhodou aplikačního rozhraní implementovaného pomocí GraphQL je, že server odesílá v odpovědi pouze ta data, o která si klient v dotazu žádá.
- **nevýhody:** malá rozšířenost, komplexní dotazy mohou představovat příliš velkou zátěž na server. Dotazy nelze cachovat, protože se mění.

## Správa sezení

Posloupnost dotazů (a odpovědí na ně) jednoho uživatele od okamžiku navštívení stránky, ža po její opuštění. Řeší se přiřazením **unikátního identifikátoru** (Session ID) uživateli při prvním navštívení stránky. Sezení **neřeší autentizaci**, pouze rozlišuje dotazy jednotlivých uživatelů (např. uživatel si v rámci sezení může nějak upravit chování stránky, např. dark mode, a stránka se tak chová do konce sezení). Protože je protokol **HTTP bezstavový** (není zachována žádná informace mezi jednotlivými dotazy) musí klient Session ID pokaždé zasílat při dotazu. To lze realizovat zasláním v rámci dotazu, např jako součást URL, což je dost nepraktické. Proto se identifikátor sezení **ukládá do Cookies**. Cookie je malý objem dat, který si server může uložit na klientovi a ten je poté zasílá při každém dotazu. Odcizení Session ID může být potenciálně nebezpečné.

## Autentizace

Autentizace je proces ověření identity uživatele (nebo jiného objektu). Autentizace vůči webovým službám je typicky řešena nějakou **tajnou informací**, kterou může znát pouze uživatel, který se autentizuje (typicky uživatelské jméno a heslo). Možnosti autentizace u webových aplikací:

- **Authorization hlavička protokolu HTTP:** Jedná se o mechanismus vestavěný pro **autentizaci** (i když název hlavičky tomu **neodpovídá**) do protokolu HTTP. **Vyžaduje** použití **HTTPS**. Používá se spíše při autentizaci vůči API, protože je nutné, aby každý dotaz obsahoval tuto hlavičku. Předávaná data mohou mít více formátů např. **Basic** a **Bearer** (údaje se předávají kódované v Base64). Při GET dotazech lze zadat údaje v url: <http://username:password@example.com/>

- **Autentizace pomocí HTML formuláře a Cookies:** Další možností autentizace je odeslat autentizační údaje v rámci POST metody na server, ten je zpracuje a vyhodnotí, vytvoří uživateli Cookie indikující, že je autentizován (Cookie je obvykle nečitelná - nějakým způsobem šifrovaná) a klient pak při každém dotazu tuto Cookie odesílá na server.
- **JSON Web Token (JWT):** Složen ze tří částí
  - Header (hlavička) – obsahující účel a použité algoritmy,
  - Payload (obsah) – JSON data obsahující informace o uživateli,
  - Signature (podpis) – informace pro ověření, že token nebyl podvržen nebo změněn cestou.

Tyto 3 části se spojí (xxxxx.yyyyy.zzzzz) a kódůjí se pomocí **base64**. Většinou se posílají v hlavičce **Authorization: Bearer base64(xxxx.yyyyy.zzzzz)**. Postup autentizace:

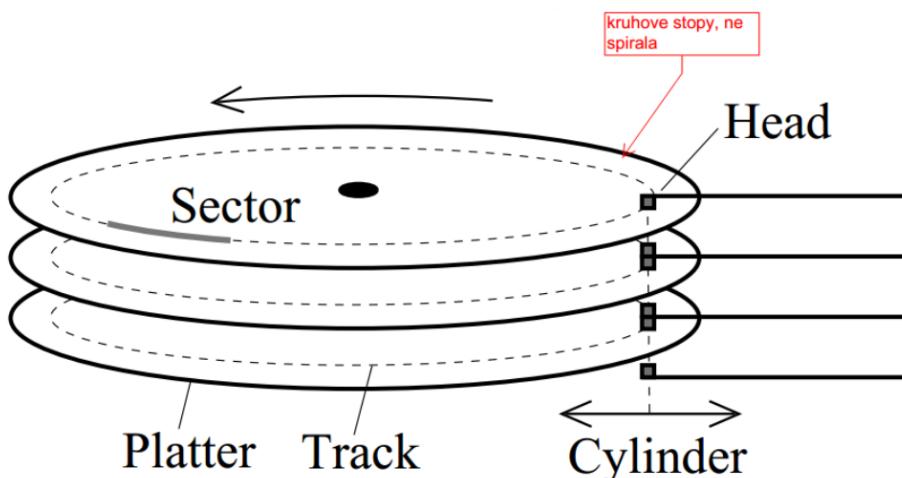
1. Klient kontaktuje **autentizační server a dodá autentizační údaje** (autentizační server může být server používané aplikace nebo nějaký jiný, např. Twitter).
  2. Autentizační server vygeneruje **podepsaný JWT token** a vrátí jej klientovi.
  3. Klient jej **zasílá při každém dotazu** na server.
- **OAuth:** OAuth řeší problém, kdy chce uživatel **umožnit aplikaci přístup k nějaké webové službě** s API, u které má účet, ale **nechce aplikaci sdělit své přihlašovací údaje** k této službě. (Příkladem je např. souhlas, že Draw.io může přistupovat ke Google Disk). Realizuje se tak, že daná aplikace **přesměruje** uživatele na **stránky webové služby**, kterou požaduje. Tam uživatel **potvrdí, že důvěřuje** této aplikaci a daná webová služba vygeneruje tzv. **Access Token**, pomocí kterého poté provádí na API webové služby dotazy.
  - **Single Sign On (SSO):** autentizační mechanismus, který umožňuje uživateli přihlásit se pomocí jediného ID a hesla do několika souvisejících, ale nezávislých systémů. Využívá k tomu **adresářové služby** (LDAP, Active Directory) a protokoly jako **NTLM** a **Kerberos**. Uživateli se pak stačí přihlásit do adresářové služby (při přihlášení na PC např. do Windows) a pomocí jmenovaných protokolů se poté provádí autentizace vůči daným webovým službám za přihlášeného uživatele na PC.

# 38. Principy a struktury správy souborů a správy paměti.

## Organizace a ukládání souborů

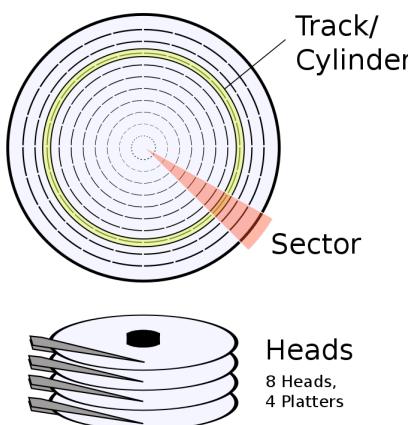
- **Soubor** - Základní organizační jednotka pro **uchovávání dat** na vnějších paměťových médiích.
- **Souborový systém** - Souhrn pravidel definujících chování a vlastnosti jednotlivých souborů a možnosti jejich další logické organizace. Také definuje způsob uložení požadovaných dat a informací k nim.
  - **FAT** - Univerzální mezi OS, je totiž skoro všude podporovaný.
  - **EXT (2,3,4)** - Pro linux.
  - **NTFS** - Pro Windows.

## Uložení dat na HDD



Disk je rozdělen na sektory, což jsou nejmenší jednotky, které lze číst/zapsat (dříve 512B dnes 4096B). Adresace sektorů:

- **Cylindr-Hlava-Sektor (CHS)** - Adresace systémem souřadnic, které jsou specifikované číslem válce (C, válec je tvořen z track na více platters), hlavou (H) a sektorem (S), který se nachází na dráze (track). Vhodné pro menší disky, jelikož větší vyžadují **proměnný počet sektorů** na stopě v závislosti na čísle válce (vzdálenosti od středu) a adresace se děje v OS.

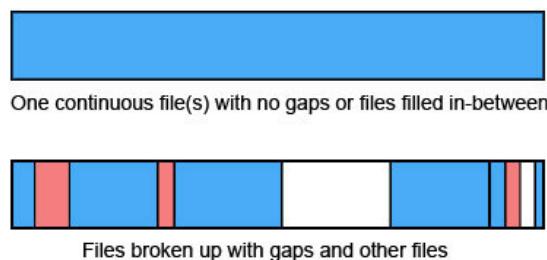


- **Logical Block Addressing (LBA)** - Adresace pomocí lineárního logického čísla sektoru (od 1 do N). Používá se v dnešní době, samotné vyhledání bloku na disku řeší nějaký obvod v disku, nikoliv OS.

## Pojmy

- **Sektor disku** - Jeho nejmenší adresovatelná jednotka (má pevnou délku).
- **Alokační blok -  $2^n$  sektorů** - Nejmenší jednotka diskového prostoru, se kterou dovoluje OS pracovat.
- **Fragmentace** - Data jsou uložena nesouvisle po částech. Zpomaluje operace prováděné s diskem. Disk lze defragmentovat - přeuspořádat uložená data, aby byla souvislá.
  - **Interní** - Fragmentace uvnitř alokovaných oblastí. Souborový systém **vyhradí pro soubor větší prostor** než je jeho velikost.
  - **Externí** - Fragmentace mezi alokovanými oblastmi, vzniká mazáním souborů - vytvořením nesouvislého uložení. Při nedostatku místa musí být soubor rozdělen (fragmentován) na části, které jsou uloženy do volných míst. Stejný problém vzniká při zvětšování souborů (za zvětšeným souborem může být uložen jiný a je nutné jej rozdělit). Může zapříčinit, že na disk soubor nelze uložit, i když je tam dostatek místa (soubor nelze ukládat po částech, nebo volná místa jsou natolik malé, že zde není možné současně uložit metadata a data - je nevhodně zvolená velikost alokačního bloku)

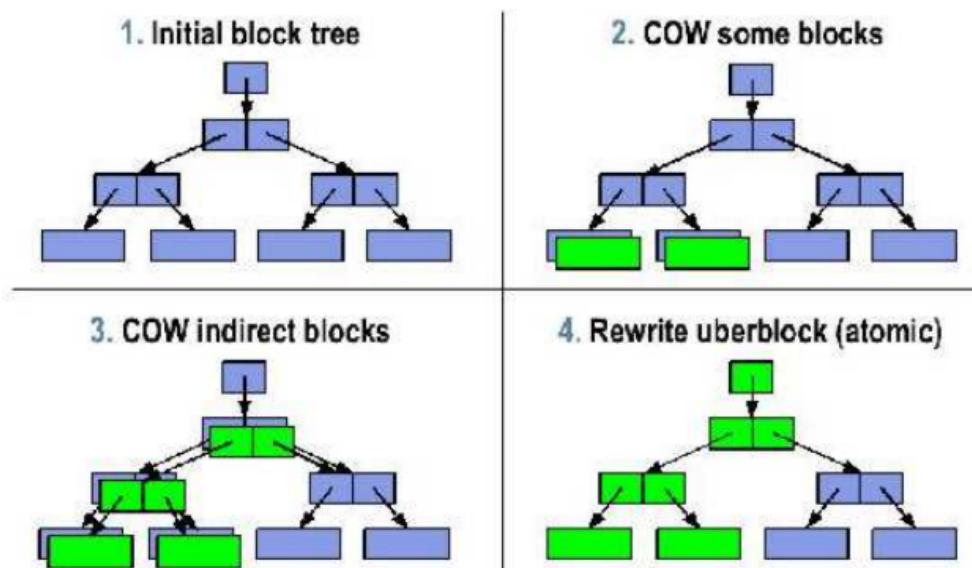
### Example of File Fragmentation



[ComputerHope.com](http://ComputerHope.com)

- **Přístup na disk (čtení a zápis)** - Přístup se musí plánovat v závislosti na aktuální pozici hlav, jednotlivé požadavky lze přeuspořádat, tak aby se hlavy musely mezi zápisy pohybovat co nejméně. Různé postupy při pohybu hlav (od středu k okraji a zpět, pouze v mezikruží, kde jsou požadovaná data, operace prováděné pouze při pohybu v jednom směru, ...) Dokončení operace (včetně chyb) je oznamováno pomocí HW přerušení)
- **Logický disk** - Dělení fyzického disku na diskové oddíly (**partition**), se kterými je možné nezávisle manipulovat (jeví se jako více fyzických disků). Tabulka MBR (**Master Boot Record**) nebo novější GPT (**GUID Partition Table**) obsahuje informace o diskových oblastech.

- **Žurnálování** - stejně jako u DB zajišťuje zachování konzistence dat na disku při zápisu. Na žurnál se zapisují operace, které budou prováděny, a po úspěšném zapsání se odstraní. Při chybě lze použít jednu ze dvou metod REDO a UNDO.
- **Copy-on-write** - nejprve zapisuje nová data a metadata na disk, poté je až zpřístupnění. Zápis dat vychází z uložení v B+ stromech. Data se zapisují postupně od listového uzlu, kde jsou uložena až po kořen (vnitřní uzly představují metadata)



### Typické parametry disku

- kapacita do 20 TB,
- doba přístupu od nízkých jednotek ms,
- přenosová rychlosť v desítkách až stovkách MB/s

### RAID (Redundant Array of Independent Disks):

Metoda pro zabezpečení dat proti selhání pevného disku.

- **RAID 0:** Disk striping - následné bloky dat jsou rozmištěny na více discích, což znamená vyšší výkonnost, ale žádnou redundanci → není to vlastně raid, nechrání před poruchou.
- **RAID 1:** Disk mirroring - data se ukládají na **2 disky**, velká redundance a výkonnost stejná jako u 1 disku.
- **RAID 2** - Data jsou na discích rozdělena po bitech a zabezpečená **Hammingovým kódem**. **Menší redundancy** dat než u RAID1.
- **RAID 3-5** - Používají paritu, dokáží se vyrovnat se **ztrátou 1 disku**.
- **RAID 6** - Používá 2 paritní bloky oproti RAID 5. dokáže se vyrovnat se **ztrátou 2 disků**.

Parita se používá se k jednoduché detekci chyb

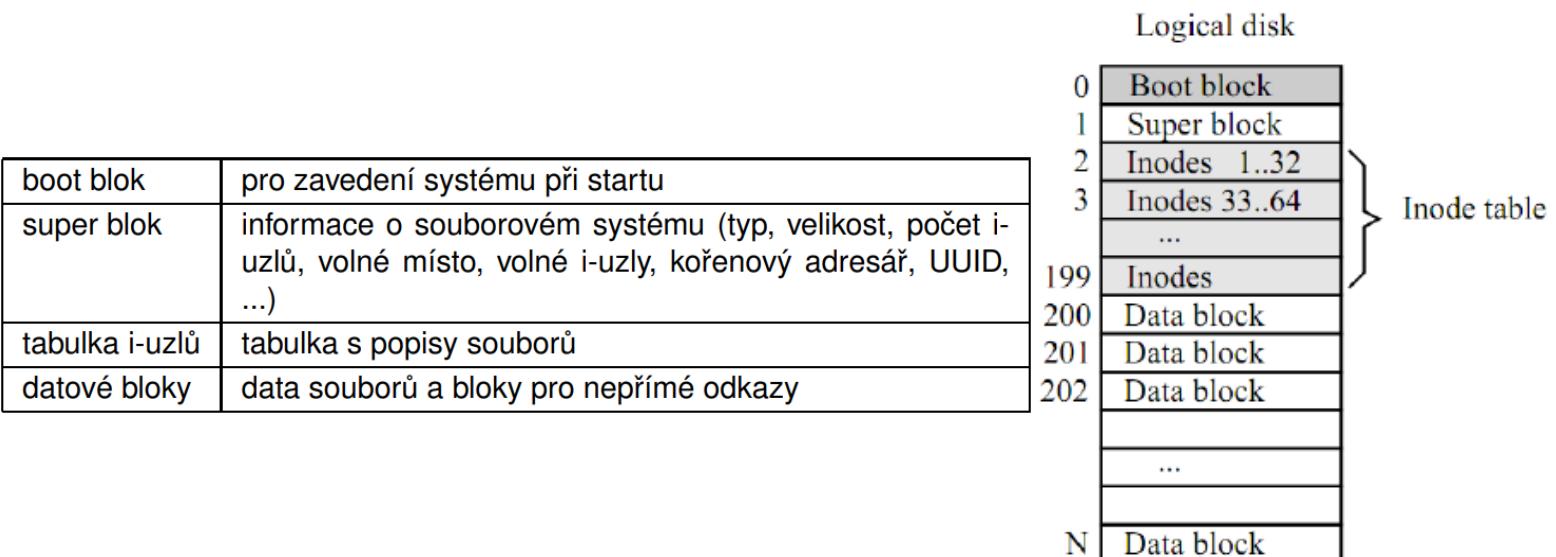
- **Lichá** - Lichý počet jedniček.
- **Sudá** - Sudý počet jedniček.

## Uložení dat na SSD

SSD jsou organizovány do **stránek** (4096 B - 4 KiB) a ty do **bloků** typicky po 128 stránkách (tj. 512 KiB). Prázdné stránky lze přepisovat jednotlivě, pro přepis stránky (editace, mazání) je nutné načíst **celý blok** (512 KiB) do VP, provést požadované změny, z disku tento blok dat **smazat a zapsat jej znova**. Problém je řešen řadičem SSD, který může sám stánky přesouvat a uvolňovat celé bloky. SSD také může obsahovat nějaké stránky nad udávanou kapacitu, které jsou použity pro přepis.

## Unixový systém souborů FS

Nejjednoduší rozdělení disku:



vylepšení:

- disk je rozdělen do skupiny **bloků**, každá skupina má **své i-uzly** a datové bloky a volné bloky. Zajišťuje lepší lokalitu dat.
- super blok s informacemi o FS je ukládán vícekrát (při poškození tohoto bloku je problematické číst z disku obnovit data, proto redundancy)

### i-uzel (inode)

Základní **datová struktura popisující soubor** v UNIX-ových souborových systémech. Obsahuje metadata, ve speciálních případech i data (např. symbolický odkaz - pokud je cesta krátká je uložena přímo v i-uzlu). Metadata tvoří:

- stav i-uzlu** (alokovaný, volný, ...),
- typ souboru** (obyčejný, adresář, **zařízení**, např. **dev/tty** - linux používá 2 abstrakce, a to soubory a procesy, symbolický odkaz, ...)
- délka souboru v bajtech**,
- mtime** = čas poslední modifikace dat,
- atime** = čas posledního přístupu,
- ctime** = čas poslední modifikace i-uzlu,
- UID** = identifikace vlastníka (číslo),

- **GID** = identifikace skupiny (číslo),
- **přístupová práva** (číslo, například **0644** znamená **rw-r-r-**).

Samotná data jsou odkazována pomocí:

- **10 přímých** odkazů na data (rychlé, ale jen pro malé soubory),
- **1 nepřímý odkaz první úrovně**,
- **1 nepřímý odkaz druhé úrovně** (tyto bloky obsahují pouze odkazy na bloky první úrovně),
- **1 nepřímý odkaz třetí úrovně** (tyto bloky obsahují pouze odkazy na bloky druhé úrovně),

Na základě **velikosti jednotlivých bloků** a **velikosti odkazů na bloky** lze poté vypočítat maximální velikost souboru v FS:

$$10 * D + N * D + N^2 * D + N^3 * D,$$

kde:

- $N = D/M$  je počet odkazů v bloku, je-li  $M$  velikost odkazu v bajtech (běžně 4B),
- $D$  je velikost bloku v bajtech (běžně 4096B).

Velikost souboru je také ovlivněna OS (např. na 32 bit je největší reprezentovatelné číslo  $2^{32}-1$ , což odpovídá 4 GiB). **i-uzel NEOBSAHUJE název souboru.**

## Jiné způsoby organizace souborů

Organizace souborů a popis uložení je implementován tak, aby byla **minimalizována režie** při práci s ním a **bylo snadné**:

- **průchod souboru**, což je spojené s rychlostí **vyhledání následujícího bloku**,
- **přesun v souboru** (seek), což je spojené s **rychlostí vyhledání prvního a určitého bloku** souboru.
- **zvětšování a zmenšování souboru**, což je spojené s **přidáním bloků** (alokací prostoru) a **odebráním bloků** (dealokací prostoru).

## Kontinuální uložení

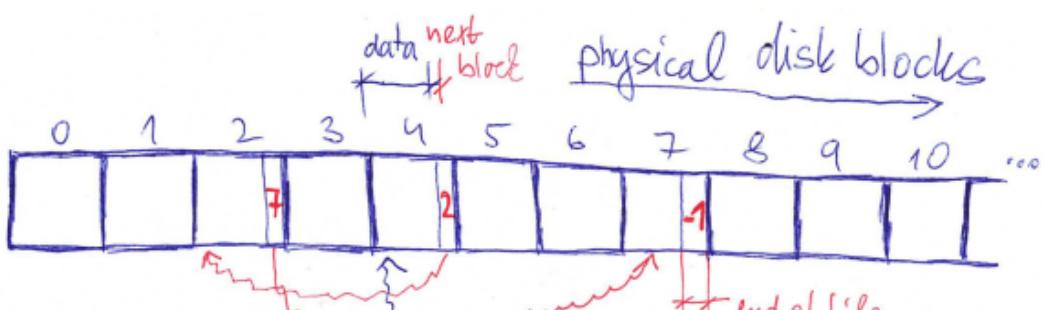
Jednoduchá Data jsou uložena na disku jako jedna **spojitá posloupnost**.

Problematické je **zvětšování souborů**, pokud se za souborem nachází další.

Problém ukládání nových souborů při **externí fragmentaci**.

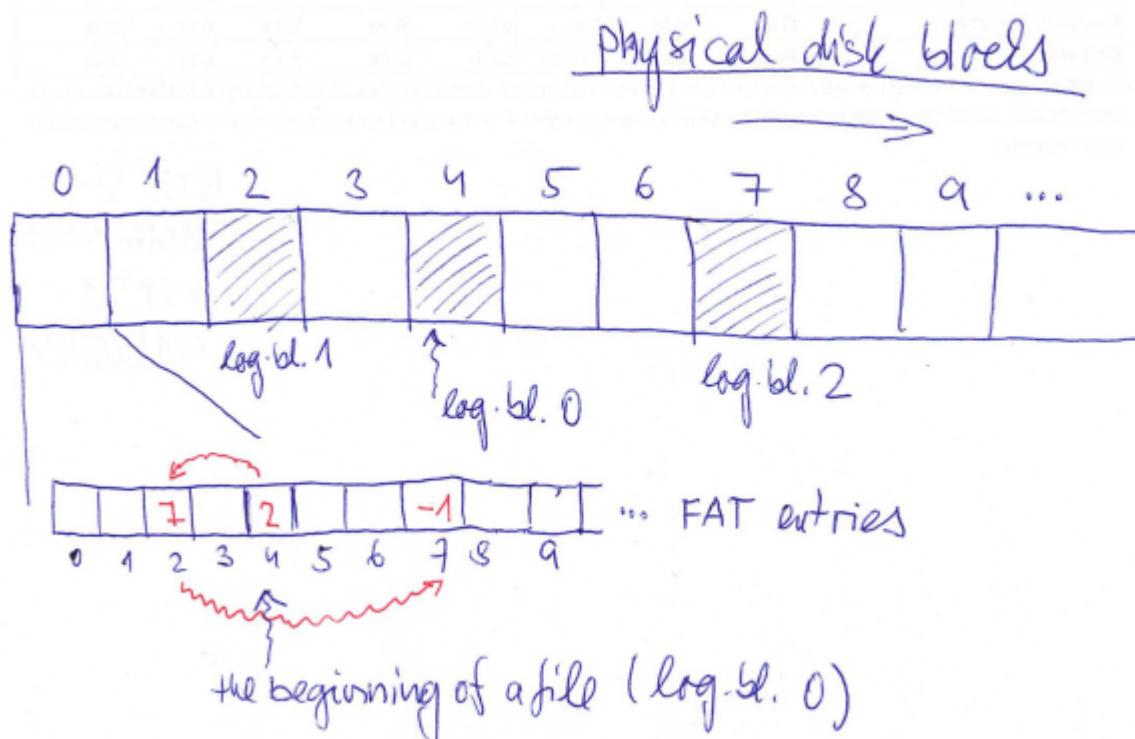
## Zřetězené seznamy bloků

Každý datový blok obsahuje kromě dat **odkaz na další blok** (nebo příznak konce souboru). Problematický je **náhodný přístup** do určité části souboru, vždy se musí **procházet sekvenčně** od začátku, stejný problém je při zápisu. Chyba v provázání kdekoliv na disku vede na ztrátu zbytku souboru. Řeší ale problém s fragmentací (bloku lze ukládat libovolně nespojitě na disku).



## File Allocation Table (FAT)

Podobný princip zřetězeným seznamům bloků. Tentokrát ale **není odkaz** na následující blok uložen **na konci datového bloku**, ale ve speciální **tabulce** (FAT - ta je uložena na začátku disku a pro jistotu 2x). Buňka tabulky obsahuje **odkaz** (index) **na jinou buňku** v tabulce a **odkaz** (index) **na blok s daty** na disku. **Částečně** řeší problém s náhodným přístupem, není nutné načítat do paměti blok po bloku (pro nalezení odkazu na další), ale samotný průchod ve FAT je **pořad sekvenční** (bude ale rychlejší).

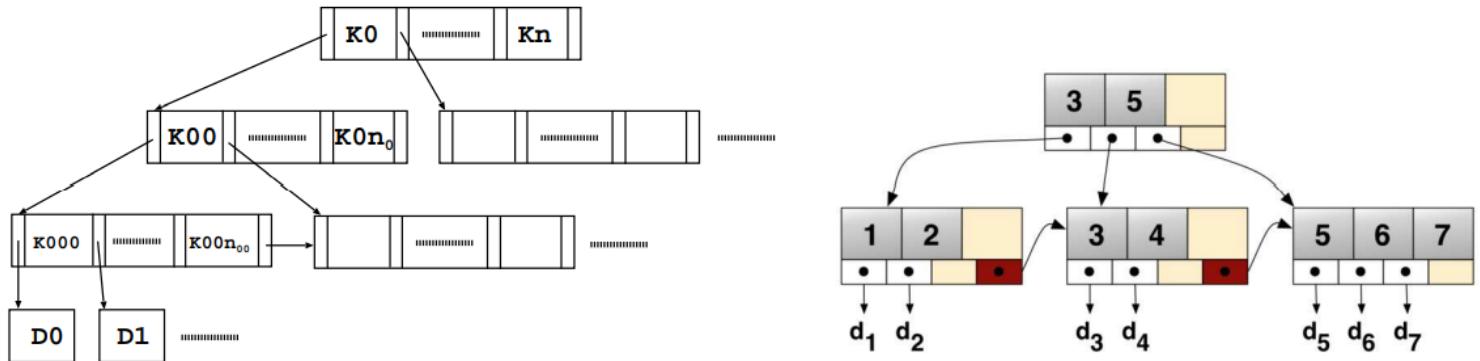


## B+ stromy

Dnes nejběžnější způsob ukládání souborů na disk (s případnými modifikacemi). Řeší problém sekvenčního i náhodného přístupu k datům souboru. Efektivnost B+ stromů je založena na **logaritmické složitosti** vyhledávání ve stromě. Struktura stromu:

- **vnitřní uzel:** je tvořen **k** klíči (identifikátory bloků - jejich sekvenční číslo) a **k+1** odkazy na bloky nižší úrovně (tedy i listové). Klíče jsou v uzlu uspořádané **od nejmenšího k největšímu** (obecně mezi nimi musí existovat nějaká relace uspořádání). Po sobě následující klíče vymezují intervaly klíčů, které jsou získatelné, pokud bude následován odkaz mezi těmito klíči. (na začátku uzlu si lze představit imaginární klíč jehož hodnota je rovna min a na konci klíč s hodnotou max+1). Tyto **intervaly jsou zleva uzavřené a zprava otevřené**, tzn. že podmínka jestli je klíč v daném intervalu vypadá následovně: **if (key\_i <= key\_hledany < key\_i+1)** tak následuj daný odkaz, jinak posuň index porovnávaných klíčů. Protože jsou klíče seřazeny, šlo by použít i vyhledávání založené na půlení intervalů.

- **listový uzel:** mají stejnou strukturu, odkazy ale vedou přímo na datové bloky (data souboru). **Poslední odkaz listového uzlu odkazuje na následující listový uzel**, což řeší sekvenční průchod souboru. Posloupnost hodnot klíčů na listové úrovni ale **nemusí být spojitá** (fragmentace), musí se zde nacházet **všechny klíče** (sekvenční čísla bloků souborů). Výběr odkazu na datový blok pak proběhne tak, že **porovnáváme** hodnotu hledaného klíče a hodnoty klíčů uložených v listovém uzlu na **rovnost**, pokud jsou si rovny, použijeme **odkaz na levo** od klíče, na kterém došlo ke shodě: **if (key\_i == key\_hledany) pak použij link\_i**, jinak porovnávej s dalším klíčem.



**B+** strom je **výškově vyvážený** a dále pro něj platí:

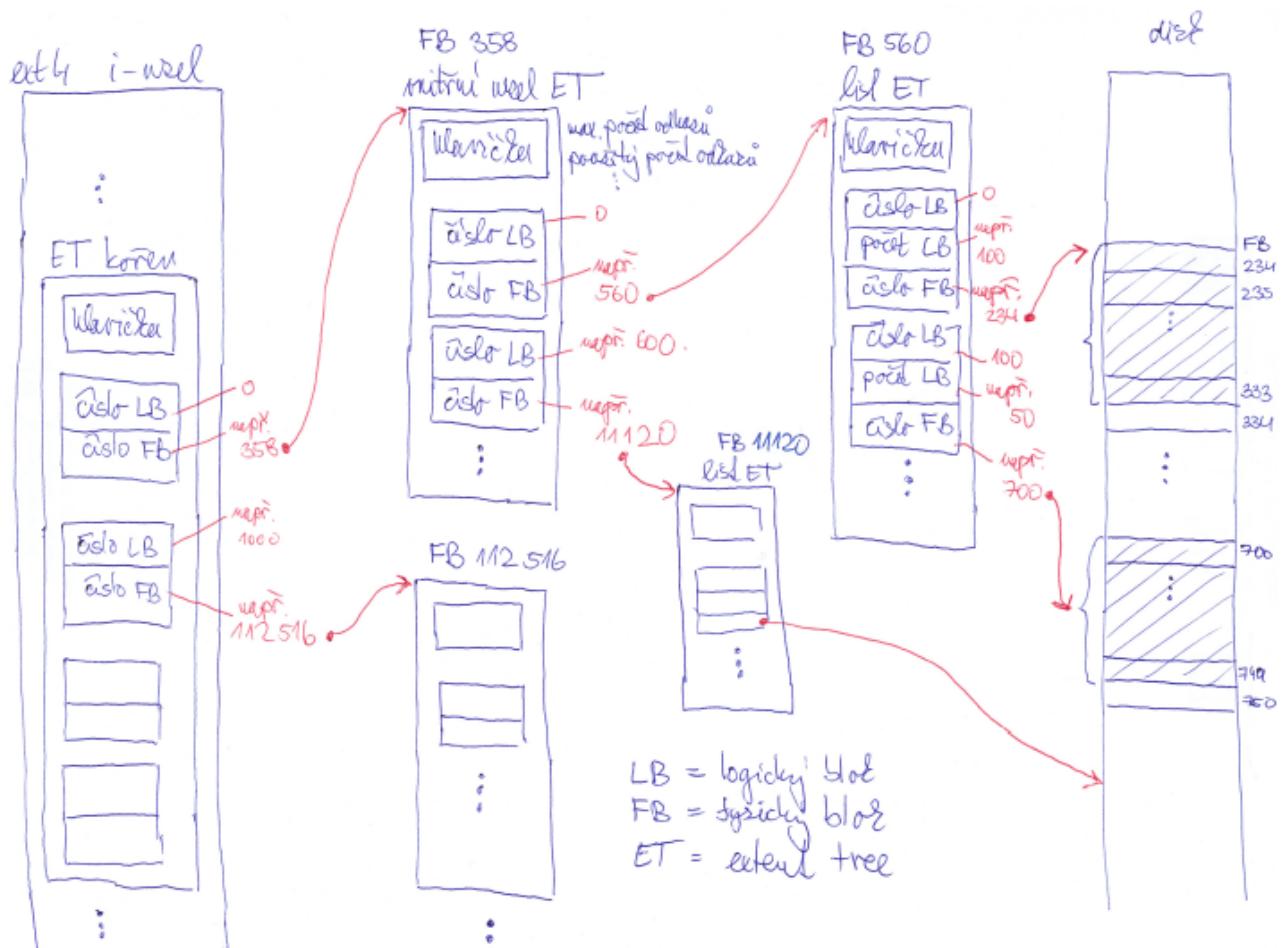
- **Limity zaplnění:** pro uzly s  $m$  odkazy (tedy klíči  $key_1$  až  $key_{m-1}$ )
  - sólo kořen 1 až  $m-1$ ,
  - kořen 2 až  $m$ ,
  - vnitřní uzel  $\lceil m/2 \rceil$  až  $m$ ,
  - list  $\lceil m/2 \rceil - 1$  až  $m-1$ .
- **Vkládání:** vkládá se na **listové úrovni** při **přeplnění** se uzel **rozštěpí** a případně se rozštěpí i uzly ve vyšších úrovních až nakonec se může rozštěpit kořen, což vede na přidání nového kořene a **zvýšení stromu** o jedna.
- **Rušení:** ruší se od listové úrovně a pokud je **porušena minimální naplněnost**, buď k **přerozdělení** odkazů **mezi sourozenci** (pokud je to možné), nebo ke **sloučení** sousedních uzlů, a případně ke sloučení uzlů ve vyšších úrovních, až nakonec může dojít ke sloučení druhé úrovně a zrušení původního kořene. Výška stromu se tak **sníží** o jedna.

## Extenty

Extenty odkazují na proměnný počet bloku, které jsou **za sebou** uloženy **logicky** (tj. v souboru) a také **fyzicky** (tj. na disku). Zrychluje práci s velkými soubory a snižuje množství metadat. Extenty se **kombinují s B+** stromy, **nelze** je ale použít v **Unixovém stromu**, protože zde není mechanismus, jak ukládat informaci o jejich proměnné délce.

## Strom extentů

Analogie B+ stromu **bez vyvažování a bez zřetězení listů**. Má omezený počet úrovní **max 5 úrovní**. Kořen v je umístěn do i-uzlu a má **max 4 odkazy** (stačí pro malé soubory nebo při málo zaplněném disku), vnitřní uzly mohou mít i více. Vnitřní **indexové** uzly se vytváří až při naplnění odkazů v kořeni, poté se ale do nově vytvořeného indexového uzlu přesunou všechny a do kořene se umístí odkaz na tento indexový uzel. Indexové uzly obsahují hlavičku, ve které je **aktuální počet odkazů a maximální počet odkazů** a dvojice **číslo logického boku a číslo fyzického bloku** (ten může být již datový nebo opět indexový).



## NTFS (New Technology File System)

Souborový systém vyvinut pro Windows NT. Využívá **Master File Table (MFT)**, ve které je aspoň jeden řádek pro každý soubor. Obsah souboru poté může být:

- uložen **přímo** v MFT daného souboru,
- odkazován z MFT pomocí **extentů**,
- odkazován z MFT pomocí **B+ stromu**, který je tvořen jinými MFT záznamy odkazovanými z **primárního MFT** záznamu.

## Organizace volného prostoru

- **bitové pole** (bitová mapa) s jedním **bitem pro každý blok**, vyhledávání pomocí bitového maskování.
- **zřetězení** volných bloků,
- **zřetězení odkazů** ve FAT, které odkazují na volné bloky,
- **B+ stromem** (adresace velikostí nebo ofsetem),
- Po extenzech.

## Deduplikace

Zamezení opětovnému ukládání stejných dat (na úrovni souboru, extentů, bloků, bytů). Může (výrazně) **ušetřit místo** např. na mail serverech (stejná příloha u několika emailů), git repozitářích (malé změny v souborech). Lze implementovat při **zápisu** nebo **dodatečně**, využívá se **kryptografického hashování** pro hledání shody (a poté případného porovnání). Problém nastává při změně jednoho ze stejných souborů, nutno rozdvojit atd. Při **malé duplikaci** může způsobovat navýšení spotřeby CPU času i místa na disku.

## Typy souborů

-	obyčejný soubor
d	adresář
b	blokový speciální soubor
c	znakový speciální soubor
l	symbolický odkaz (symlink)
p	pojmenovaná roura
s	socket

- **Adresáře** obsahuje dvojice **jméno souboru** (dříve 14 znaků, dnes až 255, nemůže obsahovat / a \0) a **číslo souboru** (typicky se jedná o **číslo i-uzlu**, případně číslo pro hledání v B+ stromu). Adresář vždy odkazuje odkazy na sebe (.) a rodiče (...). Implementace pomocí seznamů, B+ stromů, hash table.
- **Symbolický odkaz** obsahuje jako data jméno odkazovaného souboru, při otevření se vícekrát musí zpracovávat cesta. Po smazání odkazovaného souboru **symlink** zůstává ale je neplatný (jeho otevření způsobí chybu). Cykly se řeší omezení úrovně zanoření, cesta a jméno odkazovaného uzlu může být uložena přímo v i-uzlu.
- **Terminály** jsou fyzická nebo virtuální rozhraní umožňující primárně textový výstup, editaci na vstupním řádku a speciální znaky. V UNIX jsou

reprezentovány soubory (`/dev/tty`, `/dev/tty1`, ...). Více režimů zpracování znaků (**raw**, **cbreak**, **cooked**).

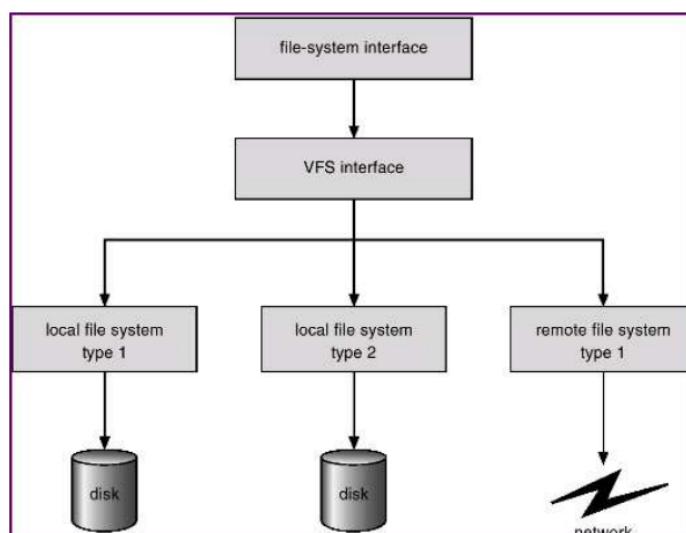
- **Roury** (pipes) umožňují **komunikaci mezi procesy**, mají **dva deskriptory** (čtecí a zápisový) V terminálu se vytváří pomocí znaku `|`. Jsou implementovány pomocí kruhového bufferu s omezenou kapacitou. Dělíme je na **pojmenované roury** a **nepojmenované roury**.
- **Sockety** Umožňují síťovou i lokální komunikaci mezi procesy (pojmenované a uložené v FS). Komunikace může být **blokující** i **neblokující**.

## Montování disků

Nový disk (a jiné zařízení) lze namontovat do libovolného adresáře již existujícího adresářového stromu. Dnes většina OS automaticky montuje disky při připojení do PC.

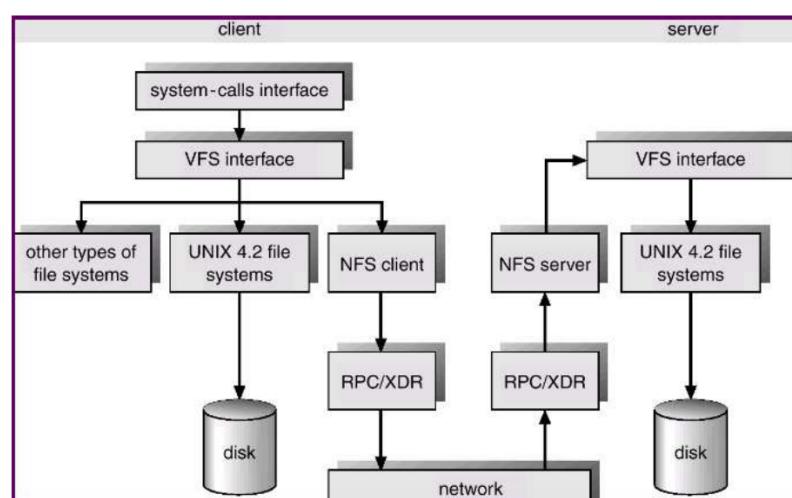
## Virtual File System (VFS)

Vytváří **jednotné rozhraní** pro práci s **různými souborovými systémy**, odděluje vyšší vrstvy OS od konkrétní implementace jednotlivých souborových operací na jednotlivých souborových systémech (FAT, ext4, NFS, ...).



## Network File System (NFS)

Zpřístupňuje soubory uložené na vzdálených systémech, zapojuje se do **VFS** a práce s ním je pak pro OS i uživatele stejná jako se soubory na disku.



## Spooling (simultaneous peripheral operations on-line)

Umožňuje **zrychlení** u pomalých **výstupních zařízení**. Výstup je proveden do **souboru** a proces, který ho zadal (nejčastěji se jedná o **tisk**), může pokračovat. Vytvořený soubor je uložen do **fronty požadavků** na výstupní zařízení (tiskárnu) a čeká až na něj přijde řada.

## Přístupová práva

Různá přístupová práva pro **vlastníka, skupinu a ostatní**.

např. **-rwx---r--** (číselně 0704), první znak znamená typ souboru (zde obyčejný soubor)

obyčejné soubory	
r	právo číst obsah souboru
w	právo zapisovat do souboru
x	právo spustit soubor jako program
adresáře	
r	právo číst obsah (ls adresář)
w	právo zapisovat = vytváření a rušení souborů
x	právo přistupovat k souborům v adresáři (cd adresář, ls -l adresář/soubor)

- vlastník: čtení, zápis, provedení,
- skupina: nemá žádná práva,
- ostatní: pouze čtení.

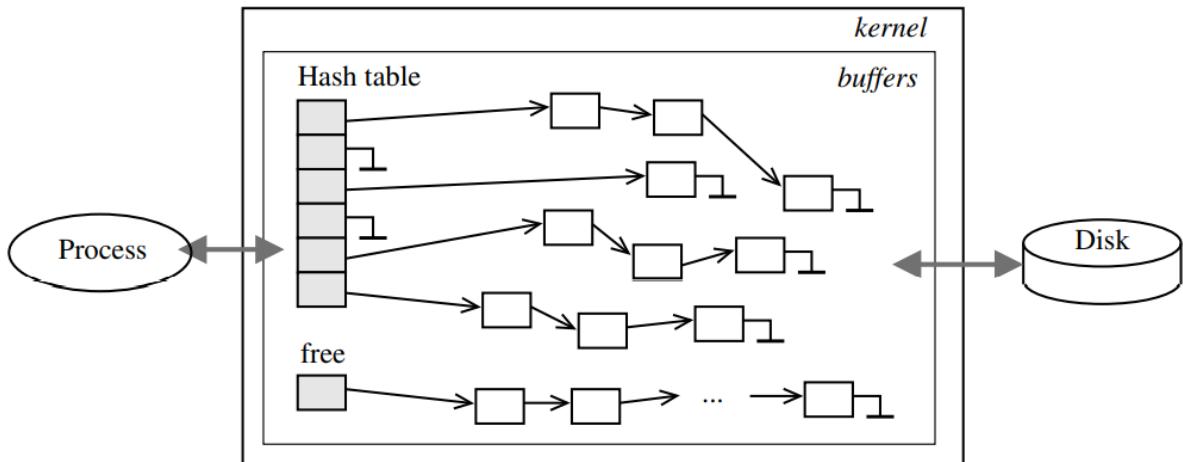
**Sticky bit** (t) je příznak, který nedovoluje rušit a přejmenovávat cizí soubory v adresáři, i když mají všichni právo zápisu (-rwxrwxrwxt).

## Práva pro procesy

- **UID** - ID uživatele co ho spustil.
- **EUID** - Efektivní UID - pro kontrolu přístupových práv.
- **GID** - ID skupiny, ve které je uživatel co proces spustil.
- **EGID** - Efektivní GID - Podobně jako EUID.
- **SUID/SGUID** - Pro propůjčení práv vlastníka uživateli.

## Práce se soubory vstup/výstup/mazání

Pro urychlení práce se soubory se používají vyrovnávací paměti (VP), které minimalizují operace s pomalými periferiemi (HDD, SSD, terminál, ...). Dílčí VP mívají **velikost datového bloku** nebo skupiny a jsou sdružovány do kolekcí pevné nebo proměnné délky. Možná implementace pomocí **hash table**.



## Čtení

Postup při prvním čtení (soubor ještě není ve VP):

1. Přidělení VP a načtení bloku (prvního nebo požadovaného)
2. Kopie požadovaných dat z VP do adresového prostoru procesu (RAM→RAM).

Dokud jsou data v RAM a dochází pouze k bodu **2**, jakmile čtením proces **překročí do jiného datového** bloku dochází k bodu **1 i 2** (pokud ještě není ve VP).

### read()

1. Kontrola platnosti **fd**.
2. Provedení kroků 1 a 2 popsaných výše dle potřeby.
3. Návratovou hodnotou je **počet doopravdy přečtených** bytů nebo -1 při chybě a nastavení errno.

## Zápis

Postup:

1. Přidělení VP a načtení bloku souboru do VP (pokud se nevytváří nový nebo zcela nepřepisuje).
2. Zápis dat do VP z adresového prostoru procesu (RAM→RAM), **nastavení příznaku modifikace (dirty bit)**.
3. **Zpožděný** zápis na disk, **nuluje příznak** (např. při uvolnění místa z RAM, ...).

### write()

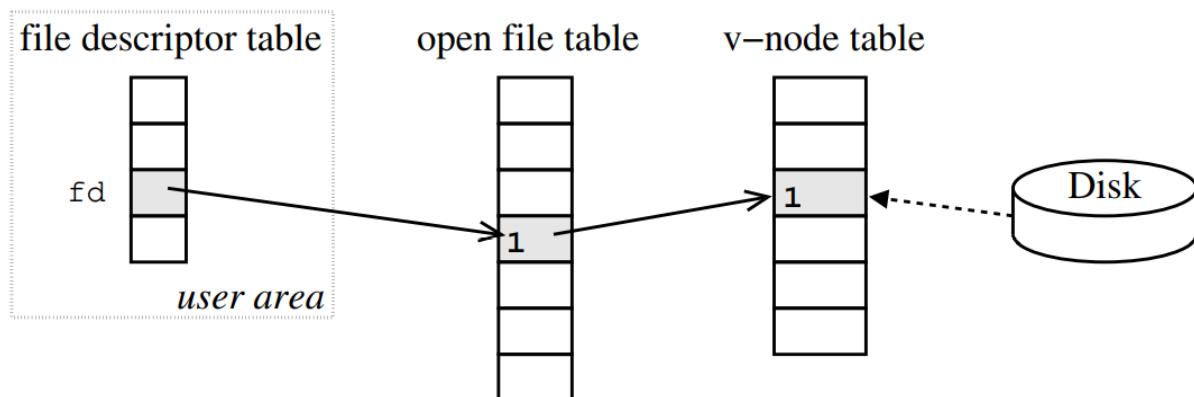
1. Kontrola platnosti **fd**.
2. Provedení kroků 1 a 2 popsaných výše dle potřeby.
3. Před zápisem dojde ke **kontrole dostupnosti dostatečného prostoru** na disk.
4. Návratovou hodnotou je **počet doopravdy zapsaných** bytů nebo -1 při chybě a nastavení errno.

## Otevření

Nejdříve (v průběhu bodu 1) se provádí kontrola na přístupová práva.

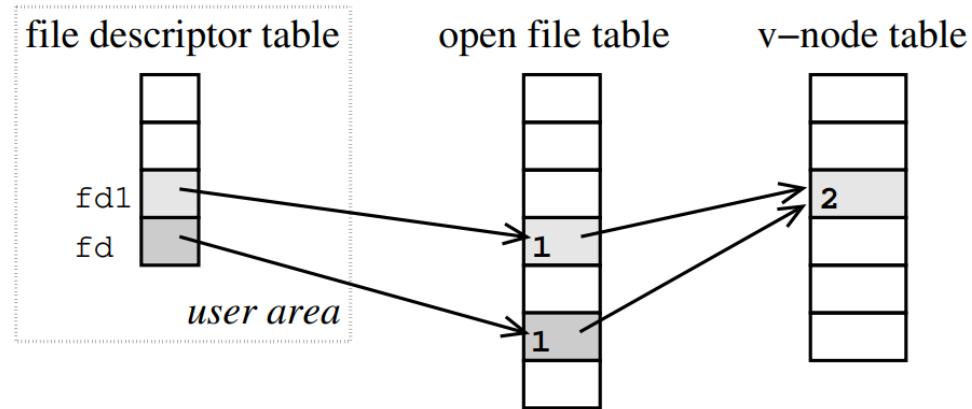
Soubor ještě **nebyl otevřen**:

1. **hledání i-uzlu** souboru: na základě cesty a jména souboru se postupně načítají **i-uzly adresářů na cestě** a datové bloky s obsahem adresářů, až je nakonec vyhledán požadovaný soubor a jeho i-uzel. Některé (časté) i-uzly mohou být už ve VP
2. V **systémové tabulce aktivních i-uzlů (inode table)** se vyhradí nová položka, do které se načte z VP **i-uzel** a stává se z něj tak **v-uzel**, což je jeho **rozšířená kopie**.
3. V **systémové tabulce otevřených souborů (open file table)** vyhradí novou položku a naplní ji:
  - a. **odkazem** na položku tabulky **v-uzlů**,
  - b. **režimem** otevření (čtení, zápis - smaže původní obsah, čtení i zápis),
  - c. **pozici** v souboru (0),
  - d. **čítačem počtu referencí** na tuto položku (1), počet referencí roste pouze při **fork** nebo **dup**, kdy dojde k **duplicaci fd** pro nový proces, procesy si mohou měnit pozici v souboru - nebezpečné...
4. V **poli deskriptorů souborů** (každý proces má vlastní, obsahují informace o **fd**) se vyhradí položka (**fd - int, index do pole**) a naplní se odkazem na položku v tabulce otevřených souborů.
5. Vrácení odkazu na **fd** nebo chyba.



Otevření pro čtení **již otevřeného** souboru (např. jiný proces nebo i ten samý):

1. **Vyhledání čísla i-uzlu**, jako při novém otevření.
2. **Vyhledání v-uzlu** v systémové tabulce (provádí se i při prvním otevření, ale při tom vyhledání selže, tabulka musí být uzpůsobena vyhledávání).
3. **Vyhrazení nového záznamu v systémové tabulce otevřených souborů** (může jít o jiný způsob otevření a také každé otevření může vyžadovat jinou pozici v souboru, odkaz v této tabulce se sdílí **pouze po fork**) s odkazem na v-uzel a **zvýšení počítadla odkazů v-uzlu**.
4. Vytvoření nového **fd**.
5. Vrácení **fd** nebo chyba.



## Přímý přístup

Pomocí lseek, postup:

1. Kontrola platnosti **fd**,
2. nastavení **pozice fd jako offset bytů**,
3. Návratovou hodnotou je **výsledná pozice** nebo -1 při chybě

Při seek za konec souboru a zapsáním vzniká řídký soubor (uprostřed má prázdný prostor, který není uložen na disku), offset může být záporný, nelze ale nastavit pozici před začátek souboru.

## Zavření souboru

Zavření souboru **nezpůsobí uložení obsahu jeho VP na disk**. Postup:

1. Kontrola platnosti **fd**.
2. Uvolnění **fd** z tabulky fd, snížení odkazů na záznam v **tabulce otevřených souborů** (open file table).
3. Pokud klesne počet referencí na záznam v tabulce otevřených souborů je záznam odebrán a snížen počet referencí na v-uzel v **tabulce otevřených i-uzlů**.
4. Pokud klesne počet referencí na v-uzel na 0, z v-uzlu se **okopíruje část, která tvoří i-uzel do VP** a v-uzel se uvolní.
5. 0 jako úspěch -1 neúspěch.

## Mazání/rušení souboru

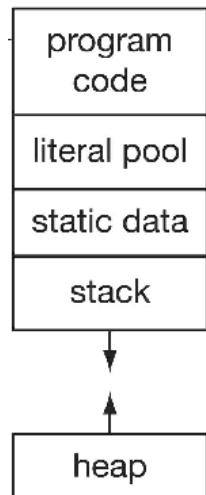
1. **Vyhodnocení cesty** a existence souboru, **kontrola přístupových práv**.
2. **Odstranění pevného odkazu** na i-uzel, v daném adresáři (vyžaduje práva pro zápis do adresáře).
3. **Zmenšení počtu odkazů** (jmén) na i-uzel (symlink v tomto počtu nejsou zahrnuty)
4. Pokud počet jmen **klesne na 0**, může být i-uzel a všechna jeho **data uvolněna**, to se provede, až soubor přestanou všichni používat (bude všemi uzavřen). Tzn. soubor jde smazat vždy (ne jak na windows, kdy kvůli tomu je

potřeba pomalu restartovat PC...), to platí i pro spustitelné soubory (program dále poběží).

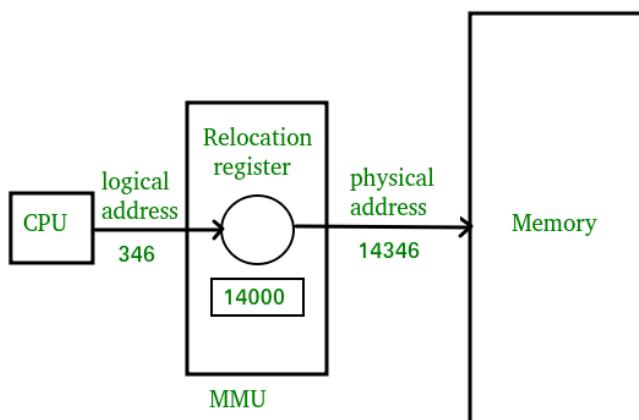
## Správa paměti

Rozlišujeme:

- **Logický adresový prostor (LAP)**: jedná se o virtuální adresový prostor, se kterým pracuje procesor při provádění kódu, každý proces i jádro jej **má svůj** a jsou **stejné** (více stejných logických adres je mapováno na různé fyzické adresy v paměti.). Pro úsek paměti se používá označení **stránka** (page).



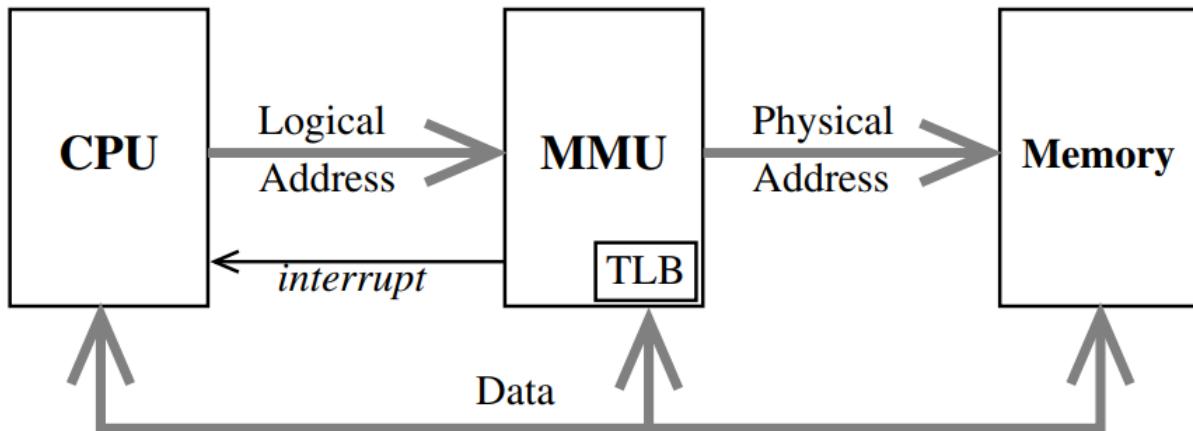
- **Fyzický adresový prostor (FAP)**: pouze **jeden pro celé PC** (společný pro všechny procesy i jádro), jedná se o adresový prostor **fyzických adres v paměti** (RAM). Pro úsek paměti se používá označení **rámec** (frame).



### Memory management unit (MMU)

HW jednotka, která zajišťuje překlad logických adres na adresy fyzické, běžně je součástí čipu (na obr. ilustrativně mimo). Pro překlad MMU využívá speciálních registrů a i hlavní paměti. Pro urychlení překladu používá různé VP, např.

**Translation Lookaside Buffer (TLB)**.



## Přidělování paměti

Přidělování **FAP** pro zmapování do **LAP** se používají úseky paměti, které jsou **konzistentní** se způsobem překladu **LAP** na **FAP**, který je podporován v HW daného systému, což jsou:

- **spojité bloky**,
- **segmenty**,
- **stránky**,
- kombinací přístupů.

Při programování (např. v C malloc) se přidělují **již namapované** úseky **LAP**.

## Contiguous Memory Allocation

Procesům jsou přidělovány **spojité bloky** paměti o určité velikosti. Pokud není aktuálně volný **dostatečně velký** spojity úsek paměti, proces buď musí čekat, nebo musí dojít k **odložení** celé paměti jiného procesu na HDD.

- **výhody**: velmi jednoduchá implementace v HW i SW.
- **nevýhody**: výrazný projev **externí fragmentace**, a to z důvodu přidělování a uvolňování paměti různé velikosti. Vznikají úseky, které mohou být příliš malé, na to aby byly využity. Dalším problémem je **zvětšování přiděleného prostoru** pokud již za úsek aktuálně přidělené paměti není místo, je nutné celý již přidělený úsek přesunout do dostatečně velké oblasti a poté až přidělenou paměť zvětšit. Lze použít nějakou **strategii** pro přidělování paměti (first fit, best fit, worst fit), to ale ubírá na jednoduchosti implementace.

## Přidělování paměti po segmentech

LAP je rozdělen na **několik segmentů**. Segmenty přiděluje různým částem procesu (zásobník, hromada, nebo i jednotlivým procedurám, ...) překladač nebo programátor. U segmentů může být dále specifikován způsob přístupu - čtení/zápis (bezpečnější, kód lze umístit do read only segmentu). **Logická adresa** je tvořena z **čísla segmentu a posunu v něm**, každý segment má číslo a velikost.

- **výhody:** poměrně jednoduchá implementace, zlepšuje externí fragmentaci, ale problém vlivem proměnné velikosti segmentů přetrvává.
- **nevýhody:** segmentování programu je řízeno při tvorbě programu, což může vést na **problémy při implementaci**. Stále neřeší **problém s externí fragmentací**, příliš velké segmenty nemusí být možné namapovat do **FAP**, nutné odkládat větší úseky paměti na HDD.

### Stránkování (přidělování paměti po rámcích)

**LAP** je rozdělen na **stránky** (pages) o **pevné velikosti** (každá stránka je stejně velká, obvykle 4096 B, což odpovídá velikosti bloku na disku), **FAP** je rozdělen na **rámce**, které jsou ale **stejně velké jako stránky**. Mapuje se vždy **jedna stránka na jeden rámec** (nelze aby se jedna stránka mapovala přes dva rámce) a **mapování je odstíněno** od implementace programu a běžícího procesu. Pro zvýšení efektivity přístupu do paměti je snaha o **mapování spojitéch posloupností rámci**, pokud je to možné.

- **výhody:** eliminuje externí fragmentaci (každý rámec lze využít), neviditelný pro běžící proces. Umožnuje **jemné odkládání** po stránkách/rámcích, proces tím ani nemusí být zpomalen, pokud danou stránku nepoužívá. Další výhodou je řízení přístupu pro každou stránku.
- **nevýhody:** složitější implementace (HW i SW), představuje větší režii (hledání, jestli je stránka namapovaná) a může být pomalejší (zejména při TLB miss). Stránkování způsobuje **interní fragmentaci** (přidělení větší části paměti, než je potřeba, zarovnávání, aby nebyly instrukce/data na rozmezí dvou stránek atd.)

### Jednoúrovňové tabulky stránek

**Nejjednodušší** ale prakticky **nepoužitelný** způsob implementace tabulek stránek. OS si udržuje informace o volných rámcích a pro každý proces včetně jádra si udržuje jeho **tabulku stránek** (page table), **adresa tabulky** je uložena v k tomu určenému **registru**. Tabulka stránek obsahuje **popis mapování** (dvojice první logická adresa stránky a první fyzická adresa rámce) a **různé příznaky** (modifikace, přístupová práva, příznak globality - může používat více procesů existuje  $2^{32}$  adres, pokud má stránka velikost **4096 B** ( $2^{12}$  B) je nutné uchovávat informaci o  $2^{20}$  stránkách, jeden záznam o mapování stránky na rámec může mít **4 B**, což znamená **4 MiB** pro uložení stránek **pro jeden proces**). Problém exponenciálně roste u 64 bit systémů.

Při provádění příkazu v programu dochází ke **dvoum přístupům do paměti - instrukce a její operandy**. To vede ke dvoum přístupům do tabulky stránek. Pro urychlení tohto procesu se používá **TLB**.

### TLB (Translation Look-aside Buffer)

Jedná se o rychlou **asociativní (obsahem adresovatelná paměť)** vyrovnávací paměť, která umožňuje **paralelně** (nebo částečně paralelně) **vyhledat** na základě

**čísla stránky** (1. log. adresy) odpovídající **číslo rámce** (1. fyz. adresu). **TLB** obsahuje **vybrané** záznamy (zejména **číslo stránky** a **číslo rámce**, příznaky nemusí všechny) z tabulek stránek, **rozhodně neobsahuje** data stránek/rámců. Výběr uložených položek v **TLB** je **zásadní** pro rychlou práci s pamětí, při **TLB hit** lze ihned pomocí vyhledaného čísla rámcí přistoupit (**1 přístup**) k fyzické paměti. Naopak při **TLB miss** (požadované číslo stránky není v TLB) je nutno provést **dva přístupy** do paměti, a to vyhledání čísla rámce v tabulce stránek a přístup do vyhledaného rámců (doba pro vyhledání v TLB je řádově menší než doba pro přístup k RAM). V praxi lze naštěstí díky **časové a prostorové lokalitě** zajistit **TLB hit** kolem **98%**, **99%**, procesory **preemptivně** načítají očekávané mapování do TLB. K **TLB miss** může teoreticky dojít **až 4x** při provádění **jedné instrukce** (instrukce je na rozmezí 2 stránek, operandy jsou na rozmezí 2 stránek). Pozor, **neplést si TLB miss s výpadkem stránku** (tj. stránce není přidělen rámcem). TLB miss lze řešit:

- HW automaticky hledá v tabulkách stránek (rychlejší ale dražší),
- řízení je předáno **SW**, které do TLB doplní požadovanou dvojici a hledání v TLB se opakuje.

Při **přepnutí kontextu** (změna procesu, který je prováděn) je nutné **změnit i položky v TLB** (oba procesy používají stejný LAP → vznik kolizí). Lze řešit označením některých stránek za **globální** (sdílené mezi procesy) nebo **přidáním identifikátoru procesu** do TLB (pak nemusí být záznamy odstraňovány, ale je složitější porovnání a vyžaduje více paměti). Záznamy z TLB musí být odstraněny i při změně mapování.

Efektivní přístupová doba:  $(\tau + \varepsilon)\alpha + (2\tau + \varepsilon)(1 - \alpha)$ , kde

- $\tau$ : vybavovací doba RAM
- $\varepsilon$ : vybavovací doba TLB
- $\alpha$ : pravděpodobnost úspěšných vyhledání v TLB (*TLB hit ratio*)

## Hierarchické tabulky stránek

Eliminuje problém s **nadměrnou velikostí tabulek stránek**, na moderních procesorech jsou tabulky stránek **4 úrovňové** (tj. strom o výšce 4, v paměti jsou uloženy jen **podstromy**, které jsou potřeba) Údaje o mapování **stránek s daty** jsou obvykle v listové úrovni, mohou být ale i v **některé z vyšších úrovní** (podle příznaku), pak má stránka a odpovídající rámcem větší velikost (2 MiB nebo někdy i 1 GiB). U hierarchických tabulek **rostet význam TLB**, při **TLB miss** je nutné vyhledávat ve stránkové hierarchii (až 4 přístupy do paměti navíc). Optimalizace např. použitím **různých TLB pro kód a pro data**.

## Hashované tabulky stránek

Tento princip je založený na **hashovací tabulce s explicitním řetězením synonym** pomocí seznamu (položky seznamu již nemusí obsahovat celé číslo stránky, pokud hashovací funkce dokáže vždy některé bity rozlišit). **Délka seznamu může být omezena**, pak je nenalezení řízeno SW (a některá překladová dvojice může být ze

seznamu odebrána). Hashovací tabulka může být **pro každý proces** nebo **sdílená** (překladové položky pak musí obsahovat i číslo procesu).

## Invertovaná tabulka stránek

Pro **každý rámec** udává, **jaký proces** do něj má namapovanou **jakou stránku** (některé rámcce mohou být aktuálně neobsazené). Index v tabulce rámců odpovídá číslu rámcce (není ale problém s velikostí, protože tabulka je jen jedna pro všechny procesy). Pro urychlení vyhledávání se používá hashování. **Problémem je sdílení paměti mezi procesy** (lze řešit mapováním regionů).

## Stránkování a segmentace na žádost

Je nemožné (u 32 bit teoreticky ano, u 64 bit ne), aby byl celý LAP jednoho procesu, natož více procesů, namapován do paměti (RAM má obvykle kolem 8-128 GB, LAP má u 64 bit procesoru  $2^{64}$  adres, což je mnohem víc než 128 GB). I když ale **není celý LAP procesu namapován** do fyzické paměti, **proces o tom neví**.

Stránky/segmenty jsou mapovány na rámcce a tím **zaváděny do RAM**, až když je to **potřeba** (proces čte/zapisuje do dané oblasti paměti). Namapování stránky na rámcce je ale **časově velmi náročná** operace (může být nutné číst data z disku), proto se tyto operace snažíme eliminovat, procesor tak **preemptivně načítá stránky** do paměti předtím, než z nich proces chce číst/zapisovat. Při využití všech rámců je nutné část namapovaných dat odložit na disk (swap area), což většinou vede na velké zpomalení systému. Informace o načtených a odložených stránkách si vede **jádro ve svých strukturách** (také v paměti, ale zde si jádro zajistí, že tam jsou neustále), **MMU** o nich **nemá informace**. K **výpadku stránky** dochází, když do ní proces **odkazuje**, ale není načtena v tabulkách stránek nebo chybí v hashovací tabulce či invertované tabulce. Při výpadku generuje **MMU přerušení**, a jádro musí výpadek stránky obsloužit (některé stránky jsou chráněny proti výpadku, např. ty, které obsahují tabulky stránek).

## Obsluha výpadku stránky

1. Kontrola, zda proces **neodkazuje mimo přidělený prostor** (o který zažádal např. použitím malloc).
2. **Alokace rámcce**: buď je nějaký **prázdný** nebo musí být **vybrána oběť** (dle nějaké strategie), pokud je navíc obsah oběti modifikován (dirty bit), musí být uložen na disk - **odložen do swap**.
3. **Inicializace stránky**: ve všech případech musí být data, která v rámcu zbyla po předešlém procesu **znehodnocena** aby proces, kterému je rámcem nově přiřazen toho **nemohl zneužít**. Pokud se jedná o kód, nějaká statická data nebo byla dřív stránka odložena na swap (byla editována), je rámcem **přepsán** tímto, jinak je **nulován**.
4. Aktualizace tabulky stránek aby odpovídala provedené změně.
5. Proces může opakovat provedení instrukce, která způsobila výpadek.

Efektivní doba přístupu do paměti:  $(1 - p)T + pD$ , kde

•  $p$ : *page fault rate* = pravděpodobnost výpadku stránky,

•  $T$ : doba přístupu bez výpadku

Může dojít k **výpadku** i **několika stránek** při provádění jedné instrukce (kód instrukce je na rozhraní dvou stránek, operandy jsou na rozhraní dvou stránek a ještě k tomu může dojít k výpadku v hierarchické struktuře tabulky stránek). Nejvyšší úroveň tabulky stránek musí být ale chráněna proti výpadku, jinak by nebylo možné výpadek obsloužit.

## Algoritmy pro výběr oběti

Výběr odkládané stránky může být:

- **lokální** v rámci procesu, u kterého došlo k výpadku,
- **globální** tj. uvolnění libovolného rámců, nehledě na to, kdo jej používá.

Typicky je ale udržován (**page daemon** - spouští se v okamžiku nedostatku) určitý počet volných rámců. Při výpadku stránky se používá **volný rámeček**. Navíc pokud byla vybrána nesprávná oběť (tj. proces požaduje stránku hned po jejím označení za oběť) je mu stránka **vrácena** a vybírá se jiná oběť. Při výběru oběti jsou **upřednostňovány nemodifikované stránky** (není je třeba odkládat na swap).

Proces musí mít vždy přidělen minimální počet rámců pro provedení jedné instrukce, jinak musí být zastaven (docházelo by k neustálým výpadkům). Počet rámců přidělených procesu se určuje na základě heuristik (např. priorita, velikost programu, frekvence výpadků, ...).

### FIFO (first in first out)

Odstraňuje stránku, která byla do paměti **zavedena před nejdelší dobou**.

- **výhody**: jednoduchá implementace,
- **nevýhody**: může odstranit stránku, která je namapovaná dlouho, ale také se často používá (eliminace použití heuristiky viz výše). Trpí Beladyho anomalií, tj. při zmenšení počtu rámců se může zvýšit počet výpadků.

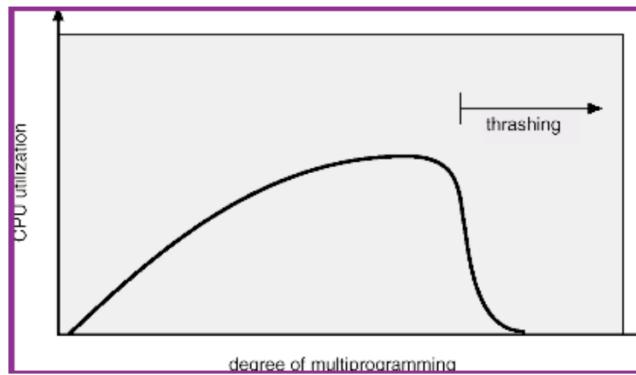
### LRU (Least Recently Used)

Odkládá **nejděle nepoužitou stránku**. Dobře approximuje ideální algoritmus výběru oběti (ten co vybere stránku, která již nebude použita, nebo použita za nejdelší dobu). Vyžaduje **podporu v HW**, nutné uchovávat **časová razítka** nebo stránky mít **seřazené** podle přístupu. Obvykle se approximují časová razítka několika bity pro každou stránku. **Jádro** periodicky stránky prochází a **nuluje bity** (shiftem doprava), při **přístupech** ke stránce se naopak nastavují **bity na 1 (MSB)**. Oběť je taková, která má v registru nejmenší hodnotu (nejvíc MSBs je na 0). Lze použít pouze jeden bit (**referenční bit**), který je periodicky nulován jádrem a nastavován při odkazu na stránku, oběť je první stránka s nulovým referenčním itemem.

- **výhody**: způsobuje malý počet výběrů nesprávné oběti.
- **nevýhody**: vyžaduje HW podporu, nefunguje pro cyklické průchody rozsáhlých polí (lze však detektovat a použít **MRU** strategii **Most Recently Used**).

## Trashing

Problém, který nastává při nadměrném vytížení RAM a nedostatku rámců. **Řešení výpadků stránek zabírá delší čas, než požadovaný výpočet**, a to mnohonásobně. Řešením je úplné pozastavení některých procesů a přesunu veškeré jejich paměti na swap area.



## Sdílení stránek

Používá se zejména pro běžné knihovny **.dll** a **.so**. Ve **FAP jsou** knihovny načteny **pouze jednou**, využívá je současně ale více procesů, více LAP je namapované na stejný FAP. Sdílení stránek také umožňuje Inter Process Communication (IPC), tj. dva procesy používají stejný úsek **FAP**. Sdílení paměťové mapovaných souborů.

## Paměťově mapované soubory

**Bloky** souborů z disku jsou **mapovány do stránek v paměti** a mohou být pak čteny/zapisovány běžným přístupem do paměti na místo `read()/write()`. Umožňují sdílený přístup k souborům.

# 39. Plánování a synchronizace procesů, transakce.

Správa procesů zahrnuje:

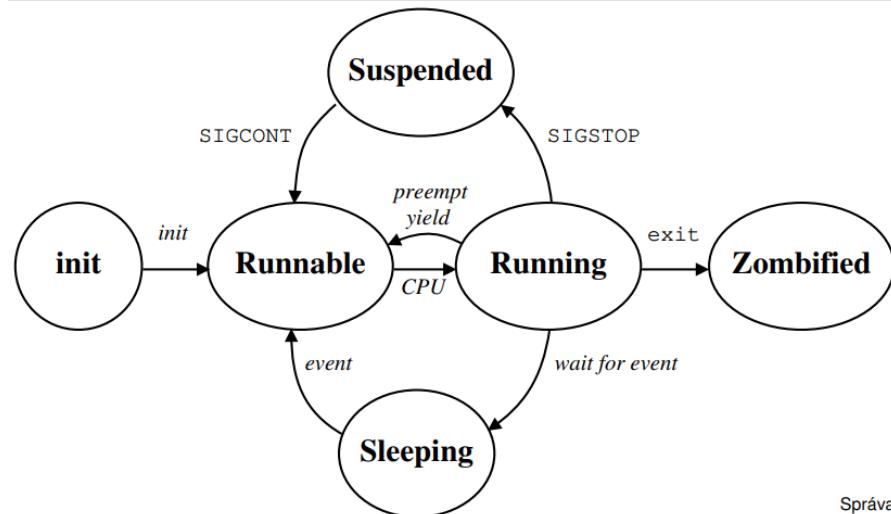
- **přepínání kontextu** (dispatcher): fyzické **odebírání** a **přidělování procesoru** procesům na základě rozhodnutí plánovače,
- **plánovač** (scheduler): rozhoduje, který proces poběží a případně jak dlouho.
- **správu paměti** (memory management): přidělování paměti procesům,
- **meziprocesovou komunikaci** (IPC): signály, sdílená paměť, roury, sockety,
- ...

## Proces

Jedná se o běžící program. V OS je identifikován a náleží mu:

- **identifikátor - PID**,
- **stav plánování** (běžící, pozastavený, čekající, připravený běžet, ukončený (zombie), ...),
- **program**, který jej řídí,
- **obsah všech registrů**,
- **zásobník** - rozpracované funkce,
- **data** - statická, hromada,
- **zdroje OS** - otevřené soubory, semafory, sdílená paměť, ...

stav	význam
Vytvořený	ještě neinicializovaný
Připravený	mohl by běžet, ale nemá CPU
Běžící	používá CPU
Mátoha	po exit, rodič ještě nepřevzal exit-code
Čekající	čeká na událost (např. dokončení read)
Odložený	"zmrazený" signálem SIGSTOP



## Process Control Block (PCB)

Je datová struktura, pomocí které je v OS **reprezentován proces** (někdy také control block nebo task struct). PCB zahrnuje (případně odkazuje na):

- **identifikátory** spojené s procesem (PID),
- **stav plánování** procesu (běžící, pozastavený, ...),
- **obsah registrů** (včetně EIP a ESP),
- **plánovací informace** (priorita),
- informace spojené se **správou paměti** (tabulky stránek, ...),
- Informace spojené s **účtováním** (doba využití procesoru, ...),
- **Využití I/O zdrojů** (otevřené soubory, používaná zařízení...).

## Části procesu v paměti v Unix

- **Uživatelský adresový prostor** přístupný procesu (kód, hromada, statická data, ...).
- **Uživatelská oblast** uložená pro každý proces spolu s **daty, kódem a zásobníkem** a částí **PCB** (PID, obsah registrů, fd, obslužné funkce pro signály, účtování, pracovní adresář, ...) v uživatelském adresovém prostoru, je **přístupná pouze jádru**.
- **Záznam v tabulce procesů**, který obsahuje informace o procesu, které jsou důležité, i když zrovna neběží (PID, stav plánování, událost, na kterou čeká, čekající signály, ...). Je **trvale uložen v jádře**.

## Kontext procesu

- **uživatelský kontext**: kód, data, zásobník, sdílená data,
- **registrový kontext**: obsah registrů,
- **systémový kontext**: systémové údaje o procesu tj. záznam v tabulce procesů, tabulka stránek, otevřené soubory procesem, ...

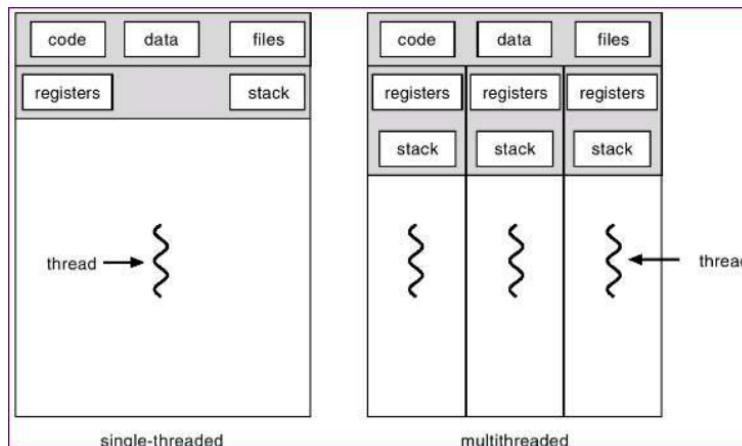
## Tvorba procesu

Nový proces vytváříme **duplikací aktuálního** pomocí funkce **fork** (vrací 0 v child procesu a PID potomka v rodičovském procesu). Duplikovaný proces je **takřka totožný**. Uživatelský adresový prostor má **stejný** obsah jako obsah rodiče, sdílí otevřené soubory, obsluhu signálu, ... Pro omezení plýtvání pamětí se používá **copy-on-write** (data se doopravdy duplikují, až když jeden z procesů začne zapisovat). Potomkovský proces se **liší** návratovou hodnotou fork (0), identifikátory, údaje o plánování a účtování, nedědí čekající signály a zámky souborů. Předkem všech uživatelských procesů je proces **init s PID=1**, tento proces přebírá návratový (tím je proces ukončen ze stavu zombie/mátoha) kód i procesů, kterým dřív skončí rodič.

Změna vykonávaného programu se provádí pomocí **exec()**.

## Vlákna

Jedná se o odlehčené vlákno, v rámci procesu jich může běžet více. Program je ve vláknech vykonáván paralelně. Vlákna mají své **vlastní registry a zásobník**, sdílí kód, data a další zdroje (otevřené soubory, signály, tabulkou stránek). Vlákna se **rychleji vytvářejí** a je **rychlejší** mezi nimi **přepínat**. Komunikace mezi vlákny je také jednodušší, protože sdílí data (nutno používat obezřetně - uzamykat). Pro paralelizaci úloh se běžně používají vlákna na místo procesů.



## Plánování procesu

Plánovač procesů (**scheduler**) rozhoduje, **který proces poběží** a případně **jak dlouho**. Existují dva druhy plánování:

- **Nepreemptivní plánování**: změnu procesu lze **provést pouze**, pokud aktuálně běžící proces **předá řízení jádru** (např. I/O operace, exit, yield), tj. dokud chce běžet, nic jej nemůže přerušit.
- **Preemptivní plánování**: změnu procesu může **jádro vynutit**, i když mu nepředá řízení, a to na základě **přerušení** (typicky od časovače, ale třeba i od disku, aj.).

Samotné **přepnutí** procesu řídí **dispečer (dispatcher)**. Plánování závisí na **prioritě** procesu a **dostupnosti zdrojů**, které potřebuje (např. paměti, dokončení IO operace, otevřání souboru, ...). **Spuštění nových procesů** se také plánuje a může být **pozdrženo** (z důvodu nedostatku paměti).

### Přepnutí procesu (kontextu)

Přepnutí kontextu provádí dispečer a znamená odebrání procesoru procesu A jeho přidělení procesu B, což zahrnuje:

1. **Úschovu stavů a registrů** procesu A do **jeho PCB**.
2. **Úprava** některých řídících **struktur v jádře** a třeba i zrušení záznamů procesu A v **TLB** (záleží na implementaci TLB).
3. **Obnova registrů** procesu B z **PCB**.
4. **Předání řízení** na adresu **procesu B**, odkud bylo dříve přerušeno.

Neukládá se a neobnovuje se **celý stav** procesu. Např. namapované stránky na rámce v paměti zůstávají, **mění se pouze ukazatel na tabulku stránek**, se kterou se bude aktuálně pracovat.

**Přepnutí** přesto trvá několik **stovek** nobo **tisíc instrukcí**, proto doba mezi přepínáním musí být dostatečně dlouhá, aby procesor nezávadil nepřiměřeně dlouho přepínáním kontextu.

## Plánovací algoritmy

Algoritmy pro plánování mohou upřednostňovat některý z požadavků na plánování procesu:

- Využití CPU
- Propustnost
- Doba běhu
- Doba čekání
- Doba odezvy

### First Come, First Served (FCFS)

Procesy čekají na přidělení procesoru ve frontě **FIFO**. Jedná se o **nepreemptivní algoritmus** (tzn. proces se musí sám vzdát procesoru žádostí jádra nebo voláním yield). Procesor je **přidělen** procesu na **začátku frontu** a proces, kterému je procesor **odebrán**, je zařazen **na konec fronty**, stejně jako nový proces.

### Round Robin

Algoritmus založený na principu **FIFO** jako FCFS, je ale preemptivní, tzn. procesům je přiděleno **časové kvantum** (dobu kterou mohou strávit výpočtem, než jim bude CPU odebráno), po **vypršení kvánu** nebo pokud se proces **vzdá** procesoru (yield nebo žádostí jádra např. o I/O) je zařazen na konec fronty a procesor je přidělen procesu na začátku fronty.

### Shortest Job First (SJF)

Algoritmus přiděluje procesor procesu, který požaduje **nejkratší dobu** pro své další provádění na procesoru **bez I/O operací**. Jedná se o **nepreemptivní** algoritmus.

**Minimalizuje** průměrnou **domu čekání** a **zvyšuje propustnost** systému, OS ale musí znát nastávající dobu procesu nebo jí aspoň dobře odhadnout. Používá se proto ve **specializovaných** systémech, kde se **opakovaně provádí podobné** úlohy. Má ale jeden zásadní problém - **stárnutí** (procesy s delším požadavkem na CPU se nemusí nikdy k CPU dostat).

### Shortest Remaining Time (SRT)

Jedná se o obdobu algoritmu **SJF**. Algoritmus je **preemptivní**, ale tím, že procesor aktuálně používá proces, který má **nejmenší dobu pro dokončení**, může k

preemci dojít pouze když dojde k **vytvoření** nového procesu s **kratším časem dokončení**, než je čas dokončení.

## Víceúrovňové plánování

Procesy jsou rozděleny do různých **skupin** (typicky dle **priority**). V rámci každé **skupiny** lze použít **jiný plánovací algoritmus**. Dále existuje **další algoritmus**, který **určuje** (nejčastěji na základě priorit), jaké skupině umožní výběr procesu (dle algoritmu dané skupiny). Může docházet ke **hledovění a inverzi priorit**.

## Víceúrovňové plánování se zpětnou vazbou

Procesy jsou rozděleny do **skupin dle priorit**. Plánování může proběhnout jednou z variant:

- Nově připravené procesy běžet jsou zařazovány do fronty s **největší prioritou** (používá se **Round Robin** - preemptivní). Jejich **priorita postupně klesá** a tím se přesouvají do front nižší úrovně, až nakonec jsou plánovány v nejméně prioritní frontě (tam se dostanou především dlouho trvající procesy).
- Proces je po spuštění zařazen do prioritní fronty dle jeho stanovené počáteční priority (statická priorita). Následně se jeho priorita mění (dynamická priorita):
  - **rosté** pokud proces často **čeká na I/O** (asi jej uživatel hodně používá, takže zajistíme jeho rychlou reakci)
  - **klesá** pokud proces **spotřebovává hodně** procesorového času.

## Completely Fair Scheduler (CFS)

Přiděluje procesor každému procesu tak, aby mu bylo přiděleno **odpovídající procento procesorového času dle jeho priority**. U každého procesu si tak musí vést údaj o **stráveném virtuálním** (vliv priority) procesorovém čase a procesor přiděluje procesu s **nejmenším** stráveným virtuálním časem. Procesy organizuje dle **stráveného času** (nejméně v kořeni) do **Red-Black Tree**. Nově vzniklým procesům musí být přidělen nějaký počáteční virtuální čas dle jeho priority (protože běžící procesy již strávili čas na CPU). Algoritmus je **preemptivní** a **časové kvantum** je procesu vypočteno **na základě priority** (respektive méně prioritním procesům ubíhá čas rychleji než prioritním). Po přepnutí kontextu je nově proces, jemuž byl odebrán procesor, zpět vložen na **odpovídající místo v RB-Tree**. Algoritmus také umožňuje plánovat procesy **více uživatelům** nebo z **více terminálů** s různou prioritou.

## Plánovače v Linux a Windows

Procesy jsou rozděleny do několika (desítky) úrovní priority. **Nejvyšší** priority mají procesy **reálného času** (plánovány pomocí Round Robin) **střední** prioritu mají **běžné procesy** (plánovány pomocí CFS) **nejnižší** prioritu mají **idle** procesy (většinou zaměstnávají CPU, když není co na práci). Procesy jsou startovány se **statickou prioritou**. Dynamická priorita se poté může měnit na základě chování procesu (zejména na Win):

- **Priorita se zvyšuje:** okno procesu je na **popředí**, do okna procesu **přichází zprávy** (myš, klávesnice, ...), proces je **uvolněn z čekání na I/O**.
- **Priorita klesá:** Proces spotřebovává moc CPU času, po vyčerpání každého kvanta.

## Inverze priorit

Je stav, kdy efektivně priorita nízko prioritního procesu přesáhne prioritu vysoko prioritního procesu. Stane se tak, pokud **nízko prioritní proces obsadí nějaký zdroj**, je mu odebrán CPU, než jej uvolní. Následně některý z **vysoce prioritních procesů potřebují tento zdroj**, ale musí na něj čekat. Nízko prioritní proces mezi tím **předbíhají jiné procesy** s vysokou a střední prioritou. Vysoko prioritní proces se tak dostane na řadu, až za dlouhou dobu (nejdřív musí být uvolněn požadovaný zdroj). Inverze priorit nemusí být problém, zvyšuje však obvykle odezvu systému a v případě **real-time procesů** pracující např. s HW může být **kritická**. Inverzi priorit lze řešit:

- procesům v **kritické sekci** je přiřazena **největší priorita**,
- procesy, na které **čeká jiný proces dědí jeho prioritu** (pokud je vyšší),
- **zákaz přerušení** pokud je proces v **kritické sekci**.

## Komplikace plánování procesů

- **Více procesorové systémy** (dnes snad všechny) vyžadují **vyvažování** výkonu jednotlivých CPU (jader), zde je ale navíc **problém s obsahem cache** atd.
- **hard real-time systémy** u kterých musí plánovač zajistit (**garantovat**) určitou **odezu**.

## Komunikace mezi procesy

**IPC = Inter-Process Communication**, provádí se pomocí:

**signály** (kill, signal, ...),  
**roury** (pipe, mknod p, ...),  
**zprávy** (msgget, msgsnd, msgrcv, msgctl, ...),  
**sdílená paměť** (shmget, shmat, ...),  
**sockety** (socket, ...).

Procesy jsou členěny na:

- **Úloha** - Skupina paralelně běžících procesů spuštěných jedním příkazem a propojených do **pipeline** (|).
- **Skupina procesů** - Množina, které je množné **poslat signál jako jedné jednotce** (může mít vedoucího - tím je implicitně první proces), procesy jsou přidávány do stejné skupiny jako rodič, lze ale změnit.
- **Sezení** - Každá skupina procesů je v **jednom sezení** (může mít vedoucího).

## Signály

Jeden z principů komunikace mezi procesy. Jedná se o **číslo (int)**, které je zasláno procesu pomocí **speciálního rozhraní**, které je pro to určeno. Lze zde najít **analogii** s přerušením na SW úrovni. Také se **dějí asynchronně** a také pro ně **definujeme obslužné funkce**. Signály jsou generovány při:

- chybách (chybná práce s pamětí, aritmetická chyba - dělení 0, ...),
- externích událostech (vypršení časovače, dokončení I/O, ...),
- na žádost procesu (např. rodič posílá signál potomkovi, signál kill, ...).

**Obsluha** signálů může být **složitá a potenciálně obsahovat chyby** (kvůli asynchronnímu vzniku signálů), při obsluze **nelze použít některé funkce**, protože by to mohlo ovlivnit normální běh procesu např. **printf()**. Pokud vzniknout dva signály hned za sebou, může být druhým přerušena obsluha prvního. Příklady: SIGHUP, SIGINT, SIGKILL, SIGALARM, SIGSTOP, SIGCONT, SIGCHILD, SIGUSR1, SIGUSR2, ... Kromě **SIGKILL** a **SIGSTOP** lze signály předefinovat. Implicitní reakce na signály jsou buď ukončení procesu (časté), ignorování signálu, zmrazení procesu, nebo rozmrazení procesu. Signály až na SIGKILL, SIGSTOP, SIGCONT lze blokovat. Nastavení **obsluhy signálů** včetně blokování se **dědí** na potomky **při fork**. Při **exec** se nastaví **implicitní obsluha**. Signály se **zasílají** (funkce **kill**) na základě **znalosti PID** případně lze zaslat všem procesům určité skupiny (např potomkům). Čekání na signály pomocí **pause()** nebo **sigsuspend()**.

## Synchronizace procesů

Synchronizaci provádíme, aby při **paralelném přístupu** ke sdíleným zdrojům (sdílená paměť, soubory, I/O zařízení, ...) nedocházelo k chybám způsobených **nesprávným pořadím** provedených dílčích **operací** různými procesy. Např. 2 procesy sdílí paměť:

proces 1 kód: <pre>N++;</pre> assembler: <pre>register = N register = register + 1 N = register</pre>	proces 2 <pre>N--; register = N register = register - 1 N = register</pre>
--	---

Pokud je **N** na začátku **1**, mělo by být **1** i po provedení následujícího kódu (oběma procesy), ale může být také **0** nebo **2**.

## Race condition

**Časově závislá chyba** (souběh) je chyba, která vzniká při **přístupu ke sdíleným zdrojům** kvůli **různému pořadí** provádění jednotlivých **paralelních výpočtů** v systému.

## (Sdílená) kritická sekce

Jedná se o **úseky v programech**, které řídí **paralelně** běžící procesy, ve kterých dochází k přístupu ke **sdíleným zdrojům**. Provádění jednoho z **úseku** jedním procesem **vyloučuje** provádění **libovoľného z týchto úseků** ostatními procesy.

## Problém kritické sekce

Jedná se o problém zajištění **korektní synchronizace** procesů na množině **sdílených kritických sekcí**. Je nutné **zajistit**:

- **vzájemné vyloučení**: **nanejvýš jeden** proces je v daném **okamžiku** v dané množině **kritických sekcí**. Případně je tam nanejvýš **k** procesů.
- **dostupnost kritické sekce**: Je-li kritická sekce **volná**, respektive je volná aspoň v určitých okamžicích, proces **nemůže čekat neomezeně dlouho** na **přístup** do ní.

Musíme se **vyhnout**:

- uváznutí (deadlock),
- blokování (blocking),
- stárnutí (hladovění - starvation).

## Data race

Časově závislá chyba nad daty nebo také souběh nad daty. K této chybě může dojít, pokud ke zdroji s **výlučným** přístupem přistupuje **více procesů bez synchronizace** (2 a více) a alespoň **jeden** k němu přistupuje **pro zápis**.

## Deadlock (uváznutí)

Jedná se o situaci, kdy **každý proces** z určité **neprázdné** množiny procesů je **pozastaven a čeká** na uvolnění nějakého zdroje s **výlučným přístupem**, který je **vlastněn** nějakým procesem z dané množiny, který **jediný jej může uvolnit**, a to až po dokončení práce s ním. (Ale tu dokončit nemůže, protože čeká). Takže každý proces čeká na jiný proces - v **grafu** čekání se vytvořila **kružnice**).

## Podmínky uváznutí (Coffmanovy podmínky)

1. **Vzájemné vyloučení** při použití **prostředků** (tj. k prostředku může v jeden čas přistupovat pouze jeden proces),
2. **vlastnictví** alespoň **jednoho** prostředu (zdroje), pozastavení **a čekání** na další (tj. proces má alokovaný nějaký zdroj, potřebuje i jiný, o něj žádá, ten není ale k dispozici, tak čeká),
3. **prostředky vrací proces**, který je **vlastní**, a to až po dokončení jejich používání (tj. pokud má proces alokovaný nějaký zdroj, např. I/O zařízení, pouze tento proces může říct, že už jej nepotřebuje, a to až v moment, kdy jej doopravdy nepotřebuje - dokončil čtení/zápis),

4. **cyklická závislost** na sebe čekajících procesů (tj. proces A čeká na proces B a proces B čeká současně na proces A).

### Řešení uváznutí

- **prevence uváznutí:** porušení jedné z **Coffmanových podmínek**,
- **vyhýbání se uváznutí:** kontrola grafu **žádostí a vlastnictví zdrojů**, zamezení vzniku kružnice.
- **detekce a zotavení:** existuje speciální proces (mimo množinu procesů, na které došlo k uváznutí), který detekuje vznik uváznutí a nějakým způsobem jej vyřeší.

### Blocking (blokování)

Blokování při přístupu do kritické sekce je situace, kdy proces, který žádá o přístup do kritické sekce **musí čekat**, i když je **kritická sekce volná** (žádný proces se nenachází v žádné z množiny těchto kritických sekcí - jinak řečeno, **žádný proces nepoužívá daný sdílený zdroj s výlučným přístupem**).

### Starvation (stárnutí)

Stárnutí je situace, kdy **proces čeká** na splnění podmínky, která **nemusí nikdy nastat**. V případě kritické sekce se jedná o splnění podmínky, která umožňuje vstup do kritické sekce. V případě plánování procesů se jedná o situaci, kdy proces nemusí být nikdy plánovačem naplánován pro běh na procesoru (předbíhají jej jiné procesy). Uváznutí (deadlock) i blokování (blocking) lze zobecnit na stárnutí.

### Livelock

Jedná se o zvláštní situaci stárnutí a v kombinaci s uváznutím. Každý proces z určité neprázdné množiny **běží** ale provádí jen **omezený úsek kódu**, ve kterém **opakovaně** žádá o přístup do **kritické sekce** (žádá o zdroj s výlučným přístupem), ve které je jiný proces z dané množiny procesů, který **jediný jí může uvolnit**, pokud by mohl pokračovat.

### Zajištění výlučného přístupu do KS - synchronizace

Výlučný přístup do KS se zajišťuje pomocí tzv. **spinlock**. Jedná se o řešení, které umožňuje **aktivní čekání**. Spinlock lze implementovat pomocí:

- **speciálních algoritmů**, které nevyžadují atomické operace. Např.
  - **Petersonův algoritmus:** Proces nastavením příznaku vyjádří svůj zájem o kritickou sekci, ale zároveň dá přednost jinému procesu. Následně čeká na uvolnění kritické sekce, nebo až mu druhý proces vrátí přednost.
  - **Bakery algoritmus L. Lamporta:** Před vstupem do KS získá proces **lístek**, jehož hodnota je větší než hodnota lístků procesů, které již

lístek mají a čekají. Pokud je hodnota stejná (kvůli tomu že lístek procesy získaly současně - souběžně četly největší hodnotu), rozhoduje se podle velikosti **PID**. Držitel **nejmenšího čísla lístku může vstoupit** do kritické sekce,

- **pomocí atomických instrukcí**, jejíž atomicitu zajišťuje HW implementace, např. instrukce **SWAP**.

**Aktivní čekání** lze využít na krátkých neblokujících sekcích bez preemce (tj. zákazem přerušení). Tyto krátké sekce se mohou nacházet např. v **implementaci operací** nad **semaforem** nebo **monitorem**, nikoliv však v uživatelském kódu. Klasickými synchronizačními problémy jsou např. **producent a konzument** (producent zapisuje do fronty, konzument odebírá), **čtenář a písář** (současně může číst libovolný počet čtenářů, při zápisu však může přistupovat pouze písář).

## Semafor

Synchronizační nástroj, který **minimalizuje** (případně úplně odstraňuje) **aktivní čekání**. Pokud proces aktuálně nemůže vstoupit do kritické sekce (tj. uzamknout semafor - uzamčení musí být **atomické**, zde se vyskytuje **spinlock** a **zákaz přerušení**), je **pozastaven** (nečeká aktivně) a umístěn do **fronty čekajících procesů** (pořadí procesů není garantováno) na uvolnění KS. Obecně mohou být semafory implementovány pomocí **celočíselné proměnné a fronty**. Proměnná udává, kolik procesů může ještě vstoupit do KS. Pokud je **0 a menší** sekce je **uzamčena**. Zvláštním (a nejpoužívanějším) případem je poté **binární semafor - mutex**, který umožňuje vstup do KS pouze **jednomu procesu**, který jediný jej může zpět odemknout.

## Monitor

Jedná se o vysokoúrovňový synchronizační prostředek. Zapouzdřuje data a poskytuje operace, které zajišťují správné provádění operací nad chráněnými daty.

- **wait**: uspí dané vlákno/process (pasivní čekání),
- **notify**: **probudí jeden** (pokud nějaký existuje, jinak prázdná operace) z čekajících procesů (dříve volaly wait), ale **pokračuje aktuální proces** (vlákno), který zavolalo notify.
- **signal**: **probudí jeden** (pokud nějaký existuje, jinak prázdná operace) z čekajících procesů (dříve volaly wait) a **pokračuje nově probuzený proces**.
- **notifyAll**: **probudí všechny** čekající procesy (pokud nějaké čekají, jinak prázdná operace).

## Prevence uváznutí (podrobněji)

**Zrušíme platnost** některé z nutných **podmínek** uváznutí (**Coffmanovy podmínky**). To lze zabudovat přímo do celého implementovaného systému a nějak ověřit (verifikace), nebo pro tento účel využívat systém přidělování zdrojů, který bude vždy

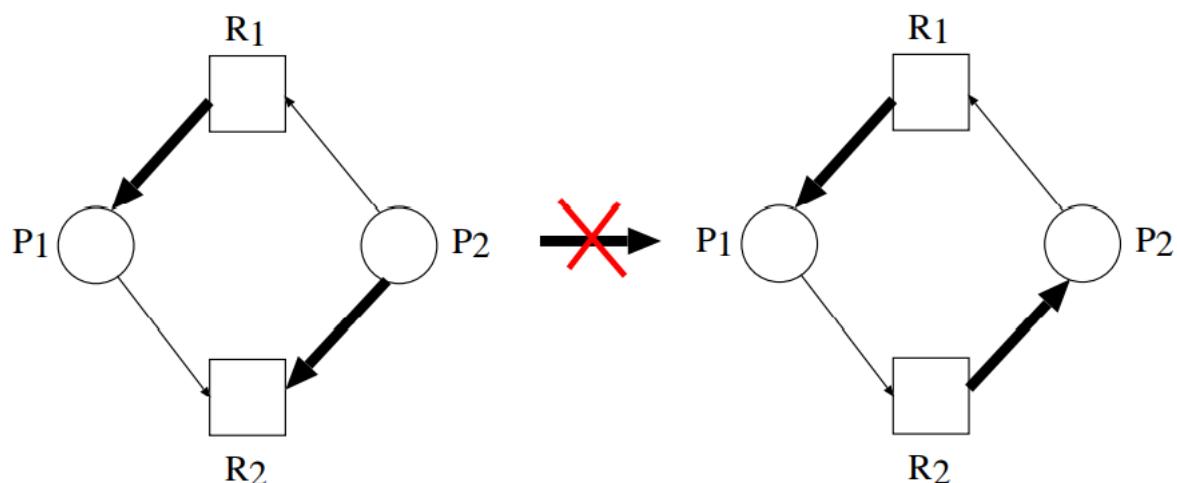
vynucovat porušení alespoň jedné podmínky. Příklady řešení pro jednotlivé podmínky:

1. **Nepoužíváme** sdílené prostředky nebo používáme takové, ke kterým **není** nutné přistupovat **výlučne** (může k nim současně přistupovat více procesů).
2. **Statická** (tj. program neběží) kontrola kódu, že proces žádá o zdroj (a tedy může být pozastaven), jen v případě, že aktuálně **žádný zdroj nevlastní**. Nebo **dynamická** kontrola uvedeného, pak je ale nutné řešit, co s takovým procesem. (Nabízí se jeho ukončení a odebrání jím vlastněných zdrojů, to ale může vést na nekonzistence a nedokončení některých úloh).
3. Při pozastavení procesu (neúspěšné žádosti o další zdroj) jsou mu **odebrány všechny jím vlastněné** zdroje. Ty jsou mu **vráceny** včetně nově žádaného zdroje, **až** jsou **všechny k dispozici** a proces je znova spuštěn. (Odebrání zdrojů v průběhu práce s nimi může způsobit nekonzistence).
4. **Očíslování prostředků** a jejich přidělování v **pořadí**, které **vylučuje** vznik cyklické závislosti (tj. **vzestupně** nebo **sestupně**).

### Vyhýbání se uváznutí (podrobněji)

Obecně lze problém řešit tak, že procesy **deklarují**, jaké zdroje **aktuálně potřebují** a na základě aktuálního **stavu přidělených** zdrojů a těchto **žádostí o zdroje** se rozhodne, jestli **může** být dané žádosti **vyhověno**, nebo proces **musí čekat**. Žádosti o zdroj je vyhověno, pokud aktuálně **nemůže vzniknout cyklická** závislost na sebe čekajících procesů a **cyklická závislost nebude možná ani v budoucnu**, a to při zohlednění té **nejhorší možné** situace.

V praxi se tento problém řeší pomocí grafových algoritmů, konkrétně pomocí **grafu alokace zdrojů**. Zdroj je **přidělen** jen tehdy, pokud **nehrozí vznik cyklu v grafu**, jinak proces musí čekat. (Vznik cyklu v grafu neznamená vznik uváznutí, pouze značí, že k uváznutí může dojít).



## Detekce uváznutí (podrobněji)

Na dané neprázdné množině procesů **může** dojít k **uváznutí**, ale existuje nějaký **další proces mimo tuto množinu**, který periodicky (nebo nějak jinak) **kontroluje**, zda k uváznutí **došlo**, a pokud ano, **provede zotavení** (zajistí, že uváznutí bude zrušeno).

**Detekce** uváznutí může být v konkrétním případě řešena **pomocí grafu vlastnictví a čekání na zdroje**. **Cyklus** v tomto grafu znamená vznik **uváznutí**.

**Zotavení z uváznutí** lze nejjednodušeji řešit **ukončením** (restartem) některého z procesů - **victim** (děje se tak např u databázi v SŘBD) a případně doplnit ukončení procesu o **rollback** (viz transakce v DB). Dále lze zotavení řešit **odebráním zdrojů** některému procesu a jejich **přidělení jinému** a poté zajistění, že tomuto procesu budou opět přiděleny včetně těch, co potřeboval původně.

## Transakce na úrovni OS

Jedná se o druh zpracování, při kterém je **skupina logických operací - transakce**, provedena jako **jeden celek**, tj. buď **vůbec nebo všechny**. Pokud se při zpracování v rámci transakce vyskytne jakákoli **chyba** a transakce nemůže být dokončena, všechny dílčí operace musejí být **vráceny do stavu před začátkem transakce**. Jejich zpracování se musí **vypořádat s výskytem poruch a paralelismem**. Definice z databází pomocí **ACID** zde nemusí být úplně přesná. Např. dosažení **trvalosti** (durability) nemusí být možné, pokud za transakci považujeme přepnutí kontextu (asi těžko zajistíme, že po dokončení a přepnutí kontextu a následném pádu systému bude přepnutí kontextu trvalé). U transakcí na úrovni OS tak zajistujeme zejména **A** (atomicity - atomicitu), tedy že transakce je provedena v celém svém rozsahu, nebo není provedena žádná z jejich částí (nemohou být při přepnutí kontextu ponechány stejné registry a pouze změněn zásobník).

Typickým **příkladem** transakčního zpracování na úrovni OS je **zápis na disk**. Využívá se zápisu do **žurnálu** a řešení chyb pomocí **REDO a UNDO**, tedy velmi podobný princip databázové transakci. Těžko ale v tomto případě zajistíme **C** (consistency, konzistence) zapsaných dat **ve smyku jejich významu**, můžeme pouze zajistit, že byty jsou zapsané přesně tak, jak vyžaduje proces.

# 40. Objektová orientace (základní koncepty, třídně a prototypově orientované jazyky, OO přístup k tvorbě SW).

**Objektově orientované programování** (OOP) je programovací paradigmum, které se začalo hojně objevovat v 80. letech minulého století. Základem tohoto paradigmatu je abstrahovat a modelovat principy reálného světa. Řeší se tak pomocí objektů a jejich vzájemné komunikace. V dnešní době je toto paradigmum jedno z nejrozšířenějších, zejména pro velké aplikace. Mezi **výhody** patří **analogie reálného a softwarového modelu, flexibilita, znovupoužitelnost**. Nevýhodou může být složitější sémantika, delší učící se křivka či režie spojená s prací s objekty. Mezi základní koncepty OOP patří:

- **Abstrakce** (Abstraction): pomocí objektů,
- **Zapouzdření** (Encapsulation): pomocí viditelnosti atributů,
- **Mnohotvárnost** (Polymorphism): abstraktní funkce a VMT,
- **Dědičnost** (Inheritance): generalizace/specializace.

## Klasifikace objektově orientovaných jazyků

- Dle přístupu k vytváření objektů:
  - **Beztrídní** OOJ (JavaScript, Lua - prototype-based, class-less),
  - **Třídní** OOJ (C#, C++ - class-based)
- Dle čistoty objektového modelu:
  - **Čisté** (Ruby, Python - vše je objekt),
  - **Hybridní** (C#, C++ - míchání s jinými paradigmaty, doplněny objekty),
- Dle platformy pro běh OO programů:
  - **Překládané** (C++ - efektivita běhu, více zdrojového textu),
  - **Interpretované** (Python, PHP - pomalé, velmi flexibilní),
  - **Částečně interpretované** (C#, Java - bytecode, mezikód, virtuální stroj)
- Dle dědičnosti (počet přímých předků):
  - **Jednoduchá** (C#, Java - oba tyto jazyky ale podporují vícenásobnou dědičnost rozhraní),
  - **Vícenásobná** (C++, Python - problematická, nutno řešit kolize)
- Dle předmětu dědění:
  - **Dědičnost implementace** (C++, C#, Java, Python, ...)
    - Třídní dědičnost: nadřída (superclass), podřída (subclass)
    - Delegace
  - **Dědičnost rozhraní** (C#, Java)

## Klasifikace nejen OOJ

- Dle způsobu určení typů:
  - **Beztypové** (Lambda kalkul - nemají žádný typ),
  - **Netypované** (Python, JavaScript - uživatel nepracuje s typem, typ ale mají a lze na vyžádání zjistit),
  - **Typované** (C++, C# - uživatel explicitně specifikuje typ, nebo musí být odvoditelný)
- Dle důslednosti kontroly typů:
  - **Silně typované** (Rust, Go - nelze provádět implicitní konverze, type-safe),
  - **Slabě typované** (C, C++ (méně než C) - implicitní konverze)
- Dle doby kontroly typů:
  - **Staticky typované** (C, C++ - při překladu, před spuštěním),
  - **Dynamicky typované** (Python, JavaScript - za běhu, run-time).

## Minimální model OO výpočtu

Minimální výpočetně úplný OO model výpočtu potřebuje pouze:

- proměnné,
- konstrukt objektu (způsob vytváření objektů),
- zasílání zpráv (komunikace mezi objekty).

## Základní koncepty OOP

Tyto koncepty by měly být více méně shodné pro každý OOJ.

### Abstrakce - Objekt

Objekt je **autonomní výpočetně úplná entita**, obvykle je tvořena **atributy** a **metodami**. Identita objektu ale nezávisí na jeho attributech. U třídních jazyků jsou objekty **instancemi tříd**.

### Kompozice

Objekt může obsahovat jako položky i jiné objekty.

### Zapouzdření

Jedná se o **uzavřenost** vůči okolním objektům, dnes je to obvykle implementováno pomocí **identifikátorů viditelnosti (public, private, protected)**. V případě **zpráv** mluvíme o přístupu přes:

- **veřejný** protokol umožňuje okolním objektům zasílat zprávy tomuto objektu.  
Zaslání zprávy veřejným protokolem může vést na:
  - **chybu** - objekt nerozumí zprávě,

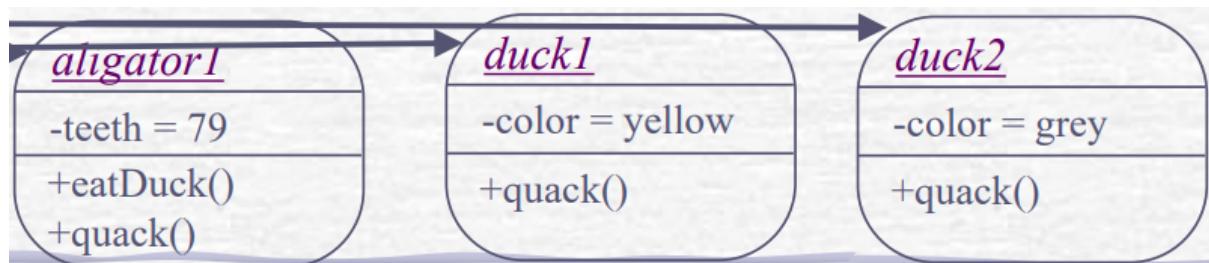
- **invokaci** odpovídající metody a **navrácení** výsledné hodnoty odesílateli.
- **interní** protokol se používá, když objekt zasílá **zprávu sám sobě** (this, self). Zaslání zprávy interním protokolem může vést na:
  - **chybu** - objekt nerozumí zprávě,
  - **přístup k atributu** (čtení nebo zapsání),
  - **invokaci** odpovídající metody a návratu výsledku,

## Zpráva

Zpráva je tvořena z **příjemce** (objekt, kterému je zpráva zaslána), **selektoru** (metody) a **argumentů**.

## Polymorfismus (Mnohotvárnost)

Vychází z toho, že **stejnou zprávu lze zaslat různým objektům** (často je to ale **omezeno** typovým systémem, aby nedocházelo k chybám typu "nerozumím zprávě") a stejné zprávy lze zasílat různým objektům jen v rámci dědičnosti). Protokol umožňuje **individuální reakci** na zaslané zprávy (volající díky zapouzdření nezná implementaci zprávou invokované metody, implementace se u různých objektů může lišit).



## Dědičnost

Vychází z toho, abychom **nemuseli** neustále **opakovat implementaci** podobných vlastností. Jedná se o jednu z hlavních předností OOP, a to **znovupoužitelnost**. Dědičnost umožňuje **sdílení** společných **položek** (atributy a metody) **od předků** a možnost definovat **individuální položky v potomcích**. Zděděný (specializovaný) objekt (třída) tak sdílí všechny atributy a metody předka (jejich použití objektem závisí na viditelnosti - musí být **public** nebo **protected**) a dále může být specializován:

- **přidáním** nových atributů a metod,
- **modifikací** metod (**override**),
- někdy lze také zakázat některé položky, většinou ale **ne**, viz dále.

## Zahrnutí typu (subsumption)

Pokud B je podtřída třídy A, lze instanci třídy B využít kdekoliv je očekávána instance třídy A. Pak se provádí přístup k **objektu třídy B přes protokol A**.

## Problém vícenásobné (třídní) dědičnosti

- nadtíry obsahují položky **stejného jména a typu**,
- nadtíry obsahují položky **stejného jména**, ale **různých typů**,
- **inicializace instancí** - pořadí volání konstruktorů,
- **uložení instancí v paměti** - instance třídy **C** (**C** je přímá podtřída **A** i **B**) lze využít **kdekoliv** očekávám instance tříd **A** nebo **B**.

## Řešení problémů vícenásobné (třídní) dědičnosti

Nejjednodušší je vícenásobnou dědičnost **zakázat**, stejně se ukázalo, že není často při programování nutná. Pokud ji ale potřebujeme:

- **Metoda stejného jména a typu**: zakázat, použít první výskyt (dle pořadí zapsání dědění v kódu), programátor musí explicitně vybrat jednu, skrytí (např. `A::m()` přístupná jen v A, ne v C).
- **Metody stejného jména a různých typů (parametrů)**: zakázat, povolit přetěžování metod (overloading), lze ale pouze u metod s rozdílnou signaturou (návratová hodnota není součást signatury), také s rozdílnými parametry.
- **Atributy stejného jména a typu**: zakázat, skrytí a existence obou (např. `A::d` přístupný jen v A, ne v C), sloučení.
- **Atributy stejného jména a různého typu**: zakázat, nebo ponechat, ale při práci s objektem musí být možné atributy vždy odlišit.

## Tvorba nových objektů

- Vytvoření **prázdného objektu** a přidání položek (atributů a metod).
- Klonování (kopie) a přidání či úprava položek. Klonovanému objektu říkáme **prototyp**. Kopie může být **mělká** (pouze 1. úroveň atributů), nebo **hluboká** (celý objekt). Princip klonování používají beztřídní OOJ.
- Vytvořením pomocí **předlohy - třídy** a naplnění atributů hodnotami. Děje se tak pomocí **konstruktoru**. Pro každý objekt je definována jeho třída - předloha. Třída také může být objektem první kategorie.

## Vzájemné vazby mezi třídami/objekty

Řeší se přes **ukazatele/reference** a **dopřednou deklaraci**. Dopředná deklarace se poté používá i pro metody a ty se definují mimo tělo funkce, aby v metodách bylo možné odkazovat atributy cyklicky závislých proměnných.

## Třídně orientované jazyky

Třídně orientované jazyky využívají pro tvorbu nových objektů - **instancí šablony**, které nazýváme **třídy**. Třída může být sama o sobě objekt nebo pouze entita, která obsahuje:

- seznam **instančních atributů** (atributů, které bude mít objekt po vytvoření) včetně **metadat**,
- **data třídních atributů** (pokud je třída taky objekt),
- implementace **instančních metod** (sdílené mezi instancemi),
- reference na její třídu,
- **statické** (!= třídní) položky - **atributy a metody** (pokud je třída entitou jazyka a není objekt).

**Instance - objekt** je poté tvořen svoují **identitou**, **referencí na třídu**, **daty instančních atributů**.

## Třída jako objekt první kategorie

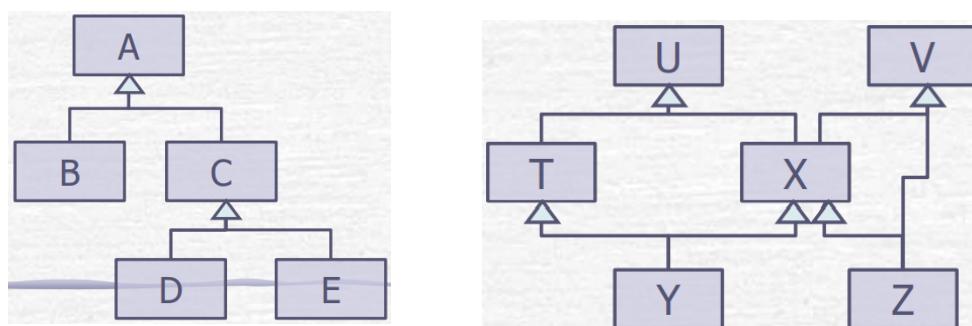
Třída je také objekt a pracuje se s ní analogicky jako s objekty, tj. **zasíláním zpráv**. Ve třídních metodách lze použít **self/this** parameter pro přístup k objektu třídy.

## Třídní dědičnost

Stejný význam jako dědičnost na úrovni objektů (ve většině jazyků používáme hlavně třídní dědičnost). Dědičnost tříd je šablonou pro dědičnost vzniklých (instanciovaných) objektů. Hlavní účel je **sdílení/znovupoužití** položek skrz třídy, **rozšíření** o nové položky a **redefinice** u specializovaných tříd.

### Hierarchie třídní dědičnosti

Matematicky můžeme mluvit o dědičnosti jako **relaci částečného uspořádání na množině tříd**. Hierarchie třídní dědičnosti je poté tvořena grafem relace dědičnosti. Levý obrázek ukazuje dědičnost u jazyků pouze s **jednoduchou** dědičností, na pravém je znázorněna vícenásobná dědičnost.



## Statické volání funkce (žádná vazba)

Jedná se o běžnou metodu/funkci, která je pouze ve jmenném prostoru třídy. Uvnitř metody **nelze** používat **self/this** parameter, není implicitně předán. Může ale používat jiné statické proměnné třídy.

## Brzská (časná) vazba

Při překladu je jasné, jaká implementace bude volána, **není** třeba mít **odkaz na metodu** v konkrétním objektu, překladač určí volání metody na základě jeho typu. V metodách lze odkazovat objekt, nad kterým je metoda volána pomocí **self/this** parametru, který je do metody implicitně předán. (ve skutečnosti překladač převede zápis `obj.call()` na `call(obj)`)

## Pozdní vazba

Umožňuje **polymorfismus** (volání virtuálních metod), volání metody nad objektem se řeší **až za běhu** dle jeho konkrétního typu. Stejně jako u brzké vazby lze v polymorfních metodách odkazovat objekt nad kterým je metoda volána (**self/this**). U překládaných OOJ se řeší pomocí **tabulky virtuálních metod**.

## Implementační problémy

Problémy implementace vychází z toho, že OOJ používají **objektovou paměť**, která je **asociativní a heterogenní** a modelem výpočtu je **zasílání zpráv** (invokace metod) objektům. Paměť počítače je ale **homogenní a neasociativní** a modelem výpočtu je **volání instrukcí** a použití **zásobníku**. Je nutné řešit:

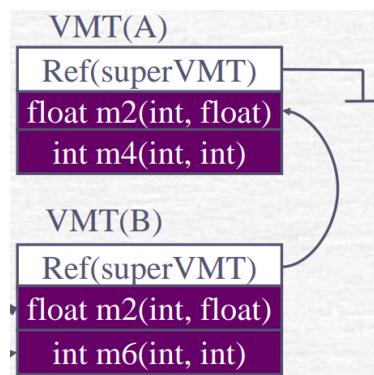
- **Uložení instancí a přístup k atributům**, tak aby byla reflektovaná dědičnost. Problematické je zejména zajistit přístup k atributům objektu při vícenásobné dědičnosti, tak aby šlo k objektu, který dědí z více objektů, přistupovat protokolem každého z obecných objektů. U jednonásobné dědičnosti to nemusí být problém, hodnoty specializovaných atributů se ukládají za hodnoty obecných atributů a při přístupu se specializované atributy ignorují (paměť se přetypuje na obecný objekt). Ale jak uložit atributy při mnohonásobné dědičnosti? Pouhé „přetypování“ nebude fungovat.
- **Uložení a invokace polymorfních metod**. Ukládání kódu metody v objektu je **nesmysl**, stačí metodu ukládat **jednou**, např. ve třídě a objekt implicitně předávat jako parametr (nultý parametr) při volání metody. Do objektu by tedy mohlo stačit uložit **odkazy na metody**, což řeší i **polymorfismus** tj. každý objekt má odkazy na takové metody, které doopravdy implementuje. Tento způsob je jednoduchý ale má 2 zásadní problémy:
  - **přístup k metodám předků** (pomocí **super/base**) po jejich redefinici - objekt by musel obsahovat odkazy na jeho redefinici metody, všechny různé redefinice metod předky a originální definici metody,
  - **plýtvání pamětí** - při vytvoření tisíce stejných objektů bude v paměti uloženy zbytečně tisíce stejných odkazů na metody.

Řešením problémů je použití **tabulky virtuálních metod** (virtual method table - VMT), kterou odkazuje objekt na místo všech jeho metod.

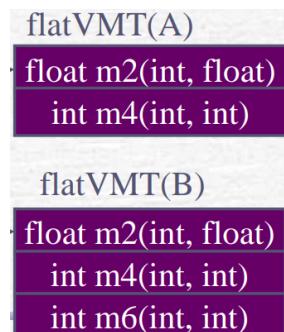
## Tabulka virtuálních metod (VMT)

Je to opravdu **tabulka virtuálních metod** (**NE** virtuální tabulka). Pro každou třídu (typ objektu) existuje **pouze jednou** a všechny instance (objekty) této třídy na ní odkazují. Dědičnost je řešena tak, že z VMT specializovaného objektu vede odkaz na VMT jeho předka a z VMT předka odkaz na VMT dalšího předka atd.

Tento princip **zanořeného** odkazování je ale **pomalý**, proto se za cenu mírného zvýšení spotřeby paměti tabulka pro danou třídu odkazuje **všechny polymorfní metody**, i když nebyly přepsány (**flat table**). Tabulka **neobsahuje** odkaz na **rodičovskou** tabulku. Rodičovskou tabulku při volání přes **self/this** dokáže určit



**překladač** na základě typu **self/this**, protože každé **tabulce přiděluje statickou adresu**.



## Vytváření instancí (objektů)

Většinou se provádí pomocí klíčového slova **new** a je tvořena ze 2 kroků:

1. **Přidělení paměti** (většinou na hromadě) na pro danou instanci (objekt).
2. **Inicializace atributů** pomocí **konstruktoru** (speciální metoda), nutné si v konkrétním jazyce zjistit, jaké je pořadí volání konstruktorů při dědičnosti.

## Rušení instancí (objektů)

Při rušení objektů se volá speciální metoda tzv. **destruktur** (opět je nutné znát pořadí volání destruktorů při dědičnosti) a může být provedena:

- **Implicitně** pomocí správce paměti (**Garbage Collector**) objekty jsou z paměti mazány, když již jsou nedosažitelné. Programátor **nemá pod kontrolou** kdy

se tak stane a jak dlouho to zabere → nelze použít např. u real time systémů s požadovanou odezvou. Algoritmy uvolňování ale obvykle **nezpůsobují** memory leaks.

- **Explicitní** použitím klíčového slova **delete** (nebo jiného) řeší **programátor** a má nad uvolňováním paměti **plnou kontrolu**, nicméně **může** vést na **memory leaks**.

## Reflektivita/Reflexe (Reflection)

Vlastnost jazyků (interpretovaných a částečně interpretovaných), která umožnuje:

- zkoumat vnitřní reprezentaci (**Introspection**) entit programu (objektů),
- měnit vnitřní reprezentaci (méně časté).

Na základě prováděných změn a zkoumání vnitřní struktury dělíme reflektivitu na:

- **Strukturální**: pracuje se **statickými strukturami** (balíčky, třídy, metody), např. zjišťujeme názvy atributů objektu.
- **Behaviorální**: pracuje s **prováděním programu** (invokace metody, přiřazení, zásobník volání), např. voláme metody na základě hodnoty řetězce, který obsahuje její název.

Reflektivita umožňuje jednodušeji vytvářet generický software. Nejlepším příkladem je serializace a deserializace. Na základě reflexe můžeme zjistit názvy atributů, získat jejich hodnoty a vytvořit např. JSON reprezentaci. Taková funkce pak může mít za parametr libovolný objekt a nepotřebujeme mít speciální funkci pro každý objekt.

## Typy vs. Třídy

- **Typ** udává množinu validních hodnot a operací nad nimi,
- **Třída** udává množina vnitřních stavů a operací nad nimi.

Typy jsou obecnější než třídy. Třída často určuje typ, ale né naopak. Typ může být určen:

- **Jméinem** např. int, Car, Person, ... Pak pokud máme funkci s parametrem **typu A**, můžeme jí volat s:
  - **instancemi** třídy **A**,
  - **instancemi** libovolné **podtřídy A** (kompatibilní podtyp).
- Pro zajištění kompatibilního typu musíme použít dědičnost - **vyžádaná dědičnost a polymorfismus** se vyskytuje také **pouze v rámci dědičnosti**.
- **výčtem položek** tj. nezáleží, jestli jde o třídu Car nebo Person. Typ se zjistí prozkoumáním položek instance - **test implementace** potřebného podtypu:
  - **staticky** během překladu (často se k tomu využívá rozhraní/interface).
  - **dynamicky** za běhu

Umožňuje **polymorfismus nezávislý na třídní dědičnosti**, např. funkce má jako parameter typ JmenoStari, která obsahuje dva atributy "jméno" a "stáří", pokud Car i Person mají (i mimo jiné) tyto atributy, lze je použít jako parametr pro volání této funkce (obdobně s metodami a kombinací metod a atributů).

- **kombinace** např. vícenásobnou dědičností nebo rozhraním

```

class A is
    int d1;
    float d2;
    float m(int);
endOfClass
subclass C of A is
    int d3;
    float m(int);
    int g(float);
endOfClass
  
```

```

  class B is
      float d2;
      int d1;
      int h(float);
      float m(int);
  endOfClass
  
```

```

static p(o : InstanceOfType(A));
static q(o : InstanceOfType(C));
α = new A;
β = new B;
γ = new C;
  
```

### Vyžádaná dědičnost

call p(α)	OK
<b>call p(β)</b>	<b>ERROR</b>
call p(γ)	OK

### Skutečné podtypy

call p(α)	OK
call p(β)	OK
call p(γ)	OK

<b>call q(β)</b>	<b>ERROR</b>	<b>call q(β)</b>	<b>ERROR?</b>
call q(γ)	OK	call q(γ)	OK

**Implementace skutečného podtypu je náročná** (změna pořadí položek atd. kód metody je ale statický a očekává položky na stejných pozicích, ...) je nutné skutečný podtyp řešit **až na běhu**. Skutečný podtyp se tak typicky vyskytuje u **interpretovaných** (nebo alespoň částečně interpretovaných) a **dynamicky typovaných** jazyků. Objekt je ve skutečnosti **slovník** s názvem položky (klíč), hodnotou, typem, aj. a interpret hledá, jestli ve slovníku existují dané položky. U **překládaných a staticky typovaných** jazyků lze podtyp nahradit **rozhraním**.

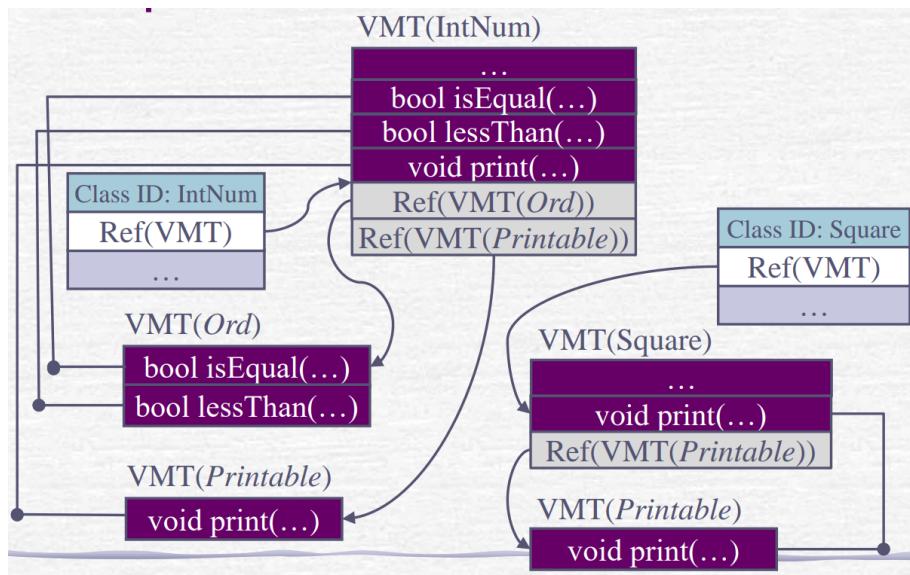
## Rozhraní

Jedná se o **náhradu vícenásobné třídní dědičnosti a skutečného podtypu**. Rozhraní je schéma, které deklaruje sadu metod. **Nelze** vytvářet **instance rozhraní**, ale rozhraní se **přiřazuje třídám**, které jej implementují (třída, které je přiřazeno rozhraní, musí implementovat všechny metody, které rozhraní deklaruje). Deklarace atributů v rozhraní je pouze teoretická a není většinou podporována (v C# lze v rozhraní specifikovat tzv. properties s { get; set; }, to ale **nejsou atributy** jedná se vlastně o funkci, která pomocí get a set získává a nastavuje daný atribut - field v C#). **Rozhraní lze kombinovat** s jednoducho **dědičností** a třída může **současně implementovat více rozhraní**. Samotná rozhraní mohou také **dědit** (částečné uspořádání na rozhraních) a **případně i vícenásobně**.

Koncept **rozhraní** umožňuje **polymorfismus** i mimo třídní dědičnost. Libovolná třída může implementovat funkce daného rozhraní libovolně. Pokud poté přistupujeme k objektu pomocí rozhraní, je nám skryto, o jaký objekt se jedná, takže volaná metoda může mít libovolnou funkcionalitu (v rámci zachování logiky programu na to musíme dávat pozor, i.e. funkci implementujeme tak aby měla očekávanou funkcionalitu např. podle jména).

## Implementace rozhraní

Objekt (instance třídy) odkazuje na více VMT, pro **každé rozhraní jiná VMT**, respektive z VMT objektu jsou odkazy na VMT realizující rozhraní (odkazující na metody, které deklaruje rozhraní).



## Beztrídní objektově orientované jazyky

U beztrídních jazyků je tvorba objektů závislá na **klonování prototypů**. Prototyp je také objekt. Např. v JS je prototypem každého objektu **Object** a ten už nemá další prototyp. Pomocí prototypů se realizuje v JS **dědičnost**, každý objekt má jeden **přímý** prototyp, ten pak může mít další přímý prototyp atd. (tzv. **prototype chain**). Obecně ale může mít objekt více přímých předků (prototypů). **Specializace** objektů je prováděna **dynamicky** za běhu (případně by mohla být prováděna staticky před překladem) přidáním nové položky (**atributu** nebo **metody**). Klonováním prototypu se **nekopírují metody** klonovaného objektu. **Atributy** jsou obecně **zkopírovány** a je jim při klonování nastavena buď zadaná hodnota, nebo implicitní hodnota (hodnoty v prototypu). Metody zůstávají pouze u prototypu. Invokace těchto metod u naklonovaného objektu se řeší pomocí **delegace** (výpočetní systém se nejdřív snaží metodu spustit na daném objektu, pokud ji nemůže najít, deleguje ji na prototyp).

**Polymorfismus** je zajištěn tak, že u naklonovaného objektu **definujeme stejně pojmenovanou metodu**. Např. JS je **dynamicky typovaný netypovaný** jazyk a implementuje **skutečný podtyp**, což umožňuje i polymorfismus i dědičnost. Pokud objekt neimplementuje danou metodu (nerozumí zprávě) a ani žádný z jeho prototypů, dochází k chybě. Příklad klonování z JS:

```
const personPrototype = {
  greet() {
    console.log('hello!');
  }
}

const carl = Object.create(personPrototype);
carl.greet(); // hello!
```

`personPrototype` je zde objekt (složené závorky je mimo jiné způsoby v JS syntax pro tvorbu objektu), jeho prototypem je tedy v tomto případě `Object` (vznikl jeho klonováním, i když to není na první pohled zřejmé). U tvorby objektu `carl` je již ale zřejmé, že se jedná o klonování objektu `personPrototype`. Objekt `carl` tak má za přímý prototyp `personPrototype` a ten má přímý prototyp `Object`. Voláním metody `greet` nad objektem `carl` vede na delegaci a volá se metoda `greet` objektu `personPrototype`.

## Delegace

Delegace je obdoba zasílání zprávy. Zpráva je v tomto případě zasílána rodiči objektu (**příjemce** je rodič objektu) a kromě názvu metody (**selektoru**) a **parametrů** obsahuje i **objekt**, kterému byla zpráva prvně adresována (**původního příjemce**).

## Sloty (v jazyce SELF)

Slot obsahuje **název položky** a **odkaz** na:

- **datový objekt**,
- **objekt s metodou**,
- **rodičovský objekt**.

## Rsys (traits v jazyce SELF)

Rys je objekt, který obsahuje pouze sdílené **metody a rodičovské sloty** (neobsahuje jiné atributy/sloty), které umožňují delegaci. Prototyp pak obsahuje datové **sloty pro atributy** a jejich **implicitní hodnoty** (použité při klonování) a **rysy**, na které jsou **delegovány** zprávy (delegace) při volání "instančních" metod. Mezi **rysem a třídou** lze nalézt v tomto směru **analogie**, třída i rys nese instanční metody, které jsou z konkrétních instancí (objektů) odkazovány.

# OO přístup k tvorbě SW

Při objektově orientované tvorbě SW využíváme skutečnosti, že **objekty reálného světa lze modelovat** (do jisté úrovně detailu) **objekty v SW**. A snažíme se využít základních vlastností OOJ, a to **abstrakci, zapouzdření, dědičnost a polymorfismus**. Např. v reálném světě potřebujeme vytvořit fakturu, což znamená, že dle nějakých konvencí ji musíme sepsat (např. vyplnit informace o dodavateli a odběrateli, položkách faktury, celkové ceně atd.). Fakturu poté můžeme odeslat buď poštou, nebo ji naskenovat a odeslat emailem.

Pokud by k nám přišel někdo, že už nechce psát faktury ručně a chce abychom mu pro to napsali SW můžeme postupovat následovně (posloupnost kroků nemusí odpovídat realitě):

1. Setkáme se s touto osobou (zákazník) a na základě jeho požadavků můžeme pomocí **UML** (standardizovaný grafický jazyk podporující objektovou

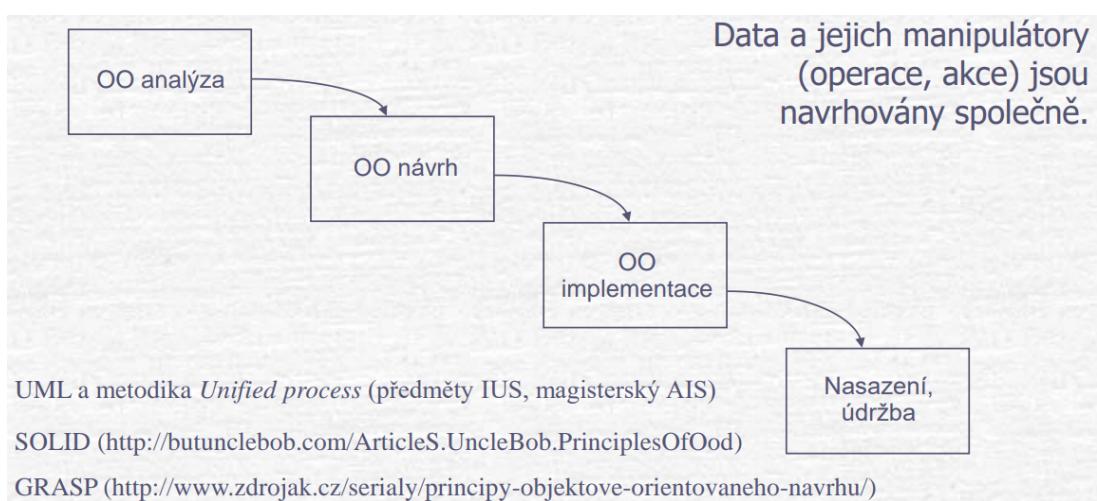
orientaci) vytvořit objektově orientovaný návrh např. pomocí **diagramů tříd** a **diagramů objektů**.

2. Následně zjistíme, jestli již **neexistuje** nějaká objektová orientovaná implementace, která řeší tvorbu faktur na základě **vytvořených diagramů**. Pokud najdeme vhodné již implementované **třídy** (objekty u beztřídního jazyka), které osabují část atributů a metod. Můžeme použít tyto - využijeme jednu z výhod OOP, a to **znovupoužitelnost**.
3. Následně z vyhledaných objektů **zdědíme** (použijeme **dědičnost**) a do zděděného objektu **doplníme** potřebné atributy a metody (provedeme specializaci).
4. Metody, které jsou již v původním objektu implementovány, ale nevyhovují našemu zadání přepíšeme (**override**) a využijeme vlastnosti OOJ **polymorfismu**.
5. Pro ukládání faktur (tj. objektů faktura) můžeme využít buď **objektové databáze**, ale mnohem častější je využití SQL databází a vrstvy mezi databázi a OOJ, která převádí relační data na objekty - **ORM (Objektově relační mapování)**

## Shrnutí

Při OO přístupu k vývoji SW se snažíme využít:

- **analogie** objektů reálného světa (mají nějaké znaky a funkce) se SW objekty (mají atributy a metody), Snažíme se zachovat vztah dat a jejich manipulátorů, respektive akcí nad nimi.
- při návrhu využíváme nástroje podporující objektový přístup, jako je jazyk **UML - formální reprezentace OOP**,
- využíváme **dekompozice** složitých objektů na jednodušší,
- při implementaci se snažíme využít **výhod OOP**, zejména **znovupoužitelnost**,
- při implementaci využíváme základní **koncepty OOP**, a to **abstrakci, zapouzdření, dědičnost a polymorfismus**.
- snažíme se využít nástroje, které umožňují převádět jiný SW na objektové paradigma, např. **ORM**.
- snažíme se využít **návrhových vzorů** (poskytují radu/návod/vzor, jak vhodně řešit často vyskytujících se problémy a umožňují lepší znovupoužitelnost, např. singleton, factory, adapter, MVC, MVVM, MVP, Facade)

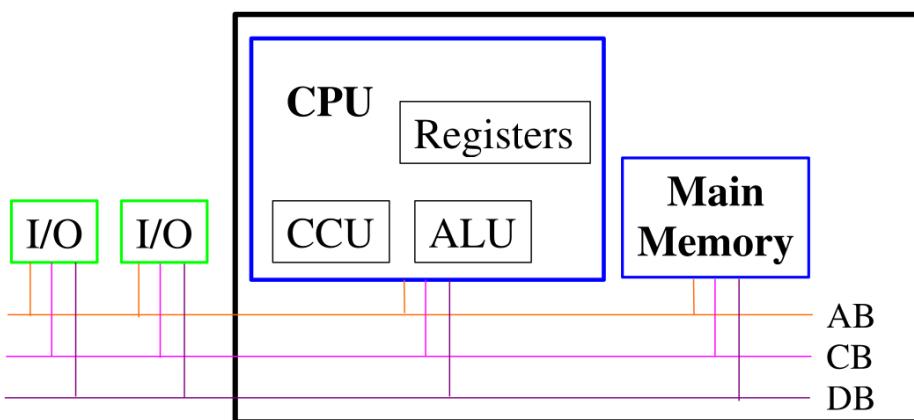




# 41. Programování v jazyku symbolických instrukcí (činnost počítače, strojový jazyk, symbolický jazyk, assembler)

Jazyk symbolických instrukcí je nízkoúrovňový programovací jazyk, který obsahuje **symbolické reprezentace strojových instrukcí**. Je definován výrobcem daného hardware, je založen na zkratkách jmen instrukcí (např. compare -> CMP)

## Činnost počítače



CPU	Central Processing Unit (procesor)
ALU	Arithmetic and Logic Unit
CCU	Central Control Unit (řadič)
I/O	Input/Output Unit
AB, CB, DB	Address Bus, Control Bus, Data Bus

Hlavními částmi počítače jsou **CPU** (skládající se z aritmeticko logické jednotky (**ALU**), **registrů**, **řadiče** a **zdroj hodin**), **paměť** a **vstupně-výstupní zařízení**, všechny tyto komponenty jsou propojeny **sběrnicemi**. Významnými registry pro základní činnost počítače jsou **akumulátor** (střadač) pro výpočty, **instruction register** (IR) obsahující současnou instrukci a **instruction pointer** register (IP pro 16-bit a EIP pro 32-bit) ukazující na současnou instrukci v paměti. Paměť tak současně obsahuje:

- **data**, se kterými se pracuje,
- **instrukce**, které tuto práci (výpočet) řídí.

## Princip činnosti PC

Princip činnosti počítače je následující (opakuje se v cyklu):

1. **fetch**: do **IR** se uloží obsah paměti, na který ukazuje **IP**,
2. aktualizuje se **IP** (na následující instrukci - přičte se počet bytů odpovídající provedené instrukci),
3. **decode**: dekóduje se IR, určí se operace a operandy,
4. **execute**: provede se daná instrukce (skoky ještě aktualizují **IP**).

V případě přerušení je obsah některých registrů (instruction pointer, flags, ...) nahrán na zásobník a po obsluze přerušení je stav těchto registrů ze zásobníků obnoven.

## Strojový kód

Kód specifický pro daný počítač, **binární jedničky a nuly**, není téměř vůbec přenositelný, v dnešní době nemožné v něm programovat. Výhodou je vysoká efektivita, nevýhodou **nepřenositelnost** a **složitost psaní**. Právě z těchto důvodů se zavádí symbolický jazyk, který je čitelnější, ovšem stále má víceméně 1:1 mapování na strojový kód (tedy výhodu efektivity) a poté dále vyšší programovací jazyky, které díky překladačům zajišťují přenositelnost.

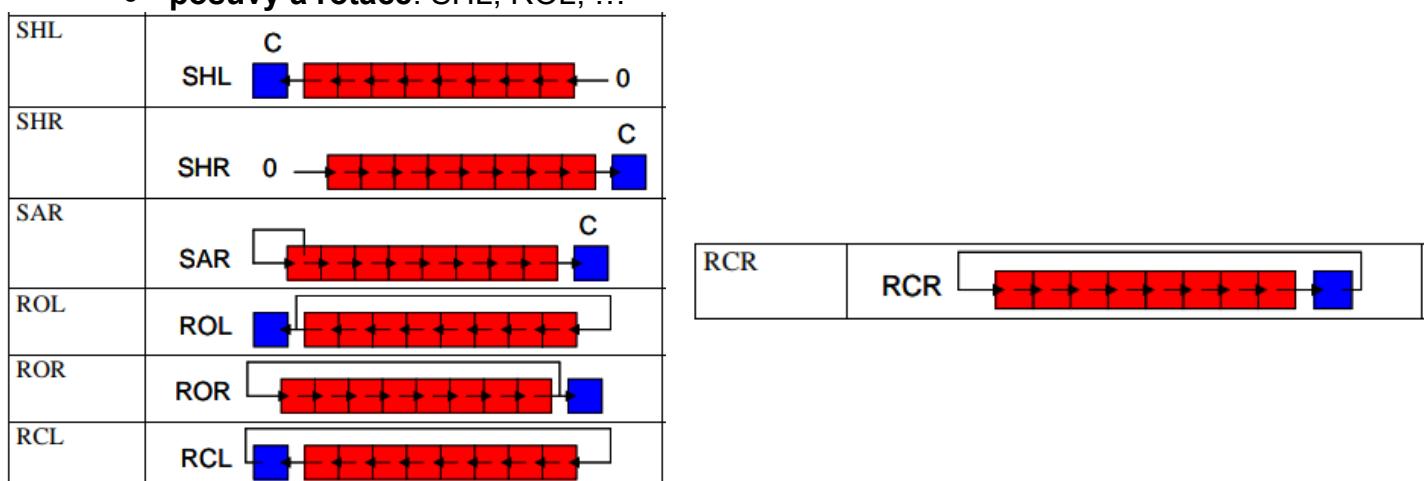
## Symbolický jazyk

Také jazyk symbolických instrukcí.

## Instrukce

Instrukce jsou **příkazy pro procesor**, obsahují jednoznačný  **operační kód** a operandy, příp. adresy operandů. Některé instrukce mohou mít ve své specifikaci napevno nastavené, s kterými registry pracují. Základními typy instrukcí jsou:

- **přenosové**: MOV, PUSH, POP,
- **aritmetické**: ADD, SUB, INC,
  - MUL - pozor,  $32b \times 32b = 64b$ , využívá se EDX:EAX pro výsledek; IMUL - znaménkové násobení,
  - DIV - umí zase dělit 64 bit číslo z EDX:EAX v případě 32-bit musíme rozšířit znaménko do EDX, jinak to nebude fungovat, výsledek je v EAX, v EDX je pak modulo - zbytek, IDIV - znaménkové dělení.
- **posuvy a rotace**: SHL, ROL, ...



- **logické**: AND, OR, NOT, TEST, XOR, ...
- **skokové**: JMP, JA, LOOP - skáče podle hodnoty counteru ECX - rozdílné od nuly znamená skoč, jinak ne. Skoky mohou prováděny **podmíněně** dle příznaků (např. přetečení).
  - skoky pro **znaménkovou** aritmetiku

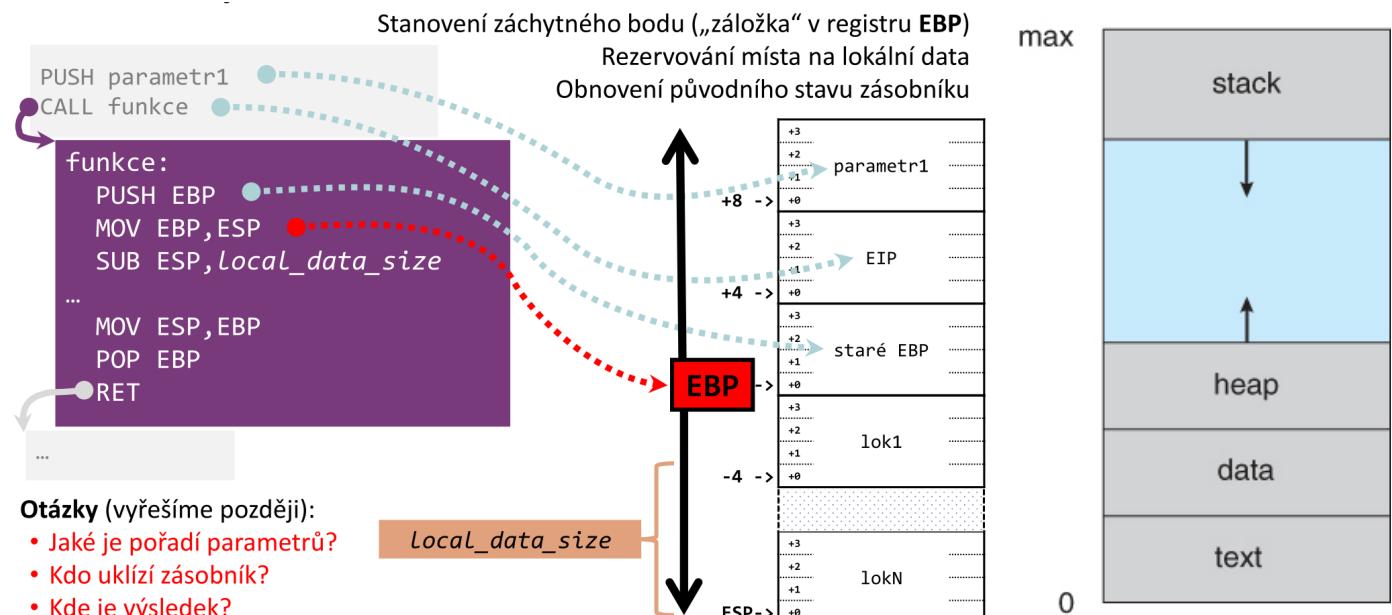
Mnemonic	Description	Condition Tested
JG, JNLE	<b>Jump if Greater, Jump if Not Less or Equal</b>	<b>ZF = 0 and SF = OF</b>
JGE, JNL	<b>Jump if Greater or Equal, Jump if Not Less</b>	<b>SF = OF</b>
JL, JNGE	<b>Jump if Less, Jump if Not Greater or Equal</b>	<b>SF ≠ OF</b>
JLE, JNG	<b>Jump if Less or Equal, Jump if Not Greater</b>	<b>ZF = 1 or SF ≠ OF</b>

- skoky pro **neznaménkovou** aritmetiku

Mnemonic	Description	Condition Tested
JA, JNBE	<b>Jump if Above, Jump if Not Below or Equal</b>	<b>ZF = 0 and CF = 0</b>
JAE, JNB	<b>Jump if Above or Equal, Jump if Not Below</b>	<b>CF = 0</b>
JB, JNAE	<b>Jump if Below, Jump if Not Above or Equal</b>	<b>CF = 1</b>
JBE, JNA	<b>Jump if Below or Equal, Jump if Not Above</b>	<b>ZF = 1 or CF = 1</b>

- **řetězové**: umožňují jednodušší a rychlejší práci s poli (automaticky **inkrementují/dekrementují** hodnotu ukazatelů v ESI a EDI). Lze je kombinovat s prefixy REP\* pro opakování dle ECX.
  - MOVS(B/W/D) - přesun hodnot odkazovaných z ESI do EDI (B po bytech, W po slovech a D po 4 bytech),
  - CMPS(B/W/D) - porovnání hodnot odkazovaných ESI a EDI,
  - SCAS(B/W/D) - porovnání hodnoty EAX s hodnotou odkazovanou EDI, lze využít pro hledání prvku v poli,
  - LODS(B/W/D) - načtení z adresy odkazované ESI do akumulátoru,
  - STOS(B/W/D) - uložení akumulátoru (AL/AX/EAX) na adresu odkazovanou z EDI.
- **řídící**: CALL, RET (umožňuje odebrat **n** slabik ze zásobníku)
  - CALL je skok, co uloží na zásobník **návratovou adresu** (RET si ji pak vezme a vrací řízení za instrukci CALL)
  - Parametry se předávají v **globálních proměnných, registrech** nebo na **zásobníku** (dle konvence volání), či kombinací.
  - Zásobníkový rámec je typická konstrukce ve volání funkcí, tvoří se "záložka" pro jednoduché vyčištění např. lokálních proměnných ze zásobníku (vyčištění znamená změnu hodnoty ukazatele na vrchol zásobníku v registru **ESP**, uložená data tam zůstávají, dokud nejsou přepsána dalším použitím zásobníku - nikdy tato data ale už nelze použít z důvodu možného vzniku přerušení). Register **EBP** pak funguje

jako reference, abychom věděli, jak moc věcí jsme už na zásobník zapsali. Při vstupu do funkce **uložíme starou referenci** (cizí referenci toho, kdo funkci volal) z **EBP** na zásobník a vytvoříme si novou referenci - aktuální vrchol zásobníku **ESP**. Místo, které budeme na proměnné potřebovat pak vyhradíme odečtením počtu bytů od **ESP**, čímž se posune vrchol zásobníku a vytvoří se požadované místo (odečítá se, protože zásobník je vzhůru nohama a jeho dno je na maximální adrese). Před výstupem z funkce pak musíme obnovit původní obsah registrů **ESP** a **EBP**.



Obecně mají instrukce **proměnnou délku** (ta je např. dána způsobem adresování operandů nebo jejich počtem). Formát instrukce na procesoru intel Pentium je takový (délka může být od **1 B** do **16 B**):

Předpony	Operační kód	ModR/M	SIB	Posunutí	Operand
0-4 slabiky	1-2 slabiky	0-1 slabika	0-1 slabika	0/1/2/4 slab.	0/1/2/4 slab.

## Konvence volání

- pascal** - parametry **uklízí volaný** (tj. uvnitř funkce), parametry se dávají na zásobník **zleva doprava**, tzn. na vrcholu zásobníku je při předání řízení do funkce hodnota **posledního parametru** (využito v Pascalu),
- cdecl** - parametry **uklízí volající** (tj. až po návratu za funkci), předávají se **zprava doleva**, tzn. na vrcholu zásobníku je po předání řízení do funkce **první parametr**. Tento způsob umožňuje funkce s proměnným počtem parametrů - první parametr specifikuje jejich počet (jazyk C),
- stdcall** - parametry **uklízí volaný**, předávají se **zprava doleva** (winapi)

## Registry

Máme mnoho registrů pro různé účely různých velikostí. Základní datové registry (například střadač), mají více možností, jak se odkazovat na jejich část, např. EAX = celý 32b registr, AX = pouze spodních 16b, AH = vrchní 8b AX, AL = spodních 8b AX

EAX	(Accumulator)	střádač
EBX	(Base)	ukazatel na data v datovém segmentu
ECX	(Counter)	čítač řetězových a smyčkových operací
EDX	(Data)	rozšíření střádače, ukazatel na I/O zařízení
ESI	(Source Index)	ukazatel na zdrojová data řetězových operací, index
EDI	(Destination Index)	ukazatel na cílová data řetězových operací, index
EBP	(Base Pointer)	ukazatel na data v zásobníkovém segmentu
ESP	(Stack Pointer)	ukazatel zásobníku

Základními registry jsou: EAX, EBX, ECX, EDX, pro práci se zásobníkem se využívá ESP (stack pointer), EBP (base pointer), řetězové instrukce využívají ESI (zdrojová data) a EDI (cílová data). Speciálním je registr **EFLAGS** obsahující příznaky, které se nastavují dle operací:

- **OF** (overflow flag) = 1 při přetečení znaménkové operace, jinak 0,
- **CF** (carry flag) = 1 při přetečení bezznaménkové operace, jinak 0,
- **SF** (sign flag) = 1 při záporném výsledku
- **ZF** (zero flag) = 1 když je výsledek 0
- **DF** (direction flag) = určuje směr řetězových instrukcí - směr průchodu polem (nastavujeme instrukcemi **CLD** - clear direction flag - nastaví na 0, což znamená zvyšování ukazatele na data po každé iteraci, **STD** - set direction flag - nastaví na 1, což způsobí snižování ukazatele na data po každé iteraci)

**DF = clear (0): increment SI a DI**

**DF = set (1): decrement SI a DI**

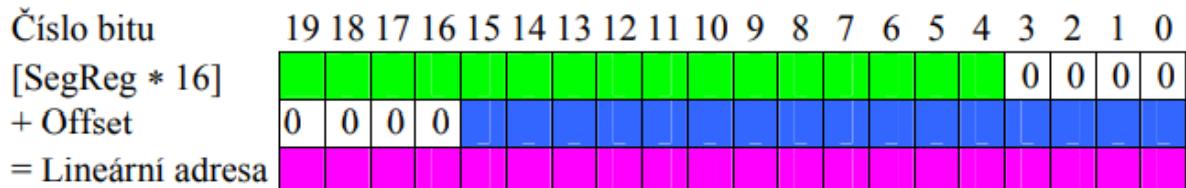
## Skoky

- **Short – 2-bytová** instrukce, která dovoluje skočit na místo v rozsahu +127 and -128 bytů od místa, které následuje za příkazem JMP.
- **Near – 3-bytová** instrukce, která dovoluje skočit v rozsahu +/- 32KB od následující instrukce v rámci běžného kódového segmentu.
- **Far – 5-bytová** instrukce, která dovoluje skočit na jakékoli místo v celém adresovém prostoru.

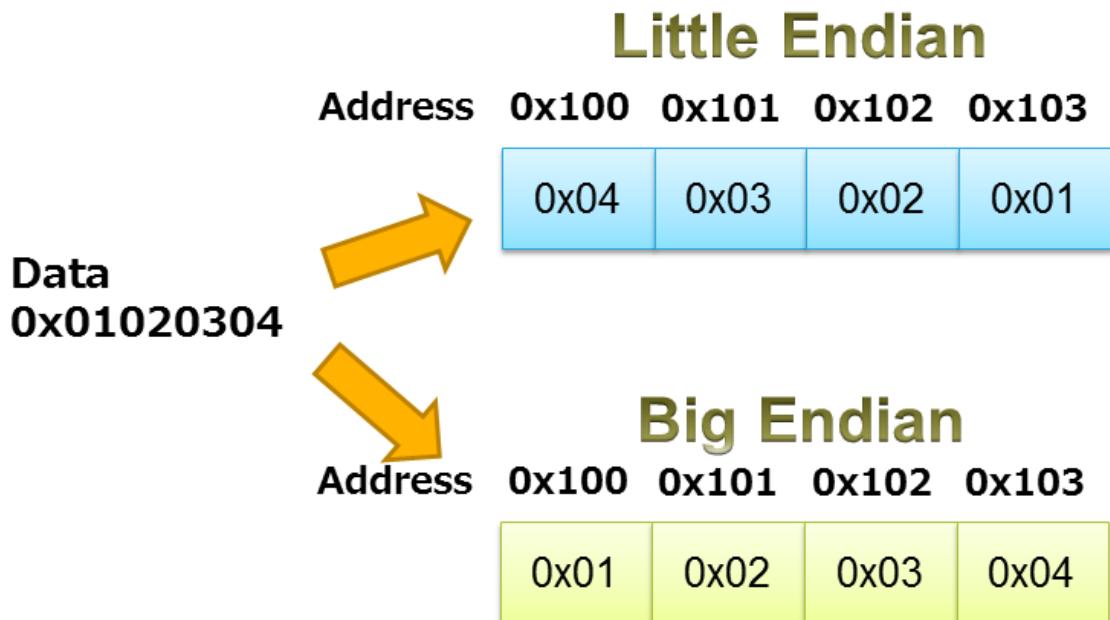
## Paměť

V základním režimu (16b) má fyzická adresa 20b, tedy pro adresaci pomocí registrů chybí 4b. K řešení tohoto problému se vytváří **segmenty** (úseky paměti o velikosti 64 KiB, které lze adresovat 16 bitů), každý segment začíná na adrese, která je násobkem 16 (je **posunutá** o 4 bity doleva). Fyzická adresa je pak kombinací

**segment:offset** (RegReg:Offset), kdy **fyzická adresa** =  $16 * \text{segment} + \text{offset}$ .



## Little a Big Endian



## FPU

Dosud tato otázka řešila pouze operace v ALU, tj. integer a unsigned aritmetika, pro operace s floating point (IEEE754 viz jiné otázky) se používají specializované jednotky (koprocesory), například **FPU** (floating point unit).

Registry FPU pracují v **zásobníkovém režimu**, vršek zásobníku je ST0. Instrukce jsou prefixovány písmenem F, například **FADD**, **FABS**, **FSIN** a mají mnoho možností operandů (implicitní operandy, explicitní operandy, popnutí zásobníku registrů atd.). Např. **FADD st1** provede  $\text{st0} = \text{st0} + \text{st1}$ .

## Assembler

Assembler je překladač jazyka **symbolických instrukcí do strojového kódu**, příkladem je **NASM**. Aby mohl být proveden překlad, potřebuje nejen samotné instrukce, ale i dodatečné informace, typicky v podobě **pseudoinstrukcí** nebo **direktiv** (direktiva je příkazem pro překladač, nikoliv instrukcí pro procesor a nejsou překládány do strojového kódu). Jedná se především o:

- Definování konstant.
- Definování místa v paměti pro uložení dat (RESB, RESW, RESQ).
- Vytváření segmentů paměti.
- Vkládání jiných souborů (INCLUDE).

Pseudoinstrukce a direktivy tak popisují strukturu paměti, například kde **začíná** a **končí program** (SECTION .text), **definici dat** (Inicializovaná - DB, DW, ... a ne inicializovaná - RESB, RESW, ...), **definici konstant**, **definice jmen** (GLOBAL, EXTERN), příp. také mohou být podporována makra, která překladač textově expanduje (jako v C) nebo nějaký další pre-processing.

Překladač dělá dvouprůchodový překlad, nejprve sestavuje **tabulku symbolů** (ukládá adresy návští a jmen), která následně při druhém průchodu využije (symbol musí být při druhém průchodu **jednoznačně** definován). Dvojí průchod je nutný například kvůli **dopředným** skokům.

## Makra

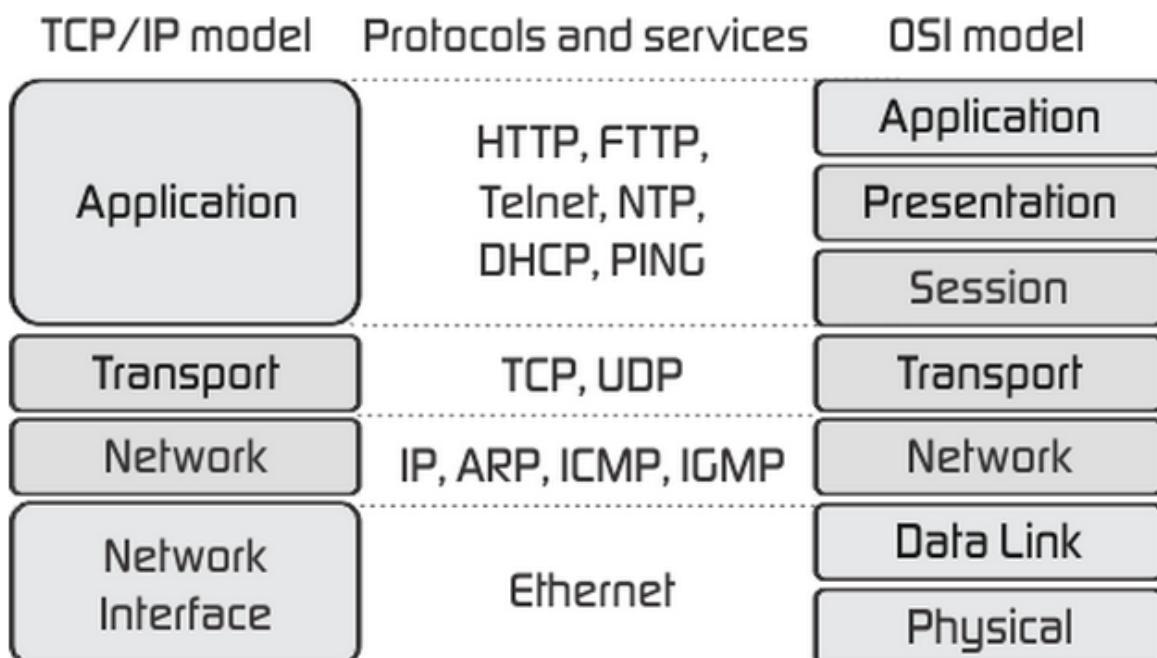
Makro je mechanismus, kterým můžeme nadefinovat posloupnost příkazů, vykonávající určitou činnost a tuto posloupnost příkazů **později vložit na různá místa programu**. Překladač tedy kód obsažený v makru kopíruje na všechna místa jeho použití. Nachází se tedy ve výsledném strojovém kódě opeakováně stejné posloupnosti jedniček a nul. Na rozdíl od funkcí, které jsou definované pouze na jednom místě a skáče se tam pomocí CALL a RET.

- **výhody**: rychlejší než funkce (nemusí se nikam skákat a vytvářet rámc),
- **nevýhody**: větší program (více kódu, což může mít špatný vliv na výkon, viz stránkování).

## 42. Služby aplikační vrstvy (web, e-mail, DNS, IP telefonie, správa SNMP, Netflow).

V modelu TCP/IP i v modelu ISO/OSI je aplikační vrstva poslední. Referenční model ISO/OSI předpokládá, že jednotlivé aplikace budou mít některé společné rysy, které se vyplatí realizovat samostatně a implementovat jen jednou. Síťový model vznikal více z praktických zkušeností. Předpokládá, že jednotlivé aplikace nebudou mít totík společného, aby se tyto části vyplatilo osamostatnit. Na rozdíl od referenčního modelu ISO/OSI očekává, že každá aplikace si sama zajistí to, co potřebuje a co jí nenabízí nižší vrstvy.

Na rozdíl od nižších vrstev (transportní, síťová a vrstva síťového rozhraní) není komunikace na aplikační vrstvě zajištěna OS (sockets) či HW počítače. Každá aplikace si musí komunikaci na této vrstvě zajišťovat sama. Proto, aby se nelišily způsoby komunikace aplikace od aplikace, používají se na aplikační vrstvě (stejně jako na jiných) standardizované protokoly. Protokol, který aplikace pro komunikaci používá, a určuje její změření.



Na základě požadavků aplikace může být aplikační protokol implementován nad UDP (rychlejší, ztrátový), nebo nad TCP (pomalejší, bezztrátový), příklady protokolů:

- **UDP:** **DNS** (překlad doménových jmen na IP), **TFTP** (přenos souborů), **SNMP** (správa sítě), **Netflow** (správa sítě), **NTP** (synchronizace času), **SIP** (signalizace VoIP), **RTP** (přenos zvuku a obrazu), **RTCP** (řídící informace pro RTP a QoS),

- **TCP:** **HTTP** (webové stránky), **SMTP** (mailtová komunikace), **LDAP** (adresářové služby, může být i přes UDP), **FTP** (přenos souborů), **POP3** (stahování el. pošty), **IMAP** (stahování a čtení el. počty).

Application	Data Loss	Throughput	Time-Sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Internet telephony/ Video conferencing	Loss-tolerant	Audio: few kbps–1Mbps Video: 10 kbps–5 Mbps	Yes: 100s of msec
Streaming stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps–10 kbps	Yes: 100s of msec
Instant messaging	No loss	Elastic	Yes and no

## Adresa

**Způsob identifikace adresáta pomocí jednoznačné informace.**

## Web (HTTP)

**World Wide Web** (také jen web) je tvořen aplikacemi, které běží na protokolu **http** (Hypertext Transfer Protocol), respektive jeho šifrované verzi **https** (Hypertext Transfer Protocol Secure). WWW je systém pro **prohlížení, ukládání a odkazování dokumentů** nacházejících se **na internetu**. Dokumenty (webové stránky) jsou uloženy na **webových serverech** a přistupujeme k nim pomocí **webového prohlížeče**. Navzájem jsou propojeny pomocí **hypertextových odkazů** zapisovaných ve formě **URL**.

## URI, URL a URN

URL a URN jsou podmnožinou URI. URI je textový řetězec s definovanou strukturou, který slouží k **přesné specifikaci zdroje** informací.

- **Uniform Resource Name (URN):** jasně identifikuje zdroj, ale neřeší jeho dostupnost. Schema URN je  $\text{URN} ::= \text{"urn:": } \langle\text{NID}\rangle \text{ ":" } \langle\text{NSS}\rangle$ , např. `urn:isbn:0451450523`
- **Uniform Resource Locator (URL):** určuje, kde je identifikovaný zdroj dostupný a mechanismus pro jeho získávání. **Adresa** URL definuje, **jak lze zdroj získat**. Nejobecněji má URL tvar  $\langle\text{scheme}\rangle : \langle\text{scheme-specific-part}\rangle$ , konkrétně pak

protokol://server.doména2.doména1:port/cesta/název?dotaz#kotva.

Pro adresaci (jako **adresu**) na webu **používáme URL**, i když v HTTP RFC je to zobecněno na URI (ale pomocí URN zdroj na internetu nenajdeme).

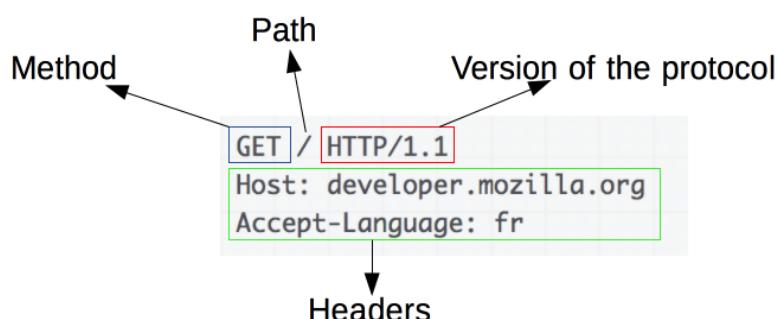
## HTTP

HTTP je protokol typu **klient-server**, což znamená, že požadavky **iniciuje příjemce** (klient), obvykle webový prohlížeč. Klienti a servery komunikují výměnou jednotlivých zpráv (na rozdíl od proudu dat). Klient zasílá požadavky/dotazy (**request**) a server na ně odpovídá odpověďí (**response**). Jedná se o **bezstavový** protokol, tj. že není zachována návaznost jednotlivých dotazů a odpovědí (řeší se ale pomocí Cookies).

### Schéma HTTP dotazu

```
<metoda> <cesta> <verze protokolu>CRLF  
<hlavička 1>CRLF  
<hlavička 2>CRLF  
<hlavička...>CRLF  
<hlavička n>CRLF  
CRLF  
<tělo dotazu>
```

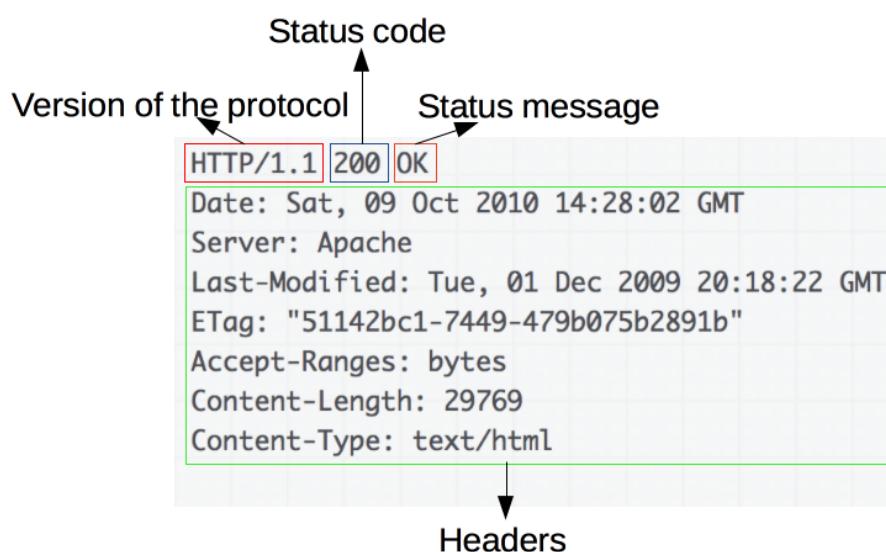
- **metoda** je:
  - **GET** - získání zdroje/dat (nemělo by se pomocí GET nikdy modifikovat),
  - **POST** - odeslání dat na server, mělo by znamenat vytvoření zdroje,
  - **PATCH** - částečná úprava zdroje,
  - **PUT** - změna aktuálního zdroje s daty v těle dotazu,
  - **DELETE** - smazání zdroje,
  - aj. (**HEAD**, **CONNECT**, **TRACE**)
- **cesta** specifikuje umístění zdroje na serveru,
- **verze protokolu** je např. **HTTP/1.1**,
- **hlavičky** jsou ve tvaru název: hodnota, nejdůležitější hlavičkou je hlavička Host, která specifikuje adresu serveru.
- **tělo** dotazu je tvořeno daty, které odesíláme na server, může jít o JSON, HTML, XML, binární data, ... specifikováno pomocí hlavičky **Content-Type** a **MIME** (application/json, text/html, application/xml) typu data.



## Schéma HTTP odpovědi

```
<verze protokolu> <status code> <status message>CRLF  
<hlavička 1>CRLF  
<hlavička 2>CRLF  
<hlavička...>CRLF  
<hlavička n>CRLF  
CRLF  
<tělo dotazu>
```

- **verze protokolu** je např. **HTTP/1.1**,
- **status code** je jeden ze skupiny:
  - **100-199**: **informační odpověď**,
  - **200-299**: odpověď hlásící **úspěch**,
  - **300-399**: odpověď hlásící **přesměrování**,
  - **400-499**: odpověď hlásící **chybu na straně klienta**,
  - **500-599**: odpověď hlásící **chybu na straně serveru**.
- **status message** dále upřesňuje status code, např.:
  - 200 OK,
  - 201 created,
  - 202 updated,
  - 400 bad request,
  - 401 unauthorized,
  - 403 forbidden,
  - 404 not found.



## E-mail

Email je mechanismus pro zasílání elektronické pošty. Je tvořen:

- **obálkou** - tu vytváří poštovní **server** (MTA - Mail Transfer Agent) a obsahuje:
  - MAIL FROM:<odesilatel@domena.com>
  - RCPT TO:<přjemce@domena.com>

- **hlavičkou** - vytváří poštovní **klient** (MUA - Mail User Agent), má tvar <název>:<hodnota> a může obsahovat (mimo jiné):
  - From, To, Subject, Cc, Bcc, Date, Return-Path, Received, ...
- **tělem** - vytváří poštovní klient a obsahuje **7-bit ASCII** data (původně, hlavička a obálka jsou také 7-bit), dnes rozšíření **MIME**.

K identifikaci adresáta (**adresa**) se u emailu používá **emailová adresa**.

## Kódování

V případě emailu se jedná o **převod** binárních nebo jiných **8-bit znaků** na **7-bit ASCII** znaky. Rozhodně se **nejedná o šifrování**.

### Quoted-printable

Nahrazují se pouze 8-bit znaky a převádějí se na **3 znaky**, a to rovnítko a 2 **hexadecimální znaky** (reprezentují index v 8-bit poli znaků), např. =F9 nebo =B9. Rovnítko musí být také kódováno jako (=3D). Teoreticky může dojít k prodloužení textu až o **200%**. Prakticky se používá při kódování textu (jazyka, např. čj, zůstává částečně čitelné), kde může mít menší prodloužení než **Base64** (záleží také na národní abecedě), nevhodné pro binární data.

### Base64

Kódují se vždy všechny znaky, a to po trojicích. **Trojice 8-bit znaků** je převedena na **čtveřici 6-bit znaků** ( $3^8 == 4^6$ ). Šestibitové znaky jsou následně převáděny na 7-bitové indexací do pole:

**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/**. Chybějící znaky do čtveřice jsou doplnovány pomocí rovnítka. Text je vždy prodloužen o **33%**, kódování je vhodné pro binární data.

## Simple Mail Transfer Protocol (SMTP)

Aplikační protokol nad TCP pro **posílání pošty** na portu **25**. S protokolem SMTP pracují 3 programy:

- **MUA – Mail User Agent**: poštovní klient, který zpracovává zprávy u uživatele.
- **MTA – Mail Transfer Agent**: server, který se stará o doručování zprávy na cílový systém adresáta. Ignoruje hlavičku a tělo zprávy, pro doručování **používá pouze obálku** (analogie s poštou).
- **MDA – Mail Delivery Agent**, program pro lokální doručování, který umísťuje zprávy do uživatelských schránek, případně je může přímo automaticky zpracovávat (ukládat přílohy, odpovídat, spouštět různé aplikace pro zpracování apod.).

Pro vyhledání serveru adresáta, na který má být předána zpráva, vyhledává MTA záznam **MX v DNS**. Dotaz na DNS obsahuje **doménové jméno** za znakem '@', DNS vrací **IP** SMTP serveru adresáta.

## **SMTP příkazy**

HELO, HELP, MAIL FROM, RCPT TO, RSET, QUIT, ...

## **Post Office Protocol (POP3)**

Protokol sloužící pro stahování elektronické pošty přes TCP port 110. Protokol je mnohem **jednodušší** než IMAP a umožňuje především **stahovat** poštu ze serveru (a zobrazovat počet příchozích zpráv). Po stáhnutí je pošta ze serveru **implicitně smazána** (může být nastaveno jinak, třeba po určité době bude pošta smazána). Implicitní mazání neumožňuje mít jednu schránku na více zařízeních a i pokud je nastaveno jinak, není protokol vhodný pro více schránek (např. přesun zpráv, označení zpráv atd. se děje pouze lokálně, na serveru se nic nemění). Protokol je vhodný, pokud na serveru máme **málo místa**.

## **Internet Mail Access Protocol (IMAP)**

Protokol sloužící pro **prohlížení** a stahování elektronické pošty přes TCP port 143. Pomocí protokolu se navazuje s emailovým serverem trvalé spojení. Zprávy není nutné stahovat celé, stačí pouze základní informace (hlavičky zpráv). Zprávu je ze serveru stažena celá, až když ji chce uživatel na klientovi číst (poté může být zase smazána). Na **serveru zprávy zůstávají neustále**, dokud je uživatel explicitně nezmaže. IMAP **reflektuje změny** provedeny uživatelem v klientovi **na server** (např. označení zprávy za přečtenou, přesun zprávy do jiné složky, přidání/odebrání štítku atd.). Je proto **vhodný** při přístupu z **více klientů** (zařízení), změny provedené na jednom jsou synchronizovány na druhý.

## **Pretty Good Privacy (PGP)**

Protokol SMTP a IMAP lze přenášet zabezpečeně přes internet (SSL, TLS, HTTPS). Tyto protokoly ale **zabezpečují obsah zprávy**, ta zůstává na emailovém serveru v čitelné formě. Zabezpečit tělo zprávy lze např. pomocí **PGP**. PGP zajišťuje:

- **integritu dat**,
- **autentizaci odesílatele** (autentizace a integrita dat zajišťují společně **neodmítnutelnost**),
- **šifrování**.

Postup zabezpečení:

1. Odesílatel vypočte **heš** zprávy a svým privátním klíčem ji **podepíše**.
2. **Komprimuje** obsah zprávy.
3. **Zašifruje** obsah zprávy vygenerovaným symetrickým klíčem.
4. Symetrický klíč **zašifruje veřejným** klíčem příjemce (dešifrovat může pouze příjemce svým privátním) a připojí jej k zašifrované zprávě.
5. Převede obsah do **Base64** (v mailu mohou být pouze 7-bit znaky, viz výše).

## Secure/Multipurpose Internet Mail Extensions (S/MIME)

Další možnost zabezpečení obsahu emailu založená na **asymetrické kryptografii** a **digitálních podpisech**.

## Domain Name System (DNS)

DNS je **globální decentralizovaný adresář** názvů počítačů a dalších identifikátorů síťových zařízení a služeb. Hlavní funkcí DNS je **překlad doménových jmen na IP adresy**. DNS rozděluje globální prostor doménových adresy (jmén) **hierarchicky**, což umožňuje delegaci. Záznamy jsou uloženy na třech typech serverů, a to **primární, sekundární a záložní**. Princip vyhledávání v DNS se nazývá **rezoluce**. Protokol DNS pro rezoluci používá primárně UDP, port 53. TCP lze použít při přenosu většího množství dat, např. při přenosu **DNS zón**.

## Prostor doménových jmen

Hierarchické uspořádání DNS záznamů do stromové struktury.

- Kořenovou doménu “.” má na starost organizace **ICANN**.
  - Top Level Domain - TLD (domény 1. řádu): zprávu těchto domén deleguje ICANN na další organizace, v ČR je to CZ.NIC. Jde o domény (.cz, .com, .eu, .org, .net, ...).
  - Domény 2. a nižších řádů: tyto domény spravují konkrétní organizace. Např. u CZ.NIC si může každý zakoupit doménu \*.cz (\* je název 2. řádu) a poté ji spravovat a případně prodávat domény \*.\*.cz. Jde o domény (seznam.cz, vut.cz, qooqle.com, ...)

## Reverzní mapování

Umožňuje na základě IP adresy vyhledat doménové jméno. Reverzní mapování zajišťují 2 podstromy **in-addr.arpa**. pro IPv4 a **ipv6.arpa**. pro IPv6. Pro IPv4 adresy se zapisuje běžným zápisem ale **zprava doleva**. U IPv6 se každá hexadecimální číslice odděluje tečkou. zápis je také **zprava doleva**.



## Záznamy DNS

Existuje velké množství DNS záznamů, zde jsou ty nejdůležitější:

- **SOA:** zahajuje záznam zónového souboru, každá zóna má **právě jeden** záznam SOA. Obsahuje mimo jiné název primárního serveru a emailovou adresu správce.

```
> dig SOA fit.vutbr.cz
fit.vutbr.cz. 14386 IN SOA guta.fit.vutbr.cz. ; primary DNS server
                                         michal.fit.vutbr.cz. ; email to the responsible person
                                         202110180          ; Serial number
                                         10800              ; Refresh - 3 hodiny
                                         3600               ; Retry - 1 hodina
                                         691200             ; Expire - 8 dní
                                         86400              ; Minimum - 1 den
```

- **NS:** určuje **autoritativní** server pro danou doménu.

```
> dig NS fit.vutbr.cz
```

```
fit.vutbr.cz.      7012    IN    NS    gate.feec.vutbr.cz.
fit.vutbr.cz.      7012    IN    NS    kazi.fit.vutbr.cz.
fit.vutbr.cz.      7012    IN    NS    guta.fit.vutbr.cz.
fit.vutbr.cz.      7012    IN    NS    rhino.cis.vutbr.cz.
```

- **A:** přímé mapování doménového jména na IPv4 adresu.

```
> nslookup isa.fit.vutbr.cz
```

```
isa.fit.vutbr.cz. 14347 IN A 147.229.176.18
```

- **AAAA:** přímé mapování doménového jména na IPv6 adresu.

```
> dig AAAA www.cesnet.cz
```

```
www.cesnet.cz. 3600 IN AAAA 2001:718:1:1f:50:56ff:feee:46
```

- **MX:** Slouží pro zjištění adresy poštovního serveru (ta může být opět jako doménové jméno), může obsahovat více záznamů rozlišených prioritou (menší číslo značí vyšší prioritu).

```
> dig MX stud.fit.vutbr.cz
```

```
stud.fit.vutbr.cz. 10384 IN MX 10 eva.fit.vutbr.cz.
stud.fit.vutbr.cz. 10384 IN MX 20 kazi.fit.vutbr.cz.
```

- **CNAME:** slouží jako alias, mapuje jej na jméno počítače (síťového zařízení), aliasy nesmí být na pravé straně záznamů a PC jich může mít více.

```
> dig email.fit.vutbr.cz
```

```
email.fit.vutbr.cz. 14400 IN CNAME hermina.fit.vutbr.cz.
hermina.fit.vutbr.cz. 3907 IN A 147.229.9.15
```

- **PTR:** mapuje IPv4 nebo IPv6 adresu na doménovou adresu, obsahuje **reverzní** mapování. Využívají speciální podstromy **in-addr.arpa.** pro IPv4 a **ipv6.arpa.** pro IPv6.

```
> dig -x 147.229.9.23
```

```
23.9.229.147.in-addr.arpa. 14400 IN PTR www.fit.vutbr.cz.
```

```
> dig -x 2001:67c:1220:809::93e5:917
```

```
7.1.9.0.5.e.3.9.0.0.0.0.0.0.0.9.0.8.0.0.2.2.1.c.7.6.0.1.0.0.2.ip6.arpa.
14400 IN PTR www.fit.vutbr.cz.
```

- **SRV**: slouží k lokalizaci služeb a serverů, např. SIP. Mají tvar `_service._protocol.domain_name.`
- ```
> dig SRV _sip._udp.cesnet.cz
_sip._udp.cesnet.cz. 1706 IN SRV 100 10 5060 cyrus.cesnet.cz.

> dig SRV _ldap._tcp.zcu.cz
_ldap._tcp.ZCU.CZ. 600 IN SRV 0 100 389 pleiades.pool.zcu.cz.

> dig SRV _ldap.vutbr.cz
_ldap._tcp.vutbr.cz. 14313 IN SRV 0 100 389 dcb.vutbr.cz.
_ldap._tcp.vutbr.cz. 14313 IN SRV 0 100 389 dca.vutbr.cz.

• a další, dohromady asi 91 DNS záznamů.
```

## Přehled DNS záznamů

| Záznam | Mapování                   | Příklad                                                                                         |
|--------|----------------------------|-------------------------------------------------------------------------------------------------|
| A      | DNS adresa → IP adresa     | tereza.fit.vutbr.cz → 147.229.9.22                                                              |
| AAAA   | DNS adresa → IPv6 adresa   | www.cesnet.cz. → 2001:718:1:101::4                                                              |
| NS     | doména → doménový server   | fit.vutbr.cz. → gate.feec.vutbr.cz.                                                             |
| MX     | doména → poštovní server   | fit.vutbr.cz. → kazi.fit.vutbr.cz.                                                              |
| SOA    | doména → info o zóně       | fit.vutbr.cz. → guta.fit.vutbr.cz.<br>michal.fit.vutbr.cz. 202010201<br>10800 3600 691200 86400 |
| CNAME  | DNS adresa → DSN adresa    | www.vutbr.cz. → piranha.ro.vutbr.cz.                                                            |
| SRV    | služba → DNS adresa + port | _sip._udp.cesnet.cz → cyrus.cesnet.cz. + 5060                                                   |
| PTR    | IP adresa → DNS adresa     | 22.9.229.147.in-addr.arpa. → tereza.fit.vutbr.cz.                                               |
| PTR    | IPv6 adresa → DNS adresa   | 4.0.0.0.0.0.0.0.0.0.0.1.0.1.0.0.0.8.1.7.0.1.0.0.2.ip6.arpa. → www.cesnet.cz.                    |

## DNS rezoluce

Jedná se o proces vyhledání odpovědi v systému DNS. Komunikace DNS je typu klient-server. Dotazy aplikací (klientů) vyřizuje systémová rutina OS tzv. resolver (protože OS může některé DNS záznamy mít ve VP a také aby to každá aplikace nemusela implementovat). Využívá stromovou strukturu jmen a kořenové DNS servery (těch je 13 a obsahují kořenovou zónu). Používají se dva způsoby rezoluce. Pro to, aby počítač mohl provést DNS rezoluci, musí nejdříve znát IP adresu DNS serveru. Ta může být zadána buď ručně (staticky) nebo jí získá od **DHCP** (IPv4) serveru (ten přiděluje: **IP adresu, masku sítě, implicitní bránu** (default gateway) a **adresu DNS serveru**). Adresu **IPv6 DNS** serveru lze získat pomocí **Router Solicitation** zprávy a odpovědi **Router Advertisement** (ICMPv6), která mimo jiné může vracet IP adresu DNS serveru.

## Rekurzivní dotaz

Rekurzivní dotaz provádí **rekurzivní server**, který po dotazu vždy **vrátí IP adresu** (pokud existuje). Při dotazu na rekurzivní server mohou nastat tyto situace:

- rekurzivní server **zná** doménové jméno (má ho v zónovém souboru nebo cache) a vrátí ho klientovi,
- rekurzivní **nezná** doménové jméno a dotazuje se jiného DNS serveru (pokud nezná žádnou část doménového jména, tak je to **kořenový server** - zjistí ze zóny **hint** (součást instalace), jinak může volat autoritativní server pro danou část doménového jména, např. při dotazu na **stud.fit.vutbr.cz**. zná již **vutbr.cz.**, tak se ptá tam), jiný DNS server může být:
  - **rekurzivní**, který vrací přímo **IP** (a dotazující se server si ji může uložit do cache),
  - **iterativní**, který buď vrací **IP**, pokud ji zná, nebo vrací **NS** záznam serveru, který ji může znát. **Rekurzivní** DNS server poté musí kontaktovat tento server.

Rekurzivní servery jsou většinou pouze servery nejblíž počítačum (klientů) poskytované od **ISP**.

- **výhody**: rychlejší než iterativní DNS servery, protože si v průběhu rezoluce mohou ukládat výsledek do cache a ten přště ihned zprostředkovat.
- **nevýhody**: bezpečnostní rizika, možnosti útoku jako **cache poisoning** nebo **DNS amplification attack** (DDoS viz <https://www.cisa.gov/uscert/ncas/alerts/TA13-088A>)

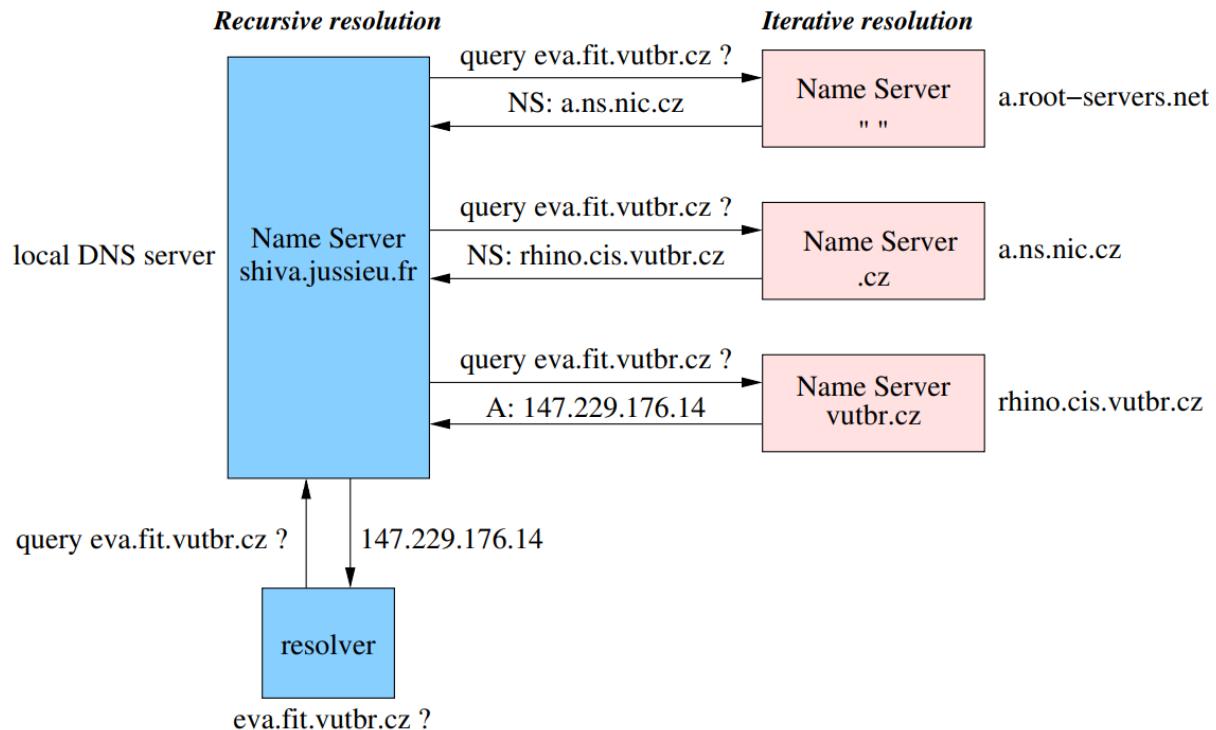
## Iterativní dotaz

Iterativní dotaz provádí **iterativní server**, který po dotazu vrátí nejlepší možnou odpověď, a to:

- **zná** mapování doménového jména na IP adresu, tak ji vrátí.
- **nezná** mapování na IP adresu
  - ale **zná DNS server**, který ji může znát (tento server je na cestě od kořene DNS stromu k hledanému uzlu) a vrací **NS** záznam (doménové jméno autoritativního serveru) tohoto serveru,
  - **nezná** ani žádný **DNS server**, který by ji mohl znát a vrací NS záznam kořenového DNS serveru.

Dotazující se (klient nebo rekurzivní server) poté musí rezoluci opakovat s takto získaným **NS** záznamem.

Obrázek rekurzivního i iterativního dotazu



## Přenos zón

Mechanismus, který umožňuje kopírovat zónové soubory z primárních DNS serverů na sekundární pro zvýšení rychlosti DNS rezoluce, zlepšení distribuce a rozdělení zátěže. Sekundární server musí aktualizovat zónové soubory s intervalu specifikovaném v záznamu **SOA** (DNS polling). Existují dva způsoby přenosu zón, oba jsou realizované přes **TCP** (rezoluce je přes UDP).

- **Celkový přenos zón (AXFR)**: primární server posílá po vyžádání sekundárním celý zónový soubor.
- **Přírůstkový přenos zón (IXFR)**: sekundární server posílá s výzvou o přenos záznamu SOA, pro který chce zónový soubor. Primární server zkонтroluje, k jakým došlo změnám a odesílá pouze záznamy, která byly změněny.

Případně může primární server notifikovat (**DNS notify**) sekundární servery o změně zóny a ty si ji poté aktualizují, pak je zajištěno, že i sekundární servery mají vždy **aktuální záznamy**.

## Druhy DNS serverů

- **Primární**: poskytují vždy autoritativní odpověď pro daný SOA záznam, pro každou doménu je vždy **pouze jeden primární** server.
- **Sekundární**: získávají primární zónové soubory od primárních serverů, slouží jako záloha, poskytuje také autoritativní odpověď, protože má **celý zónový soubor** pro daný SOA záznam.
- **záložní/cach**: Pouze přijímá dotazy, které předává dalším DNS serverům a ukládá si odpovědi do VP. Pokud má uloženou odpověď ve VP, může jí vrátit

při rezoluci, tato odpověď je ale **neautoritativní**. Záznamy z VP jsou po určitém čase (uvedeno v záznamu od primárního/sekundárního serveru) odstraňovány/aktualizovány.

Každý server může být **současně** primární, sekundární i záložní. Každou funkci ale plní pro jiné domény.

## Struktura DNS zprávy

The diagram illustrates the structure of a DNS message. It is divided into several sections:

- Header:** Contains Identification (2 bytes), Flags (2 bytes), no. of questions, no. of answer RRs, no. of authority RRs, and no. of additional RRs.
- Question:** Contains Questions for the name server.
- Answer:** Contains RRs answering the question.
- Authority:** Contains RRs pointing towards an authority.
- Additional:** Contains RRs holding additional information.

Example: answer to www.fit.vutbr.cz query

|                                                       |                |
|-------------------------------------------------------|----------------|
| 59058                                                 | qr, rd, ra, ad |
| 1                                                     | 1              |
| 4                                                     | 6              |
| www.fit.vutbr.cz IN A                                 |                |
| www.fit.vutbr.cz 13963 IN A 147.229.9.23              |                |
| fit.vutbr.cz. 13869 IN NS guta.fit.vutbr.cz.          |                |
| fit.vutbr.cz. 13869 IN NS kazi.fit.vutbr.cz           |                |
| fit.vutbr.cz 13869 IN NS rhino.cis.vutbr.cz           |                |
| fit.vutbr.cz 13869 IN NS gate.feec.vutbr.cz           |                |
| guta.fit.vutbr.cz. 14062 IN A 147.229.8.11            |                |
| kazi.fit.vutbr.cz. 13869 IN A 147.229.8.12            |                |
| rhino.cis.futbr.cz. 86211 IN A 147.229.3.10           |                |
| guta.fit.vutbr.cz. 14156 IN AAAA 2001:67c:1220:809    |                |
| rhino.cis.vutbr.cz. 43692 IN AAAA 2001:67c:1220:e000: |                |

Provádět DNS rezoluci můžeme např. pomocí nástrojů **nslookup**, **dig** a **host**.

## Struktura DNS záznamů

The diagram illustrates the structure of a Resource Record (RR):

- Name (variable length)
- Type (16 bits)
- Class (16 bits)
- TTL (32 bits)
- RDLENGTH (16 bits)
- RDATA (variable length)

Example

|                       |
|-----------------------|
| email.fit.vutbr.cz    |
| CNAME                 |
| IN (0x0001)           |
| 4106 (1 h 8 min 26 s) |
| 10                    |
| hermina.fit.vutbr.cz  |

## Bezpečnost DNS

Jedná se o veřejnou a nešifrovanou službu, která může být terčem útoků. Mezi útoky patří:

- **podvržení odpovědi**: Útočník zašle neautorizovanou odpověď na dotaz klienta před tím, než to stihne DNS server. Útočník hádá ID odpovědi. Tento útok vede na to, že klient obdrží nesprávnou IP a v případě např. http stránek může na této adrese existovat identická stránka (např. internetové bankovnictví), která je ale ovládaná útočníkem.
- **cache poisoning**: vložení nesprávné informace pro mapování DNS záznamů do VP záložních serverů (ty pak poskytují špatné adresy při rezoluci a nastává problém viz předešlý bod). Využívá se k tomuto útoku sekce **Additional**.
- (Distributed) **Denial of Service ((D)DoS)**: Snaží o přetížení DNS serveru, které zamezuje poskytovat odpovědi na legitimní dotazy. Dnes je distribuce DNS tak vysoká, že tento útok je prakticky nereálný.

## Zajištění integrity a autentizace DNS záznamů

Zajištění těchto dvou vlastností, **chrání** před útokem **podvržením** i před **cache poisoning**. Zajišťujeme je pomocí **DNSSEC** - mechanismus, který používá **asymetrickou kryptografií** a **podepisování DNS záznamů a klíčů**. Pro implementaci **DNSSEC** se využívají další DNS záznamy:

- **DNSKEY**: veřejný klíč pro ověřování podpisů,
- **RRSIG**: podpis daného záznamu,
- **NSEC, NSEC3**: odkaz na další záznam při dotazu na neexistující doménu.
- **DS**: záznam pro ověření záznamu **DNSKEY**, uložen v nadřazené doméně.

## Řetězec důvěry (chain of trust)

Implementace zajištění **autentizace a integrity dat** DNS záznamů (implementace DNSSEC). Pro vytvoření řetězce důvěry se používají dva klíče (respektive 4 každý klíč tvoří dvojice **soukromý/veřejný**):

- **Zone Signing Key (ZSK)**: slouží k podpisu dat **privátním** klíčem, která DNS server poskytuje v odpovědích. **Veřejný** klíč pro dešifrování odpovědi je poté uložen (a podepsán KSK) na dotazovaném serveru.
- **Key Signing Key (KSK)**: privátní klíč slouží pro podepsání ZSK, veřejný je uložen na serveru rodičovské domény. Resolver tak může **validovat pravost** ZSK.

Princip **řetězce důvěry** je založen na **rekurzivním podpisu** klíčů. Kořenové servery mají **veřejně známé KSK** (není je třeba ověřovat), kterými podepisují své **ZKS**.

Pomocí **ZKS** kořenových serverů jsou pak podepsány záznamy **DS** (a ostatní) obsahující **KSK** domén **1. řádu** (TLD). Domény **1. řádu** pak tímto **KSK** podepisí svůj **ZKS** a tím podepisují záznamy ze svého zónového souboru, které obsahují také **DS** záznamy s **KSK** domén **2. řádu**... a takhle to jde až k listům doménového stromu.

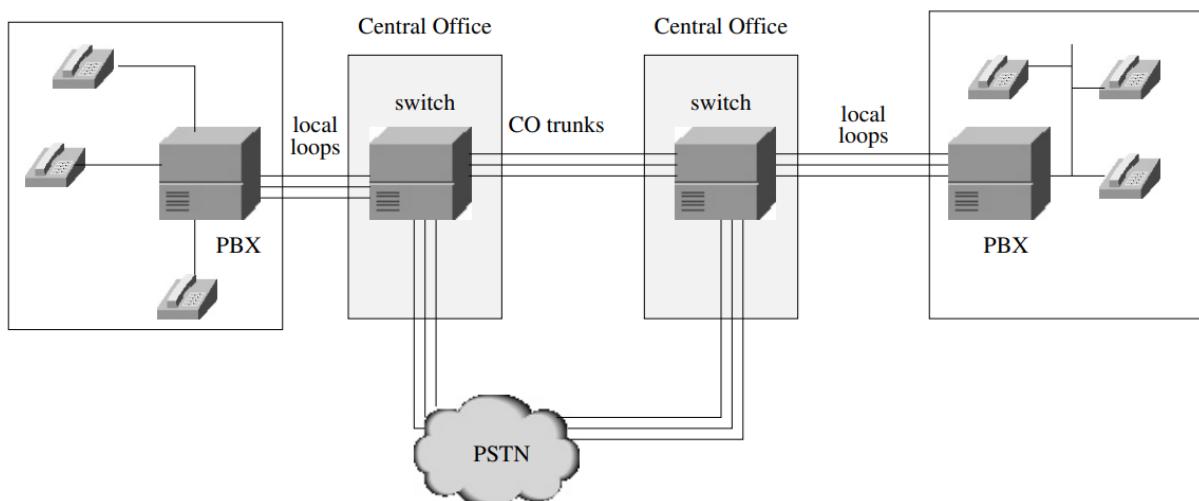
## Šifrování provozu DNS

Autentizaci a integritu dat lze ještě rozšířit o **šifrování záznamů**. To je realizováno pomocí **DNS over TLS** (DoT) a **DNS over HTTPS** (DoH). Oba vyžadují **TCP**. DNS dotazy jsou tak viditelné pouze klientovi a danému dotazovanému serveru. Řeší to sice problém s odposloucháváním, ale **neřeší to problém soukromí**. Server, na který se dotazujeme (např. Google) pořád má přehled o tom, co navštěvujeme a může to tak využít např. pro reklamy.

## IP telefonie

Také známá pod zkratkou VoIP (Voice over IP). IP telefonie implementuje způsob komunikace (tj. primárně telefonní hovory), která je normálně běžná pro síť s **přepojováním okruhů - Public Switched Telephone Network (PSTN)**. PSTN je tvořena:

- koncovými zařízeními (telefony), které jsou připojeny k **Private Branch Exchange (PBX)**.
- lokální smyčkou (spoj mezi PBX a ústřednou (Central office)),
- ústřednami (realizuje spojení s další ústřednou),
- páteřními spoji (trunks), které spojují ústředny.



Telefonní hovory pomocí **PSTN** se vyznačují:

- garantováním **šířky pásma** a spolehlivého přenosu,
- dobrou **kvalitou** přenosu u digitálních ústředen,
- napájením **koncových zařízení** (telefony fungují, i když vypadne proud - nesmí současně vypadnout i v ústředně),
- **spolehlivostí a bezchybností**, které jsou zajištěny dedikovanými spoji.

Stejně **požadavky** jsou tak očekávány i u **IP telefonie** na paketových sítích. Některé požadavky ale mohou být těžce realizovatelné, např. zajištění dostatečného přenosového pásma a s tím související kvalita hlasu (i když dnes to již se stoupající rychlostí internetu není problém) nebo zajištění napájené telefonů (nutnost použití záložních zdrojů). Dále je po IP telefonech požadována **integrace s veřejnou PSTN a mobilními sítěmi**.

## Architektura IP telefonie

Architektura IP telefonie je tvořena několika zařízeními:

- **DHCP server** - přiděluje IP adresu telefonům a ostatním zařízením, umožňuje získat adresu DNS serveru,
- **DNS server** - provádí překlad **tel. čísel a SIP URI** na IP adresy,
- **Ústředna** (Call Server/Gatekeeper) - zajišťuje registraci VoIP zařízení, které pak lze vytáčet,
- **IP telefon** (IP phone) - může se jednat o tzv. **soft phone** tj. softwarový telefon, nebo normální telefon připojený k internetu,
- **Brána** (Trunk Gateway) - zajišťuje propojení s PSTN.

## Úkoly IP telefonie

- **Převod hlasu na IP datagramy**: hlas - **analogový** signál, IP datagram - **digitální** a ještě k tomu nespojitý (nespojité myšleno, že jsou data přenášena po kusech).
- **Řízení komunikace**: komunikuje se přes ústřednu - gatekeeper (klient-server) a mezi telefony (peer-to-peer). Nejčastější je ale zahájení spojení přes ústřednu a následná peer-to-peer komunikace. Je nutné zajistit:
  - **registraci účastníků** (na ústřednách),
  - **adresování hovorů** pomocí tel. čísel a SIP URI,
  - **směrování hovorů** tj. hledání cesty, kudy budou putovat pakety,
  - **vytváření hovorů** (obvykle peer to peer spojení mezi 2 telefony) a jejich udržování.
- **Připojení** do klasického telefonního systému **PSTN** a **mobilní sítě** pomocí brány (trunk gateway).
- **Aplikační služby** jako vyhledávání uživatelů pomocí LDAP nebo WWW či navazování spojení pomocí DNS a DHCP.

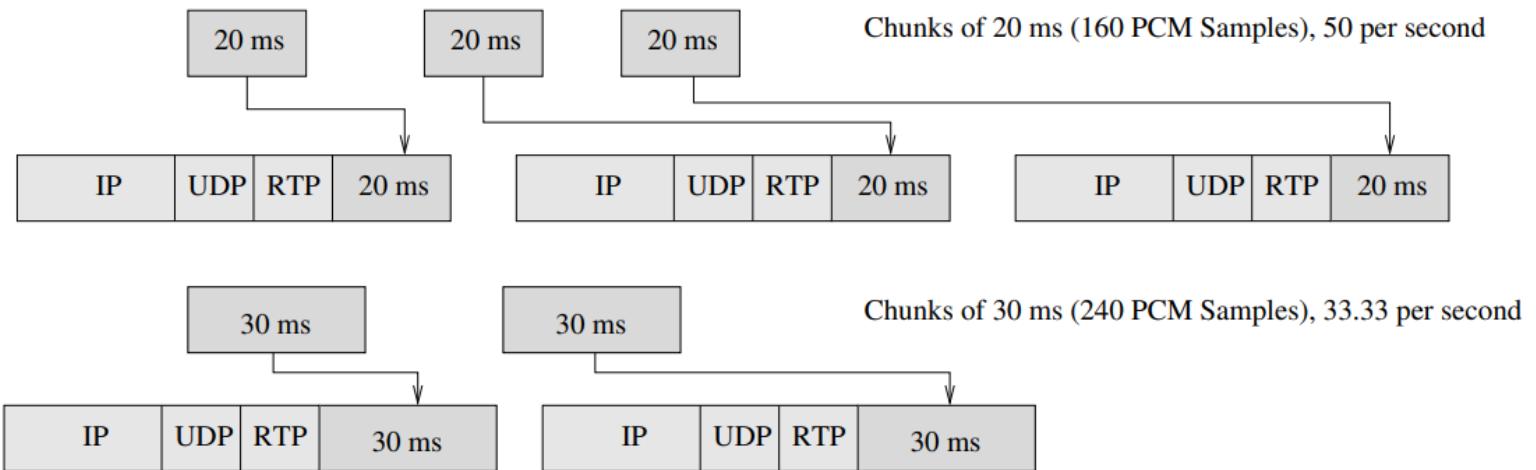
## Kódování hlasu

Hlas je nejdříve nutné **navzorkovat** a poté navzorkovaná data **rozdělit do paketů**, což se dělá pomocí **kódování hlasu** (kodeku). Kodeky využívají faktu, že lidská řeč není (obvykle) na celém rozsahu lidského slyšení a **ignorují okrajové frekvence**, tím provádí (ztrátovou) kompresi.

- Lidské **icho** vnímá na **20 Hz až 20 kHz**, **řeč** je obvykle na **200 Hz až 9 kHz** (9 kHz je ale spíš vysoký zpěv).
- U telefonní linky vzorkujeme frekvence **300 H až 3.4 kHz**, případně až **4 kHz**.
- Pokud chceme vzorkovat **4 kHz** frekvenci, musíme vzorky vytvářet **2x** rychleji (Shannonův teorém) tzn. **8 kHz**. Řekněme, že jeden vzorek má 8 bitů (1 B). Za 1 sekundu budeme muset zpracovat **8000\*8 = 64 kb**. **Šířka pásma** tak bude **64 kb/s**.

Tato šířka pásma bere v potaz pouze samotná data, v paketových sítích je musíme ale zapouzdřit. Na aplikační vrstvě je to protokol **Real-time Transport Protocol**

(RTP), který je dále zapojen do **UDP**, **IP** a **Ethernetového rámce**. Data navíc nemůžeme posílat po příliš velkých kusech (problém se ztrátou a zpožděním). Obvykle se posílají data po **20-30 ms**, což může díky hlavičkám jednotlivých protokolů způsobit navýšení potřebné šířky pásma třeba až o polovinu.



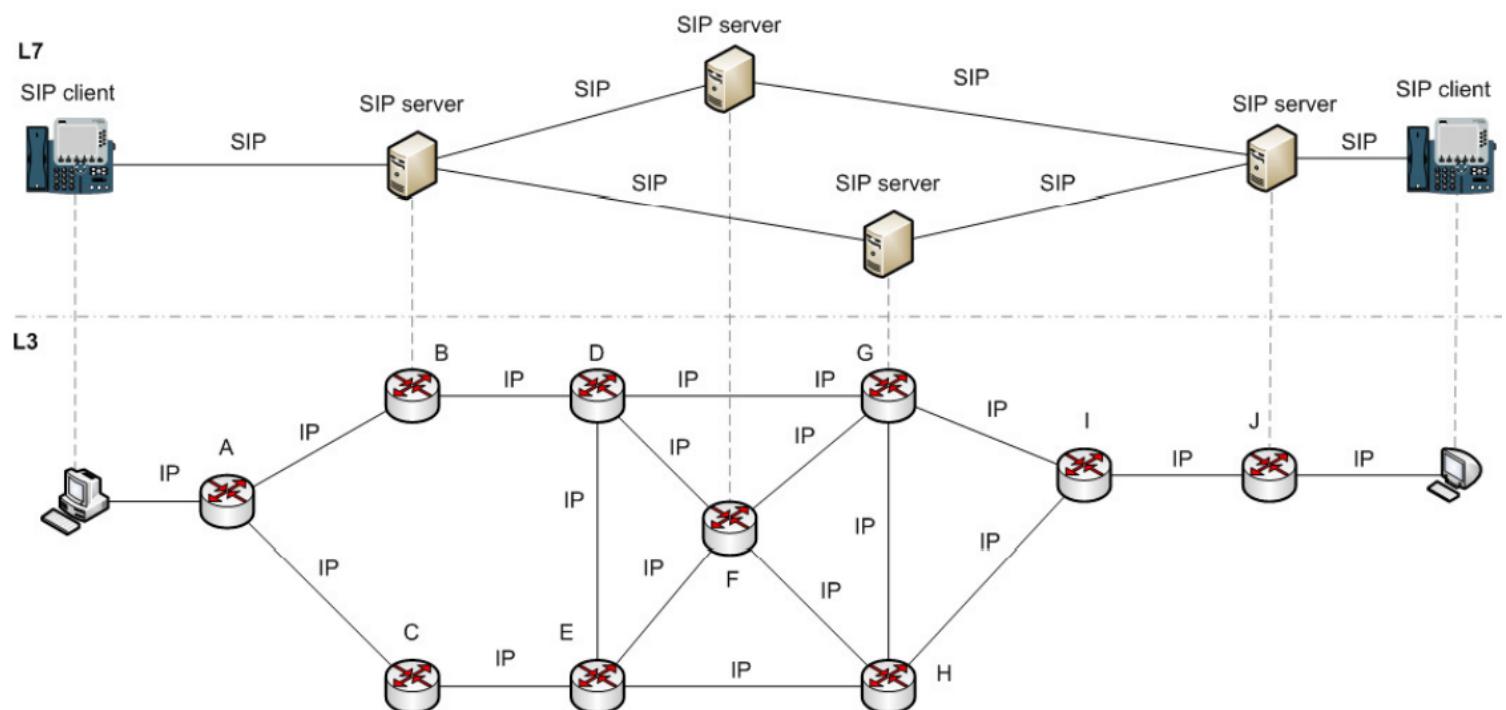
### Session Initiation Protocol (SIP)

Aplikační protokol nad UDP určený pro **signalizaci** VoIP (vytáčení atd.), neřeší přenos dat hovoru. Zajišťuje:

- registraci uživatelů (na bránu),
- navázání spojení a směrování hovorů,
- adresování pomocí URI (**sip:user@domain**) - **adresa** v IP telefonii.

Protokol **neprovádí**: správu relací po jejich vytvoření, nezajišťuje kvalitu hovoru, nezajišťuje přenos hlasových dat.

Jedná se o starý protokol a byl navržen na ISO/OSI modelu, používá tak L7 vrstvu.



## Architektura SIP

- **User Agent Server (UAS - SIP server)** - může mít jednu nebo více z funkcí:
  - **Proxy server**: analyzuje zprávy a směruje hovory,
  - **Lokalizační server**: má informace o umístění klientů,
  - **Server pro směrování**: není nutný, jedná se pouze jen o další bod spojení. Směrování se provádí buď pomocí DNS nebo staticky (předkonfigurováno). Směrovací informace jsou v **SIP hlavičce**.
  - **Registrační server**: přijímá žádosti REGISTER a aktualizuje lokaci.
- **User Agent Client (UAC - SIP klient)** - koncové zařízení, SIP telefon (fyzický nebo SW)

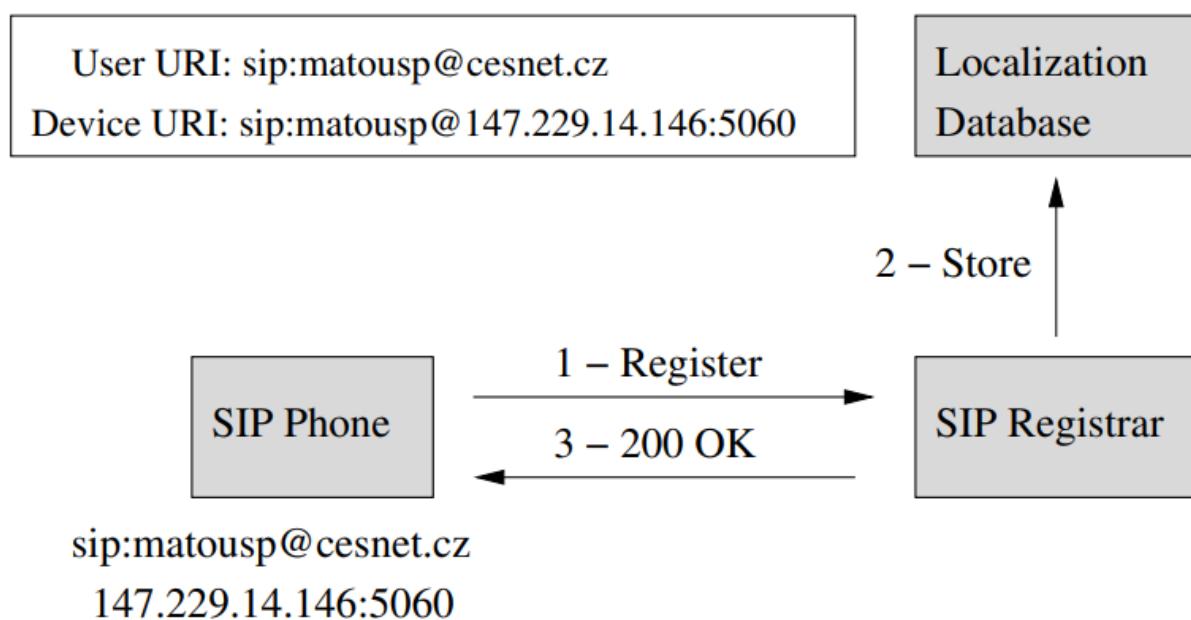
IP telefonie je pomocí SIP **globální** (zahrnuje celý internet) a **distribuovaná** (neexistuje centrální správa, SIP domény se dynamicky propojují). Systém tak umožňuje vyhledat a směrovat hovor na libovolný do sítě zapojený telefon.

## Příkazy protokolu SIP

- **REGISTER** - žádost o registraci,
- **INVITE** - zahájení komunikace vytáčení **volajícím**,
- **OK** - potvrzení komunikace **volaným**,
- **ACK** - potvrzení komunikace **volajícím**, už jde **peer-to-peer**,
- **CANCEL** - zrušení vyváření spojení,
- **BYE** - ukončení vytvořeného spojení, jde **peer-to-peer**,
- **OPTIONS** - získání možností přenosu.

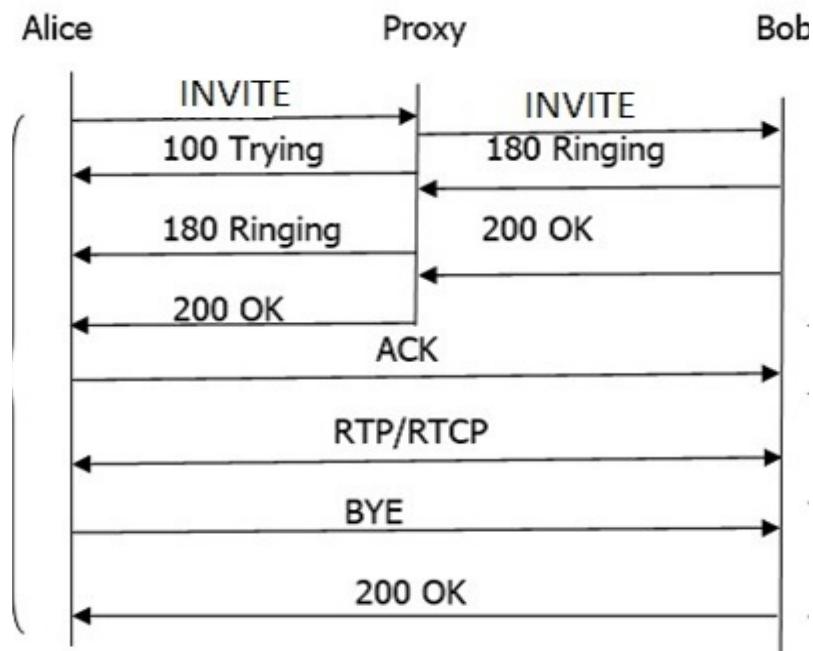
## Registrace SIP telefonu

1. SIP klient pošle svou **IP adresu a port**, na kterém běží IP telefon, SIP serveru ve své doméně.
2. SIP klient mapuje ID uživatele (user URI - `sip:user@sip.vutbr.cz`) na adresu zařízení (device URI - `sip:user@192.168.10.20:5060`).
3. Registrační server si udržuje **lokalizační údaje** (SIP URI - IP adresy) o všech svých připojených klientech ve **své SIP doméně**. Používá k tomu **lokalizační databázi**.



## Ustanovení hovoru

1. Volající zašle zprávu **INVITE**, která je směřována architekturou SIP (obecně více proxy serverů) až k příjemci.
2. Volaný (pokud chce komunikovat - tj. zvednutím sluchátka) zasílá zprávu **OK**, která je směřována architekturou SIP zpět k volajícímu.
3. Volající posílá zprávu **ACK** již přímo volanému (peer-to-peer).
4. Probíhá hovor tj. přenos zvukových dat pomocí protokolů **RTP** a **RTCP**.
5. Jeden z účastníků ukončí hovor zprávou **BYE** (peer-to-peer).
6. Druhý odpovídá s **OK**.



## Session Description Protocol (SDP)

Obsahuje **informace** potřebné pro **navázání datového spojení** pro přenos hlasu a videa (použité kodeky, bitrate, IP adresa spojení atd.). Protokol je zapouzdřen do SIP zpráv **INVITE** a **OK** při zahajování přenosu.

## Real-Time Transport Protocol (RTP)

Aplikační protokol pro **přenos hlasových a obrazových dat** pomocí protokolu **UDP**. Pro každý **směr** a typ **médií** se otevírá **samostatný RTP tok**, tzn. při videohovoru má volaný a volající otevřené mezi sebou 4 RTP toky. Hlavička protokolu mimo jiné obsahuje **typ přenášených dat, sekvenční číslo a časovou značku**, aby bylo možné data poskládat ve správném pořadí, pokud dojde k jejich přehození po cestě.

## RTP Control Protocol (RTCP)

RTCP poskytuje **řídící informace** pro RTP tok dat, ale sám žádná data nenese. Používá se k pravidelnému přenosu **kontrolních** paketů účastníkům streamované

multimediální relace. Hlavní funkcí RTP je **poskytování zpětné vazby na kvalitu služeb** (QoS) poskytovanou RTP. Pro zajištění kvality je třeba eliminovat:

- **ozvěnu** (echo),
- **zpoždění paketů**,
- **ztrátu paketů** - ztráta pod 1% znamená dobrou kvalitu, ztráta od 1% do 2.5% je akceptovatelná a ztráta převyšující 5% již má velký dopad na přenos hlasu a videa,
- **rozptyl následujících paketů** (jitter).

## SIP a DNS

- **NAPTR**: DNS záznam, který se nejčastěji používá společně s **IP telefonií**, slouží k mapování adres SIP serverů. Záznamy **NAPTR** se běžně používají pro ověření existence SIP služeb na doméně. Ok se  
`>nslookup -type=NAPTR cesnet.cz`

```
cesnet.cz. naptr = 200 50 "s" "SIP+D2T" "" _sip._tcp.cesnet.cz.  
cesnet.cz. naptr = 100 50 "s" "SIP+D2U" "" _sip._udp.cesnet.cz.
```

- **SRV**: slouží k vyhledání příslušného SIP serveru.  
`>nslookup -type=SRV _sip._udp.cesnet.cz`  
  
`_sip._udp.cesnet.cz. service = 100 10 5060 cyrus.cesnet.cz.`
- **A** nebo **AAAA**: pro vyhledání IP adresy SIP serveru.

## Zabezpečení VoIP

bezpečnostní **rizika** a jejich řešení:

- **Odposlech**: lze řešit fyzickým zabráněním přístupu k síti nebo pomocí zabezpečení spojení (IPSec, Secure RTP)
- **Viry, Spam over IP Telephony, Phishing over IP Telephony**: antivirus, filtry, vzdělání uživatelů.
- **Neautorizované použití**: vyžadovat autentizaci.
- **Výpadky napětí**: Power over Ethernet, záložní zdroje pro IP telefony a síťovou infrastrukturu (switches a routery).

## Alternativy k VoIP

- **rozdíly**: Používají proprietární nestandardní protokoly. Mohou používat rozdílnou architekturu (peer-to-peer, hybridní - jako VoIP, pouze klient-server). Proprietární správa účtů. Negarantují službu a nemusí např. poskytovat tísňová volání, což IP telefony musí. Mají omezené možnosti propojení s PSTN, mobilními sítěmi a standardním VoIP.
- **společné vlastnosti**: adresování, směrování a přenos hlasových dat a stímkování, spojené udržování spojení, vytváření účtů a registrace uživatelů a jiné služby

kromě hovorů, např. Instant Messaging, přenos souborů, přenos obrazu, sdílení plochy atd.

## Správa sítí

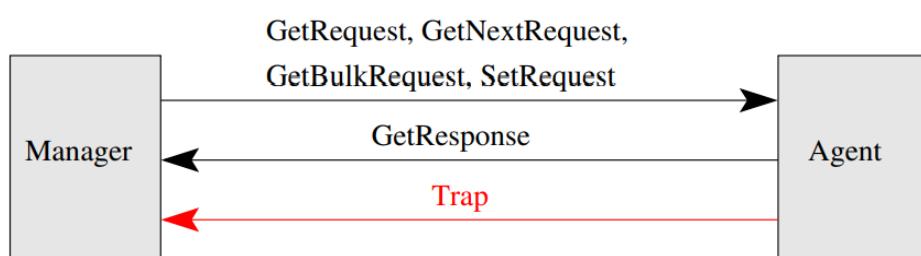
Správa sítí zahrnuje řešení problémů známých pod akronymem **FCAPS**:

- **Fault management**: zaznamenání, izolování a opravení chyby.
- **Configuration management**: zavádění konfigurací zařízení, zálohování konfigurací, zaznamenávání změn, aktualizace SW.
- **Accounting management**: zisk statistik využitívání sítě (např. IP telefonie) a stím spojené účtování.
- **Performance management**: zajištění dostatečného výkonu sítě (např. šířky pásmo pro VoIP)
- **Security management**: zabezpečení sítě (autentizace, autorizace, zabránění fyzického přístupu k zařízením atd.)

## Simple Network Management Protocol (SNMP)

SNMP je protokol pro přenos informací o zařízeních na síti, pod pojmem SNMP ale obecně myslíme celý mechanismus pasivní správy sítí (aktivní je např. ICMP ping, telnet, PortQuery, ...) za použití tohoto protokolu. SNMP zpráva sítě je tvořena:

- **Management Station**: jedná se o server na kterém běží SW, který se pomocí **protokolu SNMP** obvykle pravidelně dotazuje (polling) **SNMP agentů** na portu **161** a stahuje si od nich tak **monitorovací informace**, které ukládá a např. umožňuje nějak graficky zobrazit. Dotazy na data **MS** realizuje příkazy **Get**, **GetNext**, **GetBulk** a MS také může konfigurovat SNMP agenta pomocí příkazu **Set**.
- **SNMP Agent**: jedná se o **zařízení v síti** (switche, routery, tiskárny, počítače atd.), u síťových zařízení je nainstalován SW pro SNMP protokol obvykle od výrobců, na PC si jej musíme stáhnout. SNMP agent **sbírá informace o zařízení**, na kterém běží (např. vytížení CPU, vytížení paměti, stav toneru, počet papírů v zásobníku, počet odeslaných paketů, počet přijatých paketů atd.) a na vyzvání Management Station je jí odesílá. Nevede si však historii, tu musí zaznamenávat MS. SNMP agent může kontaktovat MS **sám** od sebe při **vzniku** nějaké závažné **chyby** (nakonfigurováno jaká chyba, resp. jaká úroveň závažnosti). Tato komunikace je realizována pomocí **asynchronní** (tj. MS si ji nevyžádala) zprávy **Trap** na portu **162**.
- **SNMP protokol**: **nestavový** protokol sloužící pro přenos informací o zařízeních na síti, jedná se o protokol typu request-response a existuje několik verzí **SNMP**, **SNMPv2** (přidává autentizaci), **SNMPv3** (přidává šifrování).



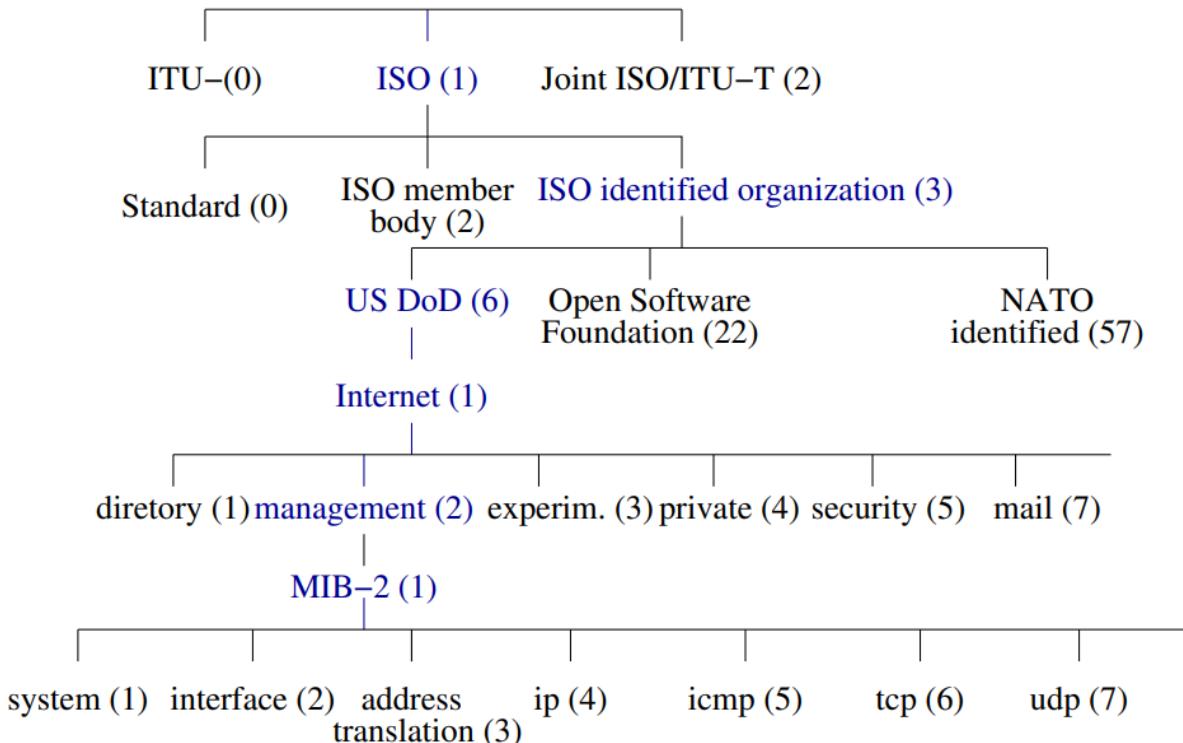
|                                      |                      |             |
|--------------------------------------|----------------------|-------------|
| SNMP header                          | Version              |             |
|                                      | Community String     |             |
|                                      | PDU type: GetRequest |             |
|                                      | Request ID           |             |
|                                      | 0                    |             |
|                                      | 0                    |             |
|                                      | Object Name 1        | Value 1     |
|                                      | Object Name 2        | Value 2     |
|                                      | Object Name 3        | Value 3     |
| PDUs                                 |                      |             |
| Variable bindings                    |                      |             |
| GetRequest,GetNextRequest,SetRequest |                      | GetResponse |
|                                      |                      | Trap        |

## Monitorované objekty

Objekty jsou uspořádané do **skupin** v **databázi objektů MIB** (Management Information Base). Databáze MIB má stromovou strukturu:

- **nelistové uzly**: reprezentují skupiny objektů,
- **listové uzly**: reprezentují konkrétní objekt.

Objekty jsou adresováný pomocí **OID**, které **reprezentuje cestu ve stromové struktuře** a má tvar čísel oddělených tečkami, kde čísla identifikují uzel na dané úrovni stromu a tvoří tak cestu. Např. **OID 1.3.6.1.2.1.4** představuje objekt **ip**.



## Definice monitorovaných objektů

Objekty jsou definovány a popisovány pomocí jazyka **Structure Management Information** (SMI). Popis objektu je tvořen:

- **jménem objektu**, což je **OID** - jednoznačný identifikátor objektu,
- **syntaxí**, která určuje (abstraktní) datový typ spojený s objektem, např (Counter32 - přeteče zpět na 0, Gauge32 - nepřeteče a setrvává na max hodnotě, OBJECT IDENTIFIER, SEQUENCE, INTEGER, IpAddress, NetworkAddress, OCTET STRING, ...)
- **kódováním pro přenos po síti** - používá se kování **BER**.

Příklad definice objektu **ipInDelivery** ve skupině **ip**:

```
ipInDelivers OBJECT-TYPE -- OID 1.3.6.1.2.1.4.9
  SYNTAX     Counter32
  MAX-ACCESS read-only
  STATUS      current
  DESCRIPTION "The total number of input datagrams successfully
               delivered to IP user-protocol (including ICMP)."
  ::= { ip 9 }
```

## Basic Encoding Rules (BER)

Jedná se o binární kódování informací po za sebou jdoucích trojicích tvořených:

- **Type**: typ objektu uložen v **1. bytu**.

8 7 6 5 4 3 2 1

| Class            | P/C | Tag number                           | Length | Value |  |
|------------------|-----|--------------------------------------|--------|-------|--|
| Identifier octet |     | Specifies the end of the value field |        |       |  |

- **Length**: délka hodnoty v bytech, implicitně je uložena pouze na 1 B (tj. na druhém bytu). Pokud nelze vyjádřit na 1 B (tj. v tomhle případě větší než 0x80), pak je délka vyjádřena v **N** následujících bytech a **N = hodnota 2. B - 0x80**.
- **Value**: hodnota záznamu.

## Nasazení SNMP

Při praktickém nasazení SNMP musíme brát v potaz:

- **kolik zařízení** bude monitorováno,
- jaká bude **frekvence sběru** dat,
- jaký bude **objem přenášených dat**.

Tyto tři faktory musí být vyváženy tak, aby nepředstavovaly významné zatížení sítě.

Dále musíme brát v potaz, že protokol **SNMP** je **zapouzdřen do UDP** a může tak docházet ke **ztrátám** datagramů (není zde potvrzení). Např. zpráva **Trap nemusí dorazit** nebo zápis konfigurace pomocí příkazu **Set se nemusí provést** atd. Systém **SNMP** je **centralizovaný** a všechna data jsou na jedné stanici (Management

Station), při poruše této stanice ztrácíme přehled o síti. Měli bychom používat **šifrovanou verzi** SNMPv3 kvůli **bezpečnosti**, pokud by se někdo naboural do sítě a zde běželo SNMP starších verzí, získá informace o téměř všech zařízeních. Program pro SNMP: **snmpwalk**.

## NetFlow

NetFlow narodil od SNMP neumožňuje monitorovat zařízení v síti a získávat informace jako (zatížení CPU a paměti, stav toneru, počet papírů v tiskárně atd.). NetFlow se zaměřuje více na monitorování komunikace na síti v podobě síťových toků, tj.:

- **kdo a s kým komunikuje** (ip adresy a jiné adresy),
- **kolik dat si posílají**,
- **kdy komunikují**,
- **jaké používají protokoly**,
- atd.

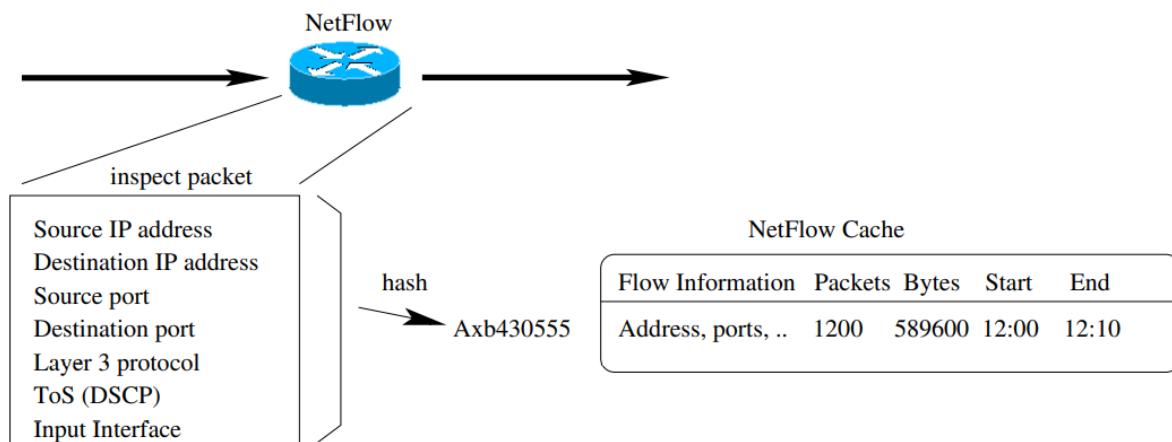
Informace o stavu zařízení a jestli funguje nelze získat pomocí NetFlow. NetFlow byl původně **proprietární** protokol vyvinutý firmou **Cisco**, dnes je již **standardizován** pomocí **RFC**. Stejně jako u SNMP nepovažujeme NetFlow pouze za protokol, ale mechanismus monitorování a správy sítě. Architektura NetFlow je tak tvořena:

- **Exportérem**: jedná se o sondu/router pro získávání statistik o tokích. Často se jedná o samostatnou sondu která pouze **odposlouchává** provoz a zaznává jej, protože řešení v rámci routeru může být příliš drahé.
- **Kolektorem**: zařízení, které ukládá záznamy o tokích. Jedná se o nějaký server (podobně jako MS u SNMP) tzn. **centralizované** řešení.
- **Protokolem**: NetFlow\_v5, NetFlow\_v9, IPFIX.
- **Nástroje pro zobrazení dat**: grafy, statistiky, výpisy atd.

## Síťový tok

**Posloupnost paketů** mající **společnou vlastnost** (např. IP adresu), které prochází určitým **bodem pozorování** za určitý **časový interval**. Záznam o toku obsahuje informace o toku, **nikoliv přenášená data**. Mezi tyto informace patří:

- zdrojová a cílová **IP adresa**,
- zdrojový a cílový **port**,
- **typ protokolu** (může být z více vrstev tj. transportní a aplikacní),
- **časové razítko** (obsahuje začátek a konec toku),
- **počet přenesených dat** v B.



## Exportér NetFlow

**Síťové zařízení** a software (často dedikované - sonda), které **monitoruje** procházející provoz. Vytváří **záznamy** o tocích a **aktualizuje** starší záznamy ve své **NetFlow cache**. **Vyhledávání** záznamů pro aktualizaci provádí **pomocí heše** informací identifikující tok (IP adresy a porty). To znamená, že pro **každou komunikaci vytváří 2 toky** (záleží na směru). Exportér také může provádět nějaké agregace dat, více viz dále. Na rozdíl od SNMP odesílájí **exportéry** data kolektoru **sami bez vyžádání** po:

- ukončení toku (TCP - **FIN, RST**),
- po **vypršení časovače**
  - **neaktivní timeout**: nebyl odeslán žádný paket v daném toku (defaultně 15 s),
  - **aktivní timeout**: odesílá data u příliš dlouhých nepřerušovaných toků, může jít např. o nějaké stahování (délka může být třeba 30 min),
- **zaplnění NetFlow cache**.

## Kolektor NetFlow

Kolektor je na síti **jeden** a přijímá pakety NetFlow z **jednoho či více exportérů**. Jeho funkce jsou poté následující:

- **Zpracování záznamů** o tocích a jejich **ukládání** na disk nebo do databáze, ukládají se binárně ve formátu optimalizovaném pro vyhledávání (před uložením mohou být data ignorována),
- **Grafické zobrazování** dat (např. nějaká **heat mapa** vytíženosti) pomocí SW, např. **Flowmon**,
- **Realizovat dotazování** nad daty pomocí např. **nfdump** (provádět agregace uložených neagregovaných dat - chci zobrazit všechny toky z jedné IP, druhá mě ale nezajímá, a sečíst objem přenesených dat - typický příklad pro účtování)
- **Automatizace** např. automatické odhalení anomalií - podezřelých toků (PC v kanceláři s pracovní dobou od 9 do 17 začne ve 22 posílat/přijímat velké množství dat → možná je to útok).

## Protokol NetFlow

Příklad paketu **verze 5**:

|                   |          |                        |           |                   |     |  |  |
|-------------------|----------|------------------------|-----------|-------------------|-----|--|--|
| Version           | Count    | sysUptime              |           |                   |     |  |  |
| Unix secs         |          | Unix nsecs             |           |                   |     |  |  |
| Flow sequence     |          | Engine Type            | Engine ID | sampling interval |     |  |  |
| Source IP address |          | Destination IP address |           |                   |     |  |  |
| Next hop          |          | Input                  |           | Output            |     |  |  |
| Packets           |          | Octets                 |           |                   |     |  |  |
| First SysUptime   |          | Last SysUptime         |           |                   |     |  |  |
| Src Port          | Dst Port | Pad1                   | Flags     | Proto             | ToS |  |  |
| Src AS            | Dst AS   | Src mask               | Dst mask  | Pad2              |     |  |  |

Paket je tvořen:

- **hlavičkou**, která obsahuje
  - počet NetFlow záznamů přenášených v datové části,
  - čas odeslání tohoto NetFlow paketu,
  - identifikaci sondy, ze které byl paket odeslán
  - atd.
- **daty** (obsah paketu) obsahuje množinu záznamů o tocích, každý záznam představuje statistiku o jednom toku.

### Vylepšení ve verzi 9 a IPFIX (IP Flow Information Export)

NetFlow verze 5 má **fixní** formát (neumožňuje např. export IPv6 adres), **verze 9** to řeší zavedením šablon. Hlavička zůstává a obsahuje informaci o **ID šablony**, podle které jsou zapsána data v datové části paketu (šablony zasílá exportér kolektoru, ten si ji pak zapamatuje a již není součástí paketů - datová část tak může obsahovat šablony, informace o datových tocích nebo oboje, **většinou ale pouze** informace o **datových tocích**). **IPFIX** je poté ještě **více flexibilní**, definuje více polí, které lze použít v šablonách, jinak se výrazně neliší.

### Použití NetFlow

- **Monitorování sítě**, plánování sítě, **bezpečnostní analýza**,
- Sledování aplikací, uživatelů a zajištění **účtování**,
- **Dlouhodobé ukládání** informací o tocích (někteří poskytovatelé to mají dané ze zákona - pak je nutné počítat poměrně s velkou paměťovou náročností objem NetFlow statistik je roven asi 1-2 % objemu provozu sítě, tj. při dení zátěži 100 GB bude třeba uložit 1-2 GB NetFlow dat).

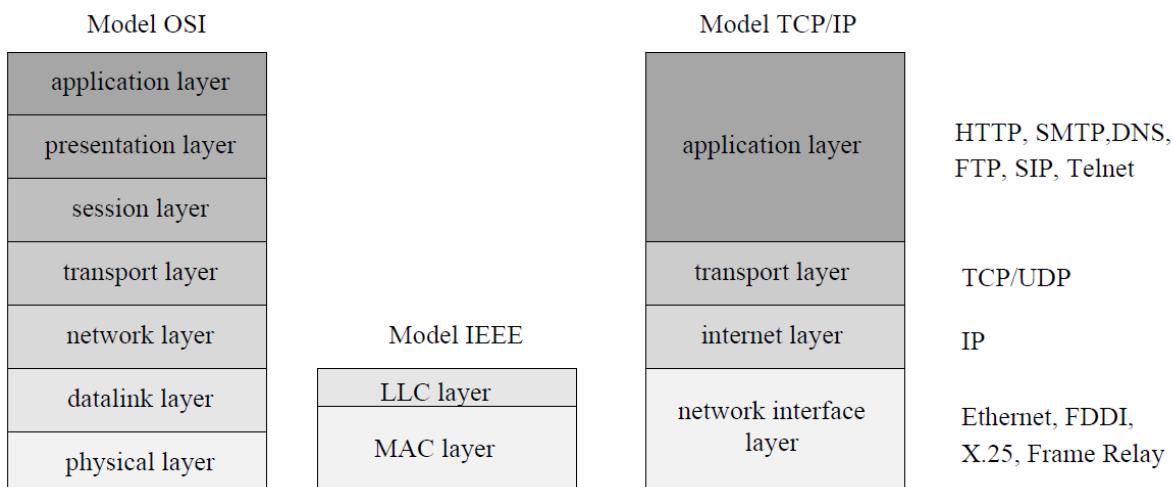
# 43. TCP/IP komunikace (model klient-server, protokoly TCP, UDP a IP, řízení a správa toku TCP)

## Model TCP-IP

Model TCP/IP je v současnosti hlavní využívaný síťový model. Má **5 vrstev** (v některé literatuře 4, kdy spodní 2 jsou spojeny do vrstvy síťového rozhraní). Je tvořen:

- **Aplikační vrstva** (L7 - používá se i pro TCP/IP), obstarává komunikaci na úrovni aplikací (viz předchozí otázka) - adresa je např. URL nebo emailová adresa,
- **Transportní vrstva** (L4) řeší komunikaci mezi **2 logickými procesy** - adresou je číslo portu,
- **Síťová vrstva** (L3) řeší komunikaci mezi **2 stroji** napříč celou sítí - adresou je IPv4 nebo IPv6 adresa,
- **Linková vrstva** (L2) řeší komunikaci dvou **síťových rozhraní** - adresou je MAC adresa,
- **Fyzická vrstva** (L1) pak přenos po médiu.

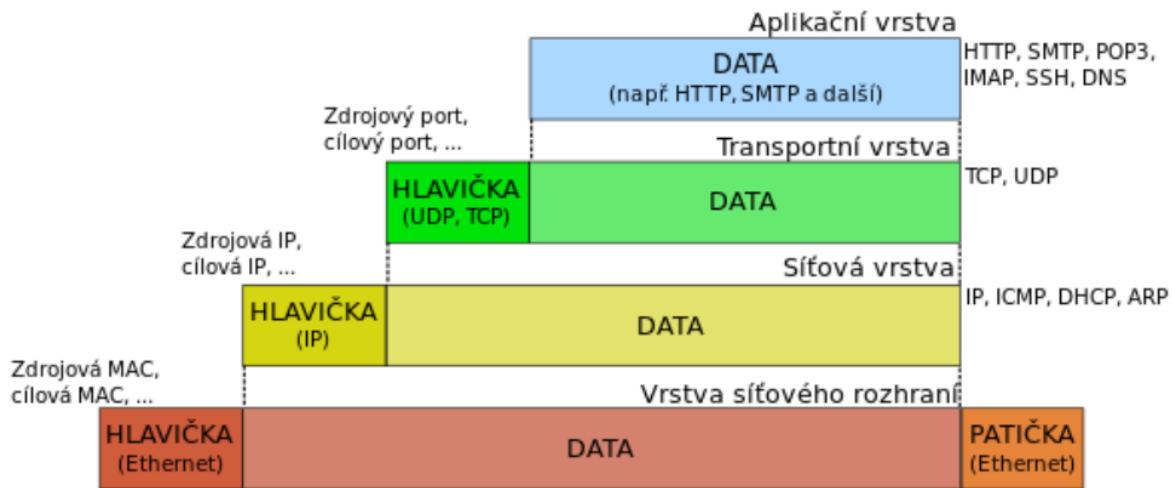
### ❖ Architektura TCP/IP



Důvodem vrstvení modelu je **oddělení logiky jednotlivých vrstev**, například L4 staví na službách L3 a své služby poskytuje vyšší vrstvě (L7). Cílem je schopnost doručit paket z libovolného uzlu do jiného uzlu za jakýchkoliv okolností.

Při průchodu paketu sítí probíhá tzv. **zapouzdření** (vzniká **PDU** - Protocol data unit), tj. čistá data jsou nejprve zabalená do L7 hlavičky, takovýto paket je zabalen do L4 hlaviček (tím vzniká UDP datagram nebo TCP segment), ten je zabalen do IP

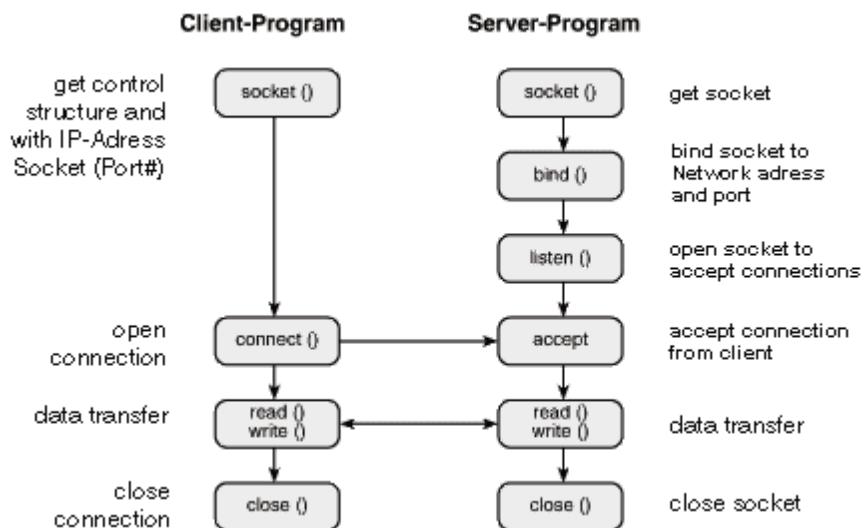
hlavičky, tak vzniká IP paket, ten je zabalen do L2 hlaviček a vzniká rámec. Při průchodu sítí síťové prvky rozbalují pouze část hlaviček dle své vrstvy, např. **switch** (L2 prvek) si přeče L2 hlavičku a při přeposlání přidá novou, podobně pak **router** (L3 prvek) nahlíží do IP hlaviček a podle nich směruje.



## Klient-server

Model klient-server je běžné komunikační schéma mezi 2 procesy. **Klient aktivně zahajuje spojení** k serveru a požaduje od něj službu (posílá mu požadavky a dostává na ně odpovědi). **Server pasivně čeká** na požadavek klienta, který následně provede a zašle odpověď odpovídající např. úspěšnosti provedení požadavku. Rozlišujeme **iterativní** a **konkurentní server**:

- **iterativní server** zpracovává jednoho klienta po druhém, nemůže zpracovávat více klientů současně, ostatní musí čekat,
- **konkurentní server** může naráz zpracovávat více klientů díky využití **dětských procesů** (funkce *fork*), kdy každý proces zpracovává požadavky jednoho klienta. Implementace je standardně následující:
  - Otevře se schránka (funkce *socket*), sváže se s portem (*bind*) a server začne poslouchat (*listen*)
  - Po přijetí spojení od klienta (*accept*) vytvoří nový proces (*fork*).
  - Synovský proces používá původní schránku, rodičovský proces ji uzavře (*close*), aby každou schránku obsluhoval jedený proces. Rodičovský proces pak ihned může zpracovat další požadavek.
  - Následně klasicky probíhá komunikace pomocí **read/write**.



- konkurentní server je **typický pro TCP**, protože lze jednoznačně přiřadit příchozí zprávu od klienta do správné schránky na základě src a dst IP adres a portů (zajišťuje OS, klient si pomocí funkce connect blokuje port - connection-oriented protocol).
- UDP typicky nebývá konkurentní, je možné tento nedostatek obejít například využitím **nového portu** pro každého klienta, UDP server klientovi řekne, ať pro další komunikaci použije **jiný port** než ten standardizovaný, protože funkce sendto může použít pokaždé náhodný port (tak funguje TFTP).

Komunikace typicky probíhá dle nějakého **protokolu**, což je soubor syntaktických a sémantických pravidel, kterými se komunikace řídí. Protokol je možné popsat:

- neformálně slovně** (např. RFC)
- formálně** (konečné automaty, gramatiky).

Protokoly typicky popisují navázání spojení, adresování, přenos dat, řízení toku komunikace.

## Dělení komunikace

Z pohledu počtu komunikujících entit rozlišujeme:

- unicast** = zpráva pro jeden uzel
- broadcast** = zpráva pro všechny uzly v síti
- multicast** = zpráva pro vybranou skupinu uzlů v síti (typicky dynamicky tvořenou)

## Transportní vrstva

Transportní vrstva vytváří logické **spojení mezi procesy**, hlavními protokoly jsou **TCP** a **UDP**. Mezi více procesy na jednom počítači a jejich komunikacemi je rozlišeno pomocí přiděleného **portu**. Ke komunikaci na L4 úrovni slouží SW prostředek **schránky** (socket). Port je 16-bitové číslo (0-65535), dělíme je:

- 0-1023, systémové = standardizované, např. http 80
- 1024-49151, uživatelské
- 49152-65535, dynamické

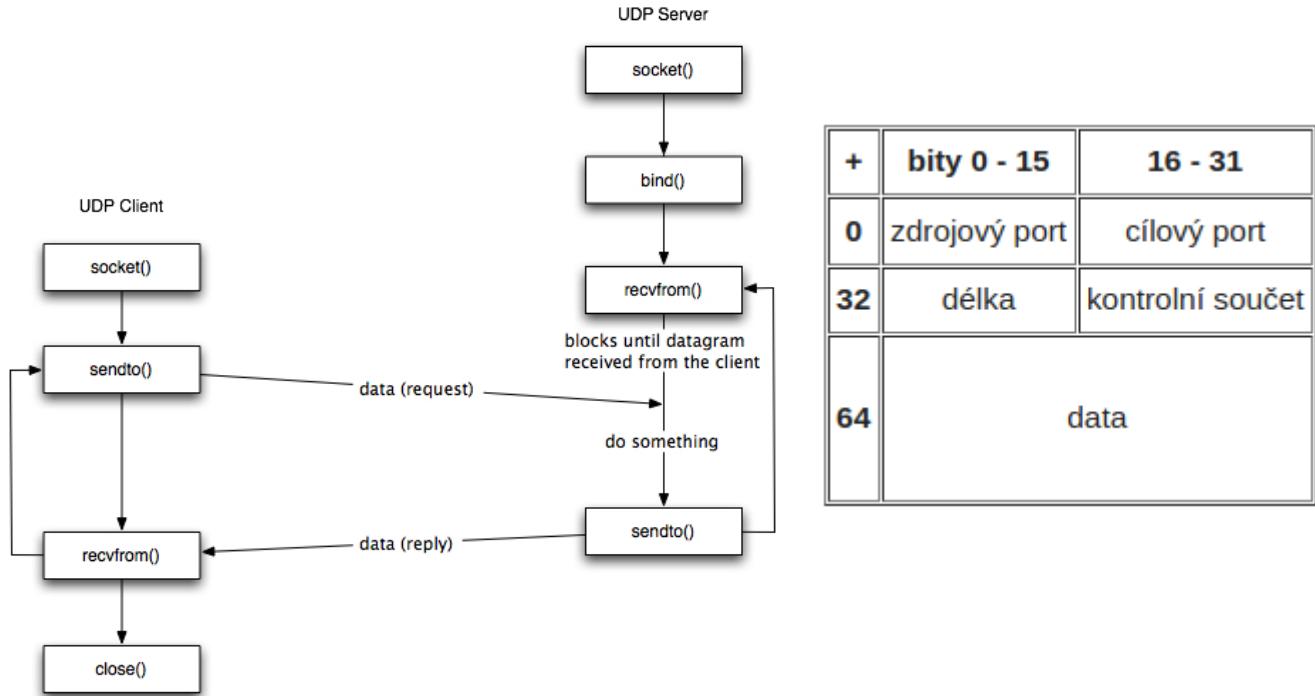
Číslo portu serveru musí být známé při navazování spojení (např. specifické číslo pro daný protokol, viz IANA). Číslo portu klienta bývá **náhodně generováno systémem**.

## User Datagram Protocol (UDP)

Connection-less protokol, **nenavazuje tedy spojení**, podobně jako na L3 doručení probíhá na bází **best-effort**. Hlavíčka je tedy velmi jednoduchá, obsahuje **zdrojový a cílový port, checksum a délku** (8B).

Protože není navázáno spojení, nezaručuje doručení, ani doručení v pořadí. Díky tomu je ovšem komunikace **rychlejší**, což je vhodné pro časově závislé aplikace

(např. přenos hlasu přes RTP, video streaming, DNS). V případě požadavků na **spolehlivost** je nutné to řešit na L7 (např. protokol TFTP pracuje nad UDP, na L7 přidává ACK datagramy, díky kterým je možné poznat, zda doručení proběhlo v pořadku).



## Transmission Control Protocol (TCP)

Spojově orientovaný protokol poskytující **spolehlivý přenos** dat (řeší ztráty paketů a pořadí, vyšší vrstvy se o případné ztrátě ani nedozví). Na začátku komunikace se vytváří spojení pomocí **3-way handshake**. Využívá pipeliningu (viz dále) a klouzavého okna, díky kterému je přenos řízen a je předcházeno zahlcení. Hlavička TCP obsahuje:

- **porty** (viz UDP),
- **checksum**,
- **sequence number** = číslo prvního B segmentu,
- **acknowledgement number** = číslo prvního B očekávaného segmentu k přijetí,
- **příznaky**.

|       | Octet | 0                                       |   |   |   |          |   |   |   | 1  |     |     |     |     |     |     |     | 2                |   |   |   |   |   |   |   | 3 |   |   |   |   |   |   |   |
|-------|-------|-----------------------------------------|---|---|---|----------|---|---|---|----|-----|-----|-----|-----|-----|-----|-----|------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit   | 7                                       | 6 | 5 | 4 | 3        | 2 | 1 | 0 | 7  | 6   | 5   | 4   | 3   | 2   | 1   | 0   | 7                | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0     | 0     | Source Port                             |   |   |   |          |   |   |   |    |     |     |     |     |     |     |     | Destination Port |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 4     | 32    | Sequence Number                         |   |   |   |          |   |   |   |    |     |     |     |     |     |     |     |                  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 8     | 64    | Acknowledgment Number (If ACK Flag Set) |   |   |   |          |   |   |   |    |     |     |     |     |     |     |     |                  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 12    | 96    | Data Offset                             |   |   |   | Reserved |   |   |   | NS | CWR | ECE | URG | ACK | PSH | RST | SYN | Window Size      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| ...   | ...   | ...                                     |   |   |   |          |   |   |   |    |     |     |     |     |     |     |     |                  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

TCP Segment Header

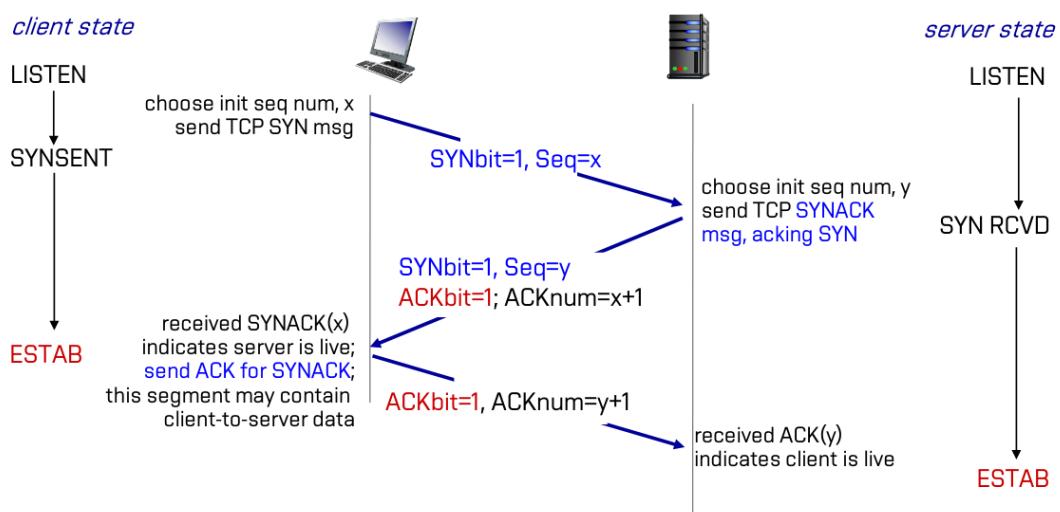
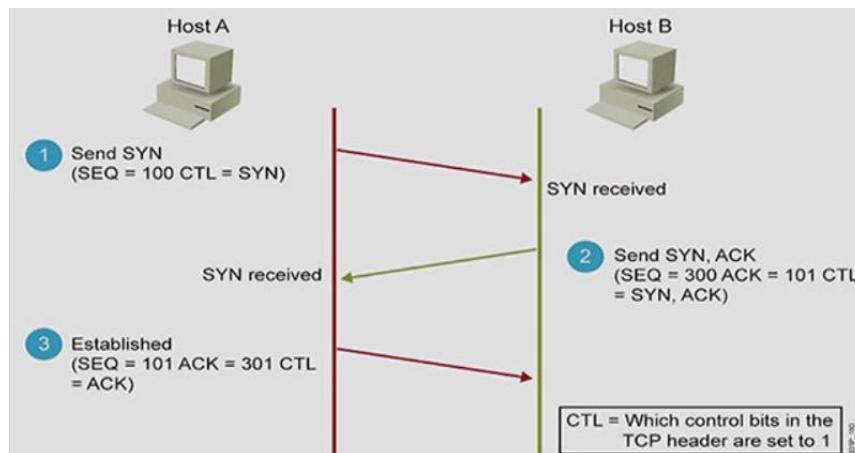
TCP rozděluje přicházející data do segmentů (je-li to třeba dle MTU) a na straně příjemce je zase skládá.

### 3-way handshake

Navázání spojení probíhá na principu 3-way handshake následovně:

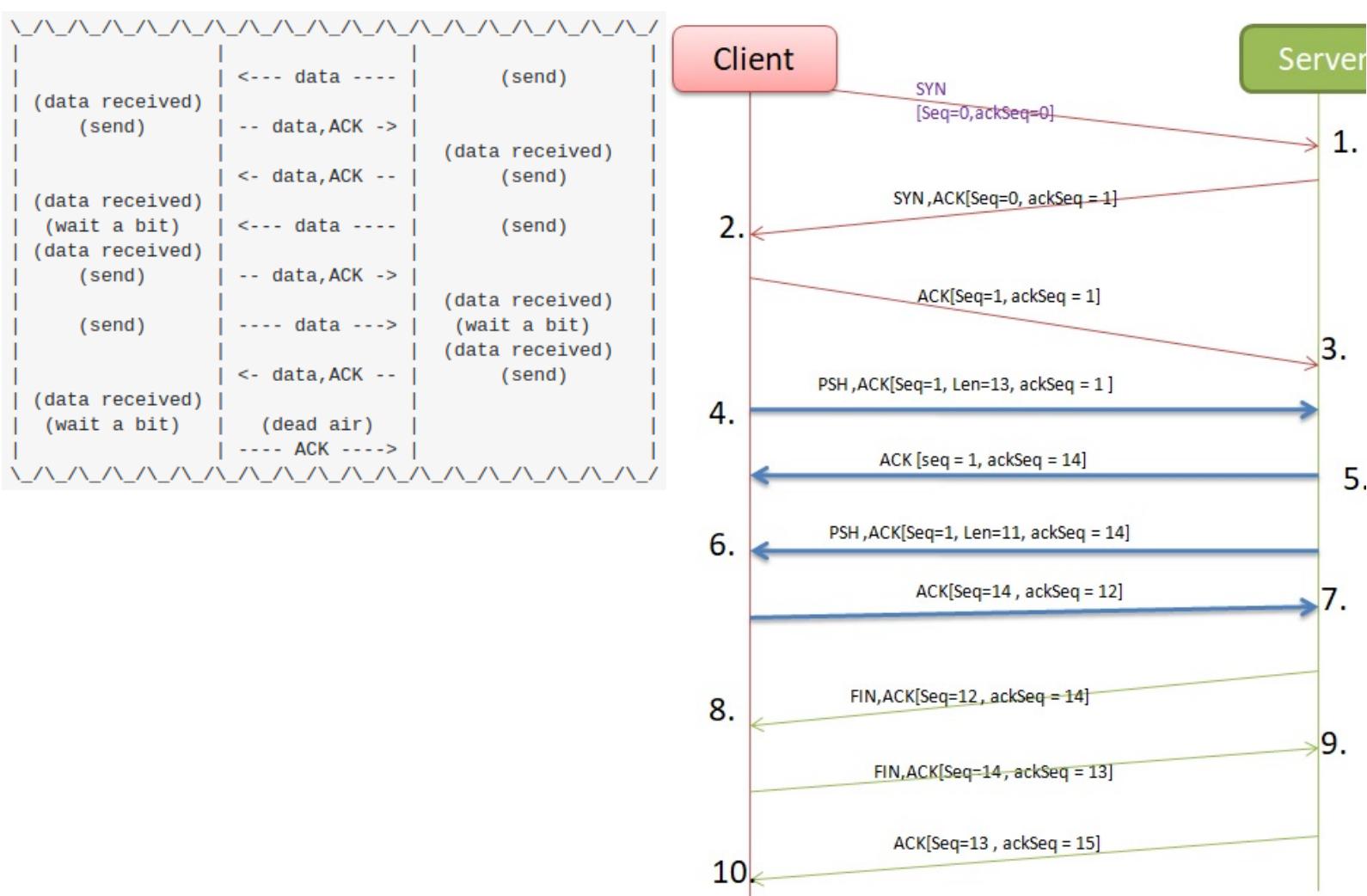
1. Klient chce zahájit spojení a zasílá paket s příznakem **SYN**, do tohoto paketu také vygeneruje náhodné **Sequence Number**, řekněme **X** (na obrázku je to 100) pro tok od klienta k serveru, Acknowledgment Number nevyplňuje.
2. Server na **SYN** paket odpovídá paketem s příznaky **SYN** a **ACK**, také generuje náhodné **Sequence Number**, řekněme **Y** (na obrázku je to 300), pro tok od serveru ke klientovi. Server navíc vyplní **Acknowledgment Number** jako **X + 1** (na obrázku  $100 + 1 = 101$ ), čímž zároveň potvrzuje přijetí prvního **SYN** paketu od klienta.
3. Klient potvrzuje zahájení komunikace pomocí **ACK** paketu reagující na **SYN+ACK** paket. Jako **Acknowledgment Number** vyplní hodnotu **Y + 1** (na obrázku  $300 + 1 = 301$ ) a inkrementuje svoje **Sequence Number**, které také odesílá.
4. Je vytvořeno spojení (respektive lze na to nahlížet jako 2 spojení) a klient se serverem si mohou vyměňovat **bezztrátově** data, klient i server jsou ve stavu **ESTABLISHED**.

2-way handshake **nestačí** kvůli možnosti znovuposlání, kdy by mohlo dojít k ponechání **napůl otevřeného spojení**, viz obrázek. Z toho důvodu se používá 3-way handshake.



## Komunikace v TCP

Při komunikaci jsou zvětšována **sériová a potvrzovací čísla** na základě počtu přijatých/odeslaných **bajtů** (potvrzuje se každý byte a případně jedním ACK paketem může být potvrzeno více paketů s daty). Klient i server by měl hned po obdržení paketu zasílat jeho potvrzení - **ACK**. V realitě to ale může být implementované v OS tak, že se čeká kolem 200 ms, jestli nedojde nějakou vyšší vrstvou k vytvoření odpovědi, kterou by zaslal v ACK paketu a zaslání ACK by tak bylo "zdarma", říká se tomu **Piggybacking** (např. klient zasílá GET dotaz na HTTP server a ten zvládne vygenerovat odpověď do 200 ms a pomocí schrány jí odesílá zpět, OS ji vezme a připojí ji k ACK paketu).



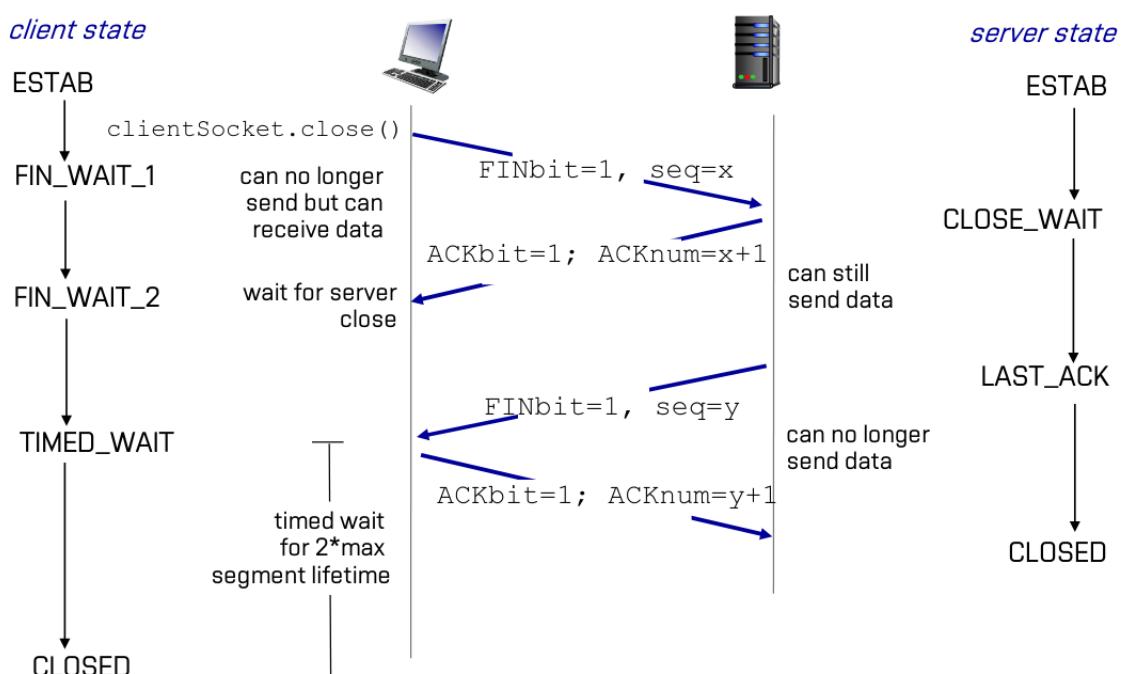
OS si navíc u každého odeslaného paketu ukládá čas jeho odeslání (spouští časovač). V případě, že **vyprší časovač**, znamená to, že se ztratil buď samotný paket s daty nebo paket potvrzující jejich přijetí (nebo ani jedno a jen je pomalá síť). Ve všech případech se **datový paket odesílá znovu**. V případě **duplicitního ACKu** se data také znova odesílají (značí to, že některý paket se ztratil, nebo že došlo k jejich prohození, druhá stanice již přijala paket následující, ale stále jí jeden paket v

řadě chybí, tak posílá ACK předchozího, aby byl opět zaslán ten chybějící paket v řadě). [TCP Duplicate Ack Explained // How to Troubleshoot Them](#)

## Ukončení spojení

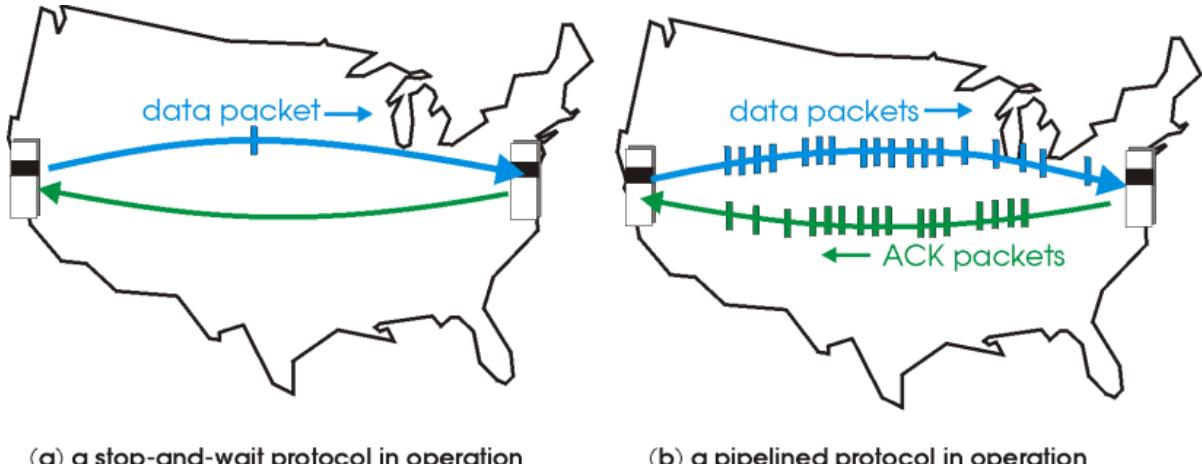
Ukončení spojení probíhá **dvoufázově** (také někdy označováno dvojice 2 way handshakes nebo také 4-way handshake):

- Počítač **A** (řekněme klient) se rozhodne ukončit spojení a zasílá paket s příznakem **FIN** (ten může i nést nějaká data) počítači **B** (řekněme server). Tím počítač **A** indikuje, že už nebude zasílat žádná **nová** data, ale pořád přijímá. (TCP je bezztrátový, může opakovat zaslání paketů, které se ztratily, a potvrzovat datové paketu od **B**). Počítač **A** přechází do stavu **FIN\_WAIT\_1**.
- Počítač **B** přijme **FIN** paket od **A** a odpovídá mu s ACK paketem, čímž přechází do stavu **WAIT\_CLOSE**. Po této operaci počítač **B** ještě může dále počítači **A** posílat rozpracovaná data (nebo případně rovnou s ACK příznakem může také zaslat FIN).
- Počítač **A** přijme **ACK** paket od **B**, přejde do stavu **FIN\_WAIT\_2** a čeká (případně znova posílá ztracené pakety nebo odpovídá na zbytek dat od serveru s ACK, nebo může taky zaslat paket RST, kterým ukončí spojení ihned, ale za cenu možné ztráty dat).
- Počítač **B** už odesal všechna data a pošle paket **FIN** pro potvrzení úplného konce komunikace. Přechází do stavu **LAST\_ACK**.
- Počítač **A** přijme **FIN**, odpovídá s paketem **ACK** a přechází do stavu **TIMED\_WAIT**, kde pouze čeká, jestli počítač **B** nezašle znova **FIN**, to nastane pouze v případě, že původní **ACK** na **FIN** se ztratil.
- Počítač **B** přijme **ACK** a přechází do stavu **CLOSED**.
- Počítači **A** vyprší čekání a přechází také do stavu **CLOSED**.



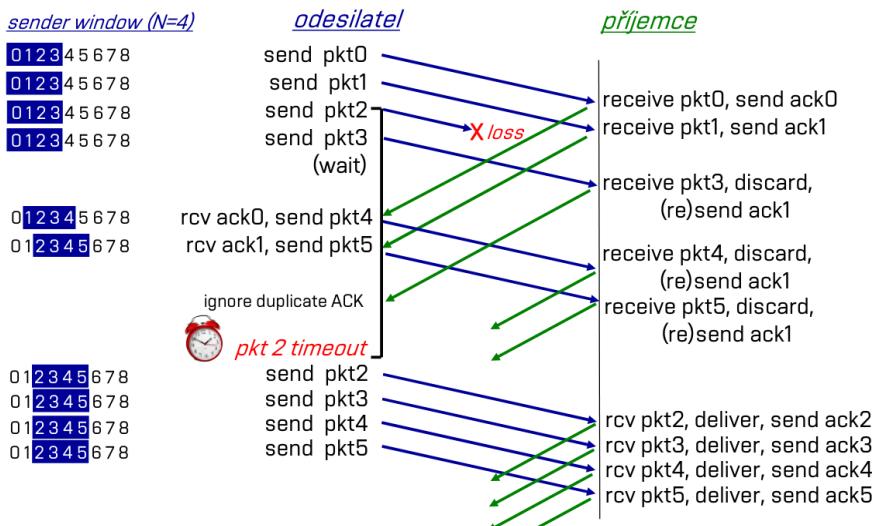
## Pipelining (zřetězené protokoly)

Na rozdíl od přístupu **Stop-and-wait**, který zašle paket, čeká na ACK a až poté zasílá další paket, dochází k odesílání více paketů naráz, aniž bychom čekali na potvrzení každého. Díky tomu je možné **maximalizovat** využití linky a **minimalizovat čekání související s potvrzováním**. Jsou k němu 2 běžné přístupy.



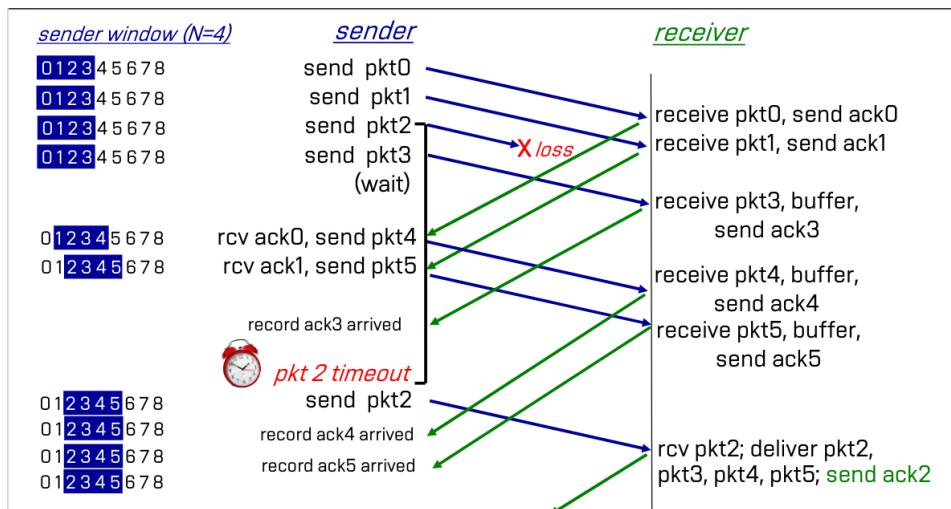
## Go-back-N

- Odesílatel má **sliding window** pro **N** paketů, každý nepotvrzený paket má časovač. V případě vypršení nejstaršího nepotvrzeného časovače nebo obdržení **duplicítního ACK** (příjemce odesílá duplicitní ACK, pokud mu přijde paket ve špatném pořadí a ten **zahazuje**), znovu proběhne zaslání celého aktuálního okna (nepotvrzený paket je na začátku okna).
- Příjemce může potvrdit **jedním ACK** více předchozích paketů (**kumulativní potvrzení**, např. příjemce zašle ACK4, ACK5 a ACK6. ACK4 a ACK5 se ztratí, ACK6 ale přijde, odesílatel tak považuje i ACK4 a ACK5 za potvrzené - odesílatel by jinak neodesílal ACK6, kdyby mu nedorazil paket 4 a 5).  
Případně někdy se ani příjemce nemusí pokoušet odesílat ACK na všechny pakety a rovnou pouze na poslední).
- výhody:** příjemce **nepotřebuje vyrovnávací paměť** pro ukládání paketů, které přišly mimo pořadí.
- nevýhody:** opakovaně se zasírají pakety, které už jednou byly zaslány a mohly úspěšně přijít, ale byly zahozeny.



## Selective Repeat (SACK)

- Odesílatel má **sliding window** pro **N** paketů, stejně jako u Go-Back-N.
- Příjemce selektivně potvrzuje přijatý paket (pro každý odesílá **individuální** potvrzení).
- Příjemce si ukládá pakety, které jsou mimo pořadí do vyrovnávací paměti.
- Odesílatel má **časovač pro každý paket** a znova posílá **jen ty nepotvrzené**, nikoliv celé okno.
- **výhody:** zasílají se opravdu pakety, které nebyly doručeny, snižuje množství paketů v síti a případnou šanci na zahlcení.
- **nevýhody:** příjemce musí implementovat vyrovnávací paměť.
- **Používá se dnes.**



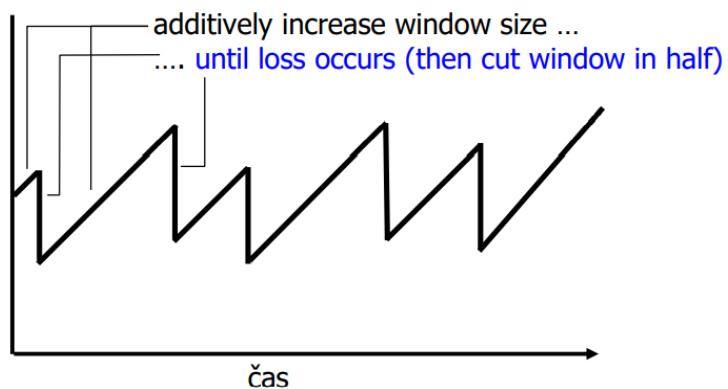
## Řízení zahlcení

Zahlcení sítě nastává, když objem přenášených dat je **větší než přenosová kapacita** linky. Směrovače mají vyrovnávací paměť (frontu), do které ukládají příchozí pakety před zpracováním. V případě, že se tato fronta zaplní (příliš mnoho paketů), začnou se **zahazovat**. Zahlcení má tendenci se zhoršovat (pokusy o znovuzaslání způsobují ještě vyšší objem dat)

TCP má implementované mechanismy řízení zahlcení, které je založeno na **sledování ztrát paketu**. V případě, že se pakety neztrácí TCP **zvyšuje rychlosť** přenosu (**zvětšuje klouzavé okno** a tím snižuje čas čekání na ACK). Pokud dojde ke ztrátě, značí to problém na lince (zahlcení) -> **snižuje rychlosť**. Snahou je co nejrychleji najít ideální velikost klouzavého okna, proto nejprve roste jeho velikost **exponenciálně** (slow start), po ztrátě paketů už jen **lineárně** (congestion avoidance). Běžné jsou 2 algoritmy - Tahoe a Reno.

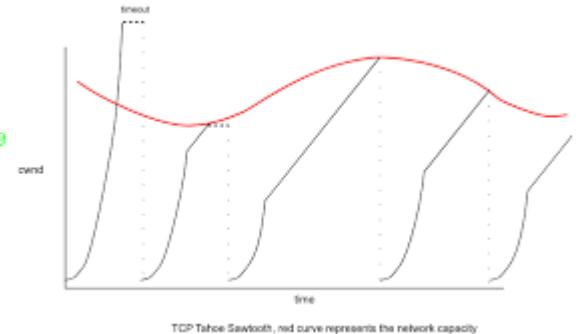
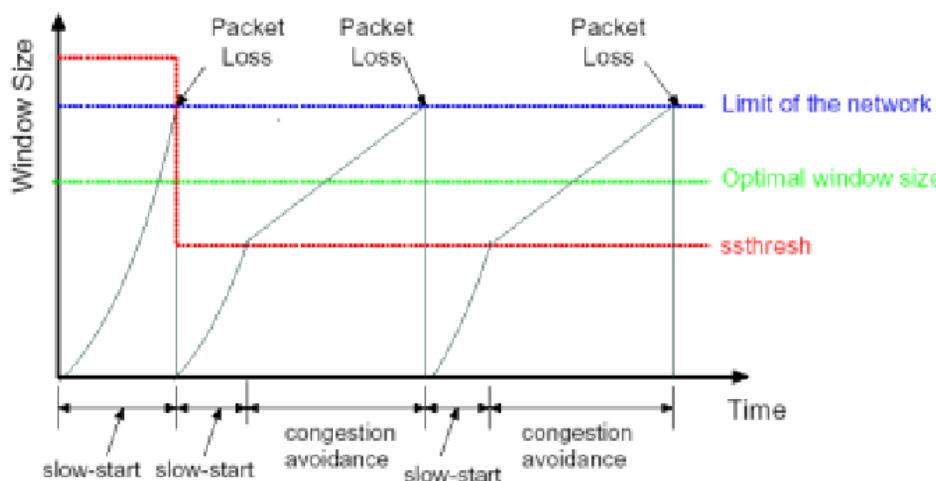
## Additive Increase, Multiplicative Decrease (AIMD)

Odesílatel zvedá rychlosť odesílání lineárně (zvětšuje velikost okna o 1). V případě ztráty sníží rychlosť na polovinu (zmenší okno na polovinu aktuální velikosti).



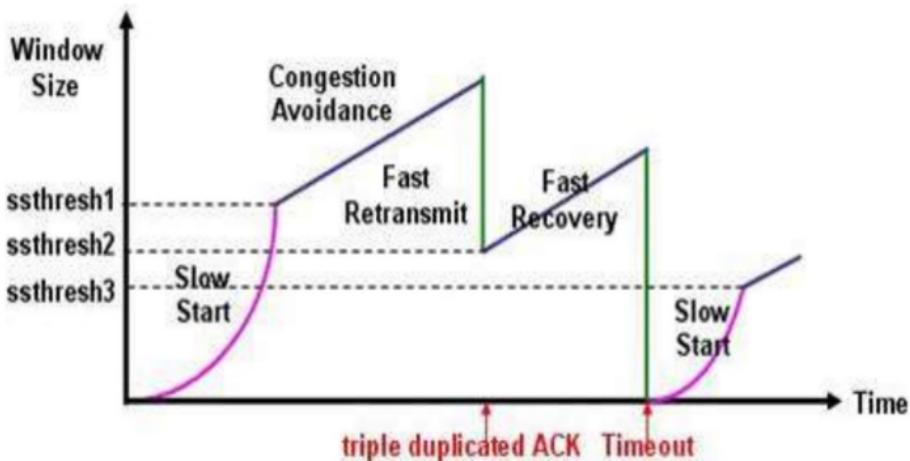
## Tahoe

Tahoe po ztrátě paketu resetuje velikost okna na **0**, pro hledání správné velikosti využívá proměnnou (slow start threshold). V případě, že je velikost okna menší než **ssthresh**, probíhá **slow start** (exponenciální nárůst okna), jinak **congestion avoidance** (lineární nárůst okna). V případě ztráty paketu se **ssthresh** zmenší na **polovinu aktuální velikosti klouzavého okna**. (obr. vlevo počítá ze stabilní rychlosti sítě, reálnější je ale obrázek vpravo, kde propustnost kolísá, lze zde také vidět změnu ssthresh)



## Reno

Reno je vylepšením Tahoe zavádí ještě jeden stav, a to **Fast Recovery** (lineární nárůst velikosti okna). U reno ztráta paketu (obdržení duplicitního ACK (konkrétně 3 duplikáty) nebo ACK mimo pořadí u SACK) nezpůsobí zásadní propad v propustnosti a velikost okna se **zmenší na polovinu**. Až v případě že dojde k **vypršení časovače**, dochází k restartování velikosti okna **na minimum**.



## NewReno

Detekuje **vícenásobnou ztrátu** paketů v jednom okně. Po první ztrátě přechází do **Fast Recovery** a při další ztrátě ve stejném okně nesnižuje velikost na polovinu. Velikost okna snižuje, až je zase ztráta mimo okno, na kterém došlo k první ztrátě, viz

<https://inst.eecs.berkeley.edu/~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf>

## Síťová vrstva

Síťová vrstva L3 se stará o doručení komunikace mezi stroji. Základními protokoly jsou **IPv4** a **IPv6** a dále protokoly s nimi související (**ICMP**, **DHCP**, apod.). Adresace probíhá pomocí IP adres (IPv4 32b, IPv6 128b). Důležitým prvkem této vrstvy je **směrovač** (router), který na základě směrovacích tabulek pakety **směruje síť**.

Síťová vrstva je nespojová (**connection-less**), tj. doručení probíhá na bázi **best-effort**. IP pakety umožňují data rozdělit na segmentu, tak aby je bylo možné odeslat přes síť a na cílové stanici opět spojit (segmentují se kvůli MTU).

Hlavička **IPv4** obsahuje (nemá fixní délku):

- verzi - verze protokolu 0x4,
- Type of Service (ToS), využívá se pro QoS (viz dále),
- délku,
- **zdrojovou a cílovou IP adresu**,
- TTL = time to live (na každém routeru se snižuje jako prevence zacyklení),
- **Offset pro fragmentaci** - udává pozici dat v původním datagramu, který byl segmentován po násobcích 8 byteů → data jsou odesílána jako **násobky 8 B**,
- **číslo protokolu vyšší vrstvy** (TCP nebo UDP).

Formát IP datagramu

| Bajty                      | 0                             | 1               | 2                         | 3                          |
|----------------------------|-------------------------------|-----------------|---------------------------|----------------------------|
| Bajt 0 až 3                | verze                         | IHL             | typ služby                | celková délka              |
| Bajt 4 až 7                | identifikace                  |                 | příznaky (3 bity)         | offset fragmentu (13 bitů) |
| Bajt 8 až 11               | TTL                           | číslo protokolu | kontrolní součet hlavičky |                            |
| Bajt 12 až 15              | zdrojová adresa               |                 |                           |                            |
| Bajt 16 až 19              | cílová adresa                 |                 |                           |                            |
| Bajt 20 až ((IHL × 4) - 1) | rozšířená nepovinná nastavení |                 |                           |                            |
| ...                        | data                          |                 |                           |                            |

## Maximum Transmission Unit (MTU)

MTU je největší jednotka, kterou je médium schopno přenést (na internetu jde obvykle o MTU ethernetu, která je 1500 B), pokud data z vyšší vrstvy přesahují MTU, nastává fragmentace paketů (což je naznačeno v hlavičce pomocí offsetu a provádí ji většinou router). Datagram sestavuje až koncová stanice. Případně router, na kterém by mělo dojít k fragmentaci může zaslat ICMP zprávu Packet Too Big a odesílatel musí provést fragmentování.

Adresování probíhá pomocí 32b IP adresy, která se skládá z **network ID** (adresa sítě) a **host ID** (adresa hosta). IP adresy sítím jsou přidělovány organizací **ICANN**, která přiděluje bloky **nadnárodním registrátorům**, ti je poté delegují **regionálním registrátorům** (RiR). Rozlišujeme **třídní** a **beztřídní** adresování.

### Třídní adresování

Dle třídy IP adresy je jasně dán, kde končí network ID v IP adrese (maska sítě). Třídu rozlišujeme dle prefixu bitů IP adresy (prvních několik 1 bitů po první 0).

Třídy IP adres

| Třída | začátek (bin) | 1. bajt | standardní maska | bitů sítě | bitů stanice | sítí                   | stanic v každé síti       |
|-------|---------------|---------|------------------|-----------|--------------|------------------------|---------------------------|
| A     | 0             | 0–127   | 255.0.0.0        | 8         | 24           | $2^7 = 128$            | $2^{24}-2 = 16\ 777\ 214$ |
| B     | 10            | 128–191 | 255.255.0.0      | 16        | 16           | $2^{14} = 16384$       | $2^{16}-2 = 65\ 534$      |
| C     | 110           | 192–223 | 255.255.255.0    | 24        | 8            | $2^{21} = 2\ 097\ 152$ | $2^8-2 = 254$             |
| D     | 1110          | 224–239 |                  |           |              | multicast              |                           |
| E     | 1111          | 240–255 |                  |           |              | vyhrazeno jako rezerva |                           |

Rozsahy IP adres a masky sítě

| Třída | 1. bajt | minimum   | maximum         | maska podsítě   |
|-------|---------|-----------|-----------------|-----------------|
| A     | 0–127   | 0.0.0.0   | 127.255.255.255 | 255.0.0.0       |
| B     | 128–191 | 128.0.0.0 | 191.255.255.255 | 255.255.0.0     |
| C     | 192–223 | 192.0.0.0 | 223.255.255.255 | 255.255.255.0   |
| D     | 224–239 | 224.0.0.0 | 239.255.255.255 | 255.255.255.255 |
| E     | 240–255 | 240.0.0.0 | 255.255.255.255 | —               |

Rozlišujeme několik speciálních adres. Pokud hostID je **0**, pak adresa je **adresou sítě**. V případě, že hostID je -1 (samé 1 bitově), pak se jedná o **broadcastovou adresu** v dané síti.

### Beztřídní adresování

VLSM = variable length subnet mask. Masku není specifikována třídou, ale je **"dynamická"**, součástí samotné IP adresy. Například **147.229.176.14/23** udává, že maska má 23 jedničkových bitů (255.255.254.0). Díky tomu je možné provádět například subnetting, kdy jednu větší síť rozdělíme na více menších podsítí

(například z jedné /24 sítě můžeme udělat 2 /25 tím, že napevno nastavíme 25. bit zleva na 1, nebo 0).

## Protokol ICMP

Protokol ICMP je podpůrný protokol pro IPv4, slouží pro aktivní diagnostiku sítě.

Důležitými typy zpráv jsou:

- echo (ping),
- host/network/port unreachable - signalizace nedoručení paketu, protože nebyl nalezen příjemce.
- TTL expired - signalizace zahození paketu, protože prošel přes příliš mnoho směrovačů.

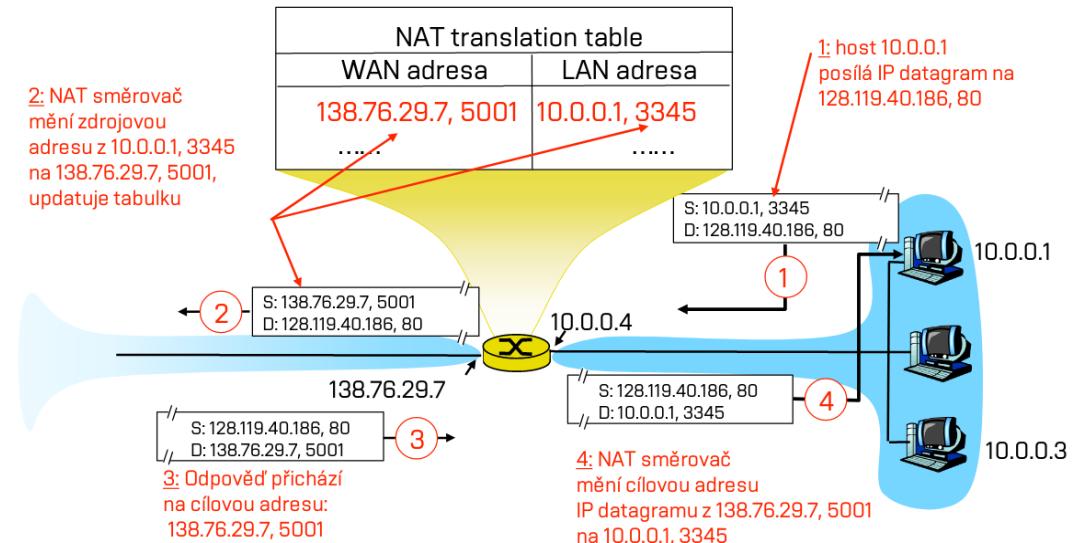
## Protokol DHCP

Slouží k dynamickému přidělování IP adres v síti, klient-server paradigm. Aplikační protokol komunikující pomocí UDP na portu 67 a 68. Komunikace probíhá následovně:

1. Nově připojené zařízení k síti vyšle DHCP **Discover** broadcast zprávu
2. Server(y) mu odpoví se zprávou DHCP **Offer**, kdy nabízí IP adresu, default gateway, DNS server
3. Klient si jednu z nabídek vybere a potvrdí ji opět broadcastově pomocí DHCP **Request**
4. Server to potvrdí pomocí DHCP **Acknowledgement**

## Network Address Translation (NAT)

NAT je mechanismus, který pomáhá „šetřit“ IP adresy. Funguje na tom principu, že v lokální síti mají stroje pouze lokálně platnou IP adresu (např. 192.168.1.113 nebo 10.0.0.0/8), ovšem z lokální sítě veškerý provoz odchází přes router, kde proběhne **překlad na reálnou adresu sítě**. Běžně se používá Port overloading:



## IPv6

Protokol IP verze 6 vznikl s motivací vyřešit nedostatek adres IPv4. Adresy mají 128b, což poskytuje mnohem **větší adresný prostor**. Má nározdíl od IPv4 **pevnou velikost hlavičky** (40 bytů) a **nepodporuje broadcast**. Umožňuje, aby jedno rozhraní mělo více než jednu IPv6 adresu.

IP adresa má zkrácený zápis, tvoříme jej následovně:

- úvodní nuly v čtveřicích hexa číslic vynechat (0000 zkrátit jako 0)
- nejdéle sekvenci nul nahradit za :: (při více stejně dlouhých sekvencích nahrazujeme tu 1.)

Podobnou roli jako ICMP pro IPv4 zastává **ICMPv6** pro IPv6, ovšem dále obsahuje ekvivalenty k **ARP** (překlad IP adres na MAC adresy) a **IGMP** (protokol pro multicast). Adresy jsou v síti přidělovány buď pomocí **DHCPv6**, nebo bezstavově pomocí **Router Advertisement** a **Router Solicitation** zpráv.

## IPSec

Jedná se o **dva protokoly**, které zajišťují **důvěrnost, integritu dat, autentizaci a ochranu proti přehrání** na úrovni síťové vrstvy, tak že vyšší vrstvy jsou již zabezpečené. Využívá se pro vytváření **VPN** (virtual private network).

### Authentication Header (AH)

Zajišťuje **integritu dat** (heš se počítá nad položkami hlavičky, které se během přenosu nemění, a dat), **autentizaci a chrání proti přehrání**. Přenášená data nejsou šifrována. Vždy musí být před ESP, pokud je také použito.

### Encapsulating Security Payload (ESP)

Zapouzdřuje a chrání data IP paketu. Zajišťuje **autentizaci, důvěrnost, integritu dat a chrání proti přehrání**. ESP pracuje ve dvou režimech:

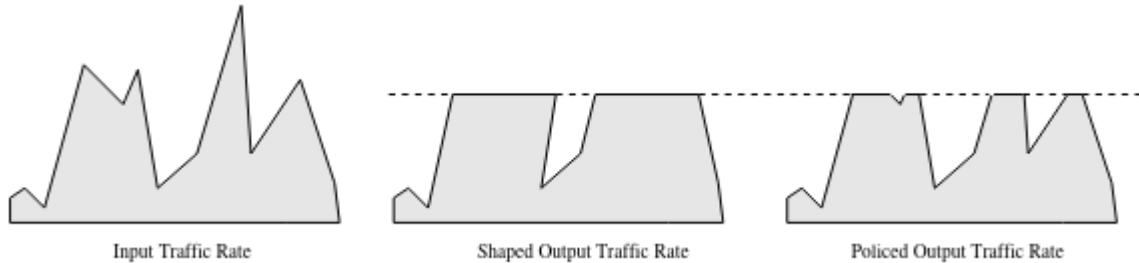
- **transportní režim**: chráněn je pouze payload (tj. vyšší vrstvy),
- **tunelovací režim**: chrání i IP hlavičku tak, že je celý paket zabalen ještě do dalšího IP paketu.

## Quality of Service (QoS)

Jak bylo zmíněno výše, IP vrstva funguje na bázi best-effort delivery, nezajišťuje tedy v základu kvalitu služeb. Směrovače se to do jisté míry snaží kompenzovat a zajistit co nejspolehlivější doručení. Základními přístupy ke zvládání "špiček paketů" na směrovači jsou shaping a policing:

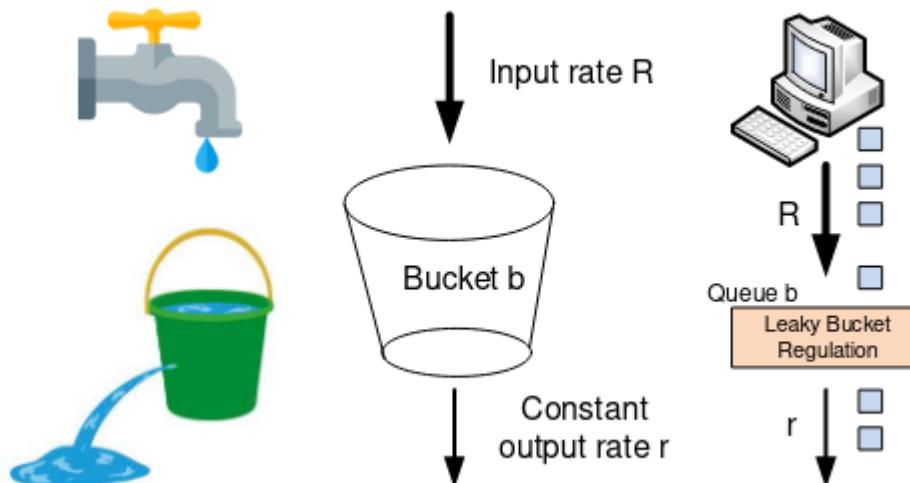
- Při **policingu** se provoz ořezává, nezpracovatelné pakety jsou zahozeny.
  - **výhody**: nezpůsobuje zpoždění a je jednodušší na implementaci,
  - **nevýhody**: způsobuje větší ztráty paketů, což může způsobovat ještě větší vytížení sítě.

- Při **shapingu** se provoz rozkládá pomocí bufferů, směrovač si zapamatuje některé pakety a odesílá je později.
  - **výhody:** méně paketů je zahozeno,
  - **nevýhody:** zanáší zpoždění (to je ale stále lepší než, když je paket zahozen a musí se odeslat znova)



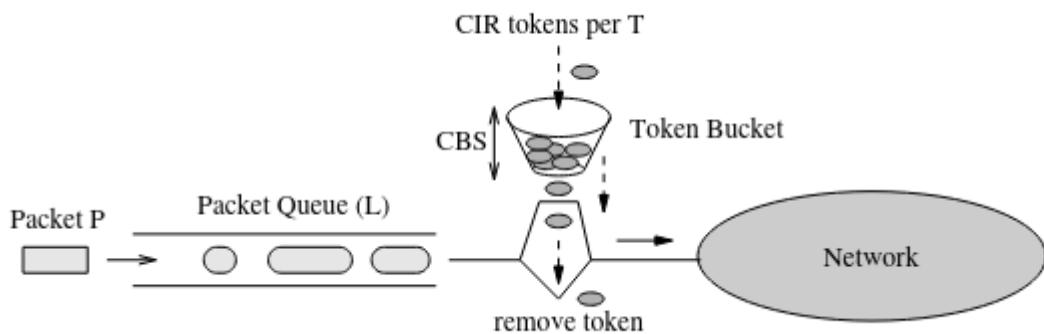
Z pohledu zpracování provozu na směrovači máme několik přístupů:

- obyčejná **FIFO fronta**,
- **prioritní FIFO fronta** (klasifikátor rozdělí provoz do front, data se zpracovávají v pořadí dle priorit front, nízko prioritní ale mohou vyhladovět). Klasifikace může být provedena podle protokolu, ToS v IP hlavičce.
- **Cyklické round robin fronty** - férové rozdělení výstupu mezi fronty, pokud je jedna fronta využívána daleko více než zbylé, zahazuje se pouze z ní, ostatní se stíhají zpracovat. To může být ale nevýhoda, řešení viz dále.
- **Weighted Fair Queues** - podobně jako round robin, ale propustnost front je dána poměrem vah (z front se odebírá v cyklech, v každém cyklu se z každé fronty odebere různý počet paketů/bytů, čím větší priorita, tím více. Nedochází ale k vyhladovění nízko prioritních).
- **Tekoucí vědro** - nezávisle na vstupní rychlosti je na výstupu vždy stejná rychlosť, v případě přetečení dojde k zahzení - způsobuje **shaping**, pokud má vědro větší kapacitu než 0 (existuje paměť - FIFO), jinak se jedná o **policing**.



- **Zásobník žetonů** - do zásobníku žetonů se sypou s **konstantní** rychlosťí žetony (až do určitého omezeného počtu). Jeden žeton představuje určité

množství dat, které lze přenést v **bytech**. Pokud je v zásobníku dostatek žetonů pro přeposlání paketu o určité délce, je **přeposlán** a žetony jsou **odebrány**. Pokud je zásobník plný, lze takhle odeslat v krátkém čase velké množství dat (burst), umí se tedy vyrovnat s krátkodobou špičkou. Poté se ale zásobník **vyprázdní** a musí se čekat, dokud se zase nenaplní žetony. V tento okamžik jsou příchozí paketu **bud' zahazovány, nebo musí čekat ve frontě**.



## IntServ

Integrované služby implementují QoS formou **rezervace zdrojů** (před každým přenosem nebo při změně cesty) pomocí protokolu **RSVP** (stanice zažádá o spojení v určité kvalitě: **garantované služby** - garanteuje vyhrazení nějakého pásma, **kontrolovaná zátěž** - kvalita provozu blížící se nezatíženému prvku, nebo **best-effort**). Protokol RSVP prakticky nelze použít na globálním internetu. Zajištění požadovaného přenosového pásma nemusí být na zatížené síti vůbec možné.

## Diffserv

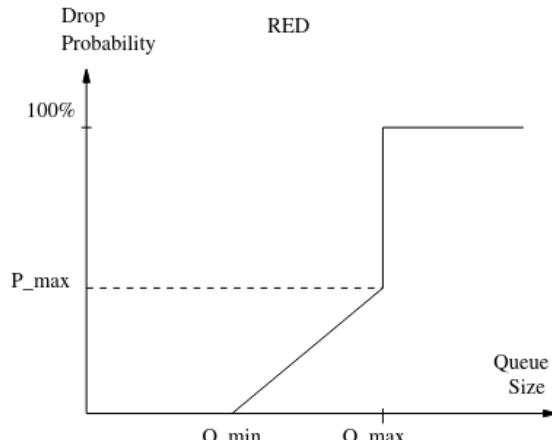
Diferencované služby fungují na principu **značení provozu** na směrovači a následné **prioritizace** dle značek. Značení probíhá do IP hlavičky, pole ToS (type of service). Podle **kategorie** poté **rozlišujeme**, zda má být paket **přednostně** zpracován nebo **pouze best-effort** (rozdělení do kategorií provádí hraniční prvky sítě, tj. směrovače). Například pro **VoIP** by byla nastaveno **přednostní doporučení** s určitou propustností, protože se jedná o časově citlivou aplikaci.

## Řízení zahlcení na L3 (RED, WRED)

Jak bylo vysvětleno výše, **TCP řeší řízení zahlcení**, problémem ovšem je, že se jedná o L4 protokol a směrovače o něm tedy neví. V případě, že komunikují stanice zároveň **UDP** a **TCP** a linka není příliš rychlá, může lehce dojít k **vyhladovění TCP**, protože UDP aplikace bude posílat naplně pakety bez omezení, zatímco TCP budou zahazovány pakety a bude se snižovat velikost klouzavého okna, proto bude zpomalovat.

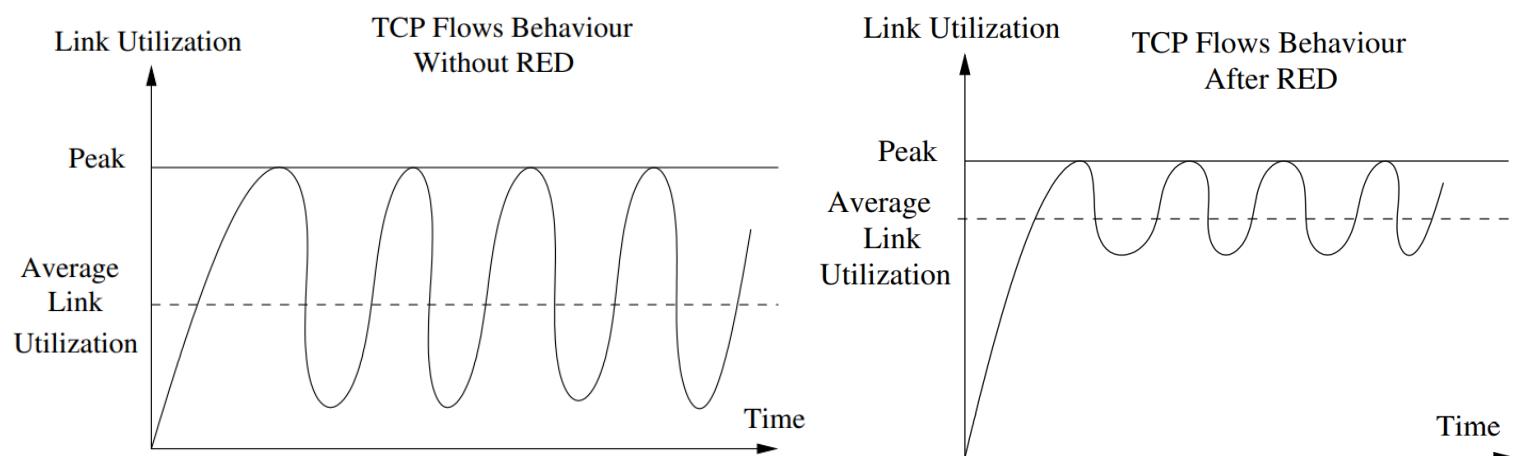
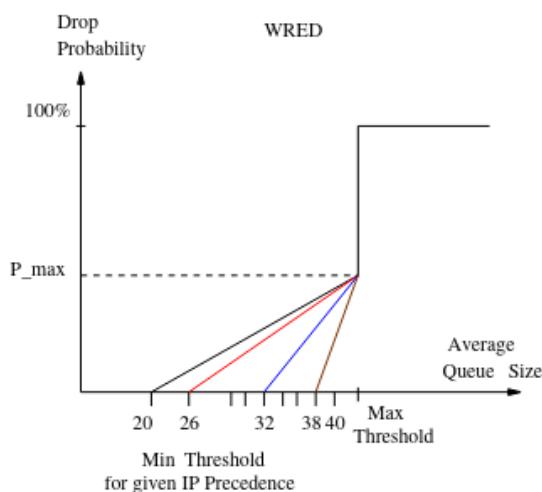
K řešení tohoto problému se na směrovačích používá algoritmus **RED = Random Early Detection**. Od určité míry zahlcení začne směrovač **preventivně zahazovat** pakety s určitou pravděpodobností. **Qmin** je zaplnění fronty, od kterého zahazujeme,

**Q<sub>max</sub>** maximální zaplnění fronty, **Q<sub>avg</sub>** současné zaplnění. Před Q<sub>min</sub> nedochází k zahazování paketů, od **Q<sub>min</sub>** roste pravděpodobnost zahzení lineárně až u **Q<sub>max</sub>** je 100% a je zahazován každý paket. Při **zvětšení Q<sub>min</sub>** se pakety začnou zahazovat později, ale hrozí úplné **zaplnění fronty**. Při **nízkém Q<sub>min</sub>** se pakety zahazují dřívěji, ale možná někdy **zbytečně** (třeba by se fronta nenaplnila).



$$\bullet P_a = P_{\max} \frac{Q_{\text{avg}} - Q_{\min}}{Q_{\max} - Q_{\min}},$$

Variantou RED je potom vážený RED (Weighted RED, WRED), kdy Q<sub>min</sub> je definováno zvlášť pro jednotlivé druhy provozu (například dle pole **ToS**), pakety s **nižší prioritou** mají **nižší Q<sub>min</sub>** a začnou být **dříve zahazovány**. Předpokládá se, že to **uvolní síť** dostatečně, aby **nemuselo** dojít k zahazování prioritnějších paketů .



# 44. Směrování a zabezpečení přenosů v počítačových sítích (algoritmy Link-State, Distance-Vector, šifrování, autentizace a integrita dat).

Směrování (routing) je **určování cest paketů** v mezi různými počítačovými sítěmi (které jsou ale na jednom internetu). Směrování zajišťují **směrovače** (routery - pomocí **IP adres** na **síťové vrstvě** - L3). Úkolem směrování je doručit paket adresátovi (který je v jiné síti), pokud možno co **nejefektivnější** cestou (tj. co nejrychleji).

- **1. poznámka:** v rámci **jedné** sítě se **nesměruje ale přepíná**, přepínání provádí přepínač (**switch**) a děje se na **linkové vrstvě** (L2). Přepínají se **rámce**.
- **2. poznámka:**
  - PDU na **linkové vrstvě** je **rámeček** (frame),
  - PDU na **síťové vrstvě** je **paket** (packet),
  - PDU na **transportní vrstvě** je:
    - **tok (stream)** pro TCP (viz SOCK\_STREAM),
    - **datagram** pro UDP (viz SOCK\_DGRAM),

Pojem **paket** je ale často také považován za data přijatá přes **TCP** a pojmenování **datagram** jako data přijatá přes **UDP** a případně jinými způsoby s možnou ztrátou.

## Třídní směrování (Classful routing)

|         | octet   | octet   | octet   | octet | Class        | High order bits | Start ip address | End ip address  |
|---------|---------|---------|---------|-------|--------------|-----------------|------------------|-----------------|
| Class A | Network | Host    | Host    | Host  | A            | 0               | 0.0.0.0          | 127.255.255.255 |
| Class B | Network | Network | Host    | Host  | B            | 10              | 128.0.0.0        | 191.255.255.255 |
| Class C | Network | Network | Network | Host  | C            | 110             | 192.0.0.0        | 223.255.255.255 |
|         |         |         |         |       | Multicast    | 1110            | 224.0.0.0        | 239.255.255.255 |
|         |         |         |         |       | Experimental | 1111            | 240.0.0.0        | 255.255.255.255 |

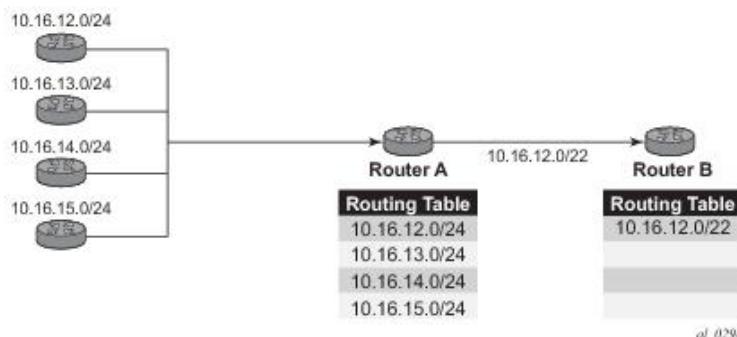
Položky ve **směrovací tabulce** (routing table) tvoří **pouze síťové adresy** o délce 8, 16 a 24 bitů, viz obrázek vlevo. Délka vychází z **masek sítí** jednotlivých tříd. Třídu lze poznat dle nejvyšších bitů, viz obrázek vpravo. V routing table je  $2^7 + 2^{14} + 2^{21} = 2\ 113\ 664$  adres (ne všechny jsou použitelné, např. broadcastové), ale i tak jich je mnoho.

## Beztřídní směrování (Classless routing)

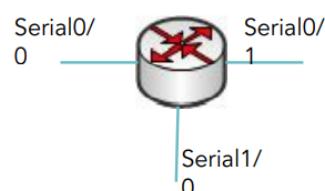
Položky ve směrovací tabulce tvoří **sítové adresy a masky sítě**. Pro každou adresu se na základě **masky sítě** nejdříve určena **sítová adresa**, až pak je provedeno směrování  $158.193.138.40 \& 255.255.255.224 = 158.193.138.32$ .

|          |          |          |          |
|----------|----------|----------|----------|
| 10011110 | 11000001 | 10001010 | 00101000 |
| AND      |          |          |          |
| 11111111 | 11111111 | 11111111 | 11100000 |
| =        |          |          |          |
| 10011110 | 11000001 | 10001010 | 00100000 |

Pro to, aby se zmenšily směrovací tabulky, **agregují** směrovače adresy sítí, pokud se pro jednu cestu **objeví více adres s rozdílnými LSB** a eliminuje se tak potřeba je rozlišovat.



Na obrázku mohl směrovač B agregovat všechny 4 IP adresy do jedné a snížit tak masku sítě na 22, protože IP adresy obsahují na bitu 9 a 8 všechny kombinace (**00, 01, 10 i 11**). To ale znamená, že když na směrovač přichází teď adresa s maskou 24, není v tabulce. Výběr cesty, kam bude takový paket poslán, pak probíhá na základě vyhledání **nejdelší shody prefixu** - adresy sítě (**Longest Prefix Match**).



Cílová adresa: 201.10.6.17

|                                             |
|---------------------------------------------|
| 1100 1001.0000 1010.0000 0110.0001 0001     |
| 0000 0100.xxxx xxxx.xxxx xxxx.xxxx xxxx /8  |
| 0000 0100.0101 0011.1xxx xxxx.xxxx xxxx /17 |
| 1100 1001.0000 1010.0000 0xxx.xxxx xxxx /21 |
| 1100 1001.0000 1010.0000 011x.xxxx xxxx /23 |
| 0111 1110.1111 1111.0110 0111.xxxx xxxx /24 |

|                  |           |
|------------------|-----------|
| 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 |

## Link-State protokoly

Jedná se např. o protokoly **Open Shortest Path First** (OSPF) a **Intermediate System to Intermediate System** (IS-IS). Link state směrování je založen na tom, že každý **směrovač** (router) zná **nejlepší cestu** (cestu s nejnižší cenou dle nějakého hodnocení - asi rychlosť přenosu) do **všech sítí**. Tzn. že každý směrovač **zná celou topologii sítě**. Dosáhneme toho:

- **Flooding:** Každý směrovač **broadcastuje** (posílá všem) Link-State paket obsahující **ceny** cest k jeho **sousedním** směrovačů. Každý směrovač si **ukládá** informace (ceny cest) o směrování od ostatních směrovačů. Pokud směrovač přijde Link-State paket, který **už jednou dostal**, již jej dále **nepřepraví** (už jej jednou přepravil). **Problémem** může být **ztráta** Link-State paketů, některý směrovač tak nemusí mít kompletní přehled o síti. Řeší se pomocí **ACK** a případného přepravování.
- Flooding je nutné provádět při **přidání** nového směrovače, při **odebrání** a případně také **periodicky**, pokud se ceny cest mohou měnit.

Jakmile zná každý směrovač celou topologii sítě, má sestavený **vážený** (ohodnocený) **graf**. Nyní musí každý směrovač vypočítat **nejlepší cestu do všech sítí** (jiných směrovačů), používá se na to **Dijkstrův algoritmus** pro hledání nejkratší cesty/cest v ohodnoceném grafu, viz okruh 25. Následně každý router ví, kam má paket směrovat na základě jeho cílové destinace. (Implementovat pak lze tak, že k jednotlivým výstupům si přiřadí všechny IP adresy, které na ten výstup vedou a provádí porovnávání - pomocí asociativní paměti (adresovatelné obsahem). Pokud navíc jsou si IP adresy podobné, může při porovnávání ignorovat některé byty).

▶ [Link-State Routing Algorithm - IP Network Control Plane | Computer Networks ...](#)

## Distance-Vector

▶ [Distance-Vector Routing Algorithm - IP Network Layer | Computer Networks Ep...](#)

Jedná se o protokoly **Enhanced Interior Gateway Routing Protocol** (EIGRP) a **Routing Information Protocol** (RIP). Každý uzel (směrovač) komunikuje **pouze se svými sousedy** (směrovači, se kterými má přímé spojení). Směrovač tak na začátku algoritmu zná cestu pouze k sousedním směrovačům (do vzdálenosti 1) a zapíše ceny těchto cest do své směrovací tabulky, do zbytku sítě nevidí (na příkladech se to vyjadřuje nekonečným ohodnocením, v realitě ani neví, že tam něco existuje). Algoritmus je **iterativní**.

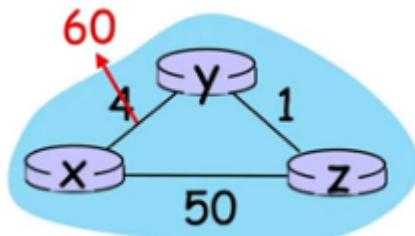
- **1. iterace:** směrovač (a všechny ostatní také) dostane směrovací tabulky od **sousedních** směrovačů, vidí tak už do **vzdálenosti 2**. Na základě získaných informací **aktualizuje svou směrovací tabulku** (přidá **nově zviditelněné** směrovače a pokud se dokáže dostat přes jiný směrovač **rychleji** k uzlu, do kterého již cestu zná, aktualizuje i tuto cestu, stejně tak při zdražení cesty).
- **2. iterace:** směrovač opět dostane směrovací tabulky od svých **sousedních** směrovačů, ty ale **mohou být jiné**, než při 1. iteraci (přibyly nově zviditelné

směrovače, zlevnili se některé cesty, zdražily se některé cesty). Směrovač opět aktualizuje svojí směrovací tabulkou, už vidí do **vzdálenosti 3**.

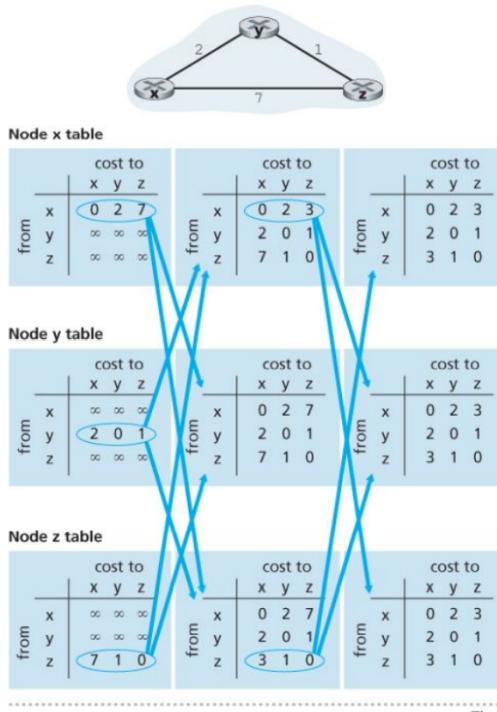
- ...
- **n. iterace:** směrovač dostane směrovací tabulky od svých **sousedů**, už ale **vidí do celé sítě** a nově získané informace **nezpůsobí žádnou změnu** v jeho tabulce. Dále už nepřeposílá svojí tabulku, protože se nezměnila. Takhle postupně přestanou odesílat své tabulky všechny směrovače, protože každý už má **ideální směrovací tabulku**.

Z výše uvedeného postupu plyne, že bude třeba **minimálně tolik iterací**, jak je dlouhá **nejdelší nejkratší cesta** (nejkratší znamená, že mezi dvěma uzly nejde již najít nejkratší cestu, nejdelší znamená, že je to pak nejdelší cesta z těchto). Reakce na změny:

- **snížení ceny cesty:** opět si směrovače iterativně vyměňují směrovací tabulky, dokud se směrování neustálí. Probíhá rychle - dobrá zpráva se šíří rychle.
- **zvýšení cesty:** iterativní šíření. Může ale dojít k zajímavému jevu, kdy dva sousední směrovače využívají cesty **přes sebe navzájem** a ty se tak v každé iteraci **zvětšují o 1**, dokud cena nestoupne natolik, že je použita jiná cesta. Např. na obrázku si bude **y** a **z** vyměňovat tabulky, dokud cena **neprekročí 50**, pak teprve použijí cestu přes **x**.



Distance-Vector je **distribuovaný algoritmus**, je problematický, pokud nějaký router **lže** nebo **špatně počítá cenu** (hlásí menší než doopravdy). Pak přes něj může být směrován provoz, který ale bude **pomalý**.



## Srovnání Link-State a Distance-Vector

### message complexity

LS:  $n$  routers,  $O(n^2)$  messages sent

DV: exchange between neighbors;  
convergence time varies

### speed of convergence

LS:  $O(n^2)$  algorithm,  $O(n^2)$  messages

- may have oscillations

DV: convergence time varies

- may have routing loops
- count-to-infinity problem

robustness: what happens if router malfunctions, or is compromised?

LS:

- router can advertise incorrect *link* cost
- each router computes only its *own* table

DV:

- DV router can advertise incorrect *path* cost (“I have a *really* low cost path to everywhere”): black-holing
- each router’s table used by others: error propagate thru network

## Zabezpečení na síti

Objektivem zabezpečením sítě chceme zajistit, aby při komunikaci dvou systémů nedocházelo k:

- **falešnému vydávání se** za jeden ze systémů,
- **odposlech** jejich komunikace,
- **zásah do komunikace** pozměněním zasílaných zpráv,
- **opakované zaslání** zachycené zprávy, která již byla předtím doručena.

### Autentizace (Authentication)

Autentizace zajišťuje, že uživatel je tím, za koho se vydává. Autentizace obvykle probíhá na základě poskytnutí nějaké **tajné informace**. Tuto informaci může znát nebo k ní mít přístup pouze uživatel, který má být autentizován. Jde například o **uživatelské jméno a heslo, biometrické údaje, bezpečnostní žeton** atd.

Autentizaci lze také zajistit **digitálním podpisem**.

### Důvěrnost (Confidentiality)

Důvěrnost zajišťuje, že obsah zpráv **nemůže číst neoprávněná osoba**. To lze zajistit buď odepřením přístupu ke zprávě, což není ale na internetu obecně možné, nebo **šifrováním** (utajením) zprávy, což sice útočníkovi umožní číst zasílaná data, ale neumožní mu získat informaci, kterou obsahují. Data pro něj **bez dešifrování budou bezcenná**. Nauka o šifrování se nazývá **kryptografie**, v informatice hovoříme především o symetrické a asymetrické kryptografii.

## Symetrická kryptografie

**Symetrická** kryptografie používá jeden **tajný klíč**, kterým odesílatel **šifruje zprávu** a příjemce ji **tím samým klíčem dešifruje**. Nejpoužívanější algoritmy pro symetrické šifrování jsou v současné době například **AES**, **ChaCha20** či **IDEA**.

Známější algoritmy jako DES, 3DES a Blowfish nejsou dnes již považované za bezpečné, byla u nich objevena bezpečnostní rizika nebo způsob, jak lze šifru překonat hrubou silou v dostatečně krátkém čase. Síla šifry však závisí především na kvalitě klíče a ta, vzhledem k tomu, že většina **symetrických klíčů** je generována jako **pseudonáhodná posloupnost bitů**, je dána jeho délkou. Za dostatečně bezpečné jsou dnes považovány klíče o délce aspoň **128 bitů**, u kterých není u běžných počítačů reálné jejich prolomení hrubou silou v přijatelně krátkém časovém úseku. Pro zajištění bezpečnosti i do budoucna je ale vhodné používat klíče o délce **256 bitů**.

## Asymetrická kryptografie

U **asymetrické kryptografie** se používá **dvojice klíčů, soukromý a veřejný**. Pro tyto klíče platí, že jsou **matematicky svázány**. Odesílatel **šifruje zprávu veřejným klíčem**, který může znát **kdokoliv**, buď je veřejně dostupný nebo jej odesílatel od příjemce získá na počátku komunikace, která není ale šifrovaná. Pro dešifrování zprávy je potřeba **soukromý klíč**, ten zná jen a **pouze příjemce**. Takto je zajištěno šifrování v jednom směru komunikace, pro opačný směr se postupuje obdobně. Obvykle si tedy na počátku komunikace respondenti vymění veřejné klíče a až poté probíhá komunikace šifrovaně. Pro asymetrickou kryptografií se používají například algoritmy **RSA** a **EIGamal**, které jsou založené na složitosti výpočtu **diskrétního logaritmu**, či algoritmus **ECC**, který pracuje na bázi **eliptických křivek**.

Stejně jako u symetrické kryptografie závisí **bezpečnost** šifry především na **délce klíče**. Soukromý klíč lze ale na základě jeho **matematické spojitosti** s veřejným klíčem rekonstruovat rychleji než při rekonstrukci hrubou silou. Proto je nutné používat delší klíče než u symetrické kryptografie. Konkrétně pro algoritmus RSA je bezpečnost klíče o délce 1024 bitů ekvivalentní s bezpečností symetrického klíče o délce 80 bitů, 2048 bitový RSA klíč odpovídá symetrickému klíči o délce 112 bitů a pro zajištění bezpečnosti odpovídající symetrickému klíči o délce 256 bitů je třeba použít RSA klíč o délce 15360 bitů, což už je prakticky nepoužitelná délka.

Algoritmus ECC je v tomto směru lepší a pro zajištění bezpečnosti na úrovni 256 bitového symetrického klíče stačí pouze 521 bitový ECC klíč

## Integrita (Data Integrity)

Integritou dat ve smyslu zabezpečení komunikace je myšleno zajištění toho, že data po cestě od odesílatele k příjemci nikdo nezmění. Při komunikaci na internetu je toto řešeno vygenerováním **charakteristiky zprávy** (message digest) o fixní délce, připojením této charakteristiky ke zprávě a jejím odesláním. Příjemce nejdříve oddělí charakteristiku zprávy a získá tak původní zprávu, stejným algoritmem jako

odesílatel vygeneruje její charakteristiku a porovná ji s obdrženou charakteristikou. Pokud jsou stejné, nikdo po cestě zprávu nezměnil.

Aby výše uvedený princip fungoval, je nutné pro generování charakteristiky použít **kryptografickou hašovací funkci**. Tato funkce zajišťuje, že je velmi náročné vygenerovat zprávu, která by **měla požadovanou charakteristiku**, stejně tak náročné je **najít dvě rozdílné zprávy se stejnou charakteristikou** a na základě charakteristiky není možné **zprávu zrekonstruovat**. Dále taková funkce musí při generování charakteristiky zprávy **zohlednit všechny její bity** a i při změně jediného bitu musí být nově vygenerovaná charakteristika od té původní natolik **odlišná**, aby se toho nedalo zneužít. Nejpoužívanější bezpečné kryptografické funkce jsou například **MD6**, **SHA-3** či **BLAKE2**.

I po splnění všech zmíněných předpokladů není stále zajištěno, že zprávu po cestě nikdo nezměnil, a to z jednoho prostého důvodu, útočník může změnit zprávu a současně i její charakteristiku. Aby k tomu nedošlo, je nutné použít šifrování, není však nutné šifrovat celou zprávu, stačí **zašifrovat charakteristiku**.

## Neodmítnutelnost (Nonrepudiation)

Neodmítnutelnost zajišťuje, že uživatel nemůže popřít provedení dané akce a poskytuje mu ujištění, že jeho zpráva s požadavkem na provedení této akce byla doručena. K zajištění neodmítnutelnosti se obvykle používá **digitální podpis**.

Nejznámějším algoritmem, který implementuje digitální podpis je **DSA**.

Digitální podpis je založen na **asymetrické kryptografii**, ale na rozdíl od zajištění důvěrnosti se zde pro **šifrování používá privátní klíč**. Privátním klíčem se nešifruje zpráva, pouze její charakteristika, která se získá stejným způsobem jako u zajištění integrity dat a má i stejnou funkci, a to zajistit, že zprávu nikdo nezměnil. Pro ověření odesílatele si příjemce obstará od odesílatele **veřejný klíč**, dešifruje zašifrovanou charakteristiku, vygeneruje charakteristiku příchozí zprávy a porovná je. Pokud jsou stejné, nedošlo po cestě ke změně zprávy, ale především odesílatel je opravdu ten, který zprávu podepsal, protože **jen on vlastní privátní klíč**, kterým byla zašifrována charakteristika.

Samozřejmě to není tak jednoduché, vygenerovat pář asymetrických klíčů si může kdokoliv a poté se **podvodně** vydávat za někoho, kým není. Aby k tomu nedocházelo, musí mít odesílatel k veřejnému klíči **certifikát**, který ověřuje jeho identitu. Certifikáty vydávají **certifikační autority**, které při tomto procesu ověří žadatelovu identitu a **podepíší jeho veřejný klíč** svým soukromým. Opět by se dalo namítat, že potenciální útočník si může vytvořit i vlastní certifikační autoritu a **certifikovat si falešné digitální podpisy**. Tomu se dá předejít jen tak, že příjemce si ověří nejen identitu vlastníka veřejného klíče, ale i **pravost certifikační autority**. Ta se zajišťuje tak zvaným **řetězcem důvěry**, který funguje na principu, že nadřazené certifikační autority podepisují klíče svým podřazeným certifikačním autoritám.

Tento řetězec končí u **kořenové certifikační autority**, která již nemůže být dále ověřena, ale vzhledem k tomu, že je pouze jedna, nedá se o její pravost pochybovat.

## Ochrana proti přehrání (replay attack)

Přehrání je druh útoku, u kterého útočník zachytí jinak **validní zabezpečenou komunikaci** a snaží se ji použít **opakovat** nebo ji pouze pozdržet. Jako ochrana proti tomuto typu útoku se nejčastěji používají **časová razítka**. Podle časového razítka se určí stáří zprávy a zprávy starší než stanovená hodnota jsou ignorovány. K tomu, aby časová razítka fungovala, musí být zajištěno, že příjemce i odesíatel mají **synchronizovaný čas** a že maximální stáří zprávy zhruba odpovídá době jejího doručení. U synchronizace času mohou být komplikací rozdílná časové pásmo, ve kterých se respondenti nacházejí. Proto se obvykle pro časová razítka používá koordinovaný světový čas. Další problém synchronizace času může být v tom, že každý respondent používá jiný zdroj hodin, který je nepřesný a k tomu nepřesný s opačnou fází, nebo že útočník zaútočí přímo při synchronizaci času a podvrhne nesprávný čas. U stanovení maximálního stáří zprávy je problematická proměnlivá doba jejího doručení, která je způsobena především změnami v zatížení internetové sítě. Z toho je patrné, že pokud bude útočník dostatečně rychlý, může se mu povést i přes použití časového razítka zprávu replikovat. Proto může být časové razítko ještě doplněno například o **sekvenční číslo** zprávy nebo o nějakou jednorázovou informaci. Jak sekvenční číslo tak jednorázová informace vyžadují **uchování stavu**, což může být problematické, protože komunikace na internetu bývá často nestavová.

## Unicast, Broadcast, Anycast a Multicast

Pro jistotu, protože to nikde nezaznělo...

### Unicast

Jedná se o komunikace jednoho odesílatele s jedním příjemcem (**one-to-one**). Komunikace je řešena standardním směrováním a jedná se o **nejpoužívanější** druh komunikace. Typická unicast komunikace je např. **HTTP**, **SMTP**, ale i **Video on Demand** (YouTube).

### Broadcast

Jde o komunikaci od jednoho bodu ke všem ostatním bodům v síti (**one-to-all**). Broadcastové IPv4 adresy mají všechny **bity adresy hosta** (host address) **rovny 1**. Pozor **IPv6 nemá broadcastové adresy**, vše je řešeno přes multicast. Na linkové vrstvě je MAC adresa pro broadcast **ff:ff:ff:ff:ff:ff**. Broadcast se používá např. u **ARP** nebo **DHCP**. Lze zasílat pouze přes **UDP**.

## Multicast

Multicast je druh komunikace od jednoho vysílajícího k více příjemcům, kteří se o to přihlásili (**one-to-many**). Odesílatel ale zasílá pouze **jeden datagram**, o jeho duplikaci a šíření se **starají síťové prvky**. Přihlašování do skupin je řešeno:

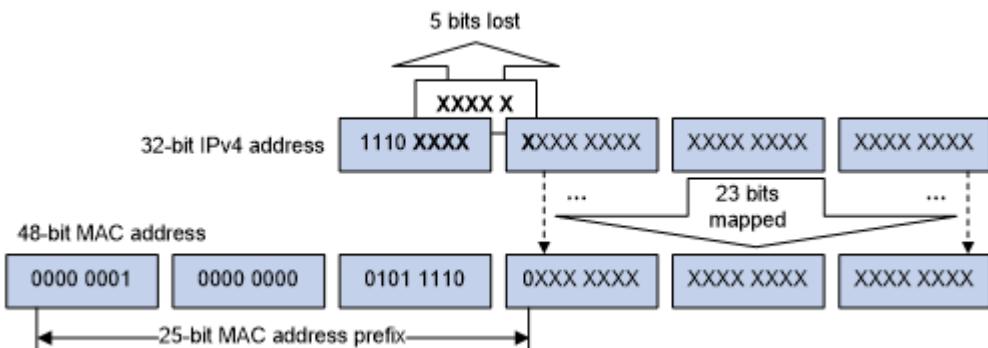
- **IPv4** pomocí protokolu **IGMP** a zpráv **Membership Report** (přihlášení se) a **Leave Group** (odhlášení se),
- **IPv6** pomocí protokolu **MLD** a zpráv **Multicast Listener Query** (test, jestli někdo ještě naslouchá), **Multicast Listener Report** (přihlášení do skupiny) a **Multicast Listener Done** (odhlášení).

Existují vyhrazené IP adresy (L3) pro multicast:

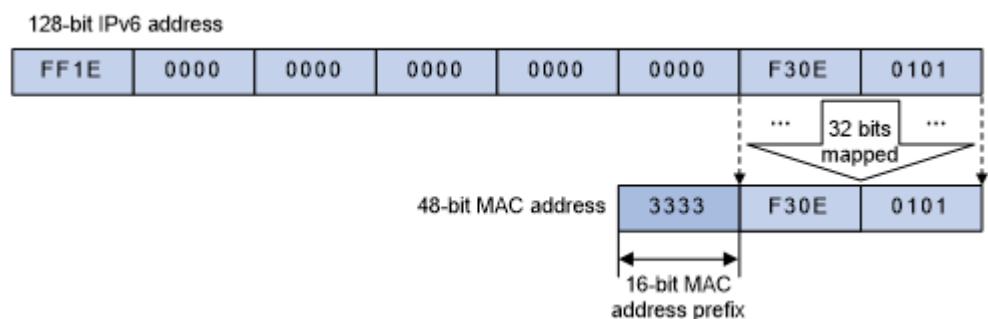
- **IPv4** blok adres D, tj. **224.0.0.0** až **239.255.255.255**,
- **IPv6** adresy s prefixem **FF00::/8**.

Multicast na L2 je řešen **mapováním IP adres** na **MAC adresy**, mapování ale není přesné:

- **IPv4** používá pro mapování 23 bitů MAC adresy, 5 bitů se nemapuje ( $32-23-4$ (blok D) = 5, tj.  $2^5 = 32$  multicast IP adres je mapováno na stejnou MAC adresu), jedná se o adresy **01:00:5E:00:00:00** až **01:00:5E:7F:FF:FF**.



- **IPv6** používá pro mapování 32 bitů MAC adresy a jedná se o adresy **33:33:00:00:00:00** až **33:33:FF:FF:FF:FF** (tedy **33:33:xx:xx:xx:xx**).



Multicast se používá pro **streamování televize a rádia** (IPTV, IP Radio) a zejména u směrovacích protokolů **RIP**, **EIGRP**, **OSPF**, **DVRMP**, ... U IPv6 nahrazuje multicast broadcast, např. při konfiguraci IP adresy pomocí **NDP**.

## Anycast

Na síti existuje **více serverů**, které poskytují **stejnou odpověď** (CDN). Typicky je anycast provoz směřován k nejbližšímu serveru, ale v případě poruchy může být použit jiný (nebo klient si případně může vybrat jaký). Anycast slouží také jako ochrana vůči DoS útokům a umožňuje **rozložení zátěže**.