

Státnicové otázky

Programové, výpočetní a informační systémy

<https://www.facebook.com/groups/344854092286840/?fref=ts>

Programovací jazyky

Datové a řídicí struktury programovacích jazyků, datové typy. Kompilace, interpretace. Příklady programovacích jazyků různých kategorií. Znalost konkrétního programovacího jazyka (podle vlastní volby).

Datové typy

Existují dva typy – jednoduché a složené.

Jednoduché datové typy

Jsou většinou zabudovány přímo do jazyka. Dělí se na ordinální a neordinální.

Ordinální datové typy

Pro každý prvek je definován předchůdce a následovník. Z posledního prvku ve většině jazyků dochází k přetečení na první.

- **Logická hodnota** – boolean, nabývá dvou hodnot (true/false, yes/no, 1/0 apod.)
- **Celé číslo** – integer, kromě matematických operací na nich lze použít i bitové operace, ty se provádí nad bitovou reprezentací čísla. Z tohoto typu lze většinou získat přirozené číslo, např. v C použitím klíčového slova unsigned.
- **Znak** – char, jeden znak (např. 'a', '@', '5'). Interně je uložen jako celé číslo. Většinou se používají znakové sady ASCII nebo Unicode.
- **Výčtový typ** – enum, lze jej považovat za variantu celočíselného typu. Jedná se o celé číslo, které může nabývat pouze určitého počtu hodnot.
enum color { red, green, blue };
Zde, color může nabývat tří hodnot.

Neordinální datové typy

U těchto typů není jednoznačně definován předchůdce a následovník.

- **Reálné číslo** – real, double, float, interně implementováno pomocí mantisy a exponentu (mantisa * 2^{exponent}). (příklad zobrazení reálného čísla v PC – standard IEEE 754 (4 Byty). 1. bit znaménkový, 8 bitů exponent (kód s posunutou nulou – báze 2^{7-1}), 23 mantisa)
- **Prázdný datový typ**
Typ který nenabývá žádných hodnot, např. typ void v C. Používá se pro označení dat nespecifikovaného typu, funkcí bez návratové hodnoty apod.
Existuje také datový typ označující prázdnou hodnotu (null).

Složené datové typy

Jsou složeny z jednoduchých datových typů. Můžou být:

1. **homogenní** – skládají se ze stejných jednoduchých nebo složených datových typů,
 2. **heterogenní** – skládají se z různých jednoduchých nebo složených datových typů.
- **Pole** – array, může být vícerozměrné. Má jednoznačně definované pořadí prvků.
 - **Textový řetězec** – string, složený z typů char. Je to vlastně pole s hodnotami char.
 - **Seznam** – list, na rozdíl od pole nemá definované pořadí prvků.
 - **Záznam (struktura)** – struct, může být složen z více typů, navenek se chová jako kompaktní celek.
typedef struct { char *name, int age } person;

Zvláštní datové typy

- **Ukazatel** – pointer, ukazuje na místo v paměti.
- **Soubor** – file, reprezentuje ukazatel na soubor (Pascal).

Datové struktury

Je to způsob, jak ukládat data tak, aby byly používány efektivně.

- **Statické datové struktury** – nemohou v průběhu výpočtu měnit počet komponent ani způsob jejich uspořádání (pole, záznam).
- **Dynamické datové struktury** – jejich rozsah se může v průběhu výpočtu měnit (ukazatel, zásobník, fronta, seznam, strom).

Patří sem výše uvedené datové typy + abstraktní datový typ.

Abstraktní datový typ

Implementačně nezávislá specifikace struktury dat s operacemi povolenými na této struktuře. Mezi základní ADT patří například mapa, zásobník, fronta, seznam, množina, strom, graf (jednoduše řečeno, není definováno, co daná struktura obsahuje).

Řídicí struktury

Rozhodují o dalším provádění programu: větví jeho běh, vytváří cykly nebo jinak mění běh programu.

1. **Posloupnost příkazů** – všechny příkazy se provedou postupně.
2. **Větvení** – příkaz se provede v závislosti na splnění/nesplnění podmínky.
3. **Cykly** – část programu se v závislosti na splnění/nesplnění podmínky provede několikrát.

1. Posloupnost příkazů

příkaz 1; příkaz 2; příkaz 3; ... příkaz n;

2. Větvení

If – nejjednodušší příklad podmínky:

```
if (podmínka) {  
    //příkazy při splnění podmínky  
} else {  
    //příkazy při nesplnění podmínky  
}
```

Switch – používaný, pokud máme více podmínek se stejným výrazem:

```
switch (n) {  
    case 1: ... ;  
    case 2: ... break;  
    case 3: ... ;  
    default: ... ;  
}
```

Příkaz **break** není povinný, ale pokud se neuvede, začnou se provádět další větve bez ověření podmínky. (Např. pokud n = 1, provedou se první dvě větve.)

default – větev která se provede vždy, pokud není splněna ani jedna z podmínek. Je nepovinná.

3. Cykly

For – cyklus s podmínkou na začátku.

```
for (int i = 0; i < 5; i++) {  
    // příkazy  
}
```

Foreach – cyklus, který projde přes všechny prvky dané struktury (např. pole) a přiřadí je do „proměnné“

```
for (int proměnná : pole) {  
    // příkazy  
}
```

While-do – cyklus s podmínkou na začátku.

```
int i = 5;  
while (i < 5) {  
    i++;  
}
```

Do-while – cyklus s podmínkou na konci. Provede se vždy alespoň jednou.

```
int i = 5;  
do {  
    i++;  
} while (i < 5);
```

Společně s cykly se často používají příkazy na přerušení běhu cyklu:

break – přejde za cyklus,
continue – přejde na začátek cyklu.

Nekonečný cyklus – za normálních okolností není ukončen, v praxi se často používá v kombinaci s předchozími příkazy na přerušení běhu (while (true); nebo for (;;)).

Skoky

Neměly by se používat z důvodu znepřehlednění kódu.

Podmíněný skok – provede se, pokud je splněna podmínka.

Nepodmíněný skok – provede se vždy.

Kompilace

Kompilátor (překladač) je nástroj sloužící k překladu algoritmů zapsaných ve vyšším programovacím jazyce do strojového jazyka. Překlad probíhá v následujících krocích (platí pro jazyk C).

1. **Preprocessing** – importuje soubory, řeší podmíněnou komplikaci, instrukce preprocesoru a makra.
2. **Compilation** – z výstupu preprocesoru a zdrojového kódu vygeneruje program ve strojovém kódu.
3. **Assembly** – vytváří objektový kód (což je v podstatě strojový kód, ale s dodatečnými informacemi o externích proměnných, kódu v knihovnách apod.).
4. **Linking** – z objektových souborů a kódu z knihoven vytváří jeden (většinou spustitelný) soubor. Řeší přiřazení adres u funkcí a proměnných.

Kompilátor

se sestává z několika částí.

1. Lexikální analyzátor

Ze vstupního souboru získá tzv. lexémy (základní prvky jazyka, např. if) a ty zašle spolu s dodatečnými informacemi (jestli je to operátor, klíčové slovo apod.) syntaktickému analyzátoru. Lexémy jsou popsány pomocí regulárních výrazů, pro rozpoznávání se používá konečný automat. Celý výstup se označuje jako **token**.

2. Syntaktický analyzátor

Tzv. parser, analyzuje, jestli je program zapsán správným způsobem (jestli každý if je ukončen složenou závorkou). Také určuje, v jakém pořadí se budou vykonávat části příkazů (výraz $1 + 2 * 3$ vyhodnotí na $1 + (2 * 3)$). Rozpoznává se bezkontextový jazyk, což je složitější a trvá delší dobu než získání lexémů. Výstupem je **parse tree**, který se předá sémantickému analyzátoru.

3. Sémantický analyzátor

Zpracovává předchozí výstup a provádí analýzu významu a korektnosti jednotlivých operací (typová kontrola, konverze). Výstupem je opět **parse tree**, ale s dodatečnými informacemi a konverzemi.

4. Překladač do mezikódu

Z **parse tree** se generuje kód (vytváří se z něj posloupnost instrukcí). Tento kód není v cílovém jazyce, ale v pomocném mezikódu. Někdy se však jako mezikód používá i assembly language.

5. Optimalizátor

Provádí další transformace kódu, jejichž cílem je zlepšení vlastností výsledného kódu (rychlejší běh, menší kód). Zahazuje mrtvý kód (např. if (false) { ... }).

6. Generátor kódu

Poslední fáze, generuje se výstupní kód v cílovém jazyce (nejčastěji strojový kód).

Interpretace

Interpret je nástroj, který umožňuje vykonávat zápis jiného programu přímo v jeho zdrojovém kódu (Java Virtual Machine). Program je pak jednoduše přenositelný mezi různými platformami. Interpreti se mohou chovat třemi různými způsoby:

1. provádějí přímo zdrojový kód (unixový shell),
2. nejprve přeloží zdrojový kód do mezikódu, který provádějí (Perl, Python),
3. přímo spustí předem vytvořený předkompilovaný mezikód, který je produktem části interpretu (Java).

Výhody interpretace

- Rychlosť vývoje – program je rychleji spuštěn, protože se nemusí kompilovat.
- Laditelnost – nejsou použity optimalizace, činnost přesně odpovídá zdrojovému kódu.
- Kompatibilita – program může být spuštěn na mnoha různých architekturách.
- Správa paměti – proměnné nejsou vázány na fyzické adresy.
- Jednoduchost – je snadnější vytvořit interpret než komplátor.

Nevýhody interpretace

- Efektivita – interpretace je pomalejší než již zkompilovaný program.

Interpret bytekódu

Bytekód je vysoce optimalizovaná a komprimovaná reprezentace zdrojového kódu. Není to však strojový kód.

Interpret abstraktního syntaktického stromu

Tzv. abstract syntax tree. Výhodou oproti bytekód interpretaci je, že udržuje globální program, strukturu a vztahy mezi instrukcemi.

Just-in-time komplilace

Reprezentace programu je za běhu komplilována do strojového kódu. Nejvíce používané u jazyků Java, Python a .NET.

Kategorie programovacích jazyků

Můžeme rozdělit několika způsoby. Asi je potřeba znát jejich syntaxe, aby to pokrylo rozsah otázky, ale vypisovat to tady bylo na dlouho, a navíc je to v jiných otázkách.

Míra abstrakce

- vyšší (Java)
- nižší (strojový kód, assembly language)

Překlad a spuštění

- komplilované (C, C++, Pascal)
- interpretované (Java, Perl, Python)

Paradigma

- imperativní – strukturované (C, Pascal), objektově orientované (Java, C++)
- deklarativní – funkcionální (Haskell, Lisp), logické (Prolog, Gödel)

Znalost programovacího jazyka

Prostě něco znejte, třeba Haskell (hue).

Objektově orientované programování

Zapouzdření, dědičnost, polymorfismus, objektové programování v imperativním jazyce, spolupráce objektů, událostmi řízené programování, výjimky. Realizace obecných principů OOP v C++ nebo Javě (podle vlastní volby).

Objektově orientované programování

Objektové programování (OOP) je model vývoje software založený na objektech a interakcích mezi nimi. Dá se říct, že na rozdíl od procedurálního modelu se objektové programování nezaměřuje na logiku, ale na data – místo pohlížení na program jako na sérii procedur se zaměřuje na objekty(data), které poskytují logiku skrze svá rozhraní.

Objekty, třídy, vlastnosti, viditelnost a interakce

Základním krokem při objektovém návrhu je identifikace objektů/entit, které budou v našem programu vystupovat – tyto objekty budou v kódu reprezentovány třídami.

Třída představuje množinu objektů se stejnými vlastnostmi (např. všechny objekty třídy Person budou mít vlastnost name). Třída deklaruje, jaké proměnné(atributy) a metody budou její instance(jednotlivé objekty) mít.

- **Proměnné(atributy)** – datové hodnoty svázané (zapouzdřené) v objektu, nesou informace o charakteristikách objektu
- **Metody** – logika poskytovaná objektem, de facto procedury/funkce, většinou pracují s proměnnými objektu, může mít vstupní parametry a může vracet hodnotu Navíc obvykle deklaruje konstruktor, tj. speciální „proceduru“, která vytvoří nový objekt dané třídy, může přijímat vstupní parametry a přímo při vytvoření objektu nastavit jeho vlastnosti.

Objekt je pak jeden konkrétní jedinec(instance) příslušné třídy. Objekt vzniká instanciací třídy, obvykle použitím konstruktoru. Pro konkrétní objekt nabývají vlastnosti deklarované jeho třídou konkrétních hodnot (např. objekt karel je třídy Person a vlastnost name má nastavenou na „Karel“).

Každý objekt a všechny jeho proměnné a metody mají nastavenou nějakou viditelnost. **Viditelnost** určuje, z jakého okruhu ostatních objektů bude daný prvek přístupný. Např. privátní(private) prvek je přístupný pouze z objektu, kterému náleží. Ty prvky (metody nebo proměnné), které jsou viditelné zvnějšku objektu, tvoří rozhraní objektu.

Objekty mezi sebou komunikují a interagují skrz toto rozhraní, tj. voláním metod nebo přístupem k viditelným proměnným ostatních objektů.

Objektové programování v imperativním jazyce

OOP obecně nespecifikuje, jak má vypadat kód, pouze to, že výsledný software se skládá nějakým způsobem z objektů a jak se skládá jejich hierarchie a jak provádí jejich interakce. OOP v imperativním jazyce znamená, že se programátor drží zásad OOP, ale jednotlivé objekty, nebo spíš třídy příše imperativně, tedy jako sekvenci příkazů. (Opravdu na tom nic víc není)

Základní principy OOP

Každý, kdo něco ví o OOP teď okamžitě vysype „zapouzdření, dědičnost, polymorfismus“. Takže jednotlivě:

Zapouzdření(encapsulation)

Zapouzdření úzce souvisí s viditelností objektů a jejich vlastností. Princip zapouzdření říká, že cokoli, co nemusí být viditelné, také nemá. Učenými slovy jde o snahu omezit možnosti interakce objektů jen na jejich specifikovaná rozhraní, a ta omezit jen na ty prvky, které musí být nezbytně viditelné.

Obvyklá praxe (aspoň v Javě) je (pokud už je samotný objekt viditelný) označit jeho proměnné za privátní, metody potřebné pro interakci s ostatními objekty nechat veřejné a všechno ostatní nastavit na nejnižší možnou úroveň viditelnosti, tj. většinou privátní.

Dědičnost(inheritance)

Dědičnost umožňuje tzv. rozšířit již existující třídu, tedy vytvořit její podtřídu(podtyp), která od svého rodiče zdědí jeho atributy a metody. Účel dědičnosti je poskytnout jednoduchý postup pro modelování hierarchie objektů, které jsou ve vztahu generalizace-specializace. Tzn. jeden objekt(podtyp) je speciálním případem druhého(nadtypu).

Potomek pak může jak rozšiřovat funkcionality rodiče přidáním nových metod nebo atributů, ale i překrýt metody rodiče, tj. deklarovat metodu se stejnou signaturou (jménem, parametry a návratovým typem), ale jinou logikou. Překrývat lze pouze tzv. virtuální metody, některé jazyky vyžadují explicitní deklaraci virtuální metody (slovíčko virtual v např. C++, C#), jiné považují metody za virtuální implicitně a umožňují definovat metody zamknuté před překrytím (např. Java slovem final). Stejně tak bývá možné zamknout i celou třídu proti rozšíření.

Pokud existuje metoda, která je potřeba ve všech podtřídách, nicméně nemá žádný smysl ji implementovat v nadřídě, je možnost označit metodu i celou třídu za abstraktní. Implementace metody je pak čistě na potomcích a mimo to daná nadříďida nejde instanciovat. Výhodou dědičnosti je fakt, že s rozšířením třídy člověk ušetří velké množství kódu, protože píše pouze tu část, která je specifická pro daného potomka, zbytek se dědí od rodiče. Nicméně dědičnost má velké množství omezení. Jedná se o statickou vazbu tříd/objektů, tudíž např. máte-li více potomků jednoho rodiče, není možné, aby jeden objekt měnil v průběhu času svůj typ z jednoho potomka na jiného. Navíc pokud jednotliví potomci zobrazují určité role, objekt nemůže zastávat více rolí zároveň, zvláště pak pokud programovací jazyk umožňuje dědit pouze z jediného rodiče.

Příklad: Máme třídu Person a její podtřídy Student a Employee. Student se pak nemůže stát zároveň zaměstnancem (Employee), protože je staticky svázán s podtřídou Student. Určitá možnost řešení by byla, pokud to jazyk podporuje, vytvořit třídu, která bude dědit z obou tříd, tedy Student i Employee, nicméně mnohem lepší je v tuto chvíli místo dědičnosti použít kompozici.

Polymorfismus(podtypový)

Podtypový polymorfismus, úzce souvisí s dědičností, resp. hierarchií objektů/tříd. U podtypového polymorfismu jde o fakt, že na místo, kde je očekávána instance nějaké třídy, je možné dosadit instanci jakékoli její podtřídy (tzv. princip subsumpce). Odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší podle

implementace. Protože je tedy náš objekt instancí podtřídy očekávané nadtřídy, jeho rozhraní je nadmnožinou rozhraní nadtřídy. Tudíž se s tímto objektem bude pracovat stejně jako s instancí jeho nadtřídy, nicméně bude dostupná pouze ta funkcionality, jejíž rozhraní se shoduje s rozhraním nadtřídy. (Takže pokud člověk v podtřídě definuje nějaké specifické metody navíc, nebude je moci použít, pokud její instanci předá namísto instance nadtřídy).

Příklad: Máme třídu Animal a její podtřídy Cat a Dog definované nějak takto:

```
class Animal {  
    talk() : string; //obecné zvíře nemá, jak se ozvat, tudíž metoda bude abstraktní }  
  
class Cat {  
    talk() : string {  
        return "Meow"; //kočka překrývá metodu talk() : string a vrací mňoukání }  
}  
  
class Dog {  
    ownerName : string; //pes má navíc atribut jméno majitele  
    talk() : string {  
        return "Woof";  
    }  
    getOwnerName() : string {  
        return ownerName; //pes navíc definuje metodu, která vrátí jméno majitele }  
}  
  
class Test {  
    getAnimalToTalk(Animal animal) : string {  
        return animal.talk();  
    }  
}
```

Pokud teď zavolám metodu getAnimalToTalk s instancí třídy Cat jako parametrem, dostanu „Meow“, s instancí třídy Dog „Woof“. Nicméně pokud bych chtěl v metodě navíc zjistit jméno majitele psa, pak bych ji musel definovat přímo s parametrem třídy Dog, protože metoda getOwnerName už není součástí rozhraní třídy animal.

Událostmi řízené programování, výjimky

Tytle dvě kapitoly píšu dohromady, protože spolu dost souvisí. Reakce na události často vyhazují výjimky, obsluha výjimek je navíc dost podobná reakci na události.

Událostmi řízené programování(event-driven programming (EDP))

EDP je paradigma programování, kde průchod programem je řízen událostmi zaregistrovanými nějakými senzory/listenersy; akcemi uživatele nebo zprávami od ostatních procesů či vláken. Druhá možná definice EDP je architektura aplikací, kde aplikace obsahuje tzv. hlavní smyčku, ve které čeká na příchozí události. Smyčka je rozdělena na dvě části: detekce a selekce události, obsluha události.

Událostmi řízená aplikace pak tedy obsahuje detektory událostí a hlavně event handlery, odpovídající za obsluhu jím přiřazených událostí pomocí definovaných metod.

Nejtypičtější použití EDP je při vytváření grafického uživatelského rozhraní, které se pak skládá z jednotlivých prvků, které reagují na podněty uživatele.

Výjimky(exceptions)

Výjimky jsou velmi silný nástroj, jak psat robustní, spolehlivé programy odolné proti chybám okolí (uživatele, systému...) a reagovat na chybové stavby. Obvykle se výjimka vyhazuje při vykonávání metody, které by program uvedlo do chybového stavu. Výjimka pak putuje programem výš do komponent, které volaly danou metodu. Na nich je, aby na ni buď zareagovaly jejím zachycením a provedly nějakou obsluhu, nebo ji propustily výš. Ve chvíli, kdy výjimka dosáhne „vrcholu“ (zpravidla metody main) a není zachycena, program spadne. Krom předdefinovaných výjimek si obvykle může programátor definovat své vlastní výjimky.

Možnosti reakce na výjimku:

- zachycení a provedení nápravy nebo aspoň zaznamenání, že k výjimce došlo
- propuštění výjimky nadřazené komponentě
- řetězení výjimek, tj. zachycení a vyhození nové výjimky, která obsahuje předchozí výjimku jako příčinu vyhození

Zásady práce s výjimkami:

- vždy nějak zareagovat na výjimku
- při zachycení výjimky provést adekvátní kroky (nenechat catch blok prázdný)
- pokud nejde na místě reagovat, propustit výjimku dál
- nesnažit se napravit výjimku přímo v metodě, která by ji mohla vyvolat, k čemu by tam pak byla...

A jak je to v Javě

Java je imperativní objektový jazyk. Není čistě objektový, protože obsahuje primitivní datové typy.

Nebudu tady moc rozebírat základní syntaxi Javy, pokud by si někdo nebyl jist, doporučuju se kouknout na <http://docs.oracle.com/javase/tutorial/java/index.html>.

Objekty, třídy, vlastnosti, viditelnost a interakce:

Jak vytvořit třídu, doufám, každý, kdo bude o Javě mluvit, ví, takže se zaměřím spíš na technické detaily.

Vytváření objektů Objekty se instancují překvapivě klíčovým slovem new a voláním konstruktu. Co možná někdo neví, je, že každá třída má implicitně definovaný veřejný bezparametrický konstruktor, kterého se jde „zbavit“ jen tak, že se překryje privátním bezparametrickým konstruktorem.

Pokud vaše třída rozšiřuje jinou třídu, pak na prvním řádku svého konstruktu vždycky musí volat adekvátní konstruktor své nadtřídy, tj. super(paramery).

Vlastnosti a viditelnost objektů Java disponuje čtyřmi typy viditelnosti:

- private – viditelný pouze uvnitř třídy
- package protected (default bez modifikátoru) – viditelný uvnitř celého balíku(package)
- protected – viditelný uvnitř třídy a v podtřídách
- public – veřejně viditelný

Krom klasických proměnných jsou k dispozici statické proměnné, potažmo i statické metody, značení slovem static. Statické proměnné jsou společné všem instancím třídy. Statické metody lze volat bez instanciaci třídy. Statická metoda se nesmí odkazovat na jiné než statické atributy třídy, obecněji nesmí se odkazovat na nic, co je specifické pro každou instanci. Java nemá konstanty. Pokud chcete definovat konstantu, definujte proměnnou jako static final.

V Javě jsou jak objektové, tak primitivní datové typy, všechny objektové typy dědí ze třídy Object, tudíž dědí mj. metody equals, hashCode a toString.

Operátor == porovnává dvě proměnné na shodu reference, tudíž pro porovnávání hodnot funguje pouze na primitivních typech a na objektech, kde je při shodě reference zaručena shoda objektů.

Při uložení objektu do proměnné se ukládá pouze reference na objekt, tudíž pokud chcete porovnat dvě objektové proměnné, použijte metodu equals, kterou daný objekt musí mít překrytu.

Principy OOP v Javě

Zapouzdření Jak na to: V zásadě všechny atributy tříd definovat jako private, s výjimkou konstant, které jsou neměnitelné, tudíž můžou mít viditelnost, jakou kdo chce. Metody, které budou potřeba pro interakci s ostatními objekty bývá dobré označit public, všecko ostatní, co není potřeba ošetřit nějak speciálně nechat nejlépe private.

Dědičnost a Interface Java dovoluje pouze jednoduchou dědičnost, tj. třída může dědit pouze z jednoho rodiče. Rozšíření třídy se definuje slovem extends. Všechny třídy v Javě jdou implicitně rozšířit, stejně jako metody jde defaultně překrýt. Obojí jde „zakázat“ použitím slova final v definici. Volání metod, případně přístup k atributům rodiče se realizuje následovně: super.metoda(parametry) resp. super.atribut

Právě kvůli možnosti přístupu k prvku rodiče je v Javě viditelnost protected. Pokud je v nějaké třídě abstraktní metoda, def. slovem abstract, musí být celá třída označena za abstraktní a nejde instanciovat.

Alternativa k dědičnosti jsou rozhraní neboli interface. Definují se stejně jako třída, jen místo slova class se použije interface. Interface obsahuje jen výčet metod, jejich implementace je na třídě, která implementuje dané rozhraní, def. slovem implements. Interface stejně jako rodič v dědičnosti představuje nadtyp třídy, která ho implementuje. Interface ovšem nejde instanciovat, navíc je možné implementovat více rozhraní jednou třídou. Rozhraní se můžou rozšiřovat.

Výhoda rozhraní oproti dědičnosti je v tom, že jednotlivé implementace nejsou nijak provázané, jediné co musí splňovat je implementace všech metod rozhraní. Nevýhoda je, že implementující třída nedědí žádný kód.

Polymorfismus V Javě funguje princip subsumpce, viz výše. Jediná věc navíc je, že všechno platí nejen při dědičnosti, ale i pro implementace Interface.

Událostmi řízené programování Krom už zmíněného vytváření GUI, se dá v Javě programovat reakce na události nějak takto:

1. Vytvoření třídy naší události rozšířením třídy EventObject, nebo použití předdefinované události
2. Vytvoření Listeneru, který událost zaregistrouje a reaguje na ni nebo ji předá jiné komponentě k obsluze.
3. Svázání listeneru s nějakým zdrojem událostí.

Celkem pochopitelný příklad je tady:

<http://coffeeonesugar.wordpress.com/2009/02/15/event-based-programming-in-java/>

Výjimky To, co je napsáno v odstavci o výjimkách platí i v Javě s tím rozdílem, že Java rozlišuje výjimky hlídané a nehlídané. Hlídané výjimky musíte v kódu nějak ošetřit, tj. zachytit je nebo deklarovat, že daná metoda může tuto výjimku vyhodit nebo propustit, jinak vás kompilátor nepustí dál.

Vyhození výjimky Výjimku buď vyhazuje JVM nebo programově jde zajistit např. takto:

throw new Exception;

Propuštění výjimky Propuštění výjimky předá odpovědnost za její zpracování nadřazené komponentě. Signalizace toho, že metoda propouští výjimku nebo že ji vyhazuje:

throws Exception v definici metody

```
Zachycení výjimky Zachycení výjimky se realizuje pomocí try-catch bloku asi takto: try {  
    metodaKteraMuzeVyvolatVýjimku(); } catch (VýjimkaKterouMetodaVyvola ex) {  
    reakceNaVýjimku; } finally {
```

```
//nepovinno jen pokud něco chcete udělat nutné když výjimku chytíte, treba zavřít soubor  
dalsiAkce();  
}
```

Řetězení výjimek Spočívá v zachycení výjimky a vyhození nové s původním jako příčinou:

```
catch (VýjimkaKterouMetodaVyvola ex) {  
    throw new NovaVýjimka(„zprava“, ex);  
}
```

Vytváření výjimek Vlastní výjimku lze vytvořit rozšířením třídy Exception pro hlídanou výjimku, nebo RuntimeException pro nehlídanou výjimku a definicí konstruktorů, které jen delegují na konstruktory své nadřídy. Exception i RuntimeException implementují rozhraní Throwable.

Základní principy počítačů

Číselné soustavy, vztahy mezi číselnými soustavami, zobrazení čísel v počítači, principy provádění aritmetických operací. Kombinační a sekvenční logické obvody. Von Neumannova architektura. Principy práce procesoru, přerušení.

Číselné soustavy

Je způsob reprezentace čísel. Podle způsobu určení hodnoty čísla z dané reprezentace rozlišujeme dva hlavní druhy číselných soustav:

1. **poziční číselné soustavy**
2. **nepoziční číselné soustavy**

Dříve se používali reprezentace z obou soustav, ale dnes se obvykle používají soustavy poziční.

Nepoziční číselné soustavy

způsob reprezentace čísel, ve kterém není hodnota číslice dána jejím umístěním v dané sekvenci číslic. Tento způsob zápisu číslic se dnes již nepoužívají a jsou považovány za zastaralé. V nejjednoduším systému stačí sečít hodnoty jednotlivých číslic. Pokud by například byly hodnoty symbolů následující: A = 1, B = 10, C = 100, D = 1000, pak vyjádřením čísla 3542 by mohl být řetězec "AABBBCCCDDDD".

- **Výhody:** jednoduché sčítání a odečítání
- **Nevýhody:** dlouhý zápis čísel, která výrazně převyšují hodnotu nejvyššího symbolu soustavy, často neobsahovaly znaky pro nulu a záporná čísla
- **Příklady:** Římské, Egyptské Řecké číslice

Poziční číselné soustavy

je dnes převládající způsob písemné reprezentace čísel. Pokud se dnes mluví o číselných soustavách, tak jsou tím myšleny poziční soustavy. V tomto způsobu zápisu čísel je hodnota každé číslice dána její pozicí v sekvenci symbolů. Každá číslice má tuto pozici dánou váhu pro výpočet celkové hodnoty čísla.

Polyadické soustavy - jsou speciálním případem pozičních soustav

Základ - počet symbolů pro číslice používaných v dané soustavě

Řád - váha číslice

zápis - číslo = součet mocnin základu vynásobených číslicemi

$$A = a^n z^n + a^{n-1} z^{n-1} + \dots + a^1 z^1 + a^0 z^0$$

$$A = 1 \cdot 10 + 2 \cdot 10^1 + 3 \cdot 10^0$$

zhuštěný zápis - běžná forma zhuštěného zápisu

$$A = a^n a^{n-1} \dots a^1 a^0$$

$$A = 123$$

$$A = 123_{10}$$

- zobecnění pro racionální čísla (zavedením záporné mocniny)

$$A = a^n z^n + a^{n-1} z^{n-1} + \dots + a^1 z^1 + a^0 z^0 + a^{-1} z^{-1} + a^{-2} z^{-2} + \dots + a^{-m} z^{-m}$$

- zobecnění pro záporná čísla - přidáme znaménka (nevhodné pro počítače)
- zobecnění pro komplexní čísla - zavedeme imaginární jednotky

Příklad: číselné soustavy se základem 2 (tj. dvě číslice 0, 1) a výpočet jeho hodnoty:

$$(10010)_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4$$

Soustavy užívané v počítačové praxi:

- **Dvojková** (binární): $z=2$; obsahuje číslice 0, 1
- **Osmičková** (oktalová): $z=8$; obsahuje číslice 0, 1, 2, 3, 4, 5, 6, 7
- **Šestnáctková** (hexadecimální): $z=16$; obsahuje 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Vztahy mezi číselnými soustavami:

číslo v soustavě o základu zk , kde zk jsou přirozené čísla, lze převézt do soustavy o základu zk jednoduše.

Převody: $2 \leftrightarrow 8$ $2 \leftrightarrow 10$ $2 \leftrightarrow 16$ $8 \leftrightarrow 16$

Převod z 10 do 2 soustavy:

$$12,2_{10} = ?_2$$

Musíme rozdělit na celou a desetinou část čísla:

celá	část	desetinná	část
12	:2	0,	22
6	0	0	4
3	0	0	8
1	1	1	6 (0,62)
0	1	1	2 (0,22)
		0	4
		0	8
		1	6
	

$$12,2_{10} = 11000011001100\dots_2$$

Převod z 2 do 10 soustavy:

$$1100,0011001100_2 = ?_{10}$$

$$\text{Celá část: } 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 12$$

Desetinná část: $0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} \dots = 0,1999\dots$ - řešíme zaokrouhlením poslední číslice rozv

Zobrazení čísel v počítači

Zobrazení celého čísla v počítači v binárním tvaru:

Zobrazení kladných čísel

- rozsah zobrazení bez použití znaménkového bitu $< 0; 2^{n-1} >$. Zobrazení na 4 bitech ($n=4$)

0	0	0	0		0
0	0	0	1		1
1	0	0	0		8
1	1	1	1		15

- rozsah při použití znaménkového bitu: $< 0; 2^{n-1} - 1 >$ nejvyšší bit určuje znaménko ($0=+; 1=-$)

Zobrazení záporných čísel

Přímý kód se znaménkem

- první bit určuje kladnost (0), respektive zápornost (1) čísla. Rozsah zobrazení je $< -2^{n-1} + 1; -0 >$

0	0	0	0		0
1	0	0	0		-0
0	1	1	1		7
0	0	1	1		3
1	0	1	1		-3

Přímý kód se znaménkem

- rozsah zobrazení: $< -2^{n-1} + 1; -0 >$, první bit určuje zápornost, záporná čísla se invertují, tedy:

0	0	0	0		0
1	1	1	1		-0
0	0	1	1		3
1	1	0	0		-3

Pozor na existenci dvou 0 - kladné a záporné!

Doplňkový kód

- por zápis kladného čísla jen převedeme číslo do dvojkové soustavy stejně, jako číslo bez znaménka.

U doplňkového kódu je vyřešen problém dvou nul. Postup pro zobrazení záporných čísel v doplňkovém kódu:

1. zobrazit kladné číslo v binární soustavě
2. prohodit 1 a 0 v zápisu binárního čísla
3. přičíst 1

Příklad 1) Převedeme číslo -55 do doplňkového kódu na 16 bitů

1. $(55) = 0000000000110111$; prohodíme jedničky a nuly a přičteme 1
2. $(-55) = 1111111111001000 + 1 = (1111111111001001)_{DK}$

Příklad 2) Převedeme číslo -1023 do doplňkového kódu na 16 bitů

$$(-1023) = 1111110000000000 + 1 = (1111110000000001)_{DK}$$

Kód s posunutou nulou

- v kódu s posunutou nulou využíváme **bázi posunutí** (standardně $2^7 - 1$). K této bázi přičteme požadované číslo a výsledek zobrazíme.

Příklad: číslo 55 zobrazíme na 8 bitů:

$$2^7 - 1 + 55 = 128 - 1 + 55 = 182 = (10110110)$$

analogicky zobrazíme záporné číslo:

$$2^7 - 1 - 55 = 128 - 1 - 55 = 72 = (1001000)$$

Zobrazení reálného čísla v jednoduché přesnosti

Zobrazení reálných nebo příliš velkých celých čísel se provádí v pohyblivé řádové čárce. Čísla jsou zobrazena ve tvaru: ČÍSLO = MzE, kde

M mantisa čísla, zobrazená v soustavě o základu z

E exponent

z ... základ pro výpočet exponentové části

Jedním z používaných formátů pro zobrazení čísel v pohyblivé řádové čárce je formát podle standardu IEEE 754 používaný v moderních počítačích.

Struktura čísla: znaménkový bit (1b) | exponent (8b) | mantisa (23b)

- Znaménkový bit - kladné číslo má znaménkový bit nulový a u záporného čísla je jedničkový bit
- Exponent - je uložen na 8 bitech v kódu s posunutou nulou, báze je $2^7 - 1 = 127$
 - příklad: exponent 4_{10} je uložen jako $4 + 127 = 131 = (10000011)_2$
- Mantisa - je znázorněna 23 bitů v přímém kódu
 - první bit (na nulté desetinné pozici) se neukládá, bere se implicitně jako 1 (.mantisa se tedy ukládá od druhého bitu)
 - myšlená desetinná čárka (tečka) je umístěna za nejvyšším bitem mantisy
 - příklad: mantisa $1,5625_{10}$ je desetinné číslo $1 + 0,5 + 0,0625 = 2^0 + 2^{-1} + 2^{-4} = 1,1001_2$ a uloží se jako $1001000000000000000000000$

Příklad: Zobrazte číslo $(-258,125)_{10}$ ve formátu IEEE (na 4 bytech):

$$258_{10} = (100000010)_2$$

$$0,125 = 0,25 \mathbf{0}; 0,25 = 0,5 \mathbf{0}; 0,52 = 1,0 \mathbf{1} (\mathbf{0}, \mathbf{125}) = (\mathbf{0}, \mathbf{001})_2$$

$$(258,125)_{10} = (100000010,001)_2$$

Nyní je nutné provést normalizaci - pomocí násobku čísla s mocninami dvojky převést do intervalu $[1, 2)$:

nenormalizované = normalizované 2^n , kde n je hledaná mocnina

normální tvar: $1,00000010001 \cdot 2^8$

$$\text{exponent: } 2^7 - 1 + 8 = 2^7 + 7 = 10000000 + 111 = (10000111)_2$$

Jelikož se předpokládá normální tvar, jednotka na místě před desetinou čárkou se nezapisuje

$$(-258,125)_{10} = (1|10000111|0000010001000000000000)_{\text{IEEE}}$$

Příklady zobrazení čísel:

Příklad 1: Převeďte číslo 11010011 z přímého kódu se znaménkem do desítkové soustavy.

$$11010011 \rightarrow (64 + 16 + 2 + 1) = -83$$

Příklad 2: Převeďte číslo 10111001 z doplňkového kódu do desítkové soustavy.

$$1011100101000111 \rightarrow (64 + 4 + 2 + 1) = -71$$

Příklad 3: Převeďte číslo 01000111 z kódu s posunutou nulou do desítkové soustavy.

$$01000111 \rightarrow (64 + 4 + 2 + 1) - 127 + 71 = -56$$

Příklad 4: Převeďte číslo 01000011100000010001000000000000 z formátu IEEE (ne 4 bytech) do desítkové soustavy.

$$0|10000111|0000010001000000000000$$

exponent: $10000111 - 127 + 135 = 8$, tedy základ je $2^8 = 256$

mantisa: $1,0000001000100000000000$ (připojení implicitního bitu)

$$1,00000010001_2 = 2^0 + 2^{-7} + 2^{-11} = 1 + 0,0078125 + 0,0048828125 =$$

$$1,00830078125_{10}$$

$$\text{Výsledek: } 256 \cdot 1,00830078125 = +258,125$$

Principy provádění aritmetických operací:

Součet doplňkového kódu

- všechny byty se sčítají stejně; vznikne-li přenos ze znaménkového bitu, tak se ignoruje; přetečení nastane, pokud se přenos do znaménkového bitu nerovná přenosu ze znaménkového bitu
- Příklad přetečení:

$$\begin{array}{r} 0 & 1 & 1 & 0 \\ \underline{0} & 1 & 0 & 1 \\ ^01 & ^0 & 1 & 1 \end{array}$$

Součet v inverzním kódu:

- problém dvou nul; nutnost provádět tzv. kruhový přenos = přičtení přenosu z nejvyššího

A binary addition diagram. On the left, there are two columns of digits: one for -0 and one for $+1$. A horizontal line with a plus sign above it separates the digits. To the right of this line is a vertical dashed line. Below the dashed line, the sum is given as 0.001 . Above the dashed line, a circled '1' is highlighted in red, with a curved arrow pointing from it to the vertical dashed line, illustrating the concept of a carry bit.

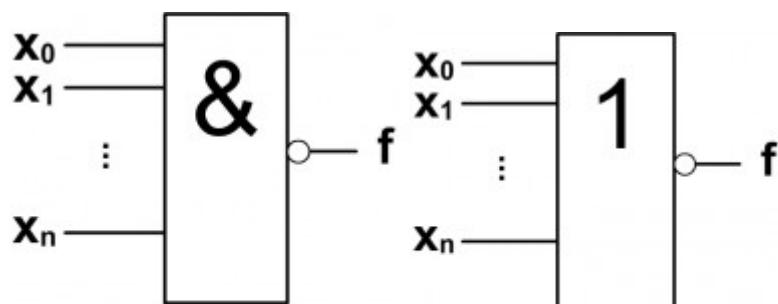
rádu:

- násobení a dělení s binárními čísly se provádějí v počítačích obvykle podle stejného algoritmu jako v dekadické soustavě.

Hrdlo XOR

- hrdlo XOR je jedním ze základních kombinačních logických obvodů, jehož výstup je exkluzivní logický součet vstupů ("bul" A nebo B). Výstup je logaritmus 1 tehdy a jen tehdy, pokud se hodnoty vstupů liší

Kombinační a sekvenční logické obvody:

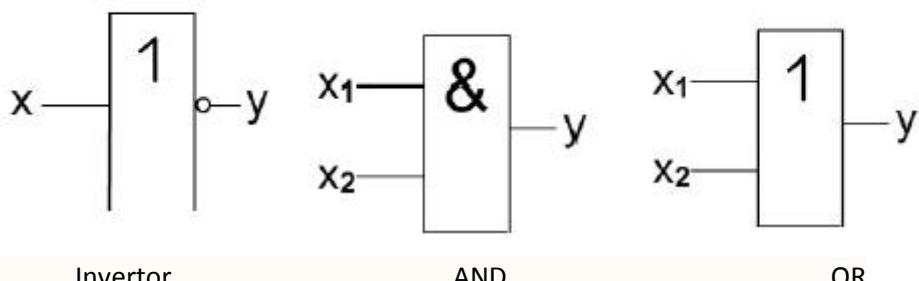


$$f = \overline{x_0 \cdot x_1 \cdot \dots \cdot x_n} \quad f = \overline{x_0 + x_1 + \dots + x_n}$$

$$y = \overline{x}$$

$$y = x_1 \cdot x_2$$

$$y = x_1 + x_2$$



Invertor

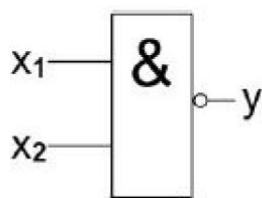
AND

OR

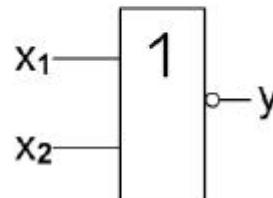
$$y = \overline{x_1 \cdot x_2}$$

$$y = \overline{x_1 + x_2}$$

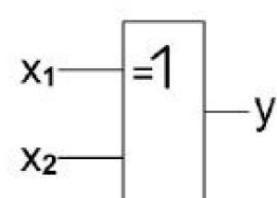
$$y = \overline{x_1 \oplus x_2}$$



NAND



NOR



XOR = nonekvivalence

Sekvenční logické obvody:

Sekvenční obvod je typ elektronického obvodu. Je složen ze dvou částí - kombinační a paměťové. Abychom mohli určit výstupní proměnné, je potřeba u sekvenčních obvodů sledovat kromě vstupních proměnných ještě jeho vnitřní proměnné = vnitřní stav. Jsou to proměnné, které jsou uchovávány v paměťových členech. Existence vnitřních proměnných způsobuje, že stejné hodnoty vstupních proměnných přivedené na vstup obvodu nevyvolávají vždy stejnou odezvu na výstupu obvodu. Sekvenční obvody dělíme na synchronní a asynchronní.

U asynchronních sekvenčních obvodů se změna vstupní proměnné promítne ihned do stavu sekvenčního obvodu. U synchronních sekvenčních obvodů je zaveden řídící synchronizační signál (hodinový signál, hodiny). Změna vstupní proměnné se promítne do stavu sekvenčního obvodu až při příchodu hodinového signálu.

Paměťová část sekvenčního obvodu je tvořena kombinačním obvodem, ve kterém byla zavedena zpětná vazba. Tomuto zapojení říkáme bistabilní klopný obvod. Jeho úkolem je převzít informaci přivedenou na vstup obvodu a uchovat tuto hodnotu, i když vstupní informace již zmizí. Typická paměťová část je klopný obvod RS.

Klopný obvod RS:

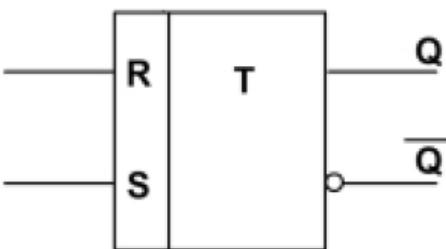
Klopný obvod RS je jedním z nejjednodušších klopných obvodů. Obvykle se zapojuje ze dvou hrdel NAND. Výstup prvního NANDu vede de jednoho ze vstupů druhého NANDu a výstup druhého NANDu vede do jednoho ze vstupů prvního NANDu.

Vstup R označuje Reset. Přivedení hodnoty logická 1 na tento vstup vynuluje hodnotu Q. Vstup S označuje Set, přivedením hodnoty logická 1 na tento vstup nastaví hodnotu Q na logickou 1. Pokud je na R a zároveň na S logická 1, jde o zakázaný stav.

Pojem zakázaný stav pochází z doby, kdy byl tento obvod realizován dvěma invertory, u kterých nebyl eliminován zpětný přenos přes jednotlivé transistory. Současné buzení obou vstupů vedlo k tomu, že se výstupní veličiny dostávaly do zakázaného pásma a tranzistory předcházely ze saturace do aktivní zóny svých charakteristik. Při používání dnešních obvyklých logických členů lze R=S=1 používat jako kterýkoli jiný stav. Pokud je na R a zároveň na S logická 0, obvod si pamatuje minulý stav výstupů.

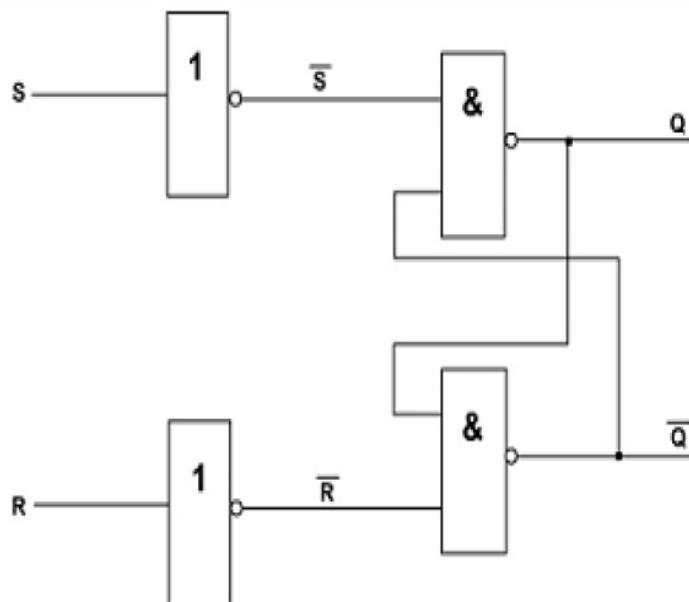
Problém nastává při přechodu z R=0, S=0 na R=1 a S=1. V tomto případě není jasné, do kterého stavu se klopný obvod překlopí (závisí na nesymetrii reálného obvodu).

Stav R=1, S=1 lze nazývat nestabilním nebo zakázaným.



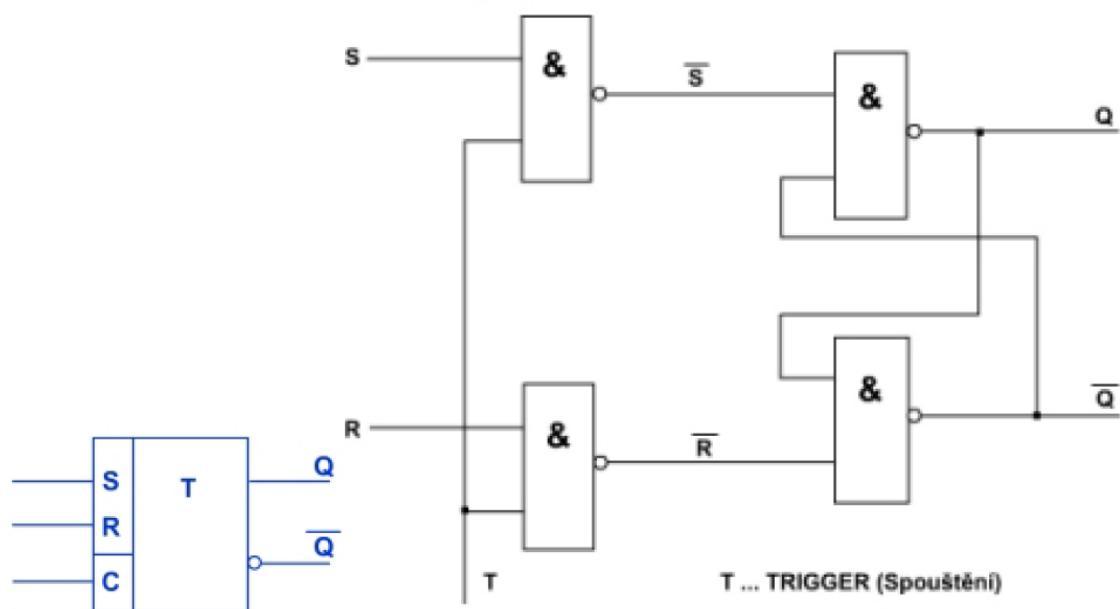
Značka:

RS může být konstruován pro řízení nulami i jedničkami. Pravdivostní tabulka a schéma zapojení obvyklého : RS řízeného jedničkami :



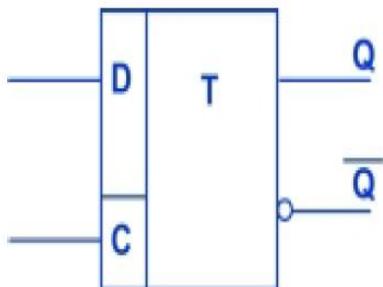
R	S	Q_i	\bar{Q}_i
0	1	1	0
1	0	0	1
0	0	Q_{i-1}	\bar{Q}_{i-1}
1	1	zakázaný stav	

Klopný obvod RS se synchronizací je používán jako základ dalších obvodů. Změna stavu je do obvodu propagována pouze, je-li hodinovým kmitočtem na vývod T přiveden patřičný impuls. Obvody mohou být řízeny hladinami (0, 1) a hránami (sestupná, vzestupná) hodinového kmitočtu.

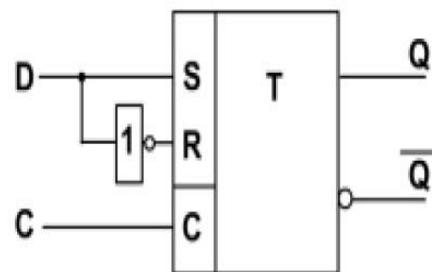


Klopný obvod D:

Vznikne doplněním obvodu RS s časovou synchronizací o invertor mezi vstupy. Tím dojde k eliminaci možných stavů na dva. D je vždy řízen synchronizací, obvod se při vedením hodinového kmitočtu přepne do stavu podle logické úrovně přivedené na vývod D. Vlastně realizuje jednobitovou paměť. Každý hodinový impulz způsobí zapamatování hodnoty vstupu.



Obrázek 2.7: Značka

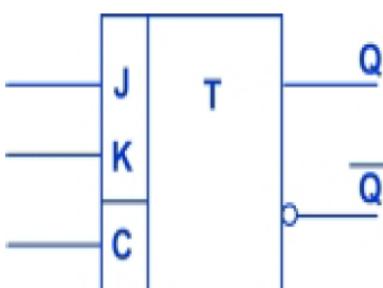


Obrázek 2.8: Zapojení a tabulka

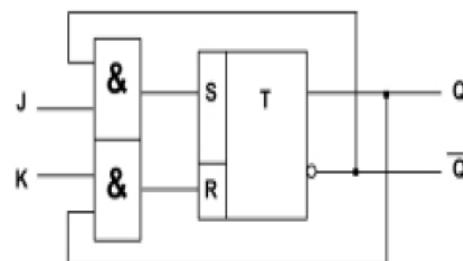
D	C	Q_i
1	1	1
0	1	0
?	—	Q_{i-1}

Klopný obvod JK:

Vývody mají obdobnou funkci jako u RS, J nastavuje hodnotu logická 1, K nastavuje hodnotu logická 0. Navíc se eliminuje zakázaný stav, při současné hodnotě 1 na obou vstupech neguje svůj aktuální stav. Vyrábí se pouze synchronní varianta.



Obrázek 2.9: Značka



Obrázek 2.10: Zapojení a tabulka

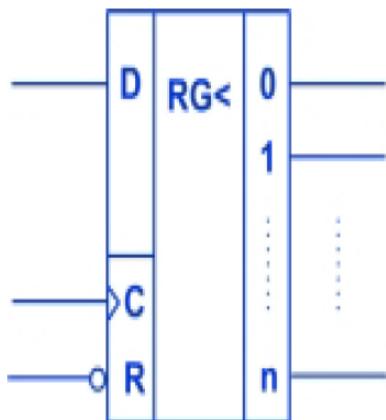
J	K	Q_i
0	1	0
1	0	1
0	0	Q_{i-1}
1	1	\bar{Q}_{i-1}

Typické sekvenční obvody

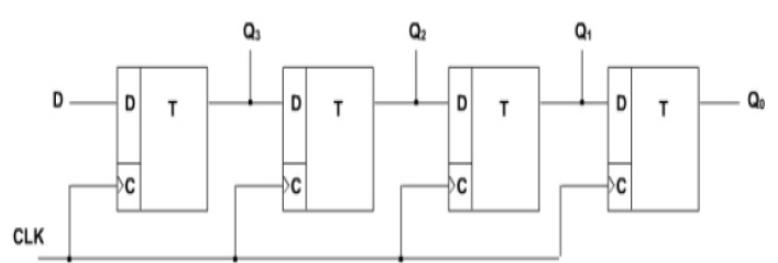
Obvody v číslicové technice a v elektronice jsou stavěny z malých stavebních kamenů. Tranzistory (nebo jiné součástky) tvoří hrdla logických obvodů, ty tvoří klopně obvody, a ty mohou tvořit další složitější obvody, a ty zase další, které později tvoří rozsáhlé zapojení, jako procesory.

Posuvný registr:

Posuvný neboli sériový registr posouvá informaci přivedenou na vstup D postupně po výstupech 0 až n. Jedním taktem signálu CLK se informace posune o jeden klopný obvod.



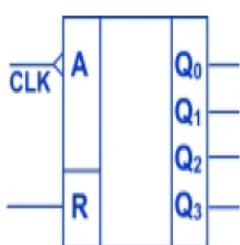
Obrázek 2.11: Značka



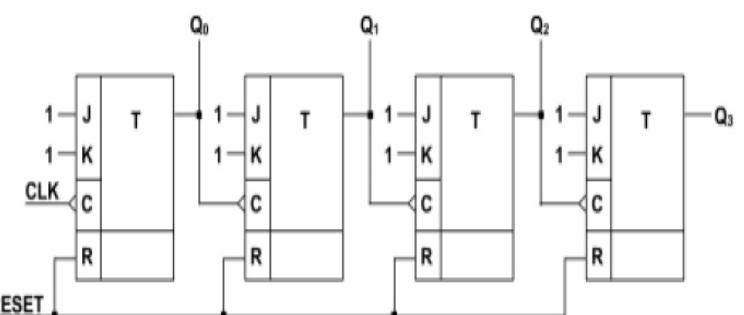
Obrázek 2.12: Zapojení registru pomocí obvodů RS

Čítač:

Čítač je zařízení, které počítá nebo odpočítává, kolikrát proběhla určitá událost nebo proces. Je synchronizován hodinovým kmitočtem.

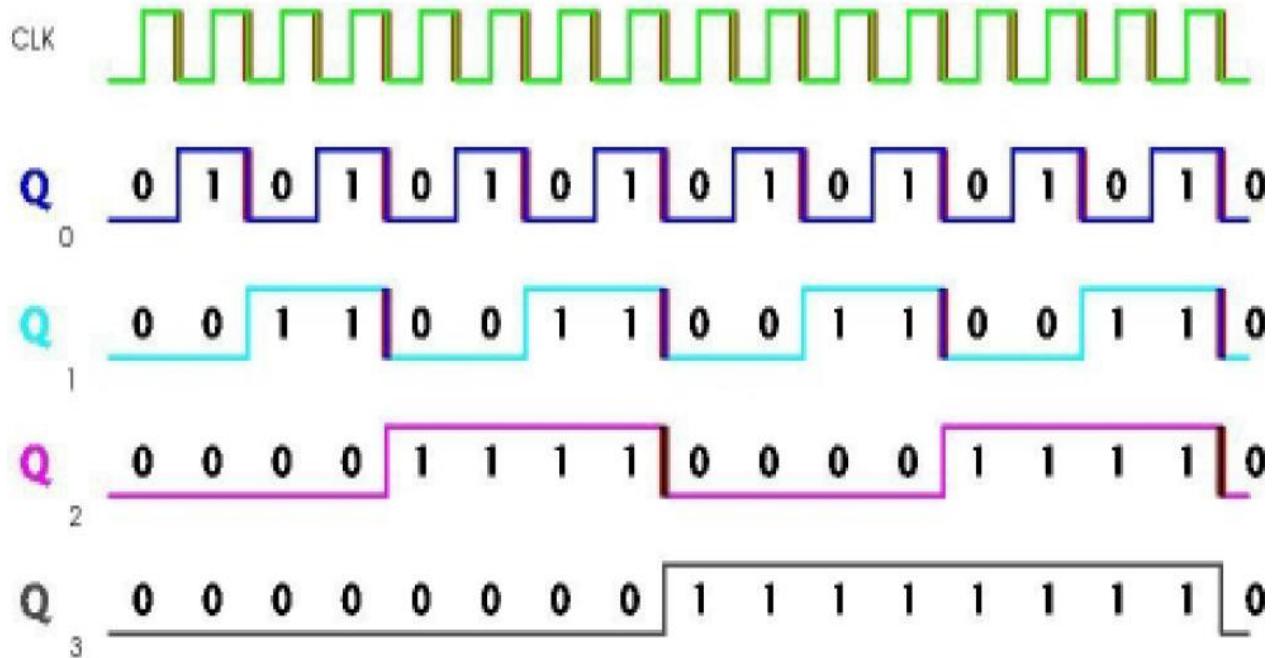


Obrázek 2.13: Značka



Obrázek 2.14: Zapojení dvojkového čítače 0 ...5 tvořené pomocí obvodů JK.

Průběh signálů na vývodech čítače.



Von Neumanova architektura

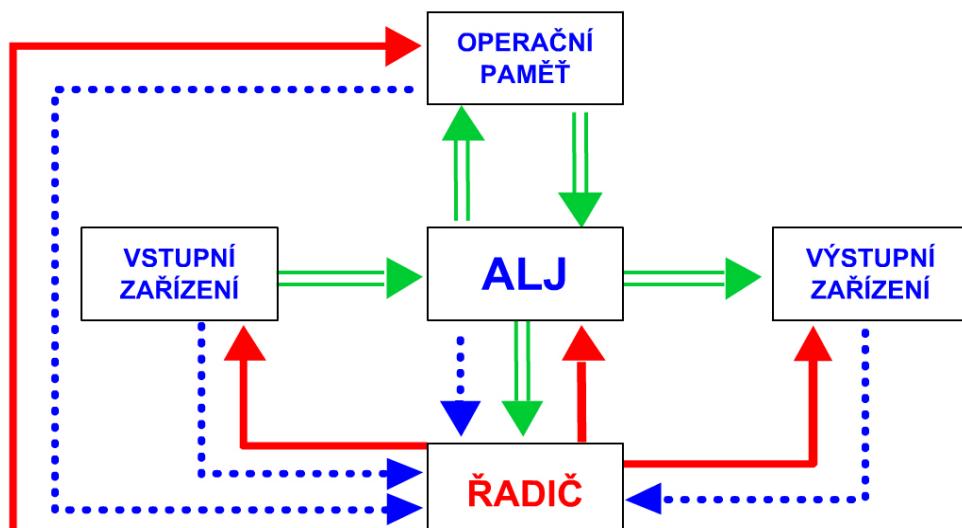
Von Neumannova architektura (koncept, či schéma) je ucelenou soustavou názorů a představ jak by měl počítač fungovat, z jakých částí se skládat, co a jak by tyto části měly dělat a jak by měly společně spolupracovat.

Tato koncepce vznikla kolem roku 1946, kde pan von Neumann stanovil principy *univerzálního počítače*, který bude pro široké použití. Základní prvky, které by měl počítač obsahovat:

1. Počítač obsahuje operační paměť, ALJ, řadič, vstupně-výstupní zařízení (V/V)
2. Předpis pro řešení úlohy je převeden do posloupnosti instrukcí
3. Údaje a instrukce jsou vyjádřeny binárně
4. Údaje a instrukce se uchovávají v paměti na místech označených adresami
5. Ke změně pořadí provádění instrukcí se používají instrukce podmíněného a nepodmíněného skoku
6. Programem řízené zpracování dat probíhá v počítači samočinně

JOHN VON NEUMANN :

1945



Principy práce procesoru

Procesory, jejich parametry a architektury:

Procesor je synchronní stroj řízený řadičem.

- **základní frekvence** = takt procesoru
- **strojový cyklus** = čas potřebný k zápisu (čtení) slova do (z) paměti (např. 2 takty)
- **instrukční cyklus** = čas potřebný pro výběr a provedení instrukce

Instrukce se skládá z operačního kódu a nepovinných adres operandů (1 nebo 2 operandy)

Součásti procesoru

- řadič nebo řídící jednotka, jejíž jádro zajišťuje řízení činnosti procesoru v návaznosti na povely programu, tj. načítání instrukcí, jejich dekódování (zjištění typu instrukce), načítání operandů instrukcí z operační paměti a ukládání výsledků zpracování instrukcí
- sada registrů (v řadiči) k uchování operandů a mezivýsledků. Řadič obsahuje celou řadu rychlých pracovních pamětí malé kapacity, tzv. registrů, které slouží k jeho činnosti. Registry dělíme na obecné (pracovní, univerzální) a řídící (např. registry adres instrukcí, stavové registry, indexregistry). Velikost pracovních registrů ke jednou ze základních charakteristik procesoru.
- jedna nebo více aritmeticko-logických jednotek (ALU = Arithmetic-Logic Unit), které provádějí s daty příslušné operace (logické, aritmetické)
- některé procesory obsahují jednu nebo více jednotek plovoucí čárky (FPU), které provádějí operace v plovoucí řádové čárce

Dělení procesorů

- podle délky operandu v bitech tj. šířka operandu, který je procesor schopen zpracovat v jednom kroku (16, 32, 64-bitový, ...)
- podle struktury procesoru RISC, CISC, jednoduché procesory (ochrana paměti), jednočipový mikropočítač (samostatně činný procesor, v periferiích), DSP (digitální signálový procesor zaměřený na zpracování signálu)
 - RISC (Reduced Instruction Set Computer) - redukovaná sada instrukcí, obsahuje hlavně jednoduché instrukce, větší instrukce vykonávány za pomoci mikroinstrukcí. Možnost lépe rozložit práci této větší instrukce příkladem může být pipeling = zřetězování.
 - CISC (Complex Instruction Set Computer) - plná sada instrukcí, větší instrukce vykonávány jako celek.
 - <http://radovan.bloger.cz/IT-internet/RISC---CISC-procesory>
- podle počtu jader

Frekvence jádra

Zásadním parametrem, který je u procesoru důležitý, je frekvence práce jeho jádra. Zdánlivě jde o banální záležitost, protože stačí spočítat kolik milionů, miliard instrukcí je procesor schopen vykonat za sekundu, tj. počet MIPS = milion operací za sekundu. Ovšem z praktického hlediska je počet MIPS např u osmibitového procesoru PIC (<http://mikrokontrolery-pic.cz/katalog/8bitove-mikrokontrolery-pic/>) a u procesoru Intel Pentium zcela nesrovnatelnou veličinou, protože instrukční sady těchto procesorů jsou zásadně odlišné a na výpočet v plovoucí čárce, který udělá Pentium v jediném taktu, může PIC potřebovat několik tisíc operací, zatímco jednoduché bitové operace zvládnou oba procesory v několika taktech.

Vyrovnnávací paměť procesoru

Důležitým faktorem celkového výkonu procesoru je tedy nyní i velikost vyrovnnávací paměti procesoru, která se označuje cache. Tato paměť bývá několika úrovňová, cache s nejrychlejším přístupem má nyní velikost 32-128 kB na jádro (Level 1 cache), další úroveň má nyní 256 kB-8MB. na jádro (Level 2 cache) a Level 3 cache 2-6 MB.

Cache

Cache je řazena mezi dva subsystémy s různou rychlostí a vyrovnnává tak rychlosť k přístupu k informacím. Cache dělíme na softwarovou a hardwarovou.

Hardwarová cache

- **Cache v řadicích jednotkách** - vyrovnnává rozdíl mezi nepravidelným předáváním/přebíráním dat počítače (sběrnice) a pravidelným tokem dat do/z magnetických hlav, jehož rytmus je dát rychlostí otáčení disku. U počítačů je cache elektronický obvod, tvořený z tranzistorů (ty tvoří bistabilní klopné obvodové) a její funkce je rozdílnou rychlosť mezi procesorem a operační pamětí. Vyšší rychlosťi je dosaženo použitím kvalitnějších tranzistorů (vyšší frekvence) než u operační paměti a cache je také blíže procesoru než operační paměť.
- **Cache paměť v procesoru:** ukládá kopie dat přečtených z adresy v operační paměti. Pokud při čtení obsahu slova z adresy v paměti je tato položka nalezena v cache paměti, je její obsah přečten z cache paměti a ne z operační paměti (z angličtiny cache hit). Mezi procesorem a cache pamětí se přenášejí jednotlivá slova, mezi cache pamětí a operační pamětí se přenášejí rámce slov o velikosti několikanásobku velikosti slova procesoru. Protože asi 90% operací procesoru je čtení paměti, většinou sekvenční, je tímto způsobem dosaženo větší propustnosti dat z operační paměti do procesoru, a tím i vyššího výpočetního výkonu. Vyrovnnávací paměť procesoru bývá dvojúrovňová (dvojstupňová), ale v dnešních procesorech můžeme nalézt i více úrovňové. Část paměti o malé kapacitě je přímo součástí procesoru a je stejně rychlá, jako vlastní procesor (značí se L1). Další paměť, která je pomalejší, ale s větší kapacitou, je mezi procesorem a operační pamětí a značí se L2. Cena pamětí stoupá spolu s rychlosťi paměti a kapacitou.

Softwarová cache

Programově vymezením určité části operační paměti pro potřeby vyrovnávací paměti (např. disková cache v operačním systému), nebo hardwarem paměťovými obvody (např. pro potřeby procesoru).

Fáze procesoru

- výběr instrukce z paměti provedení instrukcepřerušení, ...
- Výběr instrukcí je řízen registrém: čítač instrukcí; Program Counter (PC); Instruction Pointer (IP)
- Počítač pracuje ve dvojkovém doplňkovém kódu
- registry
 - A - střádač (Accumulator) - 8bitový
 - PC - čítač instrukcí - 16bitový
- paměť
 - adresovatelná jednotka = slabika
 - data - 8bitová

Přerušení

Přerušení (anglicky interrupt) je v metoda pro asynchronní obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí, vykoná obsluhu přerušení a pak pokračuje v předchozí činnosti. Původně přerušení sloužilo k obsluze hardwarem zařízení, která tak signalizovala potřebu obsloužit (tj. odebrat z vyrovnávací paměti data nebo naopak do ní další data nakopírovat, odtud označení vnější přerušení). Později byla přidána vnitřní přerušení, která vyvolává sám procesor, který tak oznamuje chyby vzniklé při provádění strojových instrukcí a synchronní softwarové přerušení vyvolávaná speciální strojovou instrukcí, která se obvykle používají pro vyvolání služeb operačního systému.

Obsluha přerušení

Přijde-li do procesu signalizace přerušení, jen v případě, že obsluha přerušení je povolena, je nejprve dokončena právě rozpracovaná úloha (strojová instrukce). Pak je na zásobník uložena adresa následující strojové instrukce, která by měla být zpracována, kdyby k přerušení nedošlo. Pak je dle tabulky přerušení vyvolána obsluha přerušení, která obslouží událost, která přerušení vyvolala. Obsluha přerušení je zpožďována tak, aby na jeho konci byl uveden stav procesoru do stavu jako před vyvoláním přerušení a aby přerušením nebyly ovlivněny jiné úlohy. Tuto funkci zpravidla zajišťuje software, zdůvodnuje vysokých rychlostí. Na konci přerušení je umístěna instrukce návrat (RET, někdy speciální IRET), která vyzvedne ze zásobníku návratovou adresu, a tak způsobí, že bude vyzvednuta následující strojová instrukce. Přerušená úloha, až na zpoždění, nepozná, že proběhla obsluha přerušení.

Tabulka přerušení umožňuje, aby procesor mohl rozlišit více různých přerušení (rozlišuje pomocí čísel), ke každému vyvolat odpovídající obsluhu přerušení (podprogramy) a aby šlo jednotlivé obsluhy umístit na libovolná místa v paměti. Obsluha přerušení je obvykle uložena v ovladači, který spolu s novým hardwarem zařízením do operačního systému instalujeme.

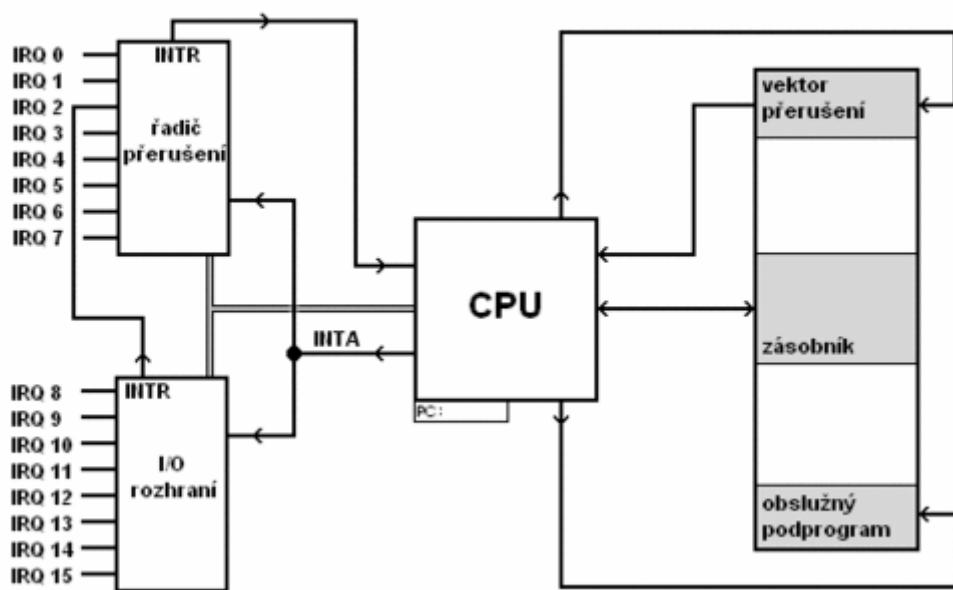
Typy přerušení

- **Vnější přerušení:** Vnější přerušení (též hardwarové přerušení) je označováno podle toho, že přichází ze vstupně-výstupních zařízení (tj. z pohledu procesoru přicházejí z vnějšku). Vstupně-výstupní zařízení má tak možnost si asynchronně vyžádat pozornost procesoru a zajistit tak svoji obsluhu ve chvíli, kdy to právě potřebuje bez ohledu na právě zpracovávanou úlohu. Vnější přerušení jsou do procesoru doručována prostřednictvím řadiče přerušení, což je specializovaný obvod, který umožňuje stanovit prioritu jednotlivých přerušení, rozdělovat je mezi různé procesory a další související akce.
- **Vnitřní přerušení:** Vnitřní přerušení vyvolává sám procesor, který tak signalizuje problémy při zpracovávání strojových instrukcí a umožňuje operačnímu systému na tyto události nevhodnějším způsobem reagovat. Jedná se například o pokus dělení nulou, porušení ochrany paměti, nepřítomnost matematického koprocesoru, výpadek stránky a podobně.
- **Softwarové přerušení:** Softwarové přerušení je speciální strojová instrukce (obvykle je jejich v procesoru k dispozici několik). Tento typ přerušení je na rozdíl od druhých dvou typů synchronní, je tedy vyvoláváno zcela záměrně umístěním příslušné strojové instrukce přímo do prováděného programu. Jedná se o podobný způsob, jako vyvolání klasického podprogramu (podprogramem je zde ISR (= obsluha přerušení) uvnitř operačního systému), avšak procesor se může zachovat jinak. Instrukce softwarového přerušení se proto využívá pro vyvolání služeb operačního systému z běžícího procesu (tzv. systémové volání). Uživatelská úloha tak sice nemůže skočit do prostoru jádra operačního systému, ale může k tomu využít softwarové přerušení. Při použití privilegovaného režimu může softwarové přerušení aktivovat privilegovaný stav.

Popis implementace přerušení

Průběh HW přerušení:

1. Vnější zařízení vyvolá požadavek o přerušení
2. I/O rozhraní vyšle signál IRQ na řadič přerušení
3. Řadič přerušení vygeneruje signál INTR - "někdo" žádá o přerušení a vyšle ho k procesoru
4. Procesor se na základě maskování rozhodne obsloužit přerušení a signálem INTA se zeptá, jaké zařízení žádá o přerušení
5. Řadič přerušení identifikuje zařízení, které žádá o přerušení a odešle číslo typu přerušení k procesoru
6. Procesor uloží stavové informace o právě zpracovávaném programu do zásobníku
7. Podle čísla typu příchozího přerušení nalezne ve vektoru přerušení adresu příslušného obsluženého podprogramu
8. Vyhledá obslužný podprogram obsluhy přerušení v paměti a vykonává ho
9. Po provedení obslužného programu opět obnoví uložené stavové informace ze zásobníku a přerušený program pokračuje dál



Průběh softwarového přerušení

- 1 Po provedení obslužného programu opět uložené stavové informace ze zásobníku a přerušený program pokračuje dál.
- 2 Zakáže se další přerušení.
- 3 Procesor zjistí vektor přerušení (podle operandu)
- 4 Nalezne obslužný program a vykonáho
- 5 Po návratu z podprogramu obnoví uložené stavové informace o přerušení programu.

Operační systémy

Architektury operačních systémů, rozhraní operačních systémů. Procesy a vlákna, synchronizace procesů a vláken, uváznutí a metody ochrany proti uváznutí. Plánování v operačních systémech. Správa a plánování činnosti procesorů.

Architektura OS

Funkce a součásti OS

- **správa procesů**
 - vytváření, ukončování, potlačování a obnovování procesů
 - mechanismy pro synchronizaci procesů, komunikaci mezi nimi a pro detekci a řešení uváznutí
- **správa operační paměti**
 - znalost, která část paměti je využívána kterým procesem
 - alokace (přidělení) a dealokace (uvolňování) paměti podle požadavků procesu
 - rozhodování o tom, který proces kdy zavést do paměti
 - virtualizace paměti
- **správa souborů**
 - vytváření a rušení souborů a adresářů
 - základní operace pro manipulaci se soubory a adresáři (čtení a zápis do souborů, seznam souborů v adresáři)
- **správa I/O zařízení**
 - snaha o skrytí specifik jednotlivých zařízení, komunikace mezi hardware a operačním systémem zajišťují speciální programy nazývané ovladače
 - některé OS zpřístupňují zařízení ve formě speciální souborů (např. v UNIXU jsou to soubory v adresáři /dev)
- **správa sekundárních pamětí**
 - typické sekundární paměti jsou disky a jejich správa je zajištěna formou souborového systému
 - typické aktivity: správa volného místa, přidělování místa, plánování činnosti disku
- **správa síťových služeb**
 - podpora pro síťové protokoly a hardware
 - nástroje pro sdílení zdrojů (distribuovaný systém souborů...)
- **ochranný systém**
 - řízení přístupu procesů a uživatelů ke zdrojům (např. zajištění, že proces může používat pouze adresy svého adresového prostoru, zabránění procesům spouštět privilegované instrukce)
- **interpret příkazů (shell)**
 - rozhraní uživatele na služby operačního systému
 - může být ve formě příkazové řádky (Command Line Interface - CLI) nebo ve formě grafického rozhraní
 - může být součástí jádra OS nebo jako samostatný program
- **ochrana a reakcia na chyby**
 - OS musí reagovať na chyby (hardware/software) tak aby sa obmedzil dopad na bežiace aplikacie
 - typy reakcii: ukoncenie procesu ktorý sposobil chybu, zopakovanie operacie, oznamenie informacie o chybe aplikacii/uzivatelovi
- **zbieranie statistik a informacii o systeme**

Druhy operačních systémů podle architektury jádra

- **Monolitické jádro**

- veškerý kód jádra běží ve stejném paměťovém prostoru (kernel space) a jednotlivé části jádra jsou silně provázány
- výhody
 - podstatně vyšší výkon než u mikrojádra
- nevýhody:
 - protože části jádra sdílení stejný paměťový prostor, může chyba v jedné části zablokovat část jinou nebo dokonce shodit celé jádro
 - silna previazanost a zavislost medzi jednotlivymi súcastami jadra
- některá monolitická jádra podporují zavádění modulů za běhu systému a není tak například nutné restartovat systém při přidání nového hardware
- příklady: UNIX, Linux

- **Mikrojádro**

- jádro systému je velmi malé a obsahuje pouze nezákladnejší funkce (např. správa prerušení, plánování procesů) a ostatní potřebné části jádra (např. správa souborového systému, ovladače zařízení, síťové služby) běží v uživatelském prostoru (user space) a jsou nazývány **servery**. Mikrokernel zaistuje iba komunikaci medzi serverami pomocou zasielania sprav.
- výhody
 - snadnejší programování systému pomocí rozdelení na samostatné logické celky a vyšší přehlednost kódu
 - rozsiritelnosť - jednoduše pridanie noveho servera
 - flexibilita - možnosť odobrat nepotrebné súčasti OS a upravit si systém podľa potreby
 - prenositelnosť - pri portovaní na nový procesor/architekturu stačí upraviť len mikrokernel
- nevýhody:
 - podstatne menší výkon než u monolitického jádra – zejména v oblasti změn kontextu a meziprocesorové komunikace (posielanie sprav je výrazne pomalsie ako zavolanie procedury)
- příklady: GNU Mach, Symbian OS, MINIX

- **Hybridní jádro**

- kombinuje vlastnosti monolitického jádra a mikrojádra za účelem získání výhod obou přístupů
- podobné mikrojádru - v adresním prostoru běží jen některé služby (např. souborový systém) aby se dosáhlo nižší režie v porovnání s mikrojádrem a ostatní služby běží v uživatelském prostoru
- příklady: Windows NT (jádro na kterém běží operační systémy Windows) a XNU (jádro systému Mac OS X)

Rozhraní OS

- Rozhraní mezi systémovým procesom a operačním systémem zajišťují tzv. systémová volání
- volania sú typicky implementované pomocou softveroveho prerušenia: na zasobník sa vložia argumenty a do specifikovaného registra sa vloží číselný kód systemovej volania. Po vyvolaní prerušenia sa procesor prepne do privilegovaneho režimu a riadenie sa predá kernelu.
- vysie programovacie jazyky volajú systémové funkcie pomocou knížnic, ktoré tvoria vrstvu nad skutočnými volaniami (napr. POSIX funkcie v jazyku C – open, write...)
- rôzne OS a HW platformy mívajú rôzne zpôsoby, ako volať OS služby – napr. POSIX v oblasti UNIXu alebo Windows API (dříve nazývané ako Win32) v oblasti Windows

Procesy a vlákna

Proces

- spuštěný program, tedy program zavedený do operační paměti (v angličtině se někdy používá synonymum task. Výraz task se může použít i pro vlákno)
- každý proces má v operační paměti tyto části:
 - zásobník – zde se ukládají informace týkající se provádění podprogramů (lokální proměnné funkcí, ukládání návratové adresy při volání funkce...)
 - datová sekce – obsahuje statická data programu a haldu (dynamicky definovaná data)
 - text programu – posloupnost instrukcí
- procesy jsou v operačním systému uspořádány hierarchicky, tj. každý proces má svého rodiče (okrem prvého procesu, např. v Linuxu je to proces 0 (idle task)) a případně své potomky
- význam procesu:
 - operační systém maximalizuje využití procesoru a minimalizuje dobu odpovědi procesu prokládáním běhu procesů (multitasking)
 - operační systém může přidělovat zdroje procesům podle vhodné politiky (priorita, vzájemná výlučnost)

Stavy procesů

- nový (**new**) – právě vytvořený proces
- připravený (**ready**) – proces čeká na přidělení času procesoru
- běžící (**running**) – některý procesor právě vykonává instrukce programu
- čekající (**blocked**) – proces čeká na dokončení nějaké události
- ukončený (**terminated**) – proces ukončil svoje provádění
- případně také odložený připravený (**ready/suspend**) a odložený čekající (**ready/blocked**) (viz odkládání procesů)

Informace o procesu

- operační systém si ukládá informace důležité pro správu procesu ve struktuře označované jako Process Control Block (PCB)
- informace obvykle obsažené v PCB:
 - identifikacne udaje
 - **PID** (Process ID) – unikátní identifikátor procesu
 - **PID rodiče**
 - **ID uzivatela**
 - stav procesů
 - obsah registrů procesoru a Program Counter (adresa intrukce procesu, která má být vykonána)
 - ukazatele na vrchol zasobníkov
 - informace potřebné pro správu paměti
 - informacie o využívanych zdrojoch
 - seznam otevřených souborů
 - účtovací informace – např. kolik času procesoru už proces spotřeboval
 - informacie planovaca
 - stav procesu (ready, running, blocked, ...)
 - priorita procesu
 - udalosti na ktore proces caka
 - ine, v zavislosti od planovacieho algoritmu (napr. doba cakania)

Přepínání procesů

- přepínání procesů je klíčovým konceptem multiprogramování (multitasking)
- prepinanie procesov zvyčajne nastava pri:
 - preruseni od casovaca
 - I/O preruseni
 - vypadku stranky - proces pristupuje na adresu ktorá sa nachadza na odswapovanej stranke
- akt přepínání procesů na procesoru se označuje jako **Context Switching**
 - operační systém musí při přepnutí procesu uložit stav procesu (včetně uložení čítače instrukcí), který je z procesoru odebírána a zavést stav nově běžícího procesu
 - přepínání kontextu představuje režijní ztrátu (během přepínání kontextu procesor nevykonává žádnou užitečnou činnost)
 - program, který řídí přepínání kontextů se nazývá dispečer nebo plánovač (angl. **scheduler**) a je součástí jádra OS

Vlákna

- Vlákno je objekt, ktorý vzniká v rámci procesu, je viditeľný pouze uvnitř procesu a je charakterizovaný svým stavem
- Vlákno predstavuje sekvenčiu instrukcií ktorá možno byť spravovaná planovačom a vykonávaná procesorom. Vlákno na rozdiel od procesu nevlastní zdroje (resources), tie sú asociované s procesom ku ktoremu dané vlákno patrí.
- každé vlákno si udržuje tyto vlastnosti současťí:
 - zásobník
 - program counter
 - registry
 - TCB (Thread Context Block)
- vlákno môže pribudovať k pamäti a ostatným zdrojom procesu (napr. súbor otvorený jedným vláknom mají k dispozícii všetky vlákna téhož procesu)
- jakmile jedno vlákno změní nelokální obsah (tj. obsah mimo zásobník), všetky ostatné vlákna téhož procesu to vidia
- výhody a využití vláken:
 - využití multiprocesorových strojov – vlákna jedného procesu môžu bieť na rôznych CPU
 - vlákno sa vytvorí a ukončí rýchleji než proces a i prepnutie medzi vlákny je rýchlejší
 - jednodušší programovanie – není potrebné používať prostredky meziprocesorovej komunikace
- hlavnou nevýhodou používania vláken je nutnosť synchronizovať dátá tak, aby sa zachovala konzistentnosť dát (musí sa zabrániť současnej modifikácii dát dvoma vláknenami) a aktualnosť dát (aby vlákno nepoužívalo neaktualné kopie dát)
- ukončenie procesu ukončuje všetky vlákna, ktorá v jeho rámci existujú
- Vlákna môžu byť implementované doma sposobmi:
 - vlákna na užívateľskej úrovni (User-Level Threads - ULT):
 - správa vláken sa provádzí prostredníctvom vláknové knihovny na úrovni užívateľskej aplikácie a jádro o jejich existencii neví a manipuluje s celými procesami
 - výhody:
 - prepojovanie medzi vlákny nepožaduje volanie služieb jádra (vlákna môžu byť používané aj nad OS ktorá samá vlákna nepodporuje)
 - prepnutie medzi vlákny je špecifické pre konkrétnu aplikáciu, ktorá si volí pro seba najvhodnejší algoritmus
 - nevýhody:
 - neefektivnosť, používa sa medzivrstva a program nemôže využívať priamo planovač OS
 - ak niektoré vlákno zavola službu jádra, tak je zablokován celý proces dokud sa služba nedokončí
 - jádro môže pridelenie procesora pouze procesu a dvom vláknam stejného procesu tak nemôžu bieť na dvoch procesorech

- vlákna na úrovni jádra (Kernel-Level Threads - KLT), používají všechny moderní OS:
 - správu vláken podporuje jádro a ty se spravují pomocí API jádra
 - výhody:
 - jádro může současně plánovat běh více vláken stejného procesu na více procesorech
 - k blokování dochází na úrovni jádra (tj. není blokován celý proces)
- příklady:
 - **GUI aplikace** - jedno vlákno vykresluje uživatelské rozhraní aplikace a druhé vlákno provádí uživatelem požadovaný úkol – aplikace je tak responsivní i pokud provádí nějakou náročnější činnost
 - **sítový server** který musí vyřizovat řadu požadavků klientů vytvoří nové vlákno pro každého připojeného uživatele

Synchronizace procesů a vláken

- Typové úlohy související s paralelním během programů:
 - **synchronizace běhu procesů** – jeden proces čeká na událost z druhého procesu
 - **komunikace mezi procesy** – způsob koordinace různých aktivit, může docházet k uváznutí (každý proces čeká na zprávu od nějakého jiného procesu) a stárnutí (dva procesy si opakovaně posílají zprávy zatímco třetí proces čeká na zprávu nekonečně dlouho)
 - **sdílení prostředků** – procesy používají a modifikují sdílená data, přičemž některé operace musí být vzájemně výlučné (např. operace zápisu mezi sebou nebo operace zápisu s operacemi čtení)

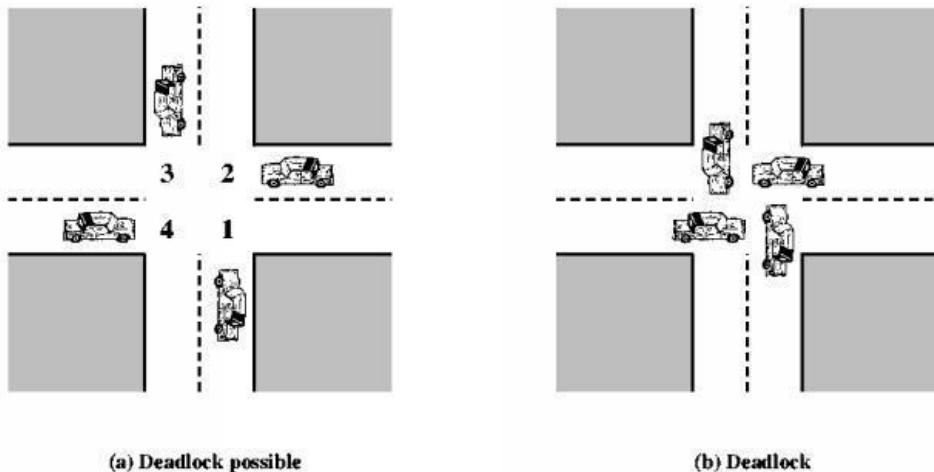
Race Condition a kritická sekce

- chyba v systému nebo procesu, ve kterém jsou výsledky nepředvídatelné a závisí na pořadí nebo načasování jednotlivých operací
- dochází k ní pokud více procesů přistupuje ke sdíleným zdrojům a manipuluje s nimi, přičemž konečnou hodnotu zdroje určuje poslední z procesů, který zdroj po manipulaci opustí
- segment kódu, v kterém proces přistupuje ke sdíleným zdrojům se nazývá kritická sekce
- podmínky řešení problému kritické sekce:
 - **podmínka bezpečnosti (safety)** – jestliže nějaký proces provádí svou kritickou sekci, nemůže svou kritickou sekci se stejným zdrojem provádět jiný proces
 - **podmínka trvalosti postupu (progress)** – jestliže žádný proces neprovádí svou kritickou sekci sdruženou s určitým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do své kritické sekce sdružené s tímto zdrojem, tak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho

- **podmínka spravedlivosti (fairness)** – pokud jeden proces usiluje o vstup do kritické sekce, tak mu v tom nesmí být zabráněno tím, že se v kritické sekci neustále střídají jiné procesy – tedy musí existovat limit, kolikrát může být povolen vstup procesu do kritické sekce v případě, že do kritické sekce chtějí vstoupit jiné procesy
- kategorie řešení kritické sekce
 - **čistě softwarové řešení** – algoritmy, jejichž správnost se nespoléhá na žádné další předpoklady, pouze vzájemná výlučnost R/W operací na paměti, dochází k aktivnímu čekání (busy waiting)
 - **hardwareové řešení** – pomocí speciální instrukce procesoru, rovněž dochází k aktivnímu čekání
 - **softwarové řešení zprostředkované operačním systémem** – služby a datové struktury zajišťuje operační systém (mutexy, semafory, zasílání zpráv...)
 - **semafor**
 - umožnuje kooperaci viacerých procesov na zaklade zasielania jednoducheho signálu
 - semafor je celociselná premenna, nad ktorou sú definované 3 operacie:
 - **inicializacia** - nastavi semafor na nezápornu hodnotu (počet volných zdrojov)
 - **wait** - zníží hodnotu semaforu o 1. Ak je hodnota po znížení zaporná, znamená to, že žiadny z pozadovaných zdrojov nie je k dispozícii a proces je zablokovany
 - **signal** - inkrementuje hodnotu semaforu (uvolní zdroj) a ak je zaporná (existuje cakajúci proces) odblokuje cakajúci proces
 - **monitor** je softverový modul ktorý možno byť používany iba jedným procesom
 - ide o špeciálnu konštrukciu konkrétneho programovacieho jazyka
 - synchronizácia prebieha pomocou **podmienených premenných** ktoré sú súčasťou monitoru a sú prístupné len zvnútora monitoru a dvoch operácií:
 - **wait(c)** - zablokuje proces pokiaľ nebude splňena podmienka reprezentovaná podmienenkou premennou **c**. Monitor je prístupný pre ďalší proces.
 - **signal(c)** - odblokuje proces ktorý caka na podmienku **c**. Ak je takých procesov viac, odblokuje sa jeden z nich, ak taký proces neexistuje nic sa nestane.

Uváznutí a metody proti uváznutí

Problém uváznutí (deadlock) – existuje množina blokovaných procesů, kde každý proces vlastní nějaký prostředek a čeká na zdroj držený jiným procesem z této množiny



Charakteristika uváznutí

- k uváznutí dojde, pokud začnou zároveň platit tyto podmínky:
 - **vzájemné vyloučení (mutual exclusion)** – sdílený zdroj může v jednom okamžiku používat pouze jeden proces
 - **ponechání si zdroje a čekání na další (hold and wait)** – proces vlastnící jeden zdroj muze čekat na zdroj, který drží jiný proces
 - **bez předbíhání (no preemption)** – zdroj lze uvolnit pouze procesem, který ho vlastní a který se jej dobrovolně vzdá
 - **kruhové čekání (circular wait)** – existuje seznam čekajících procesů (P_0, P_1, \dots, P_n) takový, že P_0 čeká na uvolnění zdroje vlastněného P_1 , P_1 čeká na uvolnění zdroje drženého P_2 , ..., P_n čeká na uvolnění zdroje drženého P_0

Metody ochrany proti uváznutí

- **prevence** – je třeba zajistit, že se systém nikdy nedostane do stavu uváznutí tak, že se zruší platnost některé nutné podmínky
- **obcházení uváznutí** – detekce potenciální možnosti vzniku uváznutí a nepřipuštění takového stavu tak, že se zamezí současně platnost všech podmínek – prostředek se nepřidělí, pokud hrozí uváznutí (může docházet ke stárnutí). Na rozdiel od prevencie sa zrusi platnosť jednej z podmienok iba pre konkretny prípad ak system vyhodnoti že hrozi deadlock.

- **detekcia uváznutí** – uváznutí povolíme, ale jeho vznik detekujeme (a řešíme, obvykle sa jeden z procesov zabije, pretože deadlock je zriedkava udalosť)
- **ignorování hrozby uváznutí** – uváznutí je věc aplikace a ne systému – tento způsob řešení většinou volí OS

Metody prevence uváznutí

- nepřímé metody
 - **ne vzájemné vyloučení** - nepoužívání sdílených zdrojů, virtualizace prostředků (např. Tiskárny), obecně se nepoužívá
 - **ne postupná alokace** – proces musí požádat o všechny požadované zdroje a obdržet je ještě před tím než se spustí jeho běh nebo o ně smí žádat pouze pokud žádný zdroj nevlastní
 - **ne předbíhání** – jestliže proces držící nějaké zdroje požaduje přidělení dalšího zdroje, který mu nelze přidělit okamžitě (není aktuálně volný), pak jsou mu odebrány všechny jím držené zdroje a proces je obnoven ve chvíli, kdy může získat původně držené zdroje a nově požadované zdroje
- přímé metody
 - **zabránění zacyklení pořadí** – zavede se úplné uspořádání typů zdrojů a každý proces požaduje prostředky v pořadí daném vznutí pořadím výčtu

Obcházení uváznutí

- systém musí mít nějaké **dodatečné apriorní informace**
- nejjednodušší a nejužitečnější model požaduje, aby každý proces udal maxima počtu prostředků každého typu, které může požadovat
- algoritmus, který řeší obcházení pak dynamicky zkouší, zda stav systému (definován počtem dostupných a přidělených zdrojů a maximem žádostí procesů) zaručuje to, že se procesy nedostanou do cyklické fronty čekání

Detekce a obnova uváznutí

- používá se graf čekání (wait-for graph) kde jednotlivé procesy představují uzly grafu a hrana z uzlu A do B existuje, pokud proces A čeká na proces B, periodicky se poté provádí algoritmus, který v grafu hledá cykly (složitost $O(n+v)$, kde n je počet uzlů a v je počet hran (Tarjan's strongly connected components algorithm))
- metody obnovy:
 - **ukončení procesu** – násilně se ukončují uváznuté procesy dokud se neodstraní cyklus (pořadí je dáno např. prioritou procesu nebo prostředky, které proces použil)
 - **nové rozdělení prostředků** – návrat do některého bezpečného stavu (stav, kdy je možné alokovat zdroje každému procesu v nějakém pořadí a nenastane uváznutí) a restart procesu z tohoto stavu

Plánování v OS

Fronty plánování procesů

- planovac si udrzuje seznamy uloh vo frontach
- procesy mezi témoto frontami migrují
- druhy front:
 - **fronta všech úloh** – seznam všech procesů v systému
 - **fronta pripravených procesů** – seznam procesů uložených v hlavní paměti a připravených k běhu
 - **fronta na přidělení zařízení** – seznam procesů čekajících na I/O operace
 - **seznam odložených procesů** – seznam procesů čekajících na přidělení místa v hlavní paměti
 - **fronta na semafor** – seznam procesů, které čekají na synchronizační událost

Plánovače v OS

- **dľouhodobý plánovač** (strategický plánovač, job scheduler)
 - rozhoduje ktoré programy sú prijaté systémom na spustenie
 - je vyvolávaný pomérne zřídka a nemusí byt rychlý – typicky pri ukončení jednoho procesu
 - rozhodne o tom, ktorou úlohu dále vybrať k zavedeniu do paměti a spuštění
- **krátkodobý plánovač** (operační plánovač, dispečer, dispatcher)
 - reprezentuje správu procesů – vybírá proces, který poběží na právě uvolněném procesoru
 - je vyvolávaný velice často (desítky, stovky milisekund) a musí proto být rychlý
 - planovanie prebieha v prípadoch keď nastane udalosť ktorá sposobi cakanie procesu alebo predstavuje prilezitosť na vykonanie preemption.
 1. proces přechází ze stavu běžící do stavu čekající
 2. proces přechází ze stavu běžící do stavu připravený
 3. proces přechází ze stavu čekající do stavu připravený
 4. proces končí
 - případy 1 a 4 se označují jako nepreemptivní plánování (plánování bez předbíhání)
 - případy 2 a 3 se označují jako preemptivní plánování (plánování s předbíháním)
- **střednědobý plánovač**
 - je súčasťou funkcie swapovania
 - vybírá, který proces lze zařadit mezi odložené procesy a který odložený proces lze zařadit mezi připravené procesy
 - souvisí s odkládáním procesů (swapping) – pokud je v operační paměti příliš mnoho procesů, je třeba uvolnit paměť dočasným odložením procesu na disk
 - náleží částečně i do správy paměti - fyzický adresový prostor (FAP) je omezený a odkládání procesů uvolňuje paměť

Algoritmy plánování

Kritéria plánování a optimalizace

- využití CPU a ostatných zdrojov – cílem je udržení CPU v kontinuální činnosti
- propustnost – počet procesů, které dokončí svůj běh za jednotku času
- doba obrátky – doba potřebná pro provedení konkrétního procesu
- doba čekání – doba, po kterou proces čekal ve frontě připravených procesů
- doba odpovědi – doba, která uplyne od okamžiku zadání požadavku do doby první reakce
- prioritá procesu
- spravedlivost

Algoritmus FCFS (First Come, First Served)

- Nepreemptivní algoritmus,
- procesy jsou řazeny ve FIFO frontě - jakmile se uvolní procesor, první proces ve frontě je přiřazen procesoru a předchozí proces se zařadí na konec fronty
- jednoduchý na implementaci a vhodný pro **dlouhé** procesy
- nevýhodou může být dlouhá průměrná doba čekání – krátké procesy jsou ovlivněny „konvojovým efektem“ (v jejich provedení brání dlouhé procesy)

Round Robin

- Každý proces dostává CPU na malou jednotku času (desítky až stovky ms) a po jejím uplynutí je běžící proces nahrazen nejstarším procesem ve frontě a sám se zařadí na její konec
- efektivita silně závisí na velikosti časové jednotky přidělované procesům
- znevýhodněny su casto cakajuce procesy, ktore nevyuziju celu svoju pridelenu casovu jednotku, napriklad procesy intenzivne pracujuce s I/O

Prioritní plánování

- Každému procesu je přiděleno prioritní číslo (obvykle nejvyšší prioritě odpovídá nejnižší číslo) a CPU se poté přiděluje procesu s největší prioritou
- dvě varianty:
 - nepreemptivní – jakmile proces získá přístup k CPU, tak nemůže být předběhnut
 - preemptivní – proces může být předběhnut procesem s vyšší prioritou
- problém stárnutí – procesy s nižší prioritou se nemusí nikdy provést, řešením je „zrání“, kdy se priorita s postupem času zvyšuje

Shortest-Job-First

- Vybírá se proces s nejkratším požadavkem na CPU a je tedy nutné znát délku příštího požadavku pro každý proces (velice zřídka se zná dopředu, proto se používá exponenciální průměrování na základě historie velikostí dávek procesu)
- používají se dvě varianty:
 - nepreemptivní – jakmile se CPU předá procesu, tak už nemůže být předběhnut jiným procesem dokud svou dávku nedokončí
 - preemptivní (nazývaná takze **Shortest-Remaining-Time-First**) – jakmile se ve frontě připravených procesů objeví proces s délkou dávky CPU, která je kratší než je doba zbývající k dokončení dávky právě běžícího procesu, tak je stávající proces nahrazen novým

Korekcia prebehla podľa knihy (dostupne v kniznici FI):

STALLINGS, William. *Operating systems: internals and design principles*. 4th ed. Upper Saddle River: Prentice-Hall, c2001, xviii, 779 s. ISBN 01-303-1999-6.

Práce s pamětí, vstupy a výstupy

Paměťová hierarchie. Práce s pamětí, logický a fyzický adresový prostor, správa paměti, virtualizace paměti, segmentace, stránkování. Ovládání vstupů a výstupů, správa a plánování činnosti V/V zařízení.

Paměťová hierarchie

Paměť: zařízení, které slouží k ukládání programů a dat, s nimiž počítač pracuje

Dělení pamětí:

1. ***registry:***

- paměťová místa na čipu procesoru
- jsou používány pro krátkodobé uchování právě zpracovávaných informací

2. ***vnitřní (interní):***

- paměti osazené většinou uvnitř základní jednotky
- realizovány pomocí polovodičových součástek
- jsou do nich zaváděny právě spouštěné programy (nebo alespoň jejich části) a data, se kterými tyto programy pracují

3. ***vnější (externí):***

- paměti realizované většinou za pomoci zařízení používajících výměnná média v podobě disků či magnetofonových pásek
- záznam se provádí většinou na magnetickém nebo optickém principu
- slouží pro dlouhodobé uchování informací a zálohování dat

Parametry pamětí:

1. ***Kapacita:***

- množství informací, které je možné do paměti uložit

2. ***Přístupová doba:***

- doba, kterou je nutné čekat od zadání požadavku, než paměť zpřístupní požadovanou informaci

3. ***Přenosová rychlosť:***

- množství dat, které lze z paměti přečíst (do ní zapsat) za jednotku času

4. ***Statičnost / dynamičnost:***

- ***statické paměti:*** uchovávají informaci po celou dobu, kdy je paměť připojena ke zdroji elektrického napětí
- ***dynamické paměti:*** zapsanou informaci mají tendenci ztrácet i v době, kdy jsou připojeny k napájení, informace v takových pamětech je tedy nutné neustále periodicky oživovat, aby nedošlo k jejich ztrátě

5. **Destruktivnost při čtení:**

- **destruktivní při čtení:** přečtení informace z paměti vede ke ztrátě této informace, přečtená informace musí být následně po přečtení opět do paměti zapsána

- **nedestruktivní při čtení:** přečtení informace žádným negativním způsobem tuto informaci neovlivní

6. **Energetická závislost / nezávislost:**

- **energeticky závislé:** paměti, které uložené informace po odpojení od zdroje napájení ztrácejí

- **energeticky nezávislé:** paměti, které uchovávají informace i po dobu, kdy nejsou připojeny ke zdroji elektrického napájení

7. **Přístup:**

- **sekvenční:** před zpřístupněním informace z paměti je nutné přečíst všechny předcházející informace

- **přímý:** je možné zpřístupnit přímo požadovanou informaci

8. **Spolehlivost:**

- střední doba mezi dvěma poruchami paměti

9. **Cena za bit:**

- cena, kterou je nutno zaplatit za jeden bit paměti

Vnitřní paměti:

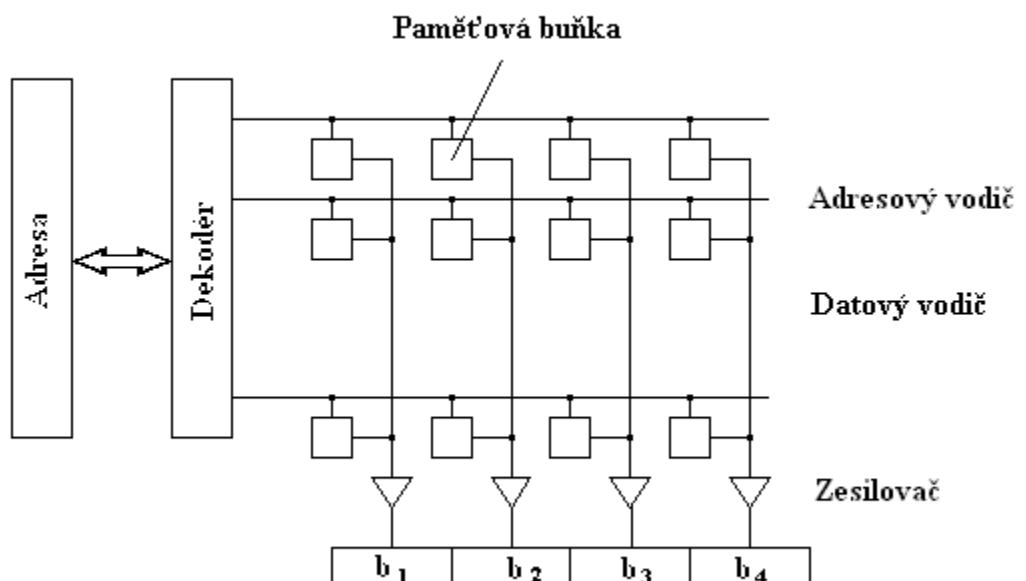
- Zapojeny jako matice paměťových buněk

- Každá buňka má kapacitu jeden bit

- Jedna paměťová buňka tedy může uchovávat pouze hodnotu logická 1 nebo logická 0

- V případě vnitřních pamětí s menší kapacitou je možné jejich strukturu znázornit následujícím schématem:

- Přístup do paměti: Při přístupu do paměti (čtení nebo zápis) je vždy udána adresa paměťového místa, se kterým se bude pracovat.



Tato adresa je přivedena na vstup dekodéru. Dekodér pak podle zadané adresy vybere jeden z adresových vodičů a nastaví na něm hodnotu logická 1. Podle toho, jak jsou zapojeny jednotlivé paměťové buňky na příslušném řádku, který byl vybrán dekodérem, projde resp. neprojde hodnota logické jedničky na datové vodiče. Informace je dále na koncích datových vodičů zesílena zesilovačem. V případě, že hodnota logická jedna projde přes paměťovou buňku, obdržíme na výstupu hodnotu bitu 1. V opačném případě je na výstupu hodnota bitu 0. Zcela analogický je postup i při zápisu hodnoty do paměti. Opět je nejdříve nutné uvést adresu paměťového místa, do kterého se bude zapisovat. Dekodér vybere adresový vodič příslušný zadané adrese a nastaví na něj hodnotu logická 1. Dále se nastaví hodnoty bitů b1 až b4 na hodnoty, které se budou do paměti ukládat. Tyto hodnoty jsou potom uloženy do paměťových buněk na řádku odpovídajícím vybranému adresovému vodiči.

Rozdělení vnitřních pamětí:

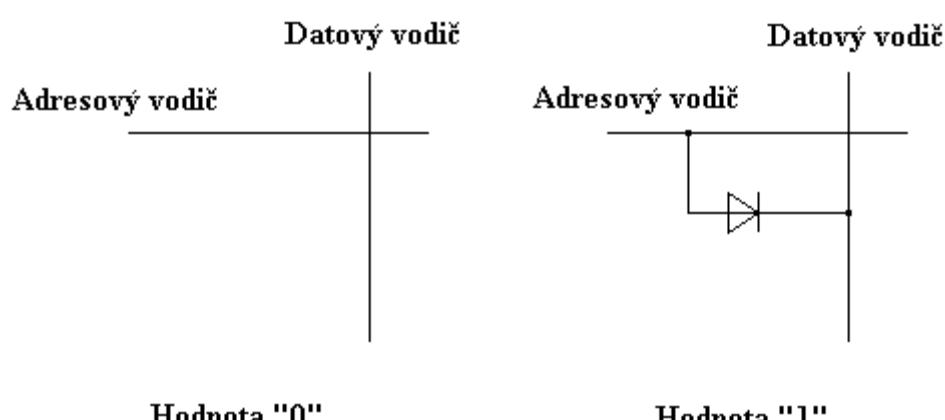
1. *paměti ROM*

- ROM - Read Only Memory
- Paměti určené pouze pro čtení uložených informací
- Informace jsou do těchto pamětí pevně zapsány při jejich výrobě
- Potom již není možné žádným způsobem jejich obsah změnit
- Jedná se o statické a energeticky nezávislé paměti
- Může být realizována jako dvojice nespojených vodičů a vodičů propojených přes polovodičovou diodu. V prvním případě nemůže žádným způsobem hodnota logická jedna přejít z adresového vodiče na vodič datový. Jedná se tedy o buňku, ve které je permanentně uložena hodnota 0.

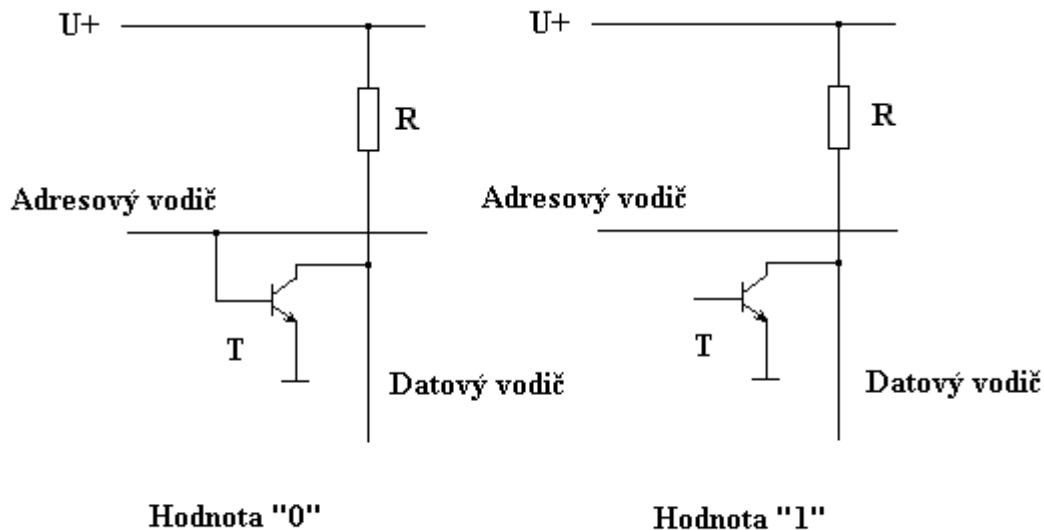
V případě druhém hodnota logická 1 přejde z adresového vodiče přes polovodičovou diodu na vodič datový. Toto zapojení představuje tedy paměťovou buňku s hodnotou 1. Dioda je zapojena tak, aby hodnota logická 1 mohla přejít z adresového vodiče na datový, ale nikoliv v opačném směru, což by vedlo k jejímu šíření po velké části paměti.

- **Realizace buňky paměti ROM pomocí polovodičové diody:**

- Jednotlivé buňky paměti ROM je také možné realizovat pomocí tranzistorů, a to jak v technologiích TTL, tak v technologiích MOS .

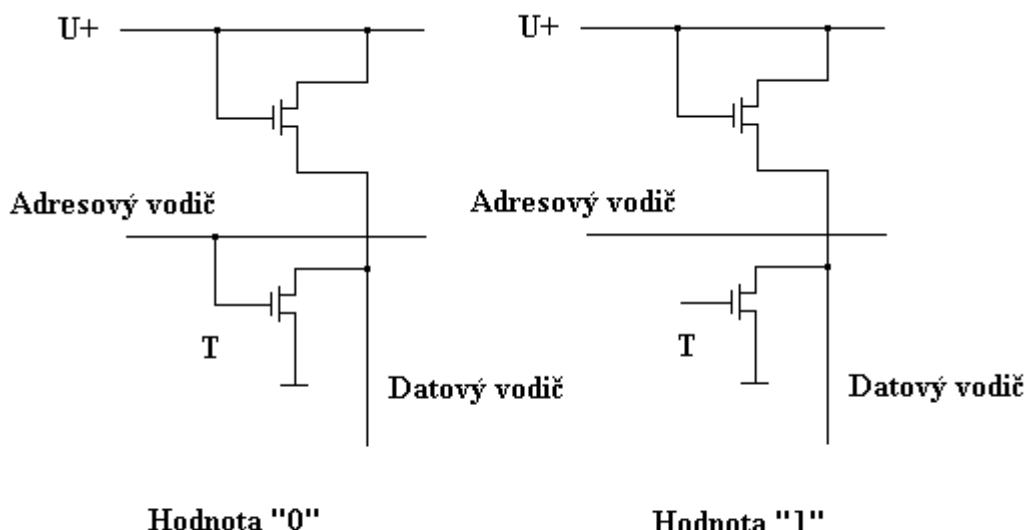


- Realizace paměťové buňky ROM pomocí tranzistoru v technologii TTL :



- V tomto případě je na datový vodič neustále přiváděna hodnota logická 1. Pokud dojde k vybrání adresového vodiče a tím k umístění hodnoty logická jedna na tento vodič, tak v případě, že je tranzistor T spojen s tímto adresovým vodičem, dojde k jeho otevření a tím k propojení datového vodiče se zemí. Na takto propojeném datové vodiči se potom objeví hodnota logická 0 a tato buňka představuje uložení hodnoty bitu 0. U buněk, jejichž tranzistor není spojen s adresovým vodičem, nemůže nikdy dojít k otevření tohoto tranzistoru a tím ani ke spojení datového vodiče se zemí. V této buňce je tedy neustále uložena hodnota 1.

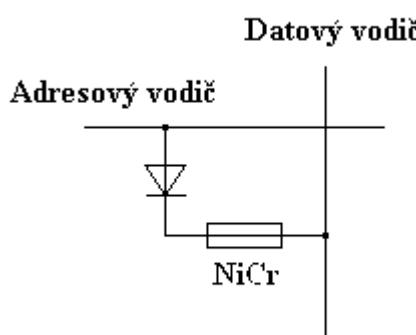
- Realizace paměťové buňky ROM pomocí tranzistoru v technologii MOS:



- Tranzistory připojené k napájecímu vodiči plní pouze úlohu rezistorů podobně jako u buňky v předešlém případě. Samotná buňka pracuje na stejném principu, který byl popsán u buňky v technologii TTL.

2. paměti PROM

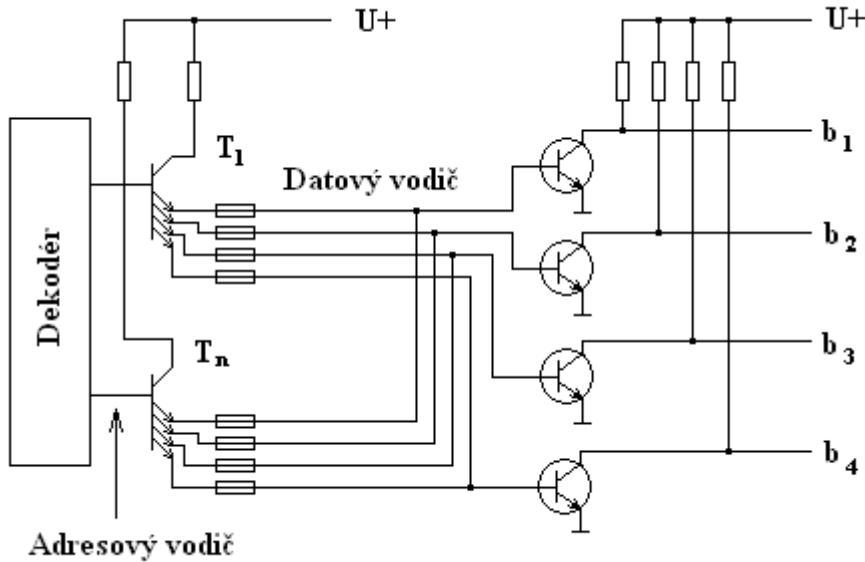
- PROM–Programable Read Only Memory
 - Neobsahují po vyrobení žádnou pevnou informaci
 - Příslušný zápis informace provádí uživatel
 - Zápis je možné provést pouze jednou a poté již paměť slouží stejně jako paměť ROM
 - Zápis informace se provádí vyšší hodnotou elektrického proudu (cca 10 mA), která způsobí přepálení tavné pojistky a tím i definitivně zápis hodnoty 0 do příslušné paměťové buňky.
 - Paměti PROM představují statické a energeticky nezávislé paměti
 - Buňku paměti je možné realizovat podobně jako u paměti ROM. Při výrobě je vyrobena matice obsahující spojené adresové vodiče s datovými vodiči přes polovodičovou diodu a tavnou pojistku z niklu a chromu (NiCr). Takto vyrobená paměť obsahuje na začátku samé hodnoty 1.
-
- Realizace paměťové buňky PROM pomocí diody:



- Paměti typu PROM se také realizují pomocí bipolárních multiemitorových tranzistorů

- Realizace paměťové buňky PROM pomocí multiemitorových tranzistorů:

- Takto realizovaná paměť PROM obsahuje pro každý adresový vodič jeden multiemitorový tranzistor. Každý z těchto tranzistorů obsahuje tolik emitorů, kolik je datových vodičů. Při čtení z paměti je opět na příslušný adresový vodič přivedena hodnota logická 1, která způsobí, že tranzistor se otevře a ve směru kolektor-emitor začne procházet elektrický proud. Jestliže je tavná pojistka průchozí, procházející proud otevře tranzistor, který je zapojen jako invertor, a na výstupu je přečtena hodnota 0. Jestliže tavná pojistka byla při zápisu přepálena, tzn. je neprůchozí, nedojde k otevření tranzistoru a na výstupu je přečtena hodnota 1.



- Paměť PROM pracující na tomto principu má po svém vyrobení ve všech buňkách zapsánu hodnotu 0 a při jejím programování se do některých buněk přepálením tavné pojistky zapíše hodnota 1.

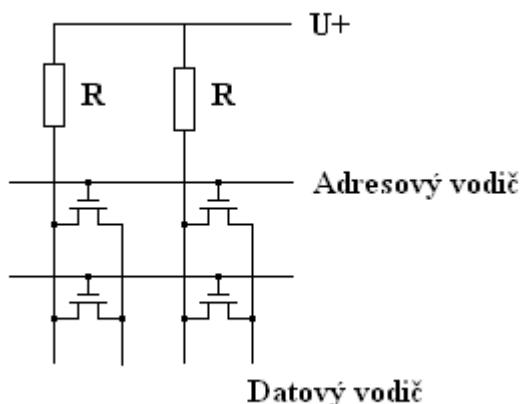
3. paměti EPROM

- EPROM–Erasable PROM
- Statické energeticky nezávislé paměti určené pro čtení i zápis informací
- Zapsané informace je možné vymazat působením ultrafialového záření
- Realizovány pomocí speciálních unipolárních tranzistorů, které jsou schopny na svém přechodu udržet elektrický náboj po dobu až několika let
- Zapojení jedné buňky paměti EPROM je podobné jako u paměti EEPROM

4. paměti EEPROM

- EEPROM – Electrically EPROM
- Mají podobné chování jako paměti EPROM, tj. jedná se o statické, energeticky nezávislé paměti, které je možné naprogramovat a později z nich informace vymazat
- Vymazání se provádí elektricky a nikoliv pomocí UV záření
- Vyrábí se pomocí speciálních tranzistorů vyrobených technologií MNOS (Metal Nitrid Oxide Semiconductor)
- Jedná se o tranzistory, na jejichž řídící elektrodě (Gate) je nanesena vrstva nitridu křemíku (Si_3N_4) a pod ní je umístěna tenká vrstva oxidu křemičitého (SiO_2)
- Buňka paměti EEPROM pracuje na principu tunelování (vkládání) elektrického náboje na přechod těchto dvou vrstev

- Realizace buňky paměti EEPROM pomocí tranzistoru MNOS - paměťová buňka EEPROM (matice 2×2):



5. paměti Flash

- Obdoba pamětí EEPROM
- Paměti, které je možné naprogramovat a které jsou statické a energeticky nezávislé
- Vymazání se provádí elektrickou cestou, jejich přeprogramování je možné provést přímo v počítači
- Paměť typu flash tedy není nutné před vymazáním (naprogramováním) z počítače vyjmout a umístit ji do speciálního programovacího zařízení
- Narozdíl od EEPROM se u pamětí flash provádí mazání nikoliv po jednotlivých buňkách, ale po celých blocích
- Paměťová buňka je tvořena tranzistorem, jehož elektroda gate je rozdělena na dvě části:
- *Control Gate*: připojená k adresovému vodiči
- *Floating Gate*: oddělená od control gate izolační vrstvou, umožňuje uložení elektrického náboje, pomocí něhož buňka uchovává hodnotu logická 0 nebo logická 1
- Flash paměti se dělí do dvou základních skupin:
 - *NOR flash*: poskytují rozhraní s vyhrazenými adresovými a datovými vodiči \Rightarrow umožňují přímý přístup k dané paměťové buňce, chovají se jako paměti, které jsou mapované do určité části adresového prostoru. Umožňují cca 10000 – 100000 smazání a následných zápisů. Mají menší hustotu paměťových buněk (dáno adresovacím mechanismem dovolujícím přímý přístup k paměťové buňce). Poskytují vyšší rychlosť při čtení, avšak jsou pomalejší při zápisu i při mazání a jsou cenově nákladnější. Jsou používány zejména pro ukládání firmwaru (BIOS, firmware pro mobilní telefony, GPS apod.) Nejsou vhodné pro ukládání větších objemů dat.
 - *NAND Flash*: jsou připojeny pomocí relativně jednoduchého rozhraní. Nevyžadují plnou šířku adresové a datové sběrnice. Data a příkazy jsou multiplexovány do 8 I/O linek, pomocí nichž jsou zasílány do interního registru

6. paměti RAM

- RAM—Random Access Memory
- Paměti určené pro zápis i pro čtení dat
- Jedná se o paměti, které jsou energeticky závislé
- Podle toho, zda jsou dynamické nebo statické, jsou dále rozdělovány na:
 - DRAM – Dynamické RAM
 - SRAM – Statické RAM

SRAM –Static Random Access Memory

- Uchovávají informaci v sobě uloženou po celou dobu, kdy jsou připojeny ke zdroji elektrického napájení
- Paměťová buňka je realizována jako bistabilní klopný obvod, tj. obvod, který se může nacházet vždy v jednom ze dvou stavů, které určují, zda v paměti je uložena 1 nebo 0
- Mají nízkou přístupovou dobu (1 – 20 ns)
- Jejich nevýhodou je naopak vyšší složitost a z toho plynoucí vyšší výrobní náklady
- Jsou používány především pro realizaci pamětí typu cache (L1, L2 i L3)
- Paměťová buňka používá dvou datových vodičů:
 - *Data*: určený k zápisu do paměti
 - *\Data*: určený ke čtení z paměti - hodnota na tomto vodiči je vždy opačná než hodnota uložená v paměti

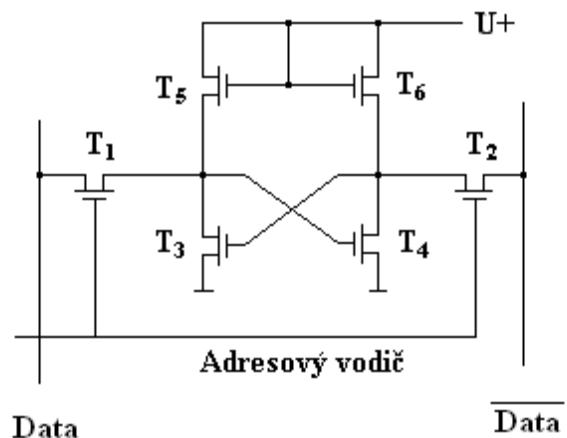
Realizace jedné buňky SRAM v technologii MOS

- U SRAM pamětí se používá dvou datových vodičů.

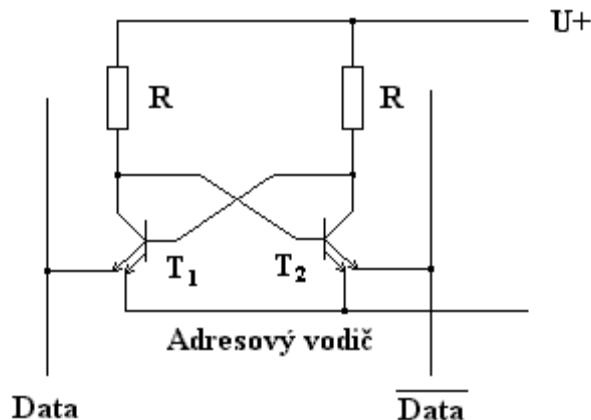
Vodič Data je určený k zápisu do paměti. Vodič označený jako \Data se používá ke čtení. Hodnota na tomto vodiči je vždy opačná než hodnota uložená v paměti. Takže na konci je nutno ji ještě negovat. Při zápisu se na adresový vodič umístí hodnota logická 1. Tranzistory T1 a T2 se otevřou. Na vodič Data se přivede zapisovaná hodnota (např. 1). Tranzistor T1 je otevřen, takže jednička na vodiči Data otevře tranzistor T4 a tímto dojde k uzavření tranzistoru T3. Tento stav obvodu představuje uložení hodnoty 0 do paměti. Zcela analogicky tato buňka pracuje i při zápisu hodnoty 1. Rozdíl je pouze v tom, že tranzistor T4 zůstane uzavřen a to způsobí otevření tranzistoru T3.

- Při čtení je opět na adresový vodič přivedena hodnota logická 1, což opět způsobí otevření tranzistorů T1 a T2. Jestliže byla v paměti zapsána hodnota 1, je tranzistor T4 otevřen (tj. na jeho výstupu je hodnota 0). Tuto hodnotu obdržíme na vodiči \DATA. Opět zcela analogicky v případě uložené hodnoty 0, kdy tranzistor T4 je uzavřen (tj. na jeho výstupu je hodnota 1).

- Paměti SRAM je možné uskutečnit i v technologii TTL



Realizace jedné buňky paměti SRAM v technologii TTL:



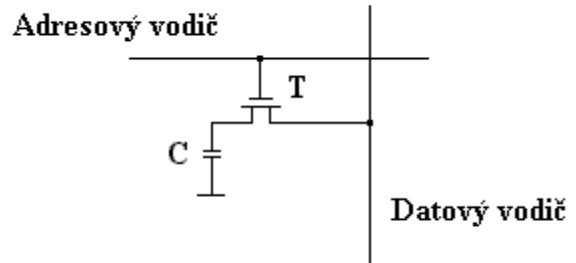
DRAM–Dynamic Random Access Memory

- Informace je uložena pomocí elektrického náboje na kondenzátoru
- Tento náboj má však tendenci se vybitit i v době, kdy je paměť připojena ke zdroji elektrického napájení
- Aby nedošlo k tomuto vybití a tím i ke ztrátě uložené informace, je nutné periodicky provádět tzv. refresh, tj. oživování paměťové buňky
- Operační paměti mají ve srovnání s jinými typy vnitřních pamětí podstatně vyšší kapacitu
⇒ nutnost jiné konstrukce
- Paměti DRAM jsou konstruovány jako matice, v nichž se jedna paměťová buňka zpřístupňuje pomocí dvou dekodérů
- Řadič operační paměti adresu rozdělí na dvě části, z nichž každá je přivedena na vstup samostatnému dekodéru (jeden dekodér vybere řádek a druhý sloupec)

Realizace jedné buňky paměti DRAM v technologii TTL:

- Při zápisu se na adresový vodič přivede hodnota logická 1. Tím se tranzistor T otevře. Na datovém vodiči je umístěna zapisovaná hodnota (např. 1). Tato hodnota projde přes otevřený tranzistor a nabije kondenzátor. V případě zápisu nuly dojde pouze k případnému vybití kondenzátoru (pokud byla dříve v paměti uložena hodnota 1).

- Při čtení je na adresový vodič přivedena hodnota logická 1, která způsobí otevření tranzistoru T. Jestliže byl kondenzátor nabitý, zapsaná hodnota přejde na datový vodič. Tímto čtením však dojde k vybití kondenzátoru a zničení uložené informace. Jedná se tedy o buňku, která je destruktivní při čtení a přečtenou hodnotu je nutné opět do paměti zapsat.



Vnější paměti

viz http://cs.wikipedia.org/wiki/Vn%C4%9Bj%C5%A1%C3%AD_pam%C4%9Bti

viz <http://www.sibl.cz/skripta/k5.htm>

Vnější paměť představuje v architektuře počítače paměť určenou k **trvalému ukládání informací** (programů a dat), její obsah se vypnutím počítače neztrácí. K vnější paměti nemá procesor počítače zpravidla přímý přístup. Srovnej s vnitřní pamětí. Vnější paměť můžeme rozdělit na stálou a výměnnou paměť. **Operační systém k přístupu do vnější paměti používá ovladače zařízení** a data jsou organizována do souborů podle pravidel použitého souborového systému. Výměnná paměť obvykle používá pro uložení dat výměnná datová média.

Mezi stálou vnější paměť počítače patří:

Pevný disk

Data jsou na povrchu pevného disku organizována do soustředných kružnic zvaných **stopy**, každá stopa obsahuje pevný anebo proměnný počet **sektorů**. Z důvodu efektivnějšího využití plochy disku je povrch většinou rozdelen do několika zón, každá zóna má různý počet sektorů na stopu. Sektor je nejmenší adresovatelnou jednotkou disku, má pevnou délku (donedávna 512 byte na sektor, nyní by se již po domluvě výrobců měly vyrábět disky s 4 KB na sektor). Pokud disk obsahuje více povrchů, všechny stopy, které jsou přístupné bez pohybu čtecí hlavičky, se nazývají **cylinder** (závit). Uspořádání stop, povrchů a sektorů se nazývá **geometrie disku**.

Adresa fyzického sektoru na disku se skládá z čísla stopy (cylindru), čísla povrchu a čísla sektoru.

Pro přístup k datům disku se používá starší metoda adresace disku CHS, která disk adresuje podle jeho geometrie (odtud název CHS - cylinder, head, sector). Hlavní nevýhodou je u osobních počítačů IBM PC omezená kapacita takto adresovaného disku (8 GB) a nutnost znát geometrii disku. U disků vyšších kapacit na rozhraní ATA, již neodpovídá zdánlivá geometrie disku skutečné fyzické implementaci (viz CHS).

Novější metoda pro adresaci disku je (u rozhraní ATA) LBA, sektory se číslují lineárně. Není třeba znát geometrii disku, max. kapacita disku je až 144 PB (144 miliónů GB).

Rozhraní SCSI používá lineární číslování sektorů disku již od své první verze. Ostatní novější rozhraní již převážně metodu jako je LBA používají.

Výměnná vnější paměť počítače:

- disketová jednotka
- CD jednotka
- DVD jednotka
- USB flash paměť
- Zip disk

Registry:

viz: http://cs.wikipedia.org/wiki/Registr_procesoru

Práce s pamětí, logický a fyzický adresový prostor, správa paměti, virtualizace paměti, segmentace, stránkování.

Obecné poznatky správy paměti

- Pro běh procesů je nutné, aby program, který ho řídí, byl umístěn v operační paměti.
- Program se převádí do formy schopné interpretace procesorem ve více krocích, jednou musí někdo rozhodnout, kde bude v operační paměti (FAP) umístěn.
- Roli dlouhodobé paměti programu plní vnější paměť.
- Vnitřní operační paměť uschovává data a programy právě běžících, resp. plánovatelných procesů.
- Správa paměti je předmětem činnosti OS, nelze ji nechat na aplikačním programování. Výkon jejich funkcí by byl neefektivní až škodlivý.
- Správa paměti musí zajistit, aby sdílení FAP mezi procesory bylo transparentní a efektivní a přitom bezpečné.

Požadavky na správu paměti

Možnost relokace programu

- programátor nemůže vědět, ze které části paměti bude jeho program interpretován
- při výměnách mezi FAP a vnější pamětí (odebrání a vrácení paměťového prostředku procesu) může být procesu dynamicky přidělena jiná (souvislá) oblast paměti než kterou opustil - swapping
- swapping umožňuje OS udržovat velký bank připravených procesů.
- odkazy na paměť v programu (LAP) se musí dynamicky překládat na skutečné adresy ve FAP

Nutnost ochrany

- procesy nesmí být schopné se bez povolení odkazovat na paměťová místa přidělená jiným procesům nebo OS
- relokace neumožňuje, aby se adresy kontrolovaly během komplikace
- odkazy na paměť se musí kontrolovat při běhu procesu hardwarem

Logická organizace

- Uživatelé tvoří programy jako moduly se vzájemně odlišnými vlastnostmi
- execute-only moduly s instrukcemi
- read-only nebo read-write - datové moduly
- private nebo public - moduly s omezením přístupu

Možnosti sdílení

- více procesů může sdílet společnou část paměti, aniž by došlo k porušení její ochrany
- sdílený přístup je lepší než udržování konzistence kopíí, které vlastní jednotlivé procesy

Pojmy

Vstupní fronta (input queue): Fronta procesů čekajících na zavedení do paměti.

Logický adresový prostor (LAP): virtuální adresový prostor, se kterým pracuje procesor při provádění kódu (každý proces i jádro mají svůj). Dán šírkou a formou adresy. Kapacita je dána šírkou adresy v instrukci.

Fyzický adresový prostor (FAP): adresový prostor fyzických adres paměti (společný pro všechny procesy i jádro). Adresa akceptovaná operační pamětí. Kapacita je dána šírkou adresové sběrnice operační paměti

Vázání adres LAP na adresy FAP

při komplikaci

- Je-li umístění ve FAP známé před překladem, komplikátor generuje *absolutní program* (obraz programu ve FAP)
- při změně umístění ve FAP se musí překlad opakovat

při zavádění

- umístění ve FAP je známé při sestavování nebo při zavádění programu
- překladač generuje *object module*, jehož cílovým adresovým prostorem je LAP
- vazby na adresy FAP provádí zavaděč

při běhu

- cílovým prostorem sestavení je LAP
- program se zavede do FAP ve tvaru pro LAP
- vázání se odkládá na dobu běhu – při interpretaci instrukce
- nutná HW podpora
- proces může měnit svoji polohu ve FAP během provádění

Přidělování souvislých oblastí

Operační paměť (FAP) je typicky rozdělena na dvě části:

sekce pro rezidentní část OS, obvykle bývá umístěna od počátku

sekce pro uživatelské procesy

Pro ochranu procesů uživatelů mezi sebou a OS při přidělování sekce procesů lze použít schéma s relokačním a mezním registrem:

relokační – hodnota nejmenší adresy sekce ve FAP

mezní – rozpětí logických adres $0..*$, přičemž logická adresa použitá v procesu musí být menší než mezní registr

Problém při přidělování více souvislých sekcí. Dynamicky vznikají a zanikají úseky dostupné paměti, které jsou roztroušené po FAP. Procesu se přiděluje takový úsek ve FAP, který uspokojí jeho požadavky. Evidenci úseků (volných i obsazených) udržuje právě OS.

Způsoby rozhodování přidělování:

First-fit – první dostatečně velký úsek paměti

Best-fit – nejmenší dostatečně dlouhý úsek paměti

Worst-fit – největší volný úsek paměti

Rychlostně i kvalitativně jsou *First-fit* a *Best-fit* lepší, než *Worst-fit*. Nejčastěji používaný je *First-fit*.

Fragmentace

Vnější – souhrn volné paměti je dostatečný, nikoliv však v dostatečně velkém souvislému bloku

Vnitřní – přidělená oblast paměti je větší, než požadovaná velikost, přebytek je nevyužitelná část paměti

Fragmentace je snižována setřásáním – přesun sekcí s cílem vytvořit velký úsek paměti.

Použitelné jen pokud je možná dynamická relokace (řeší MMU – Memory Management Unit). Provádí se v době běhu.

Výměny (Swapping)

Sekce FAP přidělená procesu je vyměňována mezi vnitřní a vnější pamětí oběma směry (*roll out, roll in*). Velmi časově náročné (srovnej přístupové doby do RAM a na HDD). Princip používaný mnoha OS ve verzích nepodporujících virtualizaci paměti – UNIX, Linux, Windows

Překryvy (Overlays)

Historická, klasická technika. V operační paměti se uchovávají pouze ty instrukce a data, která je potřeba zde uchovat „v nejbližší budoucnosti“. Vyvolávání překryvů je implementované uživatelem, není potřeba speciální podpory ze strany OS.

Stránkování

LAP procesu není zobrazován do jediné souvislé sekce FAP, zobrazuje se do po částech volných sekcí FAP

FAP se dělí na *rámce* – pevná délka (např. 512 B)

LAP se dělí na *stránky* – pevná délka shodná s délkou rámců

OS si udržuje seznam volných rámců

Pořadí přidělených rámců ve FAP nesouvisí s pořadím stránek v LAP

Překlad LAP → FAP je realizován *tabulkou stránek* (Page Table), jejíž obsah nastavuje OS

PT je uložena v operační paměti.

Zpřístupnění údaje v vyžaduje dva přístupy do paměti (do PT a pro operand)

Problém snížení efektivnosti dvojím přístupem lze řešit speciální rychlou HW cache (asociativní paměť, TLB – Translation Look-aside Buffer)

Virtuální paměti

Virtuální paměť (též virtualizace paměti) je v informatice způsob správy operační paměti počítače, který umožňuje předložit běžícímu procesu adresní prostor paměti, který je uspořádán jinak nebo je dokonce větší než je fyzicky připojená operační paměť RAM. Z tohoto důvodu procesor rozlišuje mezi virtuálními adresami (pracují s nimi strojové instrukce, resp. běžící proces) a fyzickými adresami paměti (odkazují na konkrétní adresové buňky paměti RAM). Převod mezi virtuální a fyzickou adresou je zajišťován samotným procesorem (je nutná hardwarová podpora RAM) nebo samostatným obvodem.

V současných běžných operačních systémech je virtuální paměť implementována pomocí stránkování paměti spolu se stránkováním na disk, který rozšiřuje operační paměť o prostor na pevném disku (stránkování na disk je nesprávně označováno jako swapování).

Výhody virtualizace :

- Paměť, kterou má běžící proces k dispozici, není omezena fyzickou velikostí instalované paměti.
- Je omezeno plýtvaní pamětí, kterou proces ve skutečnosti nevyužije nebo ji začne využívat až později.
- Každý běžící proces má k dispozici svou vlastní paměťovou oblast, ke které má přístup pouze on sám a nikdo jiný.
- Paměť jednotlivým procesům lze tak organizovat, že se paměť z hlediska procesu jeví jako lineární, přestože ve skutečnosti může být umístěna na různých místech vnitřní paměti i odkládacího prostoru.

Nevýhody virtualizace :

Při nedostatečné kapacitě fyzické operační paměti může dojít ke ztrátě výpočetního výkonu (thrashing). Pokud proces nemá v paměti dost stránek, často generuje přerušení výpadek stránky. To zaměstnává procesor a zpomaluje běh dalších procesů, ale hlavně tohoto procesu, neboť je v době načítání stránky pozastaven a nevykonává žádnou činnost. Dále jsou do paměti zaváděny další procesy, neboť procesor často čeká na ukončení vstupně-výstupní operace (právě zavádění a případné odkládaní stránek). To vede k dalšímu zhoršení výkonu. (Je ovšem nutné zmínit, že bez virtuální paměti by v takovém případě výpočet vůbec nemohl proběhnout.)

Principy virtualizace :

Všechny adresy, které proces používá, jsou spravovaný pouze jako virtuální - transformaci na fyzické adresy provádí správa virtuální paměti.

Každý proces po spuštění obdrží od operačního systému oblast pro uložení programu a pro data. Pokud potřebuje více operační paměti, může operační systém požádat o přidělení další paměti. Pokud se jí nedostává, operační systém odsune části obsahu operační paměti na disk a provede příslušné úpravy ve stránkovacích registrech procesů.

Procesům je bud' přidělen pevný počet rámců (ne nutně stejná hodnota pro všechny procesy, přidělený počet rámců může být úměrný velikosti programu), nebo je použito tzv. prioritní přidělování, kdy procesům s větší prioritou je dáno více rámců.

Existují dvě základní metody implementace virtuální paměti - stránkování a segmentace.

- Při stránkování je paměť rozdělena na větší úseky stejné velikosti, které se nazývají stránky. Správa virtuální paměti rozhoduje samostatně o tom, která paměťová stránka bude zavedena do vnitřní paměti a která bude odložena do odkládacího prostoru (swapu). - Při segmentaci je paměť rozdělena na úseky různé velikosti nazývané segmenty.

Používají se dvě základní politiky:

1. **Stránkování** (resp. segmentace) na žádost, kdy se stránka zavádí pouze jako důsledek přerušení typu výpadek stránky. - Předstránkování, kdy se počítá, že proces bude brzy pravděpodobně odkazovat na sousední stránku na sekundární paměti (používá např. Windows XP).

2. **Segmentace** : Segmentace je v informatice metoda správy paměti, kdy je procesu vytvořen virtuální adresní prostor začínající od nuly, čímž odpadá potřeba relokace použitého strojového kódu. Strojové instrukce používají pro adresaci logické adresy (offset). Fyzická adresa operační paměti je získána součtem segmentu registru a offsetu. Ve většině případů používá každý proces více segmentů, přičemž obsah jednotlivých segmentů odpovídá struktuře paměťového prostoru procesu.

Popis :

Ve víceúlohovém systému, který využívá multitasking, je nutné umístit do paměti více procesů najednou. Pokud jsou k dispozici nástroje na virtualizaci paměti, lze vytvořit každému z procesů jeho vlastní adresní prostor, který bude začínat od nuly. Každý proces (resp. jeho strojové instrukce) pak bude pracovat s logickými (virtuálními) adresami, které budou vyjadřovat vzdálenost od začátku tohoto vlastního vyhrazeného adresního prostoru (tj. logická adresa bude pro každý proces začínat od nuly). Logické adresy používané strojovými instrukcemi jsou pak v případě přístupu do paměti automaticky překládány pomocí MMU procesoru na fyzické adresy a zpět. Segmentace používá pro vytvoření logické (virtuální) adresy dva speciální registry - segment a offset, přičemž jejich součet vyjadřuje fyzickou adresu. Segment se nastaví předem (tzv. bázová adresa). Strojové instrukce pak pracují pouze s offsetem (logickou adresou), která vyjadřuje vzdálenost od bázové adresy (segmentu). Segmentový registr je typicky procesu nepostupný, nastavuje ho operační systém v privilegovaném režimu. Operační systém tak může bez vědomí procesu segment přemístit (při tzv. setřásání segmentu), protože logické adresy (offsety) se při přemístění nezmění (viz níže). Každý proces má k dispozici typicky více segmentů, které odpovídají logickému členění adresního prostoru procesu (např. kód programu, knihovny, data, zásobník, halda). Pro přístup do každého z těchto segmentů musí být k dispozici segmentový registr, který uchovává informaci o začátku segmentu. Nepoužívané segmenty mohou být odloženy z paměti na pevný disk (tzv. swapování) - viz virtuální paměť.

Existující implementace :

Segmentace procesoru 80286+ - Segmentace v chráněném módu na procesorech x86 od Intel 80286 výše vykazuje znaky připomínající reálný mód. Segmentace je povinná a pokud je zapnuto stránkování paměti, provádí se před ním, tedy virtuální adresa se segmentací překládá na lineární adresu, která se stránkováním překládá na fyzickou adresu.

Virtuální adresa se skládá ze dvou částí:

selektor segmentu a offset. Selektor obsahuje tři příznakové byty a index do tabulky GDT (global descriptor table) nebo LDT (local descriptor table). Selektor je uložen v segmentovém registru, kromě něj obsahuje segmentový registr i „cache“ příslušného řádku z GDT nebo LDT (zvaného descriptor) pro rychlejší vyhodnocování (k této cache nelze programově přistupovat, proto se říká že se nachází v neviditelné části registru).

Selektor - Horních 13 bitů selektoru je indexem do tabulky deskriptoru. - Další bit určuje, zda se bude hledat v GDT nebo LDT. - Dva poslední nejnižší byty určují úroveň oprávnění segmentu.

Když se segmentový registr naplní selektorem, nahraje se do jeho neviditelné části deskriptor, na který daný selektor v tabulce ukazoval.

Deskriptor - 32 bitová báze (Počáteční adresa daného segmentu), rozdělena na dolních 24 bitu a horních 8 bitu (Intel 80286 nemel horní část).

- 20-bitový limit segmentu

- **Příznaky:**

- bit „Granularity“ (značí se G). Pokud je G=0 20-bitový limit se počítá po bajtech, tzn.:Limit = 20 bitu = $1048575 \text{ bajt} = \pm 1 \text{ MB}$ segment bude mít tedy limit 1MB. Pokud bude G=1 tak se limit počítá po 4 KB, tzn.:Limit = 20 bitu = $1048575 \text{ Kb} = \pm 4096 \text{ MB}$ segment bude tedy mít limit 4GB.
- typ segmentu (kódový, datový, systémový)
- přístupová práva segmentu (READ/WRITE/EXECUTE - kombinace WRITE a EXECUTE je ovšem nenastavitelná).

Stránkování paměti :

Stránkování paměti je v informatice metoda správy paměti, kdy strojové instrukce procesu pracují s logickými adresami, které jednotka MMU (typicky součást procesoru) převádí na fyzické adresy (skutečné umístění v paměti RAM). Vzniká tak virtuální adresní prostor, který pro každý proces začíná od nuly, takže odpadá potřeba relokace použitého strojového kódu. Při převodu je adresní prostor rozdělen na stránky (rámce) stejně velikosti (typicky 4 KiB).

Popis stránkování :

Ve víceúlohovém systému, která využívá multitasking, je nutné umístit do paměti více procesů najednou. Pokud jsou k dispozici nástroje na virtualizaci paměti, lze vytvořit každému z procesů jeho vlastní lineární adresní prostor, který bude začínat od adresy nula. Každý proces (resp. jeho strojové instrukce) pak bude pracovat s logickými (virtuálními) adresami, které jsou při přístupu do paměti RAM automatický převáděny jednotkou MMU na fyzické adresy (MMU je dnes typicky umístěna v procesoru, ale může být i nezávislým obvodem). Aby nebylo nutné pro převod logických adres na fyzické udržovat velké množství informací, je adresní prostor rozdělen na stránky stejně velikosti (typicky 4 KiB, ale i 8 KiB nebo větších). Virtuální adresní prostor je tedy složen ze stránek, které odpovídají stejně velkým stránkám ve fyzické paměti RAM. Zatímco logické stránky vytvářejí souvislý lineární adresní prostor, umístění fyzických stránek je díky převodu zcela nahodile. Pro běžící proces se tak vytváří iluze, že jeho adresní prostor v operační paměti je souvislý, zatímco ve skutečnosti jsou fyzické stránky fragmentovány. Fragmentace je však procesu (resp. jeho strojovým instrukcím) skryta a není ji nutné nijak řešit (na rozdíl od segmentace paměti, kde je nutné fragmentaci odstraňovat setřásáním segmentu).

Tabulka stránek :

Stránky, ze kterých je složen adresní prostor procesu (s logickými adresami), jsou očíslovány. Stejně tak jsou očíslovány stránky ve fyzické paměti. K převodu čísla logické stránky na číslo fyzické stránky je používána tabulka stránek. Řekněme, že horní řádek tabulky obsahuje čísla logických stránek a spodní řádek tabulky obsahuje čísla fyzických stránek. Protože adresní prostor procesu s logickými stránkami je lineární, bude v horním řádku tabulky těž lineární řada čísel (tj. 0, 1, 2, 3...), která může být považována za index v tabulce (tj. pořadí záznamu) a není ji proto nutné v paměti uchovávat. Tabulka stránek, která je umístěna v operační paměti, tedy obsahuje pouze lineární seznam čísel fyzických stránek.

Překlad adresy :

Je-li velikost stránky 4 KiB, je pro odlišení jednotlivých adres uvnitř stránky potřeba 12 bitů (2¹² B = 4096 B = 4 KiB). Těchto 12 bitů je vlastně offset uvnitř stránky (tj. vzdálenost od začátku stránky), který zůstává stejný jak uvnitř logické stránky, tak uvnitř fyzické stránky. Vrchní bity pak vyjadřují číslo stránky. U 32bitových procesorů IA-32 (Intel kompatibilních) je adresa 32bitová, takže číslo stránky je 20bitové. Při překladu logické adresy na fyzickou proto zůstává spodních 12 bitů zachováno a vrchních 20 bitů logické adresy je přepsáno na hodnotu, která je uvedena v odpovídajícím záznamu v tabulce stránek. Převod logické adresy na fyzickou je tak velmi jednoduchý. Přestože by jedna položka v tabulce stránek mohla být v takovém případě jen 20bitová, nebylo by to výhodné, protože kvůli uspořádání paměti by většina čtení musela proběhnout nadvakařat. Proto jsou hodnoty zarovnány na vhodnou velikost (u IA-32 na 32 bitu) a volné bity jsou použity jako doplňující příznaky upřesňující charakter stránky (viz dále).

TLB :

Protože k převodu dochází velmi často a tabulka stránek je uložena v paměti RAM, která je v porovnaní s rychlostí procesoru relativně pomala, je v procesoru zřízena speciální hardwarová TLB cache (anglický Translation Lookaside Buffer), která čtení hodnot z tabulky stránek urychluje.

Správa tabulky stránek :

Stránkování paměti se využívá pro usnadnění multitaskingu, při kterém je spuštěno zároveň více (at' již se v běhu střídají nebo běží skutečně paralelně). Multitasking je řízen jádrem operačního systému. Každý proces má přidělen vlastní oddělený virtuální adresní prostor, a proto musí mít každý proces i vlastní tabulku stránek. Údaje o umístění tabulky stránek jsou součástí vlastností každého procesu a jsou uloženy v části paměti spravované jádrem systému, obvykle v PCB (anglický Process control block).

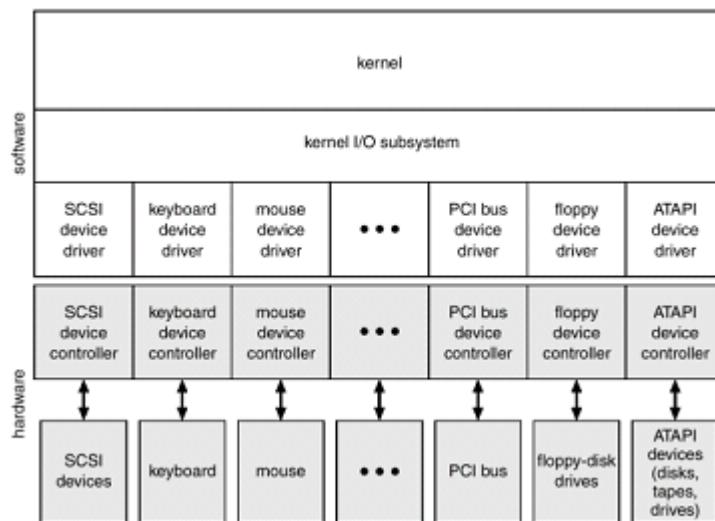
Víceúrovňové tabulky stránek :

Protože obvykle není lineární adresní prostor procesu celý využit, je mnoho pozic v tabulce stránek neobsazených. Na již zmíněné architektuře IA-32 by vytvoření nového procesu znamenalo vytvoření tabulky až o velikosti 32 MiB (4 GiB : 4 KiB x 32 bitu = 32 MiB). Z tohoto důvodu jsou tabulky stránek uspořádány do víceúrovňové stromové struktury (obvykle 2 nebo 3 úrovně) a tím není nutné souvislá neobsazena místa adresního prostoru vůbec tabulkami stránek popisovat.

Ovládání vstupů a výstupů, správa a plánování činnosti V/V zařízení.

Správa a plánování činnosti V/V zařízení

Základní představa - k počítači máme připojeny různé I/O(=V/V) zařízení a musíme ošetřit organizaci přístup k těmto zařízením(abyste například 2 procesy netiskly data na tiskárnu ve stejnou chvíli).



Techniky provádění I/O:

Polling(vyzývání)

- programovaný I/O, činné čekání, synchronní operace
- busy-wait - OS čeká aktivně na uvolnění I/O, nedělá nic jiného, než že zjišťuje stav I/O zařízení - zdržuje procesor.
- Někdy se také používá polling tak, že procesor počítá a vždy po určitém čase zkонтroluje polované zařízení zda už je ready, pokud ne, tak se vrátí k počítání - zdržuje méně.

Přerušení

- programovaný I/O, asynchronní
- označován jako interrupt, IRQ ve smyslu hardwarového přerušení
- reakce na signál od zařízení, které jím upozorňuje procesor (respektive ovladač zařízení v OS), že potřebuje obsloužit. Procesor při příchodu přerušení přestane provádět současný výpočet, uloží část svého stavu a začne vykonávat obsluhu přerušení.

DMA (direct memory access)

- náhrada programovaného I/O, není využíván procesor, ale speciální DMA řadič typicky pro velké toky dat (dnes už všechny HDD)
- Příklad: zobrazení videa - proces potřebuje data v paměti, aby je mohl dekódovat+zobrazit, procesor ale data do paměti nepřenáší - řekne DMA řadiči, že data potřebuje v paměti, DMA se postará o jejich přenos, po dokončení přenosu informuje procesor pomocí přerušení a procesor řeší ve spolupráci s grafickou kartou jejich zobrazení.

blokující/neblokující I/O:

- blokující - proces čeká na ukončení (nevyhovující), snadné použití
- neblokující - řízení procesu se vrací bez problémů, buffered I/O, implementace pomocí vláken
- asynchronní - proces souběžně s I/O, konec I/O hlášeno přerušeními, složité použití

IO subsystémy v jádru

- Dle druhu OS, nemusí být přítomny všechny

plánování (I/O plánovače)

- slouží pouze pro zvýšení výkonu (na rozdíl od plánovače procesů není nezbytný)
- některé řadiče mají vestavěný plánovač (SATA NCQ, SCSI, ...)
- CFQ = Completely Fair Queuing - obdoba round robin

buffering - zařízení A generuje data rychleji, než je zařízení B zvládne zpracovat, data pak uložíme do bufferu, a B si čte dle potřeby. I naopak(B generuje pomaleji, ale A potřebuje souvislý zdroj - nagenerejme do bufferu a A pak přečte)

caching - kopie(pouze kopie!) dat uložena v rychlé paměti - k dosažení vysokého výkonu.

spooling

V daném okamžiku může vstupní výstupní zařízení provádět jen jednu úlohu, požadavky do fronty. Pro realizaci fronty požadavků (úloh) je většinou využit souborový systém s přímým přístupem (disk, ramdisk). Např. tiskárna, magnetická páska

rezervace - exklusivní přístup k zařízení pro proces, nutná ochrana proti uváznutí)

Databáze

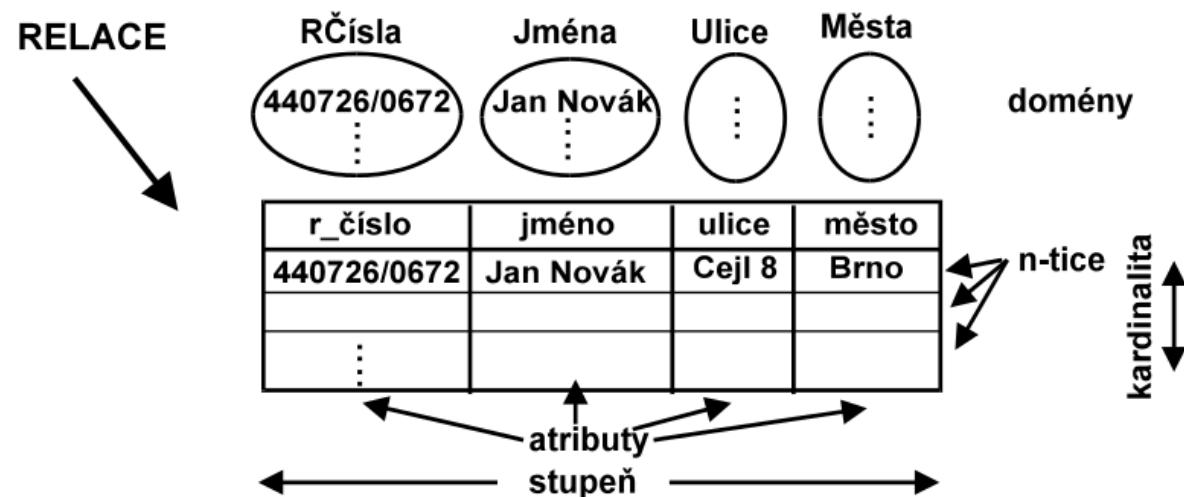
Relační model dat, relační schéma, klíče relačních schémat, relační algebra, spojování relací. Funkční závislosti, normální formy (1NF, 2NF, 3NF, Boyce-Coddova NF), vztahy mezi normálními formami. Dekompozice relačních schémat.

Relační model dat, relační schéma, klíče relačních schémat, relační algebra, spojování relací

Relační model dat

- Relační model dat je nejrozšířenějším způsobem uložení dat v databázi (relaci). Jedná se o způsob uložení v logickém smyslu.
- Matematicky je model popsán relační algebrou či relačním kalkulem (lze s nimi popsat sémantiku uživatelských relačních jazyků [jako je SQL]).
- Složky relačního modelu dat jsou
 - Relační datová struktura
 - Obecná integrativní omezení pro relační databáze
 - Manipulace s daty v relační databázi

Základní pojmy



Relace: Relace (tabulka) je základním prvkem relačního databázového modelu (relačních databází), která sdružuje data do n-tic (řádků). Tabulka je struktura záznamů s pevně stanovenými položkami (sloupci - atributy). Každý sloupec má definován jednoznačný název, typ a rozsah (doména). Záznam se stává n-ticí relace (řádkem tabulky). Pokud jsou v různých tabulkách sloupce stejného typu, pak tyto sloupce mohou vytvářet vazby mezi jednotlivými tabulkami. Tabulky se poté naplňují vlastním obsahem (daty). Kolekce více tabulek, jejich funkčních vztahů, indexů a dalších součástí tvoří relační databázi.

VLASTNOSTI RELACÍ

Neexistují duplicitní n-tice, n-tice jsou neuspořádané a hodnoty jednoduchých atributů jsou atomické.

RELACE vs. TABULKA

Relace = základní abstraktní pojem relačního modelu

Tabulka = forma znázornění relace

- v relaci nezáleží na pořadí řádků, v tabulce je dán pořadí řádků

- relace neobsahuje duplicitní n-tice, v tabulce se mohou vyskytovat duplicitní řádky

RELACE	TABULKA
Schéma relace	Záhlaví tabulky
Jméno atributu	Jméno sloupce
Atribut	Sloupec
N-tice relace	Řádek tabulky

TYPY RELACÍ

- pojmenované
 - bázové (reálné)
 - pohledy (virtuální)
 - materializované
 - pohledy (snapshot)- odvozené, ale existující
 - dočasné
- nepojmenované
 - výsledky dotazů
 - mezivýsledky

Doména: Pojmenovaná množina skalárních hodnot téhož typu neboli množina přípustných hodnot pro daný sloupec. Např. {Brno, Praha}.

Skalární hodnota: Nejmenší (atomická) sémantická vnitřně nestrukturovaná jednotka. Např. Jan Novák.

Složená doména: Doména složena z několika jednoduchých domén. Např. {Jan, Novák} – doména složena z domény Příjmení a z domény Křesní jméno.

FORMÁLNĚ:

Mějme relaci **R** nad množinou atributů $A = \{A_1, A_2, \dots, A_n\}$.

- jména atributů: A_1, A_2, \dots, A_n
- domény atributů: $D_i = \text{dom}(A_i)$
- jméno relace: **R**
- n-tice: (a_1, a_2, \dots, a_n)

Relační schéma

Schéma relace říká, jaký je název relace, kolik má sloupců a jaké jsou jejich názvy a domény.

V podstatě šablona na vytvoření relace. V databázích je schématem relace definice struktury tabulky.

FORMÁLNĚ:

Mějme relaci **R** vzniklou skrze relační schéma $R(A)$.

- relační schéma: $R(A)$ neboli $R(A_1:D_1, \dots, A_n:D_n)$

Příklad:

Relační schéma: Schéma_student = (učo, jméno, příjmení, adresa)

Relace: $R = (12345, \text{Jarda}, \text{Benda}, \text{Brno})$

Klíče relačních schémat

Klíč je součástí relačního schéma, jedná se podmnožinu atributů. Jeho úkolem je identifikovat záznam v tabulce.

Super klíč

Taková množina atributů, která jednoznačně identifikuje řádek v tabulce.

Kandidátní klíč

Minimální super klíč, tzn. po odebrání jakékoli množiny atributů by již neidentifikoval záznamy jednoznačně.

Primární klíč

Jeden z vybraných kandidátních klíčů. Zbývající kandidátní klíče nazýváme alternativní klíče. Způsob výběru klíče není v relačním modelu specifikován. Primární klíč je jednoznačný identifikátor záznamu, řádku tabulky. Primárním klíčem může být jeden sloupec či kombinace více sloupců tak, aby byla zaručena jeho jednoznačnost. Pole klíče musí obsahovat hodnotu, tzn. nesmí se zde vyskytovat nedefinovaná prázdná hodnota NULL. V praxi se dnes často používají umělé klíče, což jsou číselné či písmenné identifikátory – každý nový záznam dostává identifikátor odlišný od identifikátorů všech předchozích záznamů (požadavek na unikátnost klíče), obvykle se jedná o celočíselné řady a každý nový záznam dostává číslo vždy o jednotku vyšší (zpravidla zcela automatizovaně) než je číslo u posledního vloženého záznamu (číselné označení záznamů s časem stoupá).

Cizí klíč

Spojení jednoho nebo více sloupců se sloupcem nebo sloupci jiné tabulky. Pokud se hodnoty dotčených sloupců shodují, poté příslušný řádek cizí tabulky rozvíjí řádek zdrojové tabulky přes toto spojení. Tomu se též říká reference nebo odkaz. Cizí klíč umožňuje definovat akce, které mají nastat při pokusu o změnu nebo mazání záznamů v cizí tabulce. Například, po smazání záznamu z cizí tabulky budou ve zdrojové tabulce řádky s odpovídající hodnotou cizího klíče taktéž smazány, nebo budou jejich odkazy nastaveny na určitou (neutrální) hodnotu, nebo se smazání řádků v cizí tabulce zabrání. Omezení cizích klíčů tak představuje mechanismus pro udržení referenční integrity databáze.

Relační algebra

Čistý procedurální dotazovací jazyk vysoké úrovně. Soubor operací, které manipuluje se vztahy (soubor operací se může lišit podle definice). Předpoklad

Základní operace

Selekce (výběr): $\sigma_P(r) = \{t | t \in r \text{ and } P(t)\}$ – vybere řádky (t) podle podmínky (P) z relace (r)

Projekce: $\Pi_{A_1, \dots, A_n}(r)$ – výsledkem je „osekaná“ relace (r), bude obsahovat pouze vyjmenované sloupce

Sjednocení: $r \cup s = \{t | t \in r \text{ or } t \in s\}$ – za podmínky, že(r) a (s) mají stejnou aritu a kompatibilní domény

Rozdíl: $r - s = \{t | t \in r \text{ and not } t \in s\}$ – za stejně podmínky jako předchozí

Kartézský součin $r \times s = \{t q | t \in r \text{ and } q \in s\}$

Přejmenování: $\rho_{\text{novar}(n_A, n_B, C)}(r)$ – přejmenuje relaci (r) podle nového jména (novar), musíme také vyjmenovat atributy relace

Spojování relací

- **Přirozené spojení (natural join)**

Spojení dvou relací na základě primárních klíčů, výsledkem je nová relace obsahující atributy (i duplicitní mimo primární klíč) předešlých dvou relací.

$$r \bowtie s = \Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B=s.B \wedge r.D=s.D} (r \times s))$$

- **Vnější spojení (outer join)**

Rozšíření operace přirozeného spojení tak, aby se zabránilo ztrátě informací; spočítá operaci spojení a přidá n-tice z jedné relace, které neodpovídají n-ticím druhé a případně použije hodnotu null.

Jakékoli porovnání s null je null.

V praxi se používá varianta „left“ a „right“ to určuje zda, se do výsledku zahrnou všechny záznamy z tabulky nalevo od operátoru spojení, resp. napravo od operátoru spojení. Taktéž existuje varianta „full outer“, která do výsledků zahrne všechny záznamy z obou tabulek a tam, kde nemá daný záznam odpovídající záznam z druhé tabulky, tak se použije null.

Příklady:

	Osoba:	Adresa:																																					
	<table border="1"> <thead> <tr> <th></th><th>id</th><th>jméno</th><th>příjmení</th><th>věk</th></tr> </thead> <tbody> <tr> <td>1</td><td>Pavel</td><td>Havel</td><td>18</td><td></td></tr> <tr> <td>2</td><td>Jan</td><td>Novák</td><td>19</td><td></td></tr> <tr> <td>3</td><td>Láďa</td><td>Koupelna</td><td>99</td><td></td></tr> </tbody> </table>		id	jméno	příjmení	věk	1	Pavel	Havel	18		2	Jan	Novák	19		3	Láďa	Koupelna	99		<table border="1"> <thead> <tr> <th></th><th>id</th><th>město</th><th>ulice</th></tr> </thead> <tbody> <tr> <td>1</td><td>Brno</td><td>Jarní</td><td></td></tr> <tr> <td>3</td><td>Praha</td><td>Zelená</td><td></td></tr> <tr> <td>5</td><td>Ostrava</td><td>Černá</td><td></td></tr> </tbody> </table>		id	město	ulice	1	Brno	Jarní		3	Praha	Zelená		5	Ostrava	Černá		
	id	jméno	příjmení	věk																																			
1	Pavel	Havel	18																																				
2	Jan	Novák	19																																				
3	Láďa	Koupelna	99																																				
	id	město	ulice																																				
1	Brno	Jarní																																					
3	Praha	Zelená																																					
5	Ostrava	Černá																																					
	Osoba \bowtie Adresa:																																						
	<table border="1"> <thead> <tr> <th></th><th>id</th><th>jméno</th><th>příjmení</th><th>věk</th><th>město</th><th>ulice</th></tr> </thead> <tbody> <tr> <td>1</td><td>Pavel</td><td>Havel</td><td>18</td><td>Brno</td><td>Jarní</td><td></td></tr> <tr> <td>3</td><td>Láďa</td><td>Koupelna</td><td>99</td><td>Praha</td><td>Zelená</td><td></td></tr> </tbody> </table>		id	jméno	příjmení	věk	město	ulice	1	Pavel	Havel	18	Brno	Jarní		3	Láďa	Koupelna	99	Praha	Zelená																		
	id	jméno	příjmení	věk	město	ulice																																	
1	Pavel	Havel	18	Brno	Jarní																																		
3	Láďa	Koupelna	99	Praha	Zelená																																		
	Osoba \bowtie Adresa:																																						
	<table border="1"> <thead> <tr> <th></th><th>id</th><th>jméno</th><th>příjmení</th><th>věk</th><th>město</th><th>ulice</th></tr> </thead> <tbody> <tr> <td>1</td><td>Pavel</td><td>Havel</td><td>18</td><td>Brno</td><td>Jarní</td><td></td></tr> <tr> <td>2</td><td>Jan</td><td>Novák</td><td>19</td><td>null</td><td>null</td><td></td></tr> <tr> <td>3</td><td>Láďa</td><td>Koupelna</td><td>99</td><td>Praha</td><td>Zelená</td><td></td></tr> </tbody> </table>		id	jméno	příjmení	věk	město	ulice	1	Pavel	Havel	18	Brno	Jarní		2	Jan	Novák	19	null	null		3	Láďa	Koupelna	99	Praha	Zelená											
	id	jméno	příjmení	věk	město	ulice																																	
1	Pavel	Havel	18	Brno	Jarní																																		
2	Jan	Novák	19	null	null																																		
3	Láďa	Koupelna	99	Praha	Zelená																																		
	Osoba \bowtie Adresa:																																						
	<table border="1"> <thead> <tr> <th></th><th>id</th><th>jméno</th><th>příjmení</th><th>věk</th><th>město</th><th>ulice</th></tr> </thead> <tbody> <tr> <td>1</td><td>Pavel</td><td>Havel</td><td>18</td><td>Brno</td><td>Jarní</td><td></td></tr> <tr> <td>3</td><td>Láďa</td><td>Koupelna</td><td>99</td><td>Praha</td><td>Zelená</td><td></td></tr> <tr> <td>5</td><td>null</td><td>null</td><td>null</td><td>Ostrava</td><td>Černá</td><td></td></tr> </tbody> </table>		id	jméno	příjmení	věk	město	ulice	1	Pavel	Havel	18	Brno	Jarní		3	Láďa	Koupelna	99	Praha	Zelená		5	null	null	null	Ostrava	Černá											
	id	jméno	příjmení	věk	město	ulice																																	
1	Pavel	Havel	18	Brno	Jarní																																		
3	Láďa	Koupelna	99	Praha	Zelená																																		
5	null	null	null	Ostrava	Černá																																		
	Osoba \bowtie Adresa:																																						
	<table border="1"> <thead> <tr> <th></th><th>id</th><th>jméno</th><th>příjmení</th><th>věk</th><th>město</th><th>ulice</th></tr> </thead> <tbody> <tr> <td>1</td><td>Pavel</td><td>Havel</td><td>18</td><td>Brno</td><td>Jarní</td><td></td></tr> <tr> <td>2</td><td>Jan</td><td>Novák</td><td>19</td><td>null</td><td>null</td><td></td></tr> <tr> <td>3</td><td>Láďa</td><td>Koupelna</td><td>99</td><td>Praha</td><td>Zelená</td><td></td></tr> <tr> <td>5</td><td>null</td><td>null</td><td>null</td><td>Ostrava</td><td>Černá</td><td></td></tr> </tbody> </table>		id	jméno	příjmení	věk	město	ulice	1	Pavel	Havel	18	Brno	Jarní		2	Jan	Novák	19	null	null		3	Láďa	Koupelna	99	Praha	Zelená		5	null	null	null	Ostrava	Černá				
	id	jméno	příjmení	věk	město	ulice																																	
1	Pavel	Havel	18	Brno	Jarní																																		
2	Jan	Novák	19	null	null																																		
3	Láďa	Koupelna	99	Praha	Zelená																																		
5	null	null	null	Ostrava	Černá																																		

Funkční závislosti, normální formy (1NF, 2NF, 3NF, Boyce-Coddova NF), vztahy mezi normálními formami. Dekompozice relačních schémat.

Funkční závislosti

Funkční závislosti si lze představit jako tvrzení o reálném světě. Například plat zaměstnance závisí na tom, jakou vykonává funkci, tj. plat závisí na funkci, zapisujeme $\text{FUNKCE} \rightarrow \text{PLAT}$. Druhým příkladem by mohla být výše jízdného, která závisí na délce vlakové trasy, tj. $\text{DÉLKA_TRASY} \rightarrow \text{VÝŠE_JÍZDNÉHO}$.

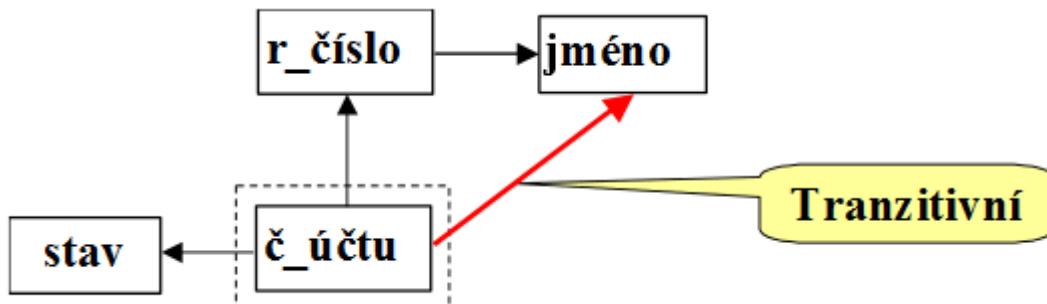
Obecně: Necht' $X \subseteq R$; $Y \subseteq R$, pak řekneme, že Y je funkčně závislé na X , píšeme $X \rightarrow Y$, když pro každou povolenou relaci $r(R)$ platí, že mají-li její dva libovolné prvky stejné hodnoty v attributech X , pak mají i stejné hodnoty v attributech Y . Funkční závislosti umožňují vyjádřit (integritní) omezení, které nelze vyjádřit pomocí super klíčů.

Využití:

- Testování relací, jsou-li povolené na dané množině funkčních závislostí. Je-li relace r povolená na množině F funkčních závislostí, říkáme, že r splňuje F .
- Definování omezení na množině povolených relací, říkáme, že F je platná na R , když všechny povolené relace na R splňují množinu F .

Tranzitivní závislost

- Atribut je funkčně závislý na jiném funkčně závislém atributu.



Normální formy

V relačním modelu jsou data uložena v tabulkách, na které má jisté požadavky. Při splnění požadavků je tabulka označována jako normalizovaná. Pokud nejsou tyto požadavky splněny, jsou označovány jako nenormalizované a proces jejich převodu na tabulky se označuje jako normalizace. Při tomto procesu dochází k odstraňování nedostatků tabulek jako je redundancy nebo možnost vzniku aktualizační anomálie, tj. nechtěného vedlejšího efektu operace nad databází, při kterém dojde ke ztrátě nebo nekonzistence dat. Postup normalizace je rozdělen do několika kroků a po dokončení každého z nich se tabulka nachází v určité normální formě. Normalizace = postupná dekompozice.

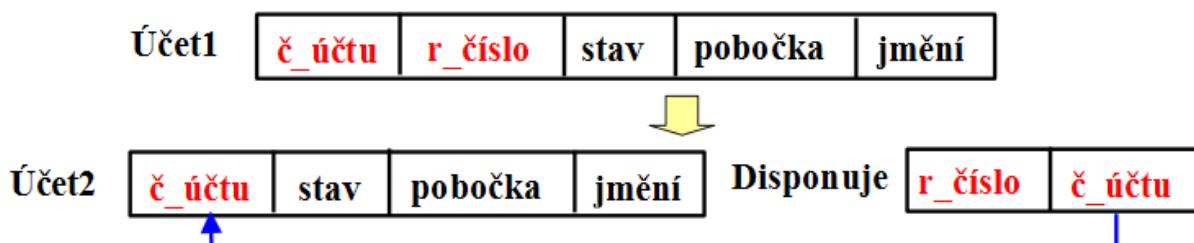
Žádoucí vlastnosti

- Bezztrátovost při zpětném spojení
- Zachování závislostí
- Odstranění redundance

1NF: Každý atribut obsahuje pouze atomické hodnoty

2NF: Každý neklíčový atribut je plně závislý na primárním klíči

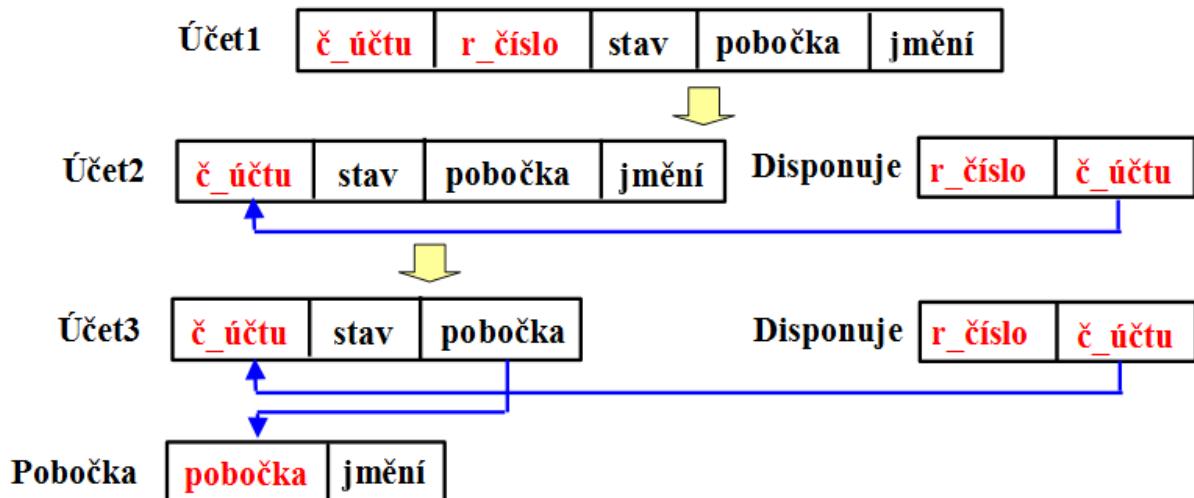
-Tabulka splňuje 2NF, právě když splňuje 1NF a navíc každý atribut, který není primárním klíčem je na primárním klíči úplně závislý. To znamená, že se nesmí v řádku tabulky objevit položka, která by byla závislá jen na části primárního klíče. Z definice vyplývá, že problém 2NF se týká jenom tabulek, kde volíme za primární klíč více položek než jednu. Jinými slovy, pokud má tabulka jako primární klíč jenom jeden sloupec, pak 2NF je splněna triviálně. Redundance je signál, který indikuje nesplnění 2NF.



3NF: Všechny neklíčové atributy musí být vzájemně nezávislé

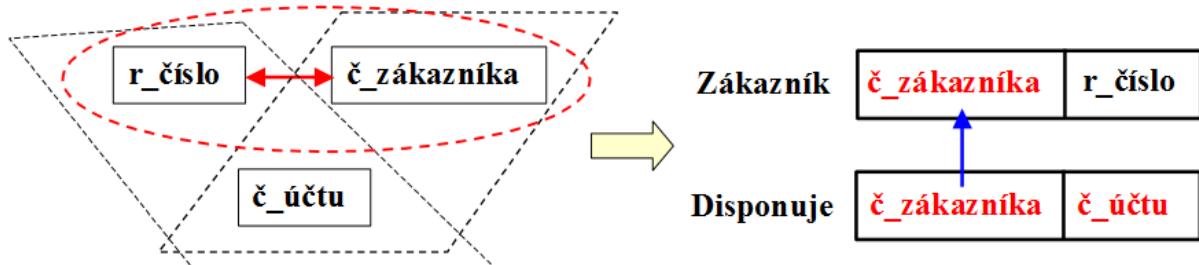
- Relační tabulky splňují třetí normální formu (3NF), jestliže splňují 2NF a žádný atribut, který není primárním klíčem, není tranzitivně závislý na žádném klíči.

č_účtu	stav	pobočka	jmění	r_číslo	č_účtu
100	100000	Jánská	10000000	600528/0275	100
130	50000	Palackého	5000000	581015/9327	100
150	150000	Palackého	5000000	600528/0275	130
				450205/3419	150



BCNF: Atributy, které jsou součástí primárního klíče, musí být vzájemně nezávislé

- Poslední prakticky užívanou formou je tzv. Boyce-Coddova normální forma (BCNF). Tabulka splňuje BCNF, právě když pro dvě množiny atributů A a B platí: A->B a současně B není podmnožinou A, pak množina A obsahuje primární klíč tabulky. Tato forma zjednodušuje práci s tabulkami, ve většině případů, pokud dobře postupujeme při tvorbě tabulek, aby splňovaly postupně 1NF, 2NF a 3NF, forma BCNF je splněna. Ovšem **POZOR, zachování závislostí a dosažení BCNF není vždy splnitelné.**



Vztahy NF

Třída schémat v BCNF je podtřídou třídy schémat 3NF. Třída relací 3NF je podtřídou třídy relací ve 2NF a ta je podtřídou relací 1NF.

- Vždy je možné provést rozklad schématu na několik schémat, která jsou v 3NF a rozklad je bezztrátový a závislosti jsou zachovány.
- Vždy je možné provést rozklad schématu na několik schémat, které jsou v BCNF a rozklad je bezztrátový, ale všechny závislosti nemusí být zachovány.

Dekompozice

Při návrhu relačních databází je potřeba nalézt dobrou množinu relačních schémat, problémem je především opakování stejné informace a dat a nemožnost vyjádřit nějakou informaci či ztráta informace. Problémy řeší dekompozice relačních schémat a normalizace.

Dekompozice:

- všechny atributy původního schématu R se musí objevit v rozkladu (R_1, R_2) : $R = R_1 \cup R_2$
- zpětné spojení musí být bezztrátové - pro všechny možné relace r na schématu R platí $r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$.

SQL

Syntaxe a sémantika příkazů. Příkazy pro dotazování a aktualizaci dat, agregační funkce, triggery a uložené procedury. Příkazy pro definici dat, integritní omezení. Transakční zpracování, jeho vlastnosti, souběžné zpracování transakcí, atomické operace. Základní principy optimalizace dotazů.

Základní pojmy

SQL – (Structured Query Language). Jedná se o standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích. Vývoj byl započat firmou IBM. Příkazy jazyka SQL se dělí na čtyři základní skupiny – příkazy pro manipulaci s daty, příkazy pro definici dat, příkazy pro řízení přístupových práv, ostatní nebo speciální příkazy. Jazyk má několik verzí a standardů.

Standardy SQL – pojmy:

- ANSI – American National Standards Institute, vydává standardy pro jazyk SQL.
- SQL-92 (SQL2) – standard přijatý v roce 1992.
- SQL-1999 (SQL3) – standard přijatý v roce 1999, obsahuje objektové prvky.
- SQL-2008 – nejnovější norma SQL, možno použít ORDER BY vně kurzoru, přidává triggery typu INSTEAD OF a příkaz TRUNCATE.

Transakce – pojem, který označuje posloupnost operací (část programu, sekvenci příkazů jazyka SQL), která přistupuje a aktualizuje (mění) data. Transakce pracuje s konzistentní databází, během spouštění transakce může být databáze v nekonzistentním stavu, ale po úspěšném dokončení transakce musí být databáze konzistentní.

DDL (Data Definition Language) – podjazyk, který slouží pro specifikaci definice schématu databáze. Pomocí tohoto jazyka můžeme definovat, vytvářet, měnit a rušit databázové struktury v databázi. Mezi tyto struktury patří tabulky, indexy, triggery a uložené procedury. Patří sem například CREATE DATABASE, CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, ...

DML (Data Manipulation Language) – podjazyk pro manipulaci s daty, slouží pro práci s obsahem tabulek. Patří sem například INSERT, SELECT, UPDATE, DELETE, ...

DCL (Data Control Language) – podjazyk pro řízení práce s daty, který obsahuje správu transakcí, nastavení práv apod. Obsahuje příkazy pro řízení provozu a údržby databáze. Patří zde příkazy GRANT, REVOKE, REVOKE CREATE USER, ALTER USER, DROP USER, ...

TCC (Transaction Control Commands) – tento jazyk obsahuje příkazy pro řízení transakcí. Patří sem například SET TRANSACTION, COMMIT, ROLLBACK, SAVEPOINT.

Databázové systémy – database + database management system = database system, např. Oracle, Postgresql, SQL server apod.

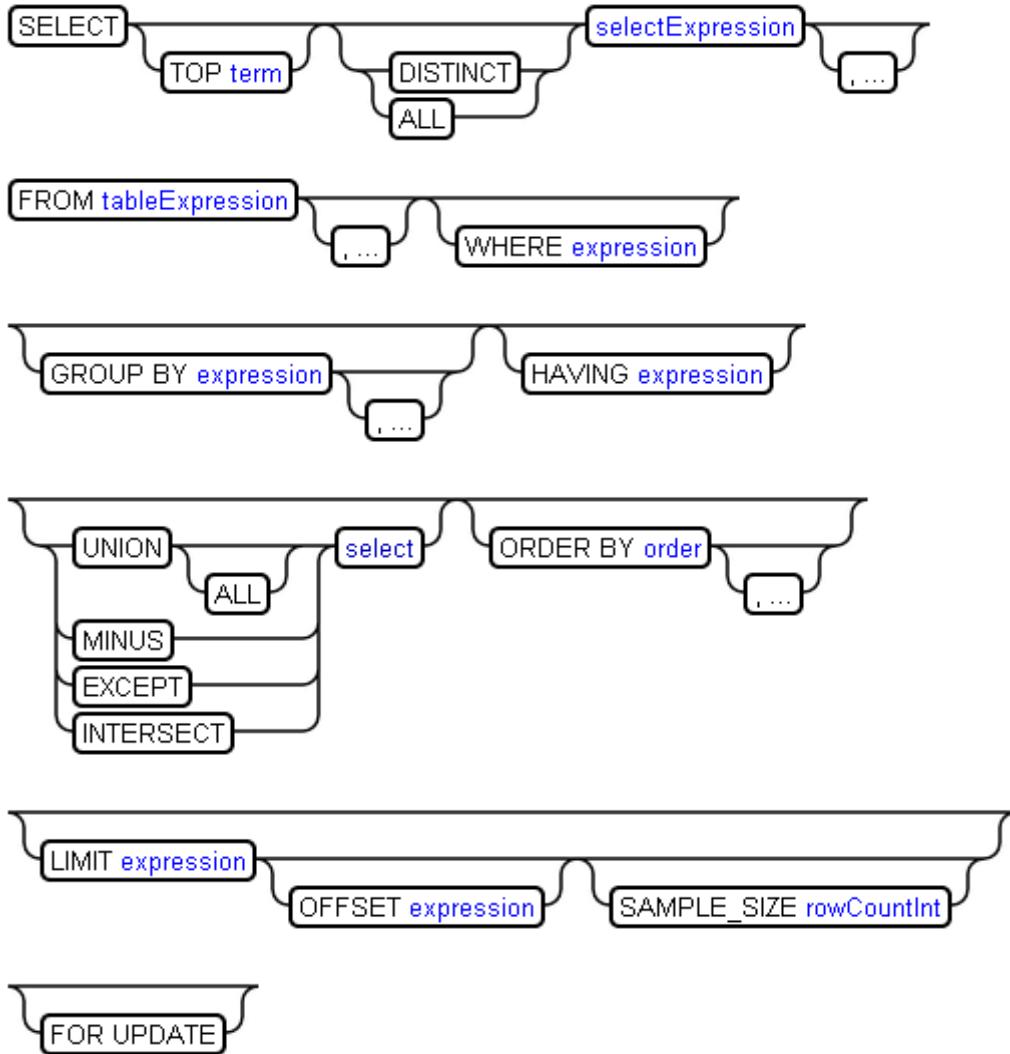
Syntaxe a sémantika příkazů

Syntaxe jazyka SQL připomíná anglický jazyk. Je dobrou zvyklostí psát příkazy *velkými písmeny* a názvy objektů obalovat speciálními znaky, aby tak nedocházelo ke kolizím mezi názvy objektů a rezervovanými klíčovými slovy (tzv. escaping). U MS SQL Serveru jsou těmito znaky hranaté závorky, v případě PostgreSQL uvozovky, MySQL používá zpětné apostrofy. Za každým příkazem může a nemusí být oddělující středník. Řádkové komentáře realizujeme dvěma pomlčkami, blokové klasicky lomítkem a hvězdičkou.

Příkazy jazyka SQL lze popsat pomocí syntaktických diagramů, které reprezentují bezkontextové gramatiky.

Příklad – příkaz SELECT

```
SELECT [ TOP term ] [ DISTINCT | ALL ] selectExpression [,...]
FROM tableExpression [,...] [ WHERE expression ]
[ GROUP BY expression [,...] ] [ HAVING expression ]
[ { UNION [ ALL ] | MINUS | EXCEPT | INTERSECT } select ] [ ORDER BY order [,...] ]
[ LIMIT expression [ OFFSET expression ] [ SAMPLE_SIZE rowCountInt ] ]
[ FOR UPDATE ]
```



Základní elementy jazyka SQL jsou:

- konstanty (101, `text`)
- Integer (5803042157)
- Number (580304.2157)
- datové typy (int, number (m,n), data, varchar(n), long, long raw)
- NULL (speciální hodnota pro prázdnou hodnotu)
- komentáře (/**/)
- objekty databázového schématu (tabulky, pohledy, indexy, sekvence)

Příkazy pro dotazování a aktualizaci dat

Jazyk pro manipulaci s daty je z uživatelského i programátorského hlediska nejčastěji využívaný, protože poskytuje určitou cestu k datům a nabízí možnosti, kterak data v databázi upravovat, mazat apod. Množina příkazů, které se využívají pro výběr, vkládání, úpravu a mazání dat v tabulkách.

- **příkaz SELECT**

- vybírá data z databáze, umožňuje výběr, agregaci a řazení dat (odpovídá projekci v relační algebře)
- příklad: `SELECT id, zakaznik, cena FROM smlouvy WHERE cena>10000 AND se_slevou=1 ORDER BY cena DESC`

- **příkaz UPDATE**

- upraví libovolný počet záznamů z právě jedné tabulky, upravené záznamy musí odpovídat definované podmínce
- aby se příkaz provedl, musí mít uživatel práva na manipulaci dat v databázi
- nové hodnoty nesmí kolidovat s podmínkami (primární klíč, jedinečný index, podmínky CHECK, NOT, NULL)
- příklad: `UPDATE T SET sloupec = sloupec + 1 WHERE podmínka`

- **příkaz DELETE**

- odstraní libovolný počet záznamů z právě jedné tabulky
- příklad: `DELETE FROM tabulka WHERE podmínka`

- **příkaz INSERT**

- vloží záznam do právě jedné tabulky
- hodnoty zadané při INSERT dotazu musí splňovat všechny podmínky pro sloupce (např. primární klíč, podmínky CHECK a NOT NULL)
- pokud nejsou splněny nebo nastane syntaktická chyba, záznam se do tabulky nevloží a databázový stroj (záleží na jeho typu) pošle chybový kód a hlášku
- příklad: `INSERT INTO tabulka (sloupec1, sloupec2) VALUES ('hodnota1', hodnota2')`

- **klíčové slovo DISTINCT**

- slouží k eliminaci duplikátů ve výsledku dotazů (ALL duplikáty ponechává)

- **klaузule WHERE**
 - odpovídá operaci selekce v relační algebře
 - pomocí této klaузule lze definovat podmínu spuštěného dotazu, podmínek může být definováno více
- **klaузule FROM**
 - odpovídá kartézskému součinu v relační algebře
- **klaузule ORDER BY**
 - uspořádá výpis sestupně (DESC) nebo vzestupně (ASC)
 - slouží k řazení výstupní dat podle daného specifika, lze řadit i ve více úrovních
- **klaузule GROUP BY**
 - umožňuje aplikovat souhrnné funkce, klaузule HAVING se aplikuje až po vytvoření skupin
 - agregace záznamů vybíraných podle příkazu SELECT
 - lze volat tzv. aggregační funkce
 - použití při získání počtu záznamů odpovídajících každé jednotlivé hodnotě, získání součtu, aritmetického průměru či jiných statistických hodnot
- **operace JOIN**
 - dovoluje vytvořit propojení mezi více tabulkami, vrací jednu relaci //odstranit
 - hlavní typy spojení – vnější, vnitřní //odstranit
 - Operace spojení vezme dvě relace a jako výsledek vrátí jednu relaci.
 - Tyto operace jsou typicky používány jako poddotazy v klaузuli from.
 - Podmínka spojení (join condition) – definuje, které n-tice ve dvou relacích si odpovídají a které atributy jsou ve výsledku spojení.
 - Typ spojení (join type) – definuje kolik je n-tic v každé relaci, které neodpovídají žádné n-tici v jiné relaci.
 - Typy spojení:
 - inner join
 - left outer join
 - right outer join
 - full outer join
 -
 -
 -

- Podmínky spojení:
 - natural
 - on <predicate>
 - using <A1,A2,...,An>

Príklad:

půjčka (pobočka-jméno, půjčka-číslo, částka)

půjčovatel (základník-jméno, půjčka-číslo)

Najděte všechny zákazníky, kteří mají v bance buď účet nebo půjčku (ale ne obojí).

```
1 SELECT zákazník-jméno
2 FROM (vkladatel NATURAL FULL OUTER JOIN půjčovatel)
3 WHERE číslo-účtu IS NULL OR půjčka-číslo IS NULL
```

- **klauzule HAVING**

- za tímto výrazem lze definovat podmínu, která operuje s agregačními funkcemi

- **klauzule IN, NOT IN**

(0)	(0)	(0)
5 in (4) = true	5 in (4) = false	5 not in (4) = true
(5)	(6)	(6)

Najděte všechny zákazníky, kteří mají v bance účet i půjčku.

```
1 SELECT DISTINCT zákazník-jméno FROM půjčovatel WHERE zákazník-jméno IN
(SELECT zákazník-jméno FROM vkladatel)
```

Najděte všechny zákazníky, kteří mají v bance půjčku, ale ne účet.

```
1 SELECT DISTINCT zákazník-jméno FROM půjčovatel WHERE zákazník-jméno NOT IN
(SELECT zákazník-jméno FROM vkladatel)
```

- **klauzule SOME, ALL**

(0)	(0)
5 < some (5) = true	5 < all (4) = false
(9)	(6)

Najděte pobočky, které mají větší aktiva než nějaká pobočka v Praze.

```
1 SELECT pobočka-jméno FROM pobočka WHERE aktiva > SOME (SELECT aktiva FROM
pobočka WHERE pobočka-město = „Praha“)
```

Najděte pobočky, které mají větší aktiva než všechny pobočky v Praze.

```
1 SELECT pobočka-jméno FROM pobočka WHERE aktiva > ALL (SELECT aktiva FROM
pobočka WHERE pobočka-město = „Praha“)
```

Agregační funkce

Jazyk SQL obsahuje vestavěné funkce, mezi ně patří především tzv. aggregační funkce. Mimo jiné zde ale patří také funkce pro práci s řetězci (SUBSTRING, LOWER), pro zjištění aktuálního data a času a další. Množina dostupných funkcí je často závislá na zvoleném databázovém systému.

Agregační funkce jsou v SQL statické funkce, pomocí kterých systém řízení báze dat umožňuje seskupit vybrané řádky dotazu (příkazem SELECT) a spočítat nad nimi výsledek určité aritmetické nebo statistické funkce.

- **funkce COUNT()**

- od ostatních funkcí se odlišuje:
 - COUNT() vrací počet záznamů, které vyhovují zadané podmínce, respektive seskupení, proto je v zásadě jedno, který ze sloupců má jako argument (často se používá hvězdičková konvence)
 - výjimkou je případ, kdy je potřeba vrátit počet unikátních hodnot určitého sloupce použitého v tabulce, pak se ke COUNT() přidává klíčové slovo DISTINCT
 - pokud je aplikována ve tvaru COUNT (název sloupce), tak je vrácen počet záznamů, které nemají ve sloupci název sloupce hodnotu NULL //odstranit'
 - pokud je aplikována ve tvaru COUNT (název_sloupce), tak je vrácen počet záznamů, které nemají ve sloupci 'název_sloupce' hodnotu NULL
- příklad: `SELECT COUNT (DISTINCT sloupec) FROM tabulka, SELECT COUNT(*) FROM tabulka`

- **GROUP_CONCAT()**

- speciální aggregační funkce, kterou nabízejí některé databázové systémy (např. MySQL)
- výsledek není počet ale výčet nalezených hodnot, oddělených čárkou (či jiným oddělovačem)
- lze ji kombinovat s klíčovým slovem DISTINCT pro eliminaci vícekrát se vyskytujících hodnot
- příklad: `SELECT sloupec1 GROUP_CONCAT (DISTINCT sloupec2 ORDER BY sloupec 2 DESC SEPARATOR ',') FROM tabulka GROUP BY sloupec1`

- **AVG()**

- vrací průměr dané hodnoty ve všech záznamech
- příklad: `SELECT AVG (průměr) FROM tabulka`

- **SUM()**
 - součet všech hodnot daného sloupce
 - příklad: SELECT SUM (součet) FROM tabulka
- **MIN(), MAX()**
 - minimální, respektive maximální hodnota daného sloupce v záznamech
 - příklad: SELECT MIN (minimunum), MAX (maximum) FROM tabulka

Všechny agregační funkce s výjimkou cot(*) ignorují n-tice s nulovými hodnotami na souhrnných atributech.

Funkce pro práci s datem a časem

CURDATE() – aktuální datum, DAY() – den, MONTH() – měsíc, YEAR() – rok, DATEDIFF() – rozdíl dvou dat ve dnech, TIMEDIFF() – rozdíl dvou časů, CURTIME() – aktuální čas, HOUR() – hodiny, MINUTE() – minuty, SECOND() – sekundy

Uložené procedury

Jsou databázové objekty, které neobsahují data, ale programy, které se nad daty v databázi mají vykonávat. Mají své parametry a lokální proměnné, které nejsou zvenku vidět.

Uložená procedura je uložená (rozuměj: uložená v databázi). To znamená, že se k ní lze chovat stejně jako ke každému jinému objektu databáze (indexu, pohledu, triggeru apod.). Lze jí založit, upravovat a smazat pomocí příkazů dotazovacího jazyka databáze.

Pro psaní uložených procedur je obvykle používán specifický jazyk konkrétní databáze, který je rozšířením jejího dotazovacího jazyka. V případě Oraclu to je procedurální jazyk PL/SQL, u SQL Serveru se jedná o T-SQL.

Jednoduché přiblížení činnosti uložené procedury spočívá v představě sestaveného podprogramu nebo funkce, která bude následně uložena jako nový objekt databáze. Uložené procedury běží na SQL serveru, a nikoli na klientu, který zadal dotaz. Uložené procedury zpravidla obsahují posloupnost příkazů jazyka SQL a příkazy pro řízení běhu, které zpracovávají tabulky z databáze.

Speciálním druhem uložené procedury je trigger.

Výhody

- pomocí uložených procedur lze vykonat složitější operace, než bychom vykonali přímo pomocí SQL
- získáváme na výkonu v porovnání s posloupností standardních SQL příkazů, při prvním provádění procedury je vytvořen prováděcí plán činnosti, plán je poté uložen do vyrovnávací paměti a následné opakování téže procedury je pak mnohem rychlejší než provádění obdobných příkazu SQL
- nepřerušitelnost provádění procedury, po spuštění je procedura provedena sekvenčně jako celek

- správná konstrukce zajišťuje minimum kolizí s integritními omezeními
- možnost nastavování a předávání parametrů umožnuje propojení více uložených procedur a tím lze vytvářet i jakési dávkové programy
- umožňují vytvořit další vrstvu zabezpečení databáze, uživatelům pak stačí přidělovat práva pro spuštění procedur
- velká část aplikační logiky může být přesunuta na databázový server, díky uloženým procedurám s vlastními proměnnými, podmínkami, cykly, popř. vyvolávání věstavěných funkcí databázového serveru

Nevýhody

- nutnost správy procedur (parametry, vhodnost použití)
- často nutnost učit se další jazyk a příkazy

Vytvoření procedury

Stejně jako u ostatních programovacích jazyků je procedura blok kódu se vstupními parametry. Vstupní hodnoty se pak v těle procedury zpracují a výsledkem může být nějaká úprava údajů z databázové tabulky, popřípadě lze odevzdat výsledek výpočtu či vytvořený řetězec.

Příklad:

```
create procedure název_procedury (seznam_parametrů) as
begin
tělo procedury
end
```

Proceduru pak spouštíme příkazem execute název_procedury;

Vstupní parametry každé procedury jsou dány svými názvy a příslušnými datovými typy. Seznam parametrů je při vytváření procedury uveden v kulatých závorkách za názvem procedury.

Spouště (Triggery)

V databázi specifikuje činnosti, které se mají provést v případě definované události nad databázovou tabulkou. Definovanou událostí může být například vložení nebo smazání dat. Často slouží pro složitější kontrolu integrity dat.

Pro aktivaci tohoto mechanismu musíme:

- Specifikovat podmínky, za jakých je spouštěč prováděn.
- Specifikovat akce, které se budou dít při jeho spuštění.

Triggery jako takové jsou definovány ve většině moderních databázových systémů, ovšem mírně se liší v sémantice svého provedení.

Klíčové rozdíly jsou zejména v:

- kdy přesně se trigger spustí
- jak proběhne (co ho může přerušit)
- jakým způsobem se řeší vzájemné volání triggerů, pokud je vůbec umožněno
- jak (a jestli vůbec) jsou ošetřeny nekonečné cykly vzájemného volání

Syntaxe triggeru (Oracle)

CREATE [OR REPLACE] TRIGGER jmeno {BEFORE | AFTER | INSTEAD OF}

udalost [FOR EACH ROW] [WHEN (podminka)]

blok

- událost je UPDATE, DELETE nebo INSERT, případně jejich kombinace (zpravidla oddělené klíčovým slovem OR)
- BEFORE triggery jsou spouštěny před provedením události, AFTER triggery až po provedení události
- Trigger je spuštěn buďto jednou při provádění příkazu (default) nebo pro každý měněný řádek (FOR EACH ROW)
- Podmínky: INSERTING, UPDATING, DELETING

Príklad použitia triggeru:

Vytvorte trigger, ktery pri zmene uvazku prepocita zamestnanci mzdu.

-- Tj. aktualujte hodnotu mzda v tabulce zamestnanci.

```
1 CREATE OR REPLACE TRIGGER uvazky_vypocti_plat
2 AFTER INSERT OR UPDATE OR DELETE ON uvazky FOR EACH ROW
3 DECLARE
4     zamid NUMBER;
5     diff NUMBER :=0;
6 BEGIN
7     IF INSERTING OR UPDATING THEN
8         zamid:=:NEW.id_zam;
9         diff:=diff+(:NEW.uvazek * :NEW.tarif + :NEW.osobni);
10    END IF;
11
12    IF DELETING OR UPDATING THEN
13        zamid:=:OLD.id_zam;
14        diff:= diff - (:OLD.uvazek * :OLD.tarif + :OLD.osobni);
15    END IF;
16
17    UPDATE zamestnanci SET mzda = mzda + diff WHERE id = zamid;
18 END;
19 /
```

Příkazy pro definici dat (Data Definition Language)

Definuje sadu příkazů, které lze použít pro vytváření, úpravu a odstraňování objektů v databázi.

Nejčastěji spravovanými objekty jsou: tabulky, pohledy, procedury, funkce.

Umožňuje specifikaci nejen množin atributů, ale také informaci o každé relaci, zahrnující:

- Schéma každé relace.
- Doménu hodnot spojenou s každým atributem.
- Omezení integrity.
- Množinu indexů, které budou udržovány pro každou relaci.
- Bezpečnostní a autorizační informace o každé relaci.
- Fyzickou strukturu uložení na disk pro každou relaci.

Příkazy

- **CREATE**

- vytvoření objektu
- syntax: CREATE TABLE název_tabulky (atribut1 doména1, ..., omezení integrity)
- příklad: CREATE TABLE název_tabulky (atribut1 doména1, ..., omezení integrity)
//odstranit
- omezení integrity může být
 - not null
 - primary key (atribut1, ..., atribut_n)
 - check (predikát)

Príklad:

```
1 CREATE TABLE uvazky (
2     id_zam NUMBER NOT NULL REFERENCES zamestnanci,
3     id NUMBER PRIMARY KEY,
4     uvazek NUMBER CHECK (uvazek > 0 AND uvazek <= 1),
5     tarif NUMBER NOT NULL,
6     osobni NUMBER DEFAULT 0 NOT NULL,
7     popis VARCHAR2(200)
8 );
```

- **ALTER**
 - úprava
 - v případě změny tabulky se může jednat o přidání/úpravu/odstranění sloupce, změny datového typu sloupce, změny relace a jiných integritních omezení
 - příklad: ALTER TABLE název_tabulky ADD atribut doména
 - může se jednat také o odstranění atributů z relace
 - příklad: ALTER TABLE název_tabulky DROP atribut
- **DROP**
 - odstranění
 - příklad: DROP název_tabulky

Doménové typy v SQL

- **char(n).** Řetězec znaků s pevnou délkou n.
- **varchar(n).** Řetězec znaků s proměnlivou délkou (maximálně n).
- **int.** Celé číslo; závislé na implementaci.
- **smallint.** Krátké celé číslo; závislé na implementaci.
- **numeric(p,d).** Číslo s pevnou desetinnou čárkou s přesností na p míst s d místy vpravo od desetinné tečky.
- **real, double precision.** Čísla s plovoucí desetinnou čárkou; závislé na implementaci.
- **float(n).** Číslo s plovoucí desetinnou čárkou s přesností nejméně na n míst.
- **date.** Datumy obsahující (4bitový) rok, měsíc a den.
- **time.** Čas v hodinách, minutách a sekundách.

Ve všech doménových typech jsou povoleny nulové hodnoty. Deklarování atributu not null je zakazuje nulové hodnoty pro daný atribut.

- V SQL-92 vytvoří konstrukce create domain uživatelské doménové typy.

Integritní omezení

Omezení integrity zabraňují poškození databáze; zajišťují, že autorizované zásahy do databáze nezpůsobí ztrátu konzistence dat. Omezení integrity testují hodnoty vkládané do databáze a kontroluje dotazy, aby bylo zajištěno, že porovnávání dávají smysl.

Omezení domény (viz. také Doménové typy v SQL)

Omezení domény je základní formou omezení integrity. Klauzule check dovoluje v SQL-92 omezit domény. Příklad: create domain doména numeric(5,2) constraint value-test check (value >= 4.00). Doména je deklarována jako dekadické číslo s pěti číslicemi, z nichž 2 jsou za desetinnou čárkou. Klauzule constraint je volitelná, užitečná pro indikaci, které omzení bylo při aktualizaci porušeno.

Referenční integrita

Zajišťuje, že hodnota, která se objeví v jedné relaci pro danou množinu atributů, se také objeví v určitých množinách atributů v jiné relaci.

Formální definice: Nechť r_1 (R_1) a r_2 (R_2) jsou relace s primárními klíči K_1 a K_2 . Podmnožina α z R_2 je cizí klíč odkazující na K_1 v relaci r_1 , jestliže pro každé t_2 v r_2 musí být n-tice v taková, že $t_1[K_1] = t_2[\alpha]$. Omezení referenční integrity: $\Pi\alpha(r_2) \subseteq \Pi K_1(r_1)$.

Referenční integrita v E-R modelu

Uvažujme množinu vztahů R mezi množinami entit E_1 a E_2 . Relační schéma pro R zahrnuje primární klíče K_1 z E_1 a K_2 z E_2 . Pak K_1 a K_2 jsou v R cizí klíče pro relační schémata E_1 a E_2 . Slabé množiny entit jsou též zdroje pro omezení referenční integrity. Proto musí relační schéma pro každou slabou množinu entit zahrnovat primární klíč množiny vztahů, na které závisí, tj. nadřazené entitní množiny.

Referenční integrita v SQL

Primární a kandidátní klíče a cizí klíče můžou být specifikovány jako část SQL příkazu create table.

- Klauzule primary key v příkazu create table zahrnuje seznam atributů, které tvoří primární klíč.
- Klauzule unique key v příkazu create table zahrnuje seznam atributů, které tvoří kandidátní klíč.
- Klauzule foreign key v příkazu create table zahrnuje seznam atributů, které tvoří cizí klíč a jméno relace odkazované cizím klíčem.

Transakční zpracování

Transakce je posloupnost operací (část programu), která přistupuje a aktualizuje (mění) data.

Transakce pracuje s konzistentní databází. Během spouštění transakce může být databáze v nekonzistentním stavu. Ve chvíli, kdy je transakce úspěšně ukončena, databáze musí být konzistentní.

Dva hlavní problémy:

- Různé výpadky, např. chyba hardware nebo pád systému.
- Souběžné spouštění více transakcí.

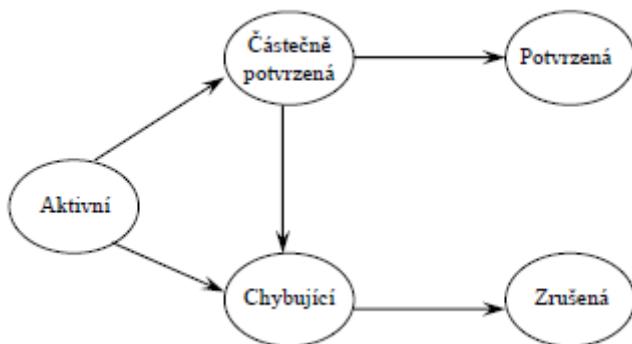
Vlastnosti transakcí (ACID vlastnosti)

- A: Atomic – celá se provede, nebo se odvolá, žádné zbytky nebo vedlejší efekty.
- C: Consistent – na konci není porušeno žádné omezení databáze.
- I: Isolated – operace jsou izolovány od ostatních transakcí.
- D: Durable – po ukončení transakce jsou data trvale uložena.

Každá transakce má určitou úroveň izolace. Pokud se při vytvoření nespecifikuje, použije se výchozí úroveň.

Stavy transakce

- **Aktivní** – počáteční stav; transakce zůstává v tomto stavu, dokud běží.
- **Částečně potvrzená (Partially Committed)** – jakmile byla provedena poslední operace transakce.
- **Chybuje (Failed)** – po zjištění, že normální běh transakce nemůže pokračovat.
- **Zrušená (Aborted)** – poté, co byla transakce vrácena (rolled back) a databáze byla vrácena o stavu před spuštěním transakce. Dvě možnosti po zrušení transakce:
 - Znovu spustit transakci – pouze pokud nedošlo k logické chybě.
 - Zamítнуть transakci.
- **Potvrzená (Committed)** – po úspěšném dokončení.



Souběžné zpracování transakcí

Více transakcí může být spouštěno současně. **Výhody jsou:**

- Zvýšené využití procesoru a disku, které vede k vyšší transakční propustnosti (jedna transakce může používat procesor a jiná disk).
- Snížená průměrná doba odezvy (krátká transakce nemusí čekat na dokončení dlouhé).

Nevýhody:

- Aby se předešlo problému s přepisováním dat, je nutné zamykání řádků či tabulek.

- Je nutná režie na řešení a předcházení deadlocku.

Deadlock

Neboli uváznutí, označuje stav, kdy transakce A čeká na prostředky, které drží transakce B a transakce B čeká na prostředky, které má uvolnit transakce A. Vždy se řeší zaříznutím jedné z transakcí, nelze čistě vyřešit a je nutné jim předcházet.

Neopakovatelné čtení označuje stav, kdy si jedna transakce přečetla záznam, provedla nějaké příkazy a pak si tento záznam přečetla znovu a už měl jiné hodnoty.

Fantom označuje data, která na první pohled „záhadně“ unikají aktualizacím.

Plány

Plány jsou posloupnosti, které určují chronologické pořadí provádění instrukcí souběžných transakcí. Plán pro množinu transakcí musí obsahovat všechny operace prováděné těmito transakcemi a musí zachovávat pořadí instrukcí stejné jako v každé jednotlivé transakci.

T_1	T_2
$\text{čti}(A)$ $A := A - 50$ $\text{zapiš}(A)$ $\text{čti}(B)$ $B := B + 50$ $\text{zapiš}(B)$	$\text{čti}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{zapiš}(A)$ $\text{čti}(B)$ $B := B + \text{temp}$ $\text{zapiš}(B)$

Serializovatelnost

Základní předpoklad – každá transakce zachovává konzistenci databáze. Tedy i sériový plán zachovává konzistenci databáze.

Plán je serializovatelný, pokud je ekvivalentní sériovému plánu. Různé formy ekvivalence plánů vedou k následujícím pojmem:

- Serializovatelnost podle konfliktu.
- Pohledová serializovatelnost.

Implementace atomičnosti a trvanlivosti

Databázový podsystém správy obnovy (recovery-management) implementuje podporu pro atomičnost a trvanlivost transakcí.

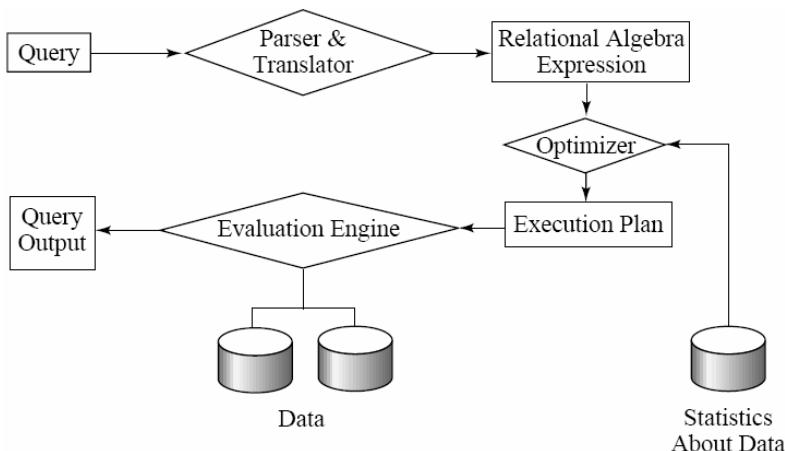
Schéma stínové databáze:

- Předpokládejme, že v jednu chvíli je aktivní pouze jedna transakce.
- Ukazatel db_pointer vždy ukazuje na současnou konzistentní kopii databáze.

- Všechny změny jsou prováděny ve stínové kopii databáze a db_pointer začne ukazovat na tuto změněnou stínovou kopii až poté, co transakce dosáhne stavu „částečně potvrzená“ a všechny změny jsou promítnuty na disk.
- V případě chyby transakce je použita stará konzistentní kopie (ukazuje na ni db_pointer) a stínová kopie může být smazána.
- Toto schéma předpokládá, že nemůže selhat disk.
- Užitečné pro textové editory, ale velmi neefektivní pro velké databáze, spuštění transakce vyžaduje vytvoření kopie celé databáze.

Optimalizace dotazů

Základní kroky ve zpracování dotazů: analýza dotazu (parsing) a překlad, optimalizace, vyhodnocení.



Analýza dotazu (parsing) a překlad:

- Přelož dotaz do interní reprezentace, která je následně přeložena do relační algebry
- Analyzátor ověří správnou syntaxi a ověří existenci relací

Optimalizace – nalezení nejlevnějšího plánu pro vykonání dotazu:

- Mějme výraz v relační algebře, tento výraz může mít několik ekvivalentních (produkují stejný výsledek) výrazů
Např. $\sigma_{zůstatek < 2500}(\pi_{zůstatek}(\text{účet}))$ je ekvivalentní výrazu:
 $\pi_{zůstatek}(\sigma_{zůstatek < 2500}(\text{účet}))$
- Jakýkoli výraz v relační algebře může být vyhodnocen mnoha způsoby.
Komentovaný výraz určující detailně postup vyhodnocení se nazývá plán pro vyhodnocení.
Např. má se použít index na atributu zůstatek k nalezení účtů se zůstatkem < 2500, nebo se má použít sekvenční průchod celého souboru a vynechat všechny účty s zůstatkem ≥ 2500 ?
- Mezi všemi možnými výrazy se snažíme najít ten, který má nejlevnější plán pro vyhodnocení. Odhad ceny plánu pro vyhodnocení je založený na statistických informacích v databázovém katalogu.

Vyhodnocení:

- Stroj pro vyhodnocení dotazu (query-execution/evaluation engine) bere na

vstupu plán pro vyhodnocení dotazu, spustí plán a vrátí výsledky dotazu.

Optimalizace = nalezení nejlevnějšího plánu pro vykonání dotazu. //môže sa odstrániť

Jedná se především o minimalizaci nákladů na:

- zdrojový čas,
- kapacitu paměti (prostor),
- programátorskou práci.

Existuje celá řada způsobů, jakými lze ovlivnit rychlosť zpracování dotazu, níže je seznam těch základních.

- **Indexy** – nejdůležitější technika urychlení dotazů, systém si vede informace o záznamech v podobě stromu.
- **Vyjmenování sloupců** – typický dotaz "SELECT * FROM [Products]" vybere všechny sloupce z tabulky, které možná ani nepotřebujeme, ale přinese také zpomalení z toho důvodu, že systém si musí nejdříve vytáhnout seznam sloupců z definice tabulky, aby je mohl zobrazit. Navíc takový dotaz není odolný proti změnám schématu tabulky.
- **Použití operátoru LIKE** – Obecně není vhodné operátor používat ve tvaru LIKE '%Karel%', v tom případě musí databáze projít všechny zbývající výskyty. To při omezení vhodnou podmínkou nemusí vadit. Tvar LIKE 'Karel%' je naprosto v pořádku a indexy v tomto případě fungují, stejně tak tvar bez procent.
- **Dočasné tabulky** – jejich použití může přinést výrazné zpomalení, pokud nejsou používány s citem a manipuluje se v nich s velkým množstvím dat.

Většina dnešních nástrojů nabízí možnost zobrazení plánu provedení dotazů, který velice usnadní jejich ladění.

Obecná pravidla pro psaní SQL dotazů:

- vyjmenovat sloupce,
- používat co nejméně klauzuli LIKE,
- používat co nejméně klauzule IN, NOT IN,
- používat klauzule typu LIMIT,
- na začátek dávat obecnější podmínky,
- výběr vhodného pořadí spojení,
- používat hinty,
- nastavit indexy.

Další pojmy

Pohledy

Pohled je databázový objekt, který uživateli poskytuje data ve stejné podobě jako tabulka. Na rozdíl od tabulky, kde jsou data přímo uložena, obsahuje pohled pouze předpis, jakým způsobem mají být data získána z tabulek a jiných pohledů.

V některých případech není vhodné, aby všichni uživatelé viděli celý logický model databáze (tj. všechny relace aktuálně uložené v databázi).

Definice pohledu: `create view v as <výraz dotazu>`, kde `<výraz dotazu>` je jakýkoliv správný výraz relační algebry. Jméno pohledu je reprezentováno proměnnou `v`.

Úrovně izolovanosti transakcí

Existují 4 druhy úrovní izolace dle standardu:

- **Read Uncommitted** – nejvíce tolerantní. Nepovoluje souběžné aktualizace, ale povoluje čtení doposud nepotvrzených změn.
- **Read Committed** – méně tolerantní, nepovoluje souběžné aktualizace ani čtení nepotvrzených změn. S jejím použitím lze dosáhnout vysoké propustnosti v databázi, ale při jejím použití nelze vyloučit neopakovatelné čtení a fantomy (viz. pojmy výše).
- **Repeatable Read** – má stejné vlastnosti jako Read Committed jen s tím rozdílem, že podporuje stabilní pohled na čtená data (tj. zamezuje neopakovatelnému čtení), ale stále mohou vznikat fantomy.
- **Serializable** – nejvyšší úroveň zabezpečení a blokování použitých prostředků. V případě častého používání transakcí se serializovatelným čtením může docházet k uváznutí a mají vliv na výkon systému, protože k jejich zpracování je třeba značené režie ze strany databázového systému.

Zdroje

- <<http://www.gjszlin.cz/ivt/esf/php/php-sql-a-funkce.php>> – SQL a funkce
<<http://statnice.dqd.cz/home:prog:ap10>> – otázka SQL z fakultní státnicové wiki
<http://cs.wikipedia.org/wiki/Agrega%C4%8Dn%C3%AD_funkce> – agregační funkce
<<http://cs.wikipedia.org/wiki/SQL>> – SQL
<<http://is.muni.cz/el/1433/jaro2012/PV003/um/PV003-slide.pdf>> – architektura RDBS (PV003)
<http://is.muni.cz/el/1433/jaro2012/PV063/um/Uvod_datove_modelovani.pdf> – uložené procedury (PV063)
<http://www.fi.muni.cz/~kripac/PV136/dusek/opt_sql_dot.pdf> – optimalizace

Počítačové sítě

Modely vrstev počítačových sítí (ISO/OSI, TCP/IP): funkcionality a součinnost vrstev, adresace. Fyzická vrstva, signály a jejich kódování, řízení přístupu k médiu. Propojování počítačových sítí. Protokoly přístupu k médiu. Síťové protokoly, přepínání a směrování, multicast. Zajištěný přenos dat, sestavení a ukončení spojení. Transportní protokoly.

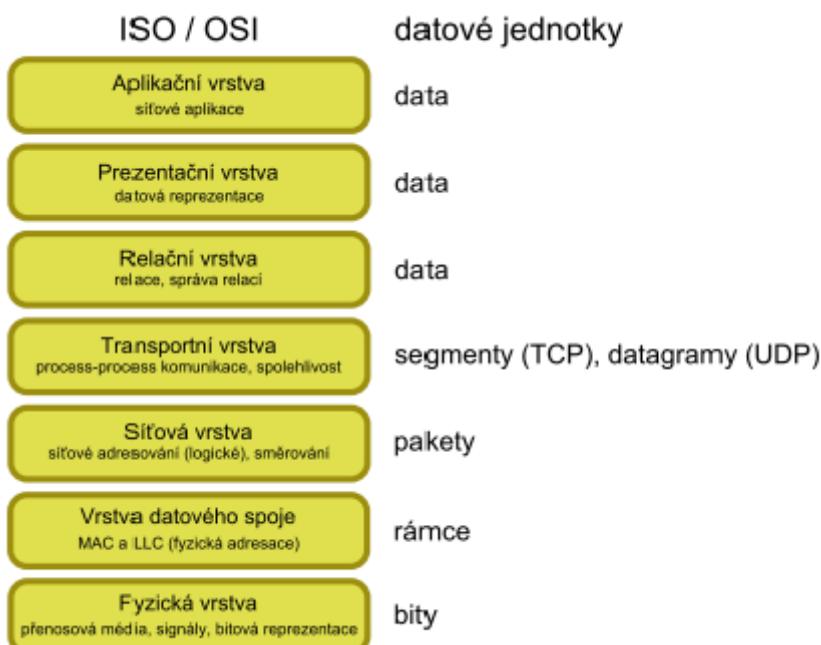
ISO/OSI Model (ISO = název organizace, OSI = jméno modelu)

7-vrstvý model navržen organizací ISO za účelem zajištění **kompatibility** (*schopnost 2 zařízení pracova dohromady, zaměnitelnost*) a **interoperability** (*schopnost kspolupráce mezi různorodými a organizačně nezávislými komponentami*) komunikčních systémů. různých výrobců.

důvody vrstevnaté architektury:

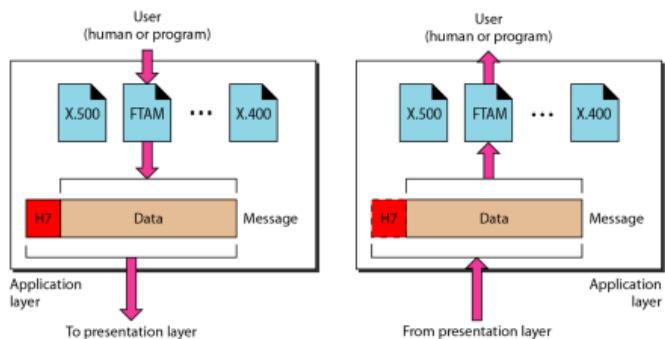
- každá z vrstev je **zodpovědná za určitou (definovanou) funkcionalitu**
 - aby mohla požadovanou funkcionalitu zajistit, přidává si do přenášených dat své řídící informace
- každá vrstva **komunikuje pouze se svými přímo sousedícími vrstvami**
 - každá vrstva využívá služeb poskytovaných vrstvou nižší a poskytuje své služby vrstvě vyšší
 - funkcionalita je izolována v rámci příslušné vrstvy (pokud dojde ke změně vrstvy, je zapotřebí upravit pouze vrstvy s ní přímo sousedící)
- z logického pohledu se komunikace odehrává pouze mezi stejnými vrstvami (tzv. peers) obou komunikujících stran; ve skutečnosti však zasílaná data prochází všemi nižšími vrstvami
- vrstvy jsou pouze abstrakcí funkcionality – skutečné implementace se více či méně liší

7 vrstev nebylo komunitou široce akceptováno ⇒ **TCP/IP model**



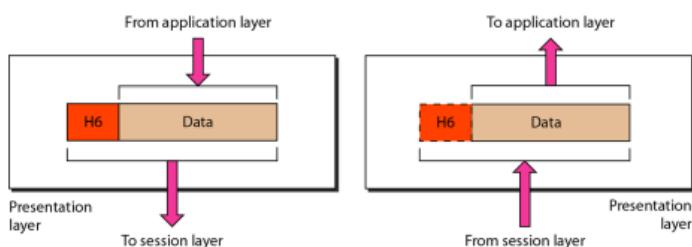
1. APLIKAČNÍ VRSTVA

- představuje rozhraní mezi uživatelem (člověkem) a počítačovou sítí
- zahrnuje *síťové aplikace/programy a síťové protokoly*
 - síťovou aplikací požadovaná data jsou balena do aplikačních protokolů a předána přezentovační vrstvě.



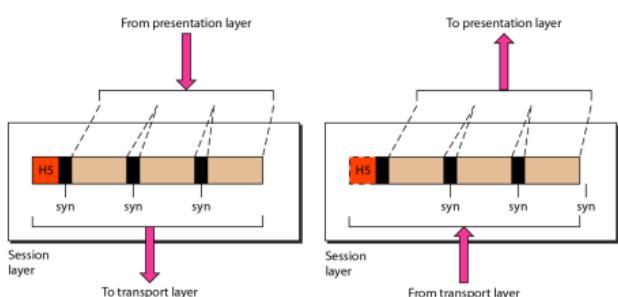
2. PREZENTAČNÍ VRSTVA

- zajišťuje jednotnou reprezentaci dat na obou komunikujících stranách
- v rámci TCP/IP modelu se předpokládá, že tato funkcionality je zajištěna samotnou aplikací.



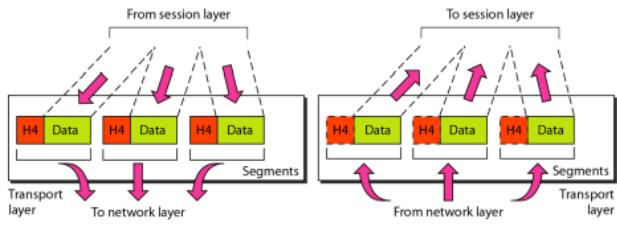
3. RELAČNÍ VRSTVA

- spravuje ustavená spojení (= relace) mezi komunikujícími aplikacemi
- v rámci TCP/IP modelu se předpokládá, že tato funkcionality je zajištěna samotnou aplikací, aplikačním protokolem



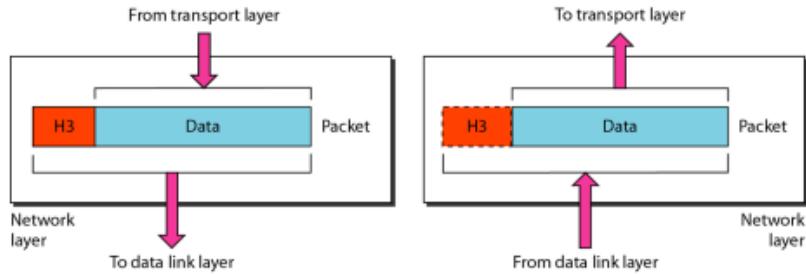
4. TRANSPORTNÍ VRSTVA

- zajišťuje identifikaci (= adresaci) a doručení dat (segmentů, datagramů) mezi dvěma komunikujícími procesy s případným zajištěním spolehlivosti přenosu.



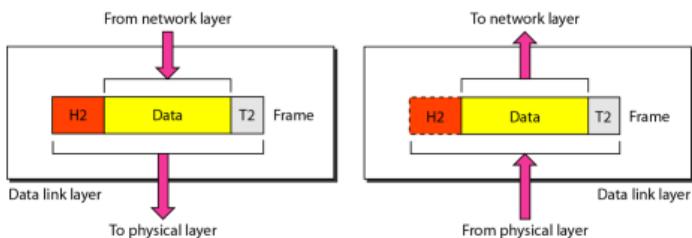
5. SÍŤOVÁ VRSTVA

- zajišťuje identifikaci (= adresaci) a doručení dat (paketů) mezi dvěma komunikujícími uzly
- součástí je také nalezení vhodné cesty mezi komunikujícími uzly (směrování)



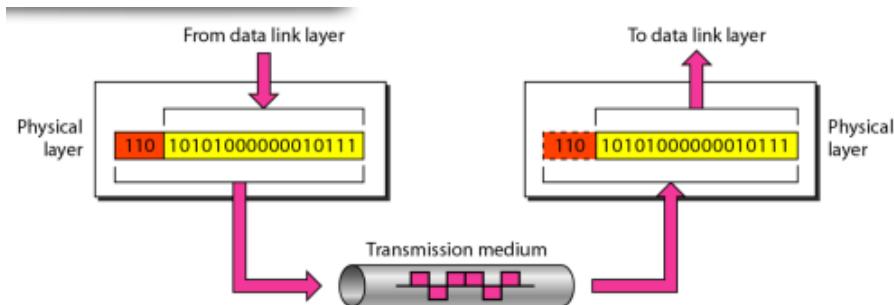
6. VRSTVA DATOVÉHO SPOJE (SPOJOVÁ VRSTVA)

- zajišťuje přenos dat (rámců) mezi dvěma komunikujícími uzly propojenými sdíleným přenosovým médiem, včetně řízení přístupu k tomuto sdílenému médiu

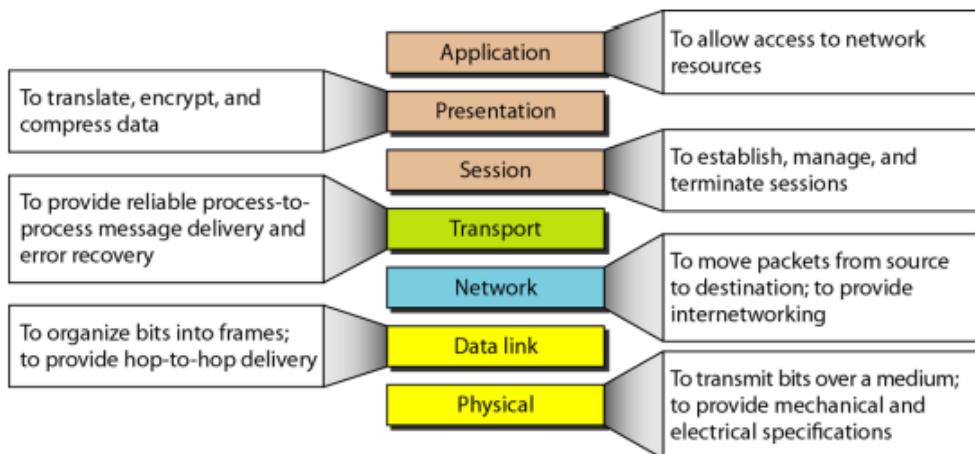


7. FYZICKÁ VRSTVA

- řídí děje v přenosovém médiu
- rozhoduje např. o vysílání / příjmů přenášených dat (bitů), kódování dat do signálů atp.

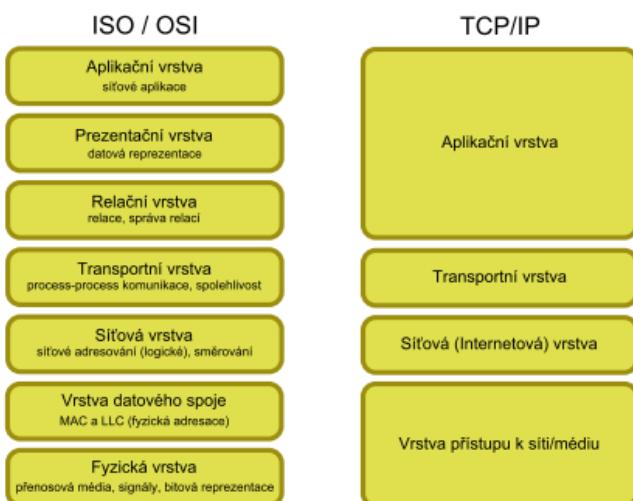


Vrstvy ISO/OSI shrnutí



TCPI/IP zjednodušený model ISO/OSI

model I:

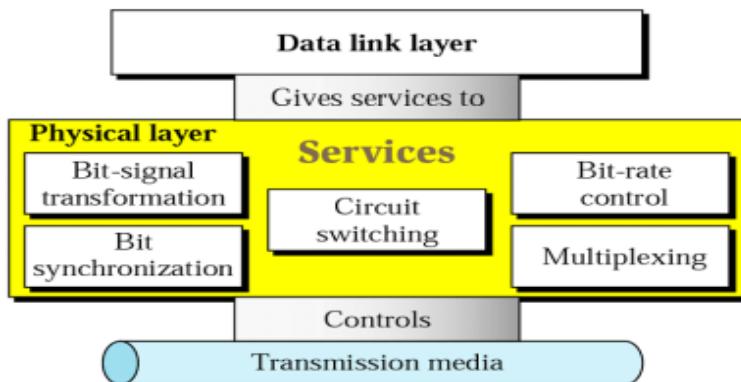


FYZICKÁ VRSTVA

- data mezi komunikujícími uzly přenášeny přenosovým médiem. Přenosové médium = pasivní entita, žádná logika řízení

- poskytuje funkcionalitu pro spolupráci s přenosovým médiem
- poskytuje služby pro *vrstvu datového spoje*
 - vrstva datového spoje předává do (získává z) fyzické vrstvy data vztahem posloupnosti 0 a 1, seskupená do rámů
 - fyzická vrstva transformuje bitový obsah rámů do signálů šířených přenosovým médiem

- řídí děje v přenosovém médiu; rozhoduje např. o:
 - vysílání / příjmu přenášených dat (signálů)
 - kódování dat do signálů
 - počtu logických kanálů přenášejících data y různých zdrojů souběžně



Obrázek: Ilustrace služeb fyzické vrstvy.

Hlavní cíl: zajistit přenos jednotlivých bitů (= obsah předaných rámců) mezi odesílatelem a příjemcem. Zprostředkovává tak logickou cestu, kterou cestují zasílané bity.

Služby

- **Bit-to-Signal Transformation** - representing the bits by a signal – electromagnetic energy that can propagate through medium
- **Bit-Rate Control** - the number of bits sent per second
- **Bit Synchronization** - the timing of the bit transfer (synchronization of the bits by providing clocking mechanisms that control both sender and receiver)
- **Multiplexing** - the process of dividing a link (physical medium) into logical channels for better efficiency
- **Circuit Switching** - circuit switching is usually a function of the physical layer (packet switching is an issue of the data link layer)

Signály

data jsou přenosovým médiem přenášeny ve formě (elektromagnetických) signálů. Data musí být na signály transformována.

Signál = časová funkce reprezentující změny fyzikálních (elektromagnetických) vlastností přenosového média

Data určená k přenosu – digitální (binární)

Signály šířené přenosovým médiem

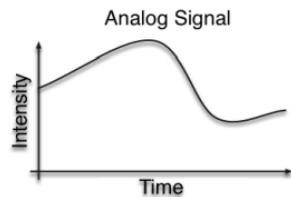
- *analogové*
- *digitální*

Média

- některá média vhodná pro analogový i digitální přenos – drátový vodič (koaxiál, kroucená dvoulinka), optické vlákno
- některá média vhodná pouze pro analogový přenos – éter

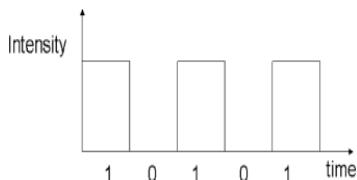
1. Analogový signál

- spojity v čase (mění se hladce)
- lze jej šířit jak vodiči, tak bezdrátovým prostředím



2. Digitální signál

- diskrétní v čase (mění se skokově)
- lze jej šířit pouze vodiči
- data diskrétní v hodnotách, např. znaky, prvky abecedy,..



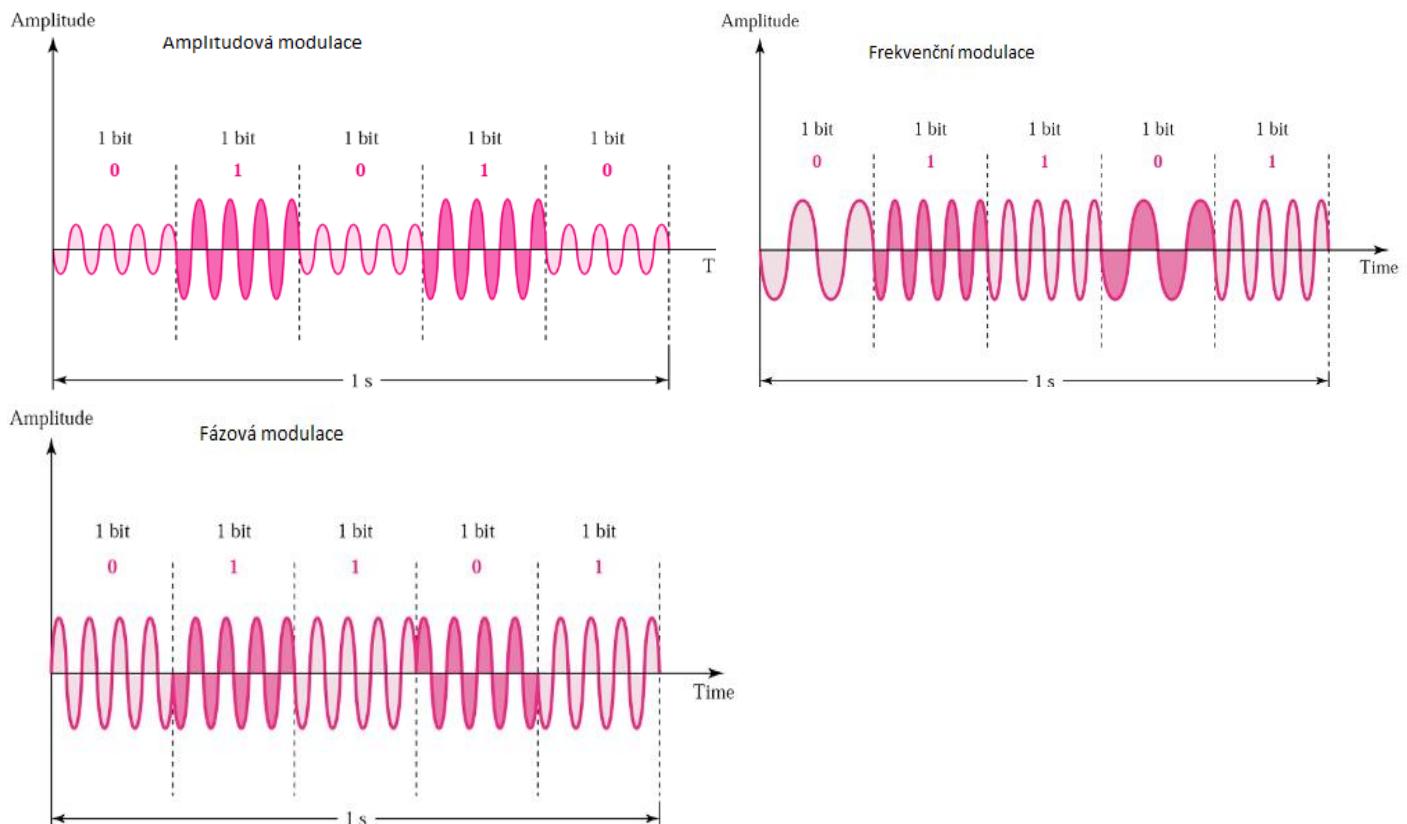
Přenos dat

digitální/binární data jsou přenášena *modulací nosného signálu digit.daty(analog) nebo transformací kódování(digitální sig.)*

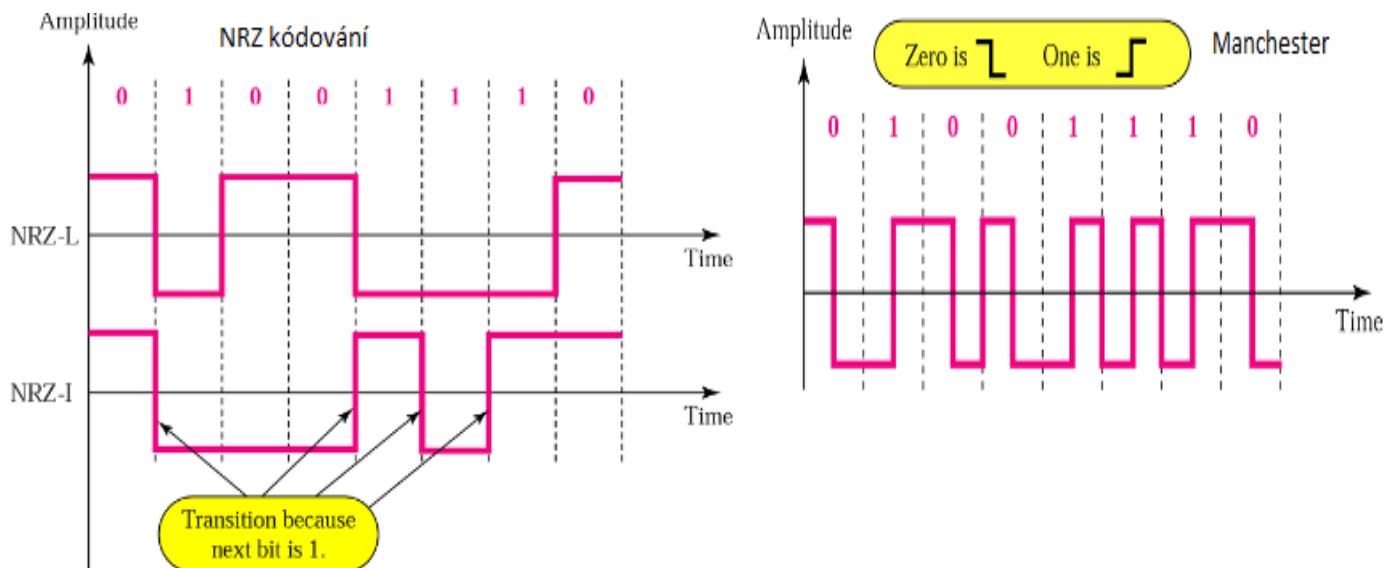
● Modulace analogového signálu

- *amplitudovou digitální modulací* - mění se amplituda nosného signálu
- *frekvenční digitální modulací* - mění se frekvence nosného signálu
- *fázovou digitální modulací* - mění se fáze nosného signálu

Pro modulaci/demodulaci slouží *modem*(MOdulátor/DEModulátor)



- Digitální přenos
 - Přímé kódování - 1 = kladná hodnota, 0 - záporná, žádná samosynchronizace
 - NRZ kódování - NRZ-L : 1 = záporná, 0 = kladná ampl.
 - žádná synchronizace
 - NRZ-1 : 1 = změna polarizace, 0 = žádná změna
 - řeší posloupnost 1
 - Manchester kódování - každý bit kódován 2 prvky signálu, snížení efektivní přenosové kapacity, plná samosynchronizovatelnost
 - 4B/5B kódování - uměle zavedená redundance pro zabezpečení synchronizace, možnost detekce chyb, substituce originálních 4-bit. bloků 5-bitovými. Nejvýše tři 0 mohou následovat po sobě, přenos pak pomocí NRZ-L



4B/5B kódování

4B	5B	4B	5B
0000	11110	1000	10010

Řízení přístupu k médiu (MAC)

- funkcionalita odpovědná za koordinaci přístupu více stanic ke sdílenému přenosovému médiu.
- Cíl - eliminace kolizí(konfliktů) při vysílání (souběžného vysílání do jediného přenosového prostředí).

- Protokoly neřízeného přístupu
 - Aloha - stanice vysílá kdykoliv má připravený rámec; kolize detekováný nepřijetím potvrzením o příjetí v časovém intervalu. Při kolizi náhodnou dobu čeká a vysílá znova, neefektivní.

- CSMA/CD - upravená Aloha, vysílá jen když zjistí klid v médiu. Současně na médiu naslouchá pro detekci případné kolice(CD= Collision Detection), aplikace v klasickém LAN Ethernetu, nepoužitelné v nevoděném médiu
- CSMA/CA - obcházení kolízí, použitelné v nevoděném médiu
- *Protokoly řízeného přístupu*
 - stanice smí vysílat jen tehdy, když k tomu získá právo od řídící stanice
 - rezervace - vysílání v předem domluvených intervalech
 - vyzývání - centrální stanice vybírá stanici, která bude vysílat
 - předávání příznaku - předává "peška" indikujícího právo k vysílání

Propojování počítačových sítí

-vzájemné propojování celých sítí i jednotlivých kabelových segmentů(hierarchie) => vznikne internetwork (internet)

internet - propojení dvou či více sítí, jméno konkrétní sítě

Důvody vzniku:

- překonání technických omezení/překážek - omezený dosah kabelů
- optimalizace fungování sítě - regulace toku, zbytečné šíření
- zpřístupnění vzdálených zdrojů
- zvětšení dosahu poskytovaných služeb - email, telefonie

1) Přepínání okruhů (Circuit Switching)

- fyzické přímé spojení mezi odesílatelem a příjemcem, paketizace, spojovaná služba

2) Přepínání paketů (Packet Switching)

- zasílání paketů
- virtuální kanály (Virtual Circuits Approach)
 - na začátku ustanovena cesta, všechny pakety jedné relace putují po stejně trase
 - použití ve: WANs, Frame Relay, ATM
 - spojovaná služba
- datagramový přístup (Datagram Approach)
 - každý paket obsuhován zcela nezávisle na ostatních
 - nespojovaná služba
 - pakety zde jsou datagramy
 - Internet

SÍŤOVÉ PROTOKOLY

Internet Protocol (IP)

- nejrozšířenější protokol síťové vrstvy , doprava dat na místo jejich určení přes mezilehlé směrovače(uzly) - *host-to-host- delivery*
 - uzly/rozhraní jednoznačně identifikovaný IP adresami
 - datagramový přístup, nespojovaná komunikace = směrování
- sám o sobě neposkytuje žádné záruky na přenos dat

- **verze IPv4** - přepravuje data bez záruky, negarantuje doručení, zachování pořadí ani vyloučení duplicit, toto je ponecháno na vyšší vrstve (protokol TCP). Při transportu dat se používá fragmentace datagramu (na zdrojovém uzlu a směrovačích), defragmentace pouze na cílovém uzlu.
- Internet Control Message Protocol (ICMP) - chyby přenosu dat

- **verze IPv6** - přináší hlavně masivní rozšíření adresního prostoru 128-bitová, možnost rozšíření skrz rozšiřující hlavičky (), podpora: přenosu reálného času, zabezpečení přenosu (IPSec - autentizace/Authentication header/ a šifronávní dat/Encapsulating Security Payload/), mobility, autokonfigurace

SMĚROVÁNÍ

- 1. Statické (neadaptivní)**
 - administrátorem ručně editované záznamy
 - -směrovač nemůže vytvářet alternativní cesty, pokud se nastavená cesta přeruší
 - jednodušší, málo flexibilní, vhodné pro statickou topologii
- 2. dynamické (adaptivní)**
 - složité (distribuované algoritmy)
 - nutnost pravidelně aktualizovat směrovací tabulky
 - nezaručuje pořadí doručení
 1. centralizované - vše řídí centrum
 2. izolované - každý sám za sebe
 3. distribuované - kooperace uzlů

Centralizované

- Routing Control Center(RCC) - každý směrovač mu posílá zprávy o své situaci(stavu), ten vypočítá optimální cesty a rozešle uzlům tabulky
- *výhody*: globální informace, ulehčení práce směrovačů
- *nevýhody*: špatná škálovatelnost, pomalé, po výpadku centra přestane aktualizovat.

Izolované

- každý uzel rozhoduje sám za sebe
- *náhodná procházka* - paket pošle po náhodně vybrané lince, vysoká robustnost
- *hot potatoe* - paket pošle do linky s nejkratší frontou
- *záplava (flooding)* - paket pošle do všech linek kromě té, po které přišel, enormní zátěž sítě, mimořádně robustní, nejlepší cesta
- *zpětné učení* - učí se z procházejících paketů, ty obsahují urazenou vzdálenost

Distribuované

- směrovací informaci si vyměňují sousedé nebo skupinky směrovačů, počítají se mapy sítě, dohoda mezi směrovači o implementaci určitého algoritmu
- pružné a robustní, vhodné pro rozlehlé sítě
- Internet

Směrovací algoritmy

1. Distance vector(DV) Bellman-Fordův algoritmus
 - sousední směrovače si v pravidelných intervalech či při změně sítě vyměňují kompletní kopie svých směrovacích tabulek
 - na základě updatů si doplňují nové informace inkrementují své distance vektor číslo
 - všechny informace jen svým sousedům
2. Link State(LS)
 - směrovače posílají informace pouze o stavu linek, na něž jsou bezprostředně připojeny
 - udržují si tak kompletní informace o topologii sítě
 - pak se počítá nejkratší cesta
 - informace o svých sousedech všem

MULTICAST - skupinová komunikace v síti

klasické řešení skupinové komunikace v síti

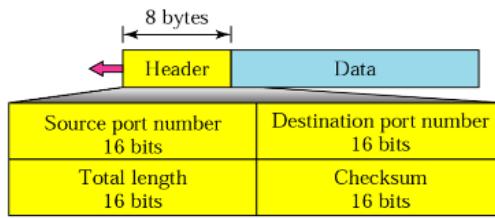
- každým spojem nejvýše jedna kopie dat
- hop-by-hop síť
- nezaručené doručení (best effort, UDP, skupinová adresace)
- rozsah šíření omezen Time To Live(TTL) pakety

- a Source Based Tree - aktivita shora od zakládajícího, periodický broadcast, ořezávání větší bez členů, TTL,
 - nevýhoda: velká režie, záplava, broadcasty
- b Core Based Tree - ustaveno jádro(body setkání), zájemce kontaktuje skupinu, aktivita zdola od příjemce, redukce broadcastu
 - nevýhoda: závislost na dostupnosti jádra

TRANSPORTNÍ PROTOKOLY

1. UDP (User Datagram Protocol)

- nejjednodušší transportní protokol poskytující **nespojovanou a nespolehlivou** (nezajištěnou) službu. Poskytuje best-effort službu
- *Přednosti:* jednoduchost, minimální režie
- žádná nutnost ustanovení spojení, uchovávání stavových informací, malá hlavička
- orientován na přenos bloků dat



- **zdrojový port (source port)** – identifikace odesílací služby/aplikace
- **cílový port (destination port)** – identifikace přijímající služby/aplikace
- **délka UDP paketu (length)** – celková délka UDP paketu
- **kontrolní součet (checksum)** – kontrolní součet UDP paketu (hlavička + data)

Procesy vyžadují jednoduchou komunikaci stylu “dotaz-odpověď” (např. DNS), protokoly s řízením toku a kontrolu chyb(Trivial File Transport Protocol), real-time přenosy, multicastové přenosy apod.

Přenos dat - aplikace předává bloky dat, které UDP opatřuje hlavičkou a předává síťovému protokolu (třeba IP)

2. TCP (Transmission Control Protocol)

- zajišťuje protokol poskytující **spojovanou** a plně **spolehlivou** (zajištěnou) službu
 - odeslaná data budou přijímací aplikaci doručena kompletní a ve správném pořadí. Oproti UDP orientovaný na přenos proudu bytů
- před začátkem nutné ustavení spojení mezi odesílací a přijímací stranou
 - handshake před začátkem, výměna všech potřebných parametrů
 - spojení rozeznatelné na koncových uzlech (end-to-end služba)
 - plně duplexní komunikace., spojení dvoubodové(point-to-point)

Přenos dat - aplikace předává TCP protokolu proud bytů, které TCP segmentuje, opatřuje hlavičkou a předává síťovému protokolu. Aplikacím poskytuje iluzi roury, která přenáší jejich data.

Velikost segmentů je omeyena hodnotou Maximum Segment Size(MSS). Segmenty následně opatřeny TCP hlavičkou a předány síťovému protokolu. Číslovánz pak jsou jednotlivé přenášené bajty.

Sestavení spojení

- full-duplexní přenos => obě strany musí iniciovat spojení
- mechanismus známý jako **třícestný handshake (three-way handshake)**

Ukončení spojení

- iniciováno jednou z komunikujících stran
- spojení musí být uzavřeno oběma stranami

Síťové aplikace a bezpečnost

Základní aplikační protokoly: doručování pošty, přenos souborů, web, jmenná služba. Principy popisu a zajištění kvality služby, použití pro multimedia. Zabezpečení síťové komunikace, autentizace a šifrování, zabezpečení na jednotlivých protokolových vrstvách.

Aplikační protokoly

Vrstva aplikací a procesů (Process/Application Layer) je nejvyšší vrstvou síťové architektury Internetu. Některé aplikační protokoly vyžadují protokol TCP (např. FTP, Telnet apod.), jiné protokol UDP (např. SNMP, BOOTP, TFTP). Některé z protokolů však mohou používat kterýkoli z transportních protokolů. Výběr protokolu se konkretizuje v rámci implementace (např. DNS). Aplikační protokoly podporují jednak čistě uživatelské aplikace, jako přenos souborů a poštovních zpráv nebo práci na vzdáleném zařízení, jednak administrativní aplikace (pro uživatele „neviditelné“), jako mapování jmen a adres, management sítě apod.

Výčet aplikačních protokolů: DHCP, DHCPv6, DNS, FTP, HTTP, IMAP, IRC, LDAP, MGCP, NNTP, BGP, NTP, POP3, RPC, RTP, RTSP, RIP, SIP, SMB, SMTP, SNMP, SOCKS, SSH, TELNET, TLS/SSL, XMPP a další.

Doručování pošty

Posílání emailových zpráv mezi počítači je uskutečňováno pomocí protokolu SMTP a softwaru typu MTA (Mail Transport Agent), což je server, který se stará o příjem a doručování emailů do schránek příjemců. Emailové zprávy je možno ukládat přímo na poštovním serveru nebo na straně klienta. K přístupu ke zprávám se používají protokoly POP a IMAP.

Simple Mail Transfer Protocol

SMTP (Simple Mail Transfer Protocol) – je internetový protokol určený pro přenos zpráv mezi stanicemi. SMTP využívá port 25. Samotný protokol zajišťuje přímé doručení zpráv do poštovní schránky příjemce, ke které může uživatel přistupovat offline. V současné době používá většina uživatelů elektronické pošty pro stahování svých e-mailů programy využívající protokol POP3 nebo IMAP4. Pro odesílání zpráv se používá pouze protokol SMTP nezávisle na použitém protokolu pro příjem zpráv.

Post Office Protocol

POP využívá verzi 3 (označení POP3). Tento internetový protokol se používá pro vyzvedávání e-mailů z poštovních schránek. Funguje na principu klient-server, tedy komunikaci začíná klient a server pouze odpovídá, jakmile klient provede inicializaci vzájemné komunikace, což je jméno a heslo, je poté možné s poštou pracovat. Klient si stáhne ze serveru zprávy, a poté si je může přečíst už bez nutnosti internetového připojení. Avšak ze vzdáleného serveru se stahnou všechny zprávy, a to také ty, které uživatel čist nechce jako je např. spam (pokud ho již nefiltruje poštovní server). Většina POP3 serverů sice umožňuje stahnout pouze hlavičky zpráv, a poté následně vybrat zprávy, které se stahnou celé, ale podpora v klientech často chybí. Tuto nevýhodu může odstranit protokol IMAP, který pracuje se zprávami přímo na určitém serveru.

Jako mnoho jiných starších internetových protokolů, POP3 původně podporoval jen nešifrované přihlašovací mechanismy. Ačkoli v POP3 je běžný jednoduchý a nezabezpečený přenos hesel, podporuje zároveň několik autentizačních metod ověřování na různých úrovních ochrany před neoprávněným přístupem k cizí poštovní schránce. Jedna taková metoda se nazývá APOP. APOP základní specifikace definuje jako „volitelný příkaz“. Užívá MD5 hash funkci pro zabezpečený přenos hesla z klienta na server. Klienti podporující APOP jsou například Mozilla Thunderbird, Eudora, Novell Evolution, IBM Lotus, MS Outlook alebo Apple Mail. Klienti mohou také šifrovat celou komunikaci užitím SSL.

Internet Message Access Protocol

IMAP využívá verzi 4 (označení IMAP4). Jedná se o poštovní protokol, který slouží pro vzdálený přístup k e-mailové schránce. Narozdíl od protokolu POP3 vyžaduje IMAP trvalé připojení k internetu, avšak nabízí pokročilé možnosti pro vzdálené správy jako jsou např. práce se složkami, přesouvání zpráv apod. Všechny zprávy a složky jsou uloženy na poštovním serveru a na počítač se stahují jen nezbytné informace, takže při zobrazení složky se stáhnou jen záhlaví určitých zpráv a jejich obsah až v případě, že si chce uživatel zprávu přečíst. U jednotlivých zpráv se uchovává jejich stav (nepřečtená, odpovězená, důležitá). Protokol navíc umožnuje připojení více klientů zároveň.

Oproti protokolu POP3 je IMAP4 velmi komplikovaným protokolem. Jeho implementace jsou značně složitější a náchylnější k chybám než implementace pro POP3. Navzdory tomu IMAP používá mnoho e-mailových serverů a klientů jako jejich standardní přístupovou metodu. Pokud nejsou ukládací a vyhledávací algoritmy na serveru bezpečně implementovány, prohledávání velké schránky může značně zatěžovat server. IMAP4 klienti mohou způsobit zpoždění při vytváření nových zpráv, u pomalých připojení. U mobilních zařízení je lepší použít Push IMAP, což je rozšířený IMAP protokol o implementaci Push e-mail.

FTP / File Transfer Protocol

Tento protokol slouží pro přenos souborů mezi počítači. Využívá protokol TCP z rodiny TCP/IP a je platformě nezávislý. Definován byl v roce 1985 jako RFC 959.

FTP patří mezi vůbec nejstarší protokoly a využívá porty TCP/20 a TCP/21. Port 21 slouží k řízení a jsou jím přenášeny příkazy FTP. Port 20 pak slouží k samotnému přenosu souborů (8 bitový). Přenos může být binární nebo ascii (textový). Při textovém přenosu dochází ke konverzi konců řádků – CR/LF (DOS, Microsoft Windows) nebo jen LF (unixové systémy), pokud jsou koncové systémy rozdílné. Při binárním přenosu není do dat nijak zasahováno.

Protokol je interaktivní a umožňuje řízení přístupu (přihlašování login/heslo), specifikaci formátu přenášeného souboru (znakově – binárně), výpis vzdáleného adresáře atd. V současné době už není považován za bezpečný a z tohoto důvodu pro něj byla definována některá rozšíření (RFC 2228). V protokolu je použit model klient-server. FTP server poskytuje data pro ostatní počítače. Klient se k serveru připojí a může provádět různé operace (výpis adresáře, změna adresáře, přenos dat atd.). Operace jsou řízeny sadou příkazů, které jsou definovány v rámci FTP protokolu, proto je možné vytvořit klienta pro jakékoli prostředí nebo operační systém. Existuje mnoho programů pro FTP servery i klienty a mnoho je jich taky volně dostupných.

Výhody a nevýhody

- hesla a soubory jsou ve standardním protokolu zasílána jako běžný text (nejsou šifrovány)
 - snižuje bezpečnost (ohrožuje jméno, heslo, ale i přenášená data)
 - existují rozšíření FTP protokolu, která tento nedostatek odstraňují

2. používají se 2 TCP spojení (dva kanály - první TCP spojení je řídící, druhé datové pro vlastní přenos dat)

- je-li použit firewall, protokol vyžaduje jeho speciální podporu (aktivní FTP přenos)
- podpora aktivního přenosu nefunguje u šifrovaného řídícího spojení
- pasivní přenos tento nedostatek odstraňuje

3. FTP server má delší odezvy

- nemožnost sloučit přenos více (malých) souborů do jednoho zvyšuje časovou režii i zátěž serveru
- serverová část je jednodušší, než běžný HTTP server (neplatí pro odlehčené HTTP servery)
- na rozdíl od HTTP má protokol širší možnosti (nastavení práv, mazání, upload, rekurzivita)

4. v některých sítích je povolen pouze protokol HTTP

- FTP je v současné době méně používáno

Web

World wide web, zkráceně web, je označení pro aplikace aplikačního protokolu HTTP (popsaným níže) (Hypertext Transfer Protocol). Princip webu spočívá v soustavě propojených hypertextových dokumentů. Dokumenty umístěné na počítačových serverech jsou adresovány pomocí URL (Uniform Resource Locator), jehož součástí je i doména a jméno počítače. Název naprosté většiny těchto serverů začíná zkratkou www, i když je možné používat libovolné jméno vyhovující pravidlům URL.

Protokol HTTP je dnes již používán i pro přenos jiných dokumentů, než jen souborů ve tvaru HTML a výraz World Wide Web se postupně stává pro laickou veřejnost synonymem pro internetové aplikace.

Autorem Webu je Tim Berners-Lee, který jej vytvořil při svém působení v CERNu. Navrhl jazyk HTML a protokol HTTP, napsal první webový prohlížeč WorldWideWeb a koncem roku 1990 spustil první webový server na světě info.cern.ch. V říjnu roku 1994 založil World Wide Web Consortium (W3C), které dohlíží na další vývoj Webu.

HTTP a HTTPS specifikátory

Toto schéma specifikátorů (<http://> nebo <https://>) v URIs (Uniform Resource Identifier) odkazují na Hypertext Transfer Protocol a také na HTTPS, a tak definují komunikační protokol, který se použije pro žádost a odpověď. HTTP protokol je základem pro fungování World Wide Webu. Přidáním šifrovací vrstvy vznikl HTTPS protokol, který je vhodný pro přenos důvěrných informací, jako jsou hesla či bankovní údaje, které mají být přeneseny přes veřejný internet.

Ukládání do vyrovnávací paměti (caching)

Pokud se uživatel vrátí na webovou stránku, kterou v poslední době navštívil, tak nebude potřeba data stránky opět načítat z internetu. Skoro všechny internetové prohlížeče mají cache (vyrovnávací paměť) nedávno načtených stránek, obvykle se tyto data nacházejí na pevném disku počítače. HTTP obvykle zasílá pouze žádost o data, která se od poslední návštěvy změnila. Pokud jsou dočasně uložena data na pevném disku stále aktuální, tak se znova použijí. Načítání dat do vyrovnávací paměti redukuje internetový přenos. O platnosti jednotlivých dat je rozhodováno odděleně, zvlášť pro obrázky, CSS styl, JavaScript, HTML a další obsah webové stránky. Proto i na internetových stránkách s

velmi dynamickým obsahem, není nutné pokaždé načítat veškerá data znova. Weboví designeři považují za užitečné oddělit tedy styl stránky, obsah(HTML document) a JavaScript(alebo iné interaktívne zložky) do samostatných souborů, tak aby nebylo nutné vždy načítat vše, ale pouze ten soubor, který není aktuální. Obsah webové stránky se mění mnohem častěji než její styl. Toto pomáhá zkrátit dobu načítání stránky z internetu a snižuje nároky na webový server. Okrem toho, že to znižuje provoz na síti je takéto rozdelenie dôležité aj kvôli rôznorodosti prehliadačov (rôzna podpora pre rôzne verzie). Nie je neobvyklé mať aj viacero stupňov štýlov / interaktívnych zložiek, ktoré sa vyberajú podľa druhu a verzie klientovho prehliadača.

Existují i další součásti, které lze načítat do vyrovnavací paměti. Firemní a akademické firewally často načítají internetové zdroje, o které žádá jeden uživatel, ve prospěch ostatních uživatelů. (Viz též caching proxy server.) Některé internetové vyhledávače také uchovávají obsah webových stránek ve vyrovnavací paměti. Na rozdíl od zařízení, která jsou zabudovaná ve webových serverech a mohou určit, kdy byla data aktualizována a kdy je potřeba je znova odeslat. Designeři dynamických webových stránek mohou kontrolovat HTTP hlavičky, které se odesílají uživateli jako odpověď na žádost odeslané uživatelem, takže přechodná a citlivá data nejsou ukládána. Internetové bankovnictví a zpravodajské weby tohoto často využívají. Data požadovaná pomocí HTTP 'GET' jsou ukládána, pokud jsou splněny další podmínky, data získaná v reakci na 'POST' jsou závislá na datech odeslaných, takže se neukládají.

Pojmy World Wide Web a Internet nejsou vzájemně zaměnitelné.

HTTP

HTTP (Hypertext Transfer Protocol) je internetový protokol určený pro výměnu hypertextových dokumentů ve formátu HTML. Používá obvykle port TCP/80, verze 1.1 protokolu je definována v RFC 2616. Tento protokol je spolu s elektronickou poštou tím nejvíce používaným a zasloužil se o obrovský rozmach internetu v posledních letech.

V současné době je používán i pro přenos dalších informací. Pomocí rozšíření MIME umí přenášet jakýkoli soubor (podobně jako e-mail), používá se společně s formátem XML pro tzv. webové služby (spouštění vzdálených aplikací) a pomocí aplikačních bran zpřístupňuje i další protokoly, jako je např. FTP nebo SMTP.

HTTP používá jako některé další aplikace tzv. jednotný lokátor prostředků (URL, Uniform Resource Locator), který specifikuje jednoznačné umístění nějakého zdroje v Internetu.

Samotný protokol HTTP neumožňuje šifrování ani zabezpečení integrity dat. Pro zabezpečení HTTP se často používá TLS spojení nad TCP. Toto použití je označováno jako HTTPS.

Protokol funguje způsobem dotaz-odpověď. Uživatel (pomocí programu, obvykle internetového prohlížeče) pošle serveru dotaz ve formě čistého textu, obsahujícího označení požadovaného dokumentu, informace o schopnostech prohlížeče apod. Server poté odpoví pomocí několika řádků textu popisujících výsledek dotazu (zda se dokument podařilo najít, jakého typu dokument je atd.), za kterými následují data samotného požadovaného dokumentu. Pokud uživatel bude mít po chvíli další dotaz na stejný server (např. proto, že uživatel v dokumentu kliknul na hypertextový odkaz), bude se jednat o další, nezávislý dotaz a odpověď. Z hlediska serveru nelze poznat, jestli tento druhý dotaz jakkoli souvisí s předchozím. Kvůli této vlastnosti se protokolu HTTP říká bezestavový protokol – protokol neumí uchovávat stav komunikace, dotazy spolu nemají souvislost. Tato vlastnost je nepřijemná pro implementaci složitějších procesů přes HTTP (např. internetový obchod potřebuje uchovávat informaci o identitě zákazníka, o obsahu jeho „nákupního košíku“ apod.). K tomuto účelu

byl protokol HTTP rozšířen o tzv. HTTP cookies, které umožňují serveru uchovávat si informace o stavu spojení na počítači uživatele.

DNS / Jmenná služba

DNS (Domain Name System) je hierarchický systém doménových jmen, který je realizován servery DNS a protokolem stejného jména, kterým si vyměňují informace. Jeho hlavním úkolem jsou vzájemné převody doménových jmen a IP adres uzlů sítě. Později ale přibral další funkce (např. pro elektronickou poštu či IP telefonii) a slouží dnes de facto jako distribuovaná databáze síťových informací.

Protokol používá porty TCP/53 i UDP/53, je definován v RFC1035. Servery DNS jsou organizovány hierarchicky, stejně jako jsou hierarchicky tvořeny názvy domén. Jména domén umožňují lepší orientaci lidem, adresy pro stroje jsou však vyjádřeny pomocí adres 32bitových (IPv4 - A záznam) nebo 128bitových (IPv6 - AAAA záznam). Systém DNS umožňuje efektivně udržovat decentralizované databáze doménových jmen a jejich překlad na IP adresy. Stejně tak zajišťuje zpětný překlad IP adresy na doménové jméno - PTR záznam.

Jak DNS funguje

Prostor doménových jmen tvoří strom. Každý uzel tohoto stromu obsahuje informace o části jména (doméně), které je mu přiděleno a odkazy na své podřízené domény. Kořenem stromu je tzv. kořenová doména, která se zapisuje jako samotná tečka. Pod ní se v hierarchii nacházejí tzv. domény nejvyšší úrovně (Top-Level Domain, TLD). Ty jsou buď tematické (com pro komerci, edu pro vzdělávací instituce atd.) nebo státní (cz pro Česko, sk pro Slovensko, jo pro Jordánsko atd.). Strom lze administrativně rozdělit do zón, které spravují jednotliví správci (organizace nebo i soukromé osoby), přičemž taková zóna obsahuje autoritativní informace o spravovaných doménách. Tyto informace jsou poskytovány autoritativním DNS serverem. Výhoda tohoto uspořádání spočívá v možnosti zónu rozdělit a správu její části svěřit někomu dalšímu. Nově vzniklá zóna se tak stane autoritativní pro přidělený jmenný prostor. Právě možnost delegování pravomocí a distribuovaná správa tvoří klíčové vlastnosti DNS a jsou velmi podstatné pro jeho úspěch. Ve vyšších patrech doménové hierarchie platí, že zóna typicky obsahuje jednu doménu. Koncové zóny přidělené organizacím připojeným k Internetu pak někdy obsahují několik domén – například doména kdesi.cz a její poddomény výroba.kdesi.cz, marketing.kdesi.cz a obchod.kdesi.cz mohou být obsaženy v jedné zóně a obhospodařovány stejným serverem.

Složení doménového jména

Celé jméno se skládá z několika částí oddělených tečkami. Na jeho konci se nacházejí domény nejobecnější, směrem doleva se postupně konkretizuje. Část nejvíce vpravo je doména nejvyšší úrovně, např. wikipedia.org má TLD org. Jednotlivé části (subdomény) mohou mít až 63 znaků a skládat se mohou až do celkové délky doménového jména 255 znaků. Doména může mít až 127 úrovní. Bohužel některé implementace jsou omezeny více.

DNS servery (name servery)

DNS server může hrát vůči doméně (přesněji zóně, ale ve většině případů jsou tyto pojmy zaměnitelné) jednu ze tří rolí:

Primární server je ten, na němž data vznikají. Pokud je třeba provést v doméně změnu, musí se editovat data na jejím primárním serveru. Každá doména má právě jeden primární server.

Sekundární server je automatickou kopí primárního. Průběžně si aktualizuje data a slouží jednak jako záloha pro případ výpadku primárního serveru, jednak pro rozkládání zátěže u frekventovaných domén. Každá doména musí mít alespoň jeden sekundární server.

Pomocný (caching only) server slouží jako vyrovnávací paměť pro snížení zátěže celého systému. Uchovává si odpovědi a poskytuje je při opakování dotazů, dokud nevyprší jejich životnost. Odpověď pocházející přímo od primárního či sekundárního serveru je autoritativní, čili je brána za správnou. Z hlediska věrohodnosti odpovědí není mezi primárním a sekundárním serverem rozdíl, oba jsou autoritativní. Naproti tomu odpověď poskytnutá z vyrovnávací paměti není autoritativní.

Root servery

Kořenové jmenné servery (root name servers) představují zásadní část technické infrastruktury Internetu, na které závisí spolehlivost, správnost a bezpečnost operací na internetu. Tyto servery poskytují kořenový zónový soubor (root zone file) ostatním DNS serverům. Jsou součástí DNS, celosvětově distribuované databáze, která slouží k překladu unikátních doménových jmen na ostatní identifikátory. Kořenový zónový soubor popisuje, kde se nacházejí autoritativní servery pro domény nejvyšší úrovně. Tento kořenový zónový soubor je relativně malý a často se nemění – operátoři root serverů ho pouze zpřístupňují, samotný soubor je vytvářen a měněn organizací IANA.

Pojem root server je všeobecně používán pro 13 kořenových jmenných serverů. Root servery se nacházejí ve 34 zemích světa, na více než 80 místech. Root servery jsou spravovány organizacemi, které vybírá IANA. Následující tabulka zobrazuje těchto 13 root serverů:

Název root serveru	Operátor
A	VeriSign Global Registry Services
B	University of Southern California - Information Sciences Institute
C	Cogent Communications
D	University of Maryland
E	NASA Ames Research Center
F	Internet Systems Consortium, Inc.
G	U.S. DOD Network Information Center
H	U.S. Army Research Lab
I	Autonomica/NORDUnet
J	VeriSign Global Registry Services
K	RIPE NCC
L	ICANN
M	WIDE Project

Popis a zajištění kvality služby

Quality of Service (QoS) má základy použití v telefonních a počítačových sítích, kde jsou nutné určité požadavky na kvalitu přenosu dat, např. audio / video konference přes internet. Proto protokoly QoS mají za úkol zajištění vyhrazení a dělení dostupné přenosové kapacity, aby nedocházelo díky zahlcení sítě ke snížení kvality služeb.

Charakteristika

Pomocí QoS se může např. nastavit maximální nebo minimální přenosové pásma pro určitá data, prohlásit některý provoz za prioritní nebo rozdělit provoz do kategorií podle nastavených parametrů. QoS se tedy snaží poskytovat uživatelům služby s předem garantovanou kvalitou, aby nedocházelo ke zpoždění, ztrátovosti nebo plýtvání s dostupnou šírkou pásma.

V lokálních sítích je provoz víceméně bezproblémový, protože disponují dostatečnou kapacitou, která je typicky vyšší, než schopnost počítačů generovat nebo zpracovávat přicházející síťový provoz (viz gigabitový Ethernet vs. rychlosť zpracovávania dat u pevného disku). Nasazení QoS pro zajištění kvality služeb proto nastupuje typicky až hranici lokální sítě LAN, kde je spojení pokračuje do Internetu typicky linkou s nižší datovou propustností.

Důvod nasazení QoS

V běžných počítačových sítích se používá konkurenční přístup k přenosovému médiu (např. síť Ethernet, ale i bezdrátové síť Wi-Fi) a v nich tzv. best-effort services, kdy se přenášejí data tak, jak přicházejí, což typicky funguje dobře, protože požadavky na přenos jsou nižší, než dostupná kapacita. Dojde-li však k zaplnění přenosové kapacity, začnou se plnit vyrovnávací paměti (buffery), čímž se aktivní prvky v počítačové síti snaží překlenout chvílkové zahlcení přenosového média. Tím dochází ke zvětšování latence (zpoždění), protože jednotlivé datagramy ve frontách čekají na vyřízení. K vyčerpání přenosové kapacity může například snadno dojít, když uživatel začne stahovat z Internetu větší objemy dat, které na delší dobu plně využijí kapacitu jeho linky k providerovi.

Některé síťové služby, které pracují s daty v reálném čase (např. VoIP nebo přenos multimedialních dat jako je IPTV, ale i počítačové hry) fungují při vyšší latenci špatně. Dochází ke zpoždění, výpadkům, kolísání přenosové rychlosti (jitter) a celkově ke zhoršení kvality poskytované služby. V tuto chvíli nastává okamžik, kdy je možné situaci řešit nasazením QoS. Pomocí QoS může správce sítě pro některý síťový provoz nastavit vyšší prioritu, takže je přenášen i v případě zahlcení přenosového média prioritně, beze ztrát a bez zvýšení latence.

QoS typicky používá i provider, který používá tzv. agregaci, což znamená, že svým zákazníkům „naslibuje“ vyšší přenosové kapacity, než má reálně k dispozici. Za běžných okolností to nemusí vadit, protože ne všichni jeho zákazníci využívají svoje připojení k Internetu naplno a ve stejný čas. Proto dochází k zahlcení konektivity providera typicky hlavně ve špičkách. Provider může pomocí diferencovaných služeb zajistit, že zákazníci sice zaregistrují jisté zpomalení, ale nedojde ke stavu kdy by bylo téměř nemožné jeho služby využívat.

Činnost QoS

Pokud je použito QoS, je nutné provoz některých počítačů (uživatelů, služeb) omezit ve prospěch jiných. Omezování se typicky provádí zahazováním paketů (anglicky packet droping), protože na něj mají povinnost síťové aplikace reagovat na zahlcení sítě okamžitým snížením rychlosti přenosu (anglicky congestion control). Například pro protokol TCP obsahuje tuto regulaci povinně přímo v sobě a v běžných počítačích je proto realizována jako součást TCP/IP stacku (součást jádra operačního systému). QoS je proto typicky realizována v uzlech sítě (routery) metodou zahazování datagramů, které přesahují předem nastavené parametry datových toků.

Metody QoS

V sítích se v dnešní době používají především tři typy mechanismů QoS:

Best-effort services: metoda největší snahy, která má QoS nastaven na nulu a snaží se každý paket co nejrychleji a nejfektněji přenést k cíli

Differentiated services (DiffServ): pakety se rozdělují do kategorií, což se může zaznamenat do hlavičky paketu, a zachází se s nimi podle předdefinovaných parametrů

Integrated services (IntServ)

IEEE 802.1p

IEEE 802.1Q

IEEE 802.11e

a další..

Kvalita Provozu

V paketových sítích, je kvalita služeb ovlivněna různými faktory, které mohou být rozděleny na "lidské" a "technické" faktory. Do lidských faktorů patří: stabilita služeb, dostupnost služeb, zpoždění, uživatelské informace. Do technických faktorů patří: spolehlivost, škálovatelnost, účinnost, udržovatelnost, stupeň služby, aj.

Aplikace

- Definované kvality služby mohou být žádoucí nebo nutné pro některé druhy síťového provozu, například:
- Streaming media (Internet Protocol Television (IPTV, Audio přes Ethernet, Audio přes IP)
- IP telefonie také známý jako Voice over IP (VoIP)
- Videokonferenční hovory
- Circuit Emulation Service
- Bezpečnostní aplikace, jako jsou vzdálené operace, kde problémy s dostupností mohou mít kritické dopady na bezpečnost
- Online hry
- Průmyslové řídící systémy jako je například protokol Ethernet / IP, které se používají pro řízení strojů v reálném čase

- Těmto typům služeb se říká "neelastické", což znamená, že vyžadují určitou minimální úroveň šířky pásma a určitou maximální čekací dobu na funkci. Naopak elastické služby mohou aplikace využít i když je k dispozici malá, nebo velká šířka pásma. Hromadné aplikace pro přenos souborů, které spoléhají na TCP jsou obecně elastické.

Zabezpečení síťového provozu

S připojením do sítě ztrácí počítač svoje soukromí, které mu zajišťovalo i bezpečnost.

Prvky zabezpečení síťového provozu:

Firewall

Firewall je síťové zařízení, které slouží k řízení a zabezpečování síťového provozu mezi sítěmi s různou úrovní důvěryhodnosti a zabezpečení. Zjednodušeně se dá říct, že slouží jako kontrolní bod, který definuje pravidla pro komunikaci mezi sítěmi, které od sebe odděluje. Tato pravidla historicky vždy zahrnovala identifikaci zdroje a cíle dat (zdrojovou a cílovou IP adresu) a zdrojový a cílový port, což je však pro dnešní firewalls už poměrně nedostatečné – modernější firewalls se opírají přinejmenším o informace o stavu spojení, znalost kontrolovaných protokolů a případně prvky IDS (Intrusion Detection System (IDS, tj. systém pro odhalení průniku). Firewalls se během svého vývoje řadily zhruba do následujících kategorií: Paketové filtry, Aplikační brány, Stavové paketové filtry. Stavové paketové filtry s kontrolou známých protokolů a popř. kombinované s IDS.

Protokol SSL/TLS

Secure Sockets Layer, SSL (doslova vrstva bezpečných socketů) je protokol, resp. vrstva pracující mezi vrstvu transportní (např. TCP/IP) a aplikační (např. HTTP), která poskytuje zabezpečení komunikace šifrováním a autentizací komunikujících stran. Následovníkem SSL je protokol Transport Layer Security (TLS).

Využití

Protokol SSL se nejčastěji využívá pro bezpečnou komunikaci s internetovými servery pomocí HTTPS, což je zabezpečená verze protokolu HTTP. Po vytvoření SSL spojení (session) je komunikace mezi serverem a klientem šifrovaná, a tedy zabezpečená.

Obvyklá využití SSL certifikátů: on-line obchody, které přijímají objednávky a údaje platebních karet; www portály a projekty s administrací pro zabezpečení hesel a dat; komunikace s obchodním partnerem (výměna důvěrných informací); zabezpečení přístupu k poště mimo firemní síť (Exchange, ...); zpracování citlivých osobních údajů;

dodržení regulačních ustanovení (legislativa), která vyžadují zabezpečené přenosy.

Princip fungování

Ustavení SSL spojení funguje na principu asymetrické šifry, kdy každá z komunikujících stran má dvojici šifrovacích klíčů – veřejný a soukromý. Veřejný klíč je možné zveřejnit a pokud tímto klíčem kdokoliv zašifruje nějakou zprávu, je zajištěno, že ji bude moci rozšifrovat jen majitel použitého veřejného klíče svým soukromým klíčem.

Ustavení SSL spojení (SSL handshake, tedy „potřásání rukou“) pak probíhá následovně:

- Klient pošle serveru požadavek na SSL spojení, spolu s různými doplňujícími informacemi (verze SSL, nastavení šifrování atd.).
- Server pošle klientovi odpověď na jeho požadavek, která obsahuje stejný typ informací a hlavně certifikát serveru.
- Podle přijatého certifikátu si klient ověří autentičnost serveru. Certifikát také obsahuje veřejný klíč serveru.
- Na základě dosud obdržených informací vygeneruje klient základ šifrovacího klíče, kterým se bude šifrovat následná komunikace. Ten zašifruje veřejným klíčem serveru a poše mu ho.
- Server použije svůj soukromý klíč k rozšifrování základu šifrovacího klíče. Z tohoto základu vygenerují jak server, tak klient hlavní šifrovací klíč.
- Klient a server si navzájem potvrdí, že od teď bude jejich komunikace šifrovaná tímto klíčem. Fáze handshake tímto končí.
- Je ustaveno zabezpečené spojení šifrované vygenerovaným šifrovacím klíčem.
- Aplikace od teď dál komunikují přes šifrované spojení. Například POST požadavek na server se do této doby neodešle.
- Během první fáze ustanovení bezpečného spojení si klient a server dohodnou kryptografické algoritmy, které budou použity. V dnešní implementaci jsou následující volby:
 - pro výměnu klíčů: RSA, Diffie-Hellman, DSA nebo Fortezza;
 - pro symetrickou šifru: RC2, RC4, IDEA, DES, 3DES nebo AES;
 - pro jednocestné hašovací funkce: MD5 nebo SHA.

Chyby

- Důvěra k mnoha CA

V PC je předinstalováno několik CA (Certificate authority - Certifikační autorita). Těmto se při prohlížení stránek <https://> automaticky důvěruje a uživatel si to ani nemusí uvědomit. Chyba je v tom, že v úložišti pro CA může být i nějaká testovací CA. Ta samozřejmě důvěryhodná není!

- Podepsaný certifikát je dobrý certifikát

Další častá chyba je, že uživatelé předpokládají, že podepsaný certifikát je správný certifikát. To je samozřejmě špatně, neboť certifikát mohl být podepsán útočníkovou CA. Je důležité kontrolovat, kdo certifikát vydal.

- Návrat k TCP

Uživatel má vepsat URL adresu, která má na začátku <https://>, ale připojení se nezdaří (např. prohlížeč napíše, že vypršela doba na připojení). Uživatel si řekne, že někde nastala chyba (třeba že se překleplo) a vepíše adresu znova bez „s“. Tedy zůstane mu jen <http://> a SSL se nepoužije.

- Povolení nebezpečných šifer

Existuje několik šifrovacích algoritmů. Některé jsou bezpečné a jiné nikoli. Chyba je v tom, že někde může být povoleno použít již překonané šifrovací algoritmy. Uživatel by si měl tedy zkonto rovat, že používá ty (v současné době) bezpečné.

- Certifikační autority

CA jsou komerční společnosti, které certifikují klientské žádosti a potvrzují identitu žadatele. Získané informace pak připojují k vydanému certifikátu. V dnešní době se používají různé úrovně ověření majitele domény. Od jednoduchého potvrzení odkazu v zaslанém e-mailu až po detailnější autorizaci včetně telefonického ověření.

Nejznámější certifikační autority: THAWTE, VeriSign, GeoTrust

- Doplňující informace

Standardní port pro komunikaci přes HTTPS/SSL je 443, standardní port HTTP je 80.

HTTPS/SSL dokáže zajistit důvěrnost dat jen na cestě od klienta k serveru (a naopak). Je na provozovateli serveru, jak potom s důvěrnými daty po rozšifrování naloží. Výjimkou není uložení v nešifrované podobě do nechráněné databáze.

IPSec

IPsec (IP security) je bezpečnostní rozšíření IP protokolu založené na autentizaci a šifrování každého IP datagramu. V architektuře OSI se jedná o zabezpečení již na síťové vrstvě, poskytuje proto transparentně bezpečnost jakémukoliv přenosu (kterékoliv síťové aplikaci). Bezpečnostní mechanismy vyšších vrstev (nad protokoly TCP/UDP, kde pracují TLS/SSL, SSH apod.) vyžadují podporu aplikací. IPsec je definován v několika desítkách RFC vydaných IETF, základními jsou 2401 a 2411.

Princip činnosti

Vytváří logické kanály – Security Associations (SA), které jsou vždy jednosměrné, pro duplex se používají dvě SA.

Bezpečnostní rozšíření vypadá následovně:

- Ověřování – při přijetí paketu může dojít k ověření, zda vyslaný paket odpovídá odesílateli či zda vůbec existuje.
- Šifrování – obě strany se předem dohodnou na formě šifrování paketu. Poté dojde k zašifrování celého paketu krom IP hlavičky, případně celého paketu a následně bude přidána nová IP hlavička.
- Základní protokoly (jsou často používány zároveň, protože se vzájemně doplňují):
- Authentication Header (AH) – zajišťuje autentizace odesílatele a příjemce, integritu dat hlavičce, ale vlastní data nejsou šifrována.
- Encapsulating Security Payload (ESP) – přidává šifrování paketů, přičemž vnější hlavička není nijak chráněna a není zaručena její integrita.

S/MIME

S/MIME (Secure/Multipurpose Internet Mail Extensions) je v současnosti používaný k zabezpečení elektronické pošty jako standard pro veřejný klíč šifrování a podepisování. Je to speciální verze protokolu MIME - S/MIME (Secure MIME).

S/MIME poskytuje služby pro kryptografické zabezpečení elektronických zpráv aplikace: autentizaci, integritu zpráv, nepopiratelnost původu (pomocí digitálních podpisů), soukromí a zabezpečení dat (pomocí šifrování). S / MIME specifikuje MIME typ application/pkcs7-mime pro přenos dat do obálek (což je prakticky zašifrování otevřeného textu), kde je celý (upravený) MIME subjekt, zabalený a šifrovány do jednoho objektu, který je následně vložen do application/pkcs7-mime MIME subjektu.

Pro přípravu poštovní zprávy je nejdříve nutné zabezpečit data, a to pomocí protokolu CMS (Cryptographic Message Standard), poté je potřeba data překódovat pomocí normy Base64 do sedmibitové podoby. A nadále získaná data podepsat, nebo je vložit do obálky. Na pořadí těchto dvou kroků nezáleží, avšak se doporučuje nejdříve data podepsat a až poté je zašifrovat. CMS je protokol určený pro zabezpečených zpráv (RFC 2630).

Pozn.: Standard MIME definuje strukturu elektronických zpráv, které mohou obsahovat různá multimediální data. Vznikl pro potřeby elektronické pošty, ale byl převzat i dalšími internetovými službami, např. službou WWW. MIME zavádí popisnou část zprávy, tzv. hlavičku, která umožňuje přesně deklarovat typ přenášených dat a tím usnadnit jejich další zpracování. Jedná se zejména o označení typu a podtypu přenášených dat. Syntakticky je hlavička tvořena seznamem klíčových slov, kterým jsou přiřazeny normou definované hodnoty. Jako příklad můžeme uvést identifikátor typu s názvem Content-type, který může na základě obsahu zprávy nabývat např. hodnot application/msword, image/jpeg, audio/wav, atd.

Autentizace a šifrování

Autentizace je proces ověření identity subjektu. Autentizace je ověření identity uživatele služeb nebo původce zprávy. Používají se tyto základní metody pro zjištění identity:

- podle toho, co uživatel **zná** (zná správnou kombinaci uživatelského označení a hesla nebo PIN – Personal Identification Number)
- podle toho, co uživatel **má** (nějaký technický prostředek, který uživatel vlastní – hardwarový klíč, smart card, privátní klíč apod.)
- podle toho, čím uživatel **je** (uživatel má biometrické vlastnosti, které lze prověřit – otisk prstu, snímek oční duhovky či sítnice apod.)
- podle toho, co uživatel **umí** (umí správně odpovědět na náhodně vygenerovaný kontrolní dotaz)

Hesla a PINy

Autentizace pomocí hesla je nejjednodušším způsobem autentizace v současné době. Je používána ve velkém množství aplikací. Jako příklad můžeme uvést SMTP, POP3 a IMAP protokoly pro připojování k e-mailovým serverům, různe IRC, apod. Protokol spočívá v tom, že Alice (A, první účastník komunikace) pošle Bobovi (B analogicky k A) heslo. Bob má někde v databázi uložena hesla všech svých komunikačních partnerů a po příjmu hesla si najde příslušný záznam patřící Alici a porovná zaslalané heslo s kopí ve svém záznamu.

Heslo typicky bývá řetězec dlouhý 6-10 znaků, v ideálním případě netriviální (odolný proti možnému slovníkovému útoku, či útoku hrubou silou), ale uživatelem snadno zapamatovatelný. Uživatel předkládá systému heslo (sdílené tajemství) společně se svou identifikací - uživatelským jménem (loginem). Systém tyto autentizační údaje kontroluje s daty uloženými k danému uživateli. Prokázání znalosti tajemství je vyhodnoceno systémem jako korektní prokázání identity.

Běžní uživatelé si většinou nejsou vědomi (ne)bezpečnosti, kterou jejich hesla reprezentují. Dnešní systémy spravující hesla proto umožňují kontrolu bezpečnosti vkládaných hesel (včetně populárních indikátorů vhodnosti), příp. uživateli vygenerují heslo s požadovanými parametry. Požadavky kladené na tato hesla jsou pak součástí bezpečnostních politik systému. Stinnou stránkou tohoto přístupu ale je, že uživatel si heslo bude obtížně pamatovat a často zapomínat.

Jako bezpečné heslo (jakkoliv je pojmen relativní) lze považovat to, jehož prolomení obvyklými technikami je časově náročné. Momentálně odporúčaná délka hesla je aspoň 10 znakov, přičemž by mali byť použité male až veľké písmená, číslice, prípadne špeciálny znak.

Doporučovaným způsobem pro zvyšování bezpečnosti hesla je zvětšování základní množiny znaků před prodlužováním (problém zapamäteľnosti hesla vs jeho sily).

PINy poskytují jinou možnost posílení bezpečnosti. V tomto případě omezujeme počet pokusů, které máme k dispozici pro uhádnutí hodnoty PINu. Pokud se v daném počtu pokusů netrefíme, tak systém zablokuje přístup uživateli a je nutné použít nějaký složitější mechanizmus na odblokování PINu a tím vynulování počtu chybných pokusů. Tímto druhým mechanizmem může být mnohem delší PIN (někdy označován jako PUK), nebo např. osobní kontakt se zákaznickým centrem, které bude vyžadovat předložení např. identifikačních dokladů před tím, než bude PIN odblokován.

Díky tomuto omezení je možné značně zjednodušit formu a délku PINu v porovnání s heslem. Obvyklý PIN je složen pouze z číslic a jeho délka bývá 4-8 znaků. V mnoha případech si uživatelé mohou PIN sami měnit podle potřeby. U nás je to obvyklé např. u mobilních telefonů, v jiných zemích je možné měnit PIN i pro platební karty.

Bohužel, mechanizmus omezení počtu pokusů není vhodné obecně použít pro hesla (zejména pak, je-li login veřejně známý či snadno odvoditelný), protože by reálně hrozil útok odmítnutí služby (Denial of Service). Jestliže by vám chtěl někdo znemožnit přístup do systému, prostě by několikrát zadal správné stejné jméno a chybné heslo.

Nutným předpokladem pro fungování tohoto mechanizmu je nutnost fyzického vlastnictví autentizačního předmětu (tokenu), jedná se vlastně tedy o tzv. dvoufaktorovou autentizaci. Bez vlastnictví autentizačního předmětu pak není možné PIN vůbec zadat. Tímto předmětem může být mobilní telefon, SIM karta, nebo kreditní karta.

Autentizační tokeny

Tokeny jsou, zjednodušeně řečeno, zařízení, která mohou uživatelé nosit neustále s sebou a jejichž vlastnictví je nutné pro to, aby se mohli autentizovat do systému. Mají buď specifické fyzické vlastnosti (tvar, elektrický odpór, elektrickou kapacitu, ...), nebo obsahují specifické tajné informace (např. kvalitní heslo nebo kryptografický klíč), nebo jsou dokonce schopny provádět specifické (obvykle kryptografické) výpočty.

Asi nejčastějším autentizačním tokenem současnosti jsou karty. Můžeme je dělit na několik typů - typicky podle jejich obsahu a schopností. Úplně nejjednodušší jsou karty s magnetickým proužkem (obsahují obvykle neměnnou informaci, kterou lze ale kdykoliv přepsat, dnes už nepovažované za bezpečné), složitějšími a dražšími jsou čipové karty (dokáží provádět nad uloženými/zaslánými daty různé operace). Téměř každý, kdo má bankovní účet, tak vlastní alespoň jednu platební kartu. Každý kdo má mobilní telefon, pak vlastní čipovou kartu ve formě SIM karty.

Dalším obvyklým typem tokenu je tzv. autentizační kalkulátor. Samotné kalkulátory mohou být založeny buď na tajemství, které je uloženo v kalkulátoru a v autentizačním serveru, nebo na synchronizovaných hodinách. Důležitou vlastností kalkulátorů je způsob komunikace s uživatelem - klasické komunikační rozhraní typicky zahrnuje pouze klávesnici a displej, speciální optická rozhraní či infračervený port umožňují navíc kalkulátoru komunikovat přímo s počítačem.

V posledních letech se poměrně rozšířily také tzv. USB tokeny. Pojem "token" zde však byl použit pro zařízení, která v drtivé většině případů neposkytují bezpečné úložiště dat, a jsou tedy pro účely autentizace zcela nevhodná. I zde samozřejmě existují výjimky (specializované USB tokeny), které typicky využívají stejnou technologii jako čipové karty. Cena takového tokenu je ale výrazně vyšší, a množství dat, které dokáží bezpečně uchovat, se už nepohybuje v řádech megabajtů či gigabajtů, ale pouze v řádech kilabajtů.

Biometriky

Biometrické techniky můžeme použít na dvě rozdílné aplikace: na autentizaci neboli verifikaci identity a na identifikaci. Autentizace/verifikace je proces, při kterém subjekt předkládá tvrzení o své identitě (např. vložením karty nebo zadáním identifikátoru) a na základě takto udané identity se srovnávají aktuální biometrické charakteristiky s uloženými charakteristikami, které této identitě odpovídají podle záznamů autentizační databáze. Odpovídáme na otázku: "Je to opravdu ta osoba, za kterou se sama vydává?" Při identifikaci (nebo také vyhledání) naopak člověk identitu sám nepředkládá, systém prochází všechny (relevantní) záznamy v databázi, aby našel patřičnou shodu a identitu člověka sám rozpoznal. Systém odpovídá na otázku: "Kdo to je?" Je zřejmé, že identifikace je podstatně náročnější proces než verifikace. Se zvyšujícím se rozsahem databáze se přesnost identifikace snižuje a rychlosť klesá.

Biometrických technologií existuje mnoho a jsou založeny na měření fyziologických vlastností lidského těla (např. otisk prstu nebo geometrie ruky) nebo chování člověka (např. dynamika podpisu nebo vzorek hlasu), přičemž se jedná o měření automatizovaným způsobem. Některé technologie jsou teprve ve stádiu vývoje (např. analýza pachů), avšak mnohé technologie jsou již relativně vyzrálé a komerčně dostupné (např. otisky prstů nebo systémy porovnávající vzorek oční duhovky). Systémy založené na fyziologických vlastnostech jsou obvykle spolehlivější a přesnější než systémy založené na chování člověka, protože měření fyziologických vlastností jsou lépe opakovatelná a nejsou ve velkém míře ovlivněna daným (psychickým, fyziologickým) stavem jako např. stres nebo nemoc.

Nejvýznamnější rozdíl mezi biometrickými a tradičními technologiemi je odpověď systému na autentizační požadavek. Biometrické systémy nedávají jednoduché odpovědi typu ano/ne. Heslo buďto je "abcd" nebo ne, magnetická karta s číslem účtu "1234" jednoduše je nebo není platná. Podpis člověka však není vždycky naprostě stejný, stejně tak pozice prstu při snímání otisku se může trochu lišit. Biometrický systém proto nemůže určit identitu člověka absolutně, ale místo toho řekne, že s určitou pravděpodobností (vyhovující autentizačním/identifikačním účelům) se jedná o daného jedince.

Mohli bychom samozřejmě vytvořit systém, který by vyžadoval pokaždé téměř 100% shodu biometrických charakteristik. Takový systém by však nebyl prakticky použitelný, neboť naprostá většina uživatelů by byla téměř vždy odmítнутa, protože výsledky měření by byly vždy alespoň trochu rozdílné. Abychom tedy udělali systém prakticky použitelný, musíme povolit určitou variabilitu biometrických charakteristik. Současné biometrické systémy však nejsou bezchybné, a proto čím větší variabilitu povolíme, tím větší šanci dáváme podvodníkům s podobnými biometrickými charakteristikami.

Složitější autentizační schémata

Probíhá-li autentizace uživatele v zabezpečeném výpočetním prostředí, jsou i přenášená autentizační data (tajné informace nezbytné pro korektní autentizaci - např. PINy, hesla, šifrovací klíče) v bezpečí. To však neplatí pokud se uživatel autentizuje ke vzdálenému systému. Autentizační data jsou pak totiž přenášena nezabezpečeným prostředím (např. počítačovou sítí, která není pod naší kontrolou) a mohou být snadno odposlechnuta a zneužita pro neoprávněný přístup ke vzdálenému systému. Pouhé hašování (tj. zpracování vhodnou jednosměrnou funkcí) či šifrování autentizačních dat není samo o sobě vhodným řešením - autentizační data sice zůstanou utajena, ale pro neoprávněný přístup k systému stačí příslušný haš (tj. výsledek hašování) či zašifrovaná autentizační data.

Proto se používají složitější autentizační schémata - tzv. autentizační protokoly - která umožňují demonstrovat znalost sdíleného tajemství, aniž by během autentizace poskytla případnému útočníkovi (ať již pasivnímu či aktivnímu) jakoukoliv užitečnou informaci využitelnou pro další (neoprávněnou) korektní autentizaci a následný (neoprávněný) přístup k systému. Tyto protokoly jsou většinou budovány s využitím základních kryptografických primitiv (symetrické či asymetrické kryptosystémy, kryptografické hašovací funkce apod.) a pracují na principu výzva-odpověď. Základní myšlenkou tohoto přístupu je ověřování správnosti a čerstvosti (nebyl dříve odposlechnut) autentizačního požadavku. Ten je typicky zaslán jako odpověď na unikátní výzvu, a demonstruje tak znalost nějakého sdíleného tajemství, které je kryptografickými prostředky aplikováno na onu autentizační výzvu. Na tomto principu fungují například mnohé autentizační kalkulátory.

Většina běžně používaných autentizačních protokolů však vyžaduje předem ustavené sdílené tajemství - např. šifrovací klíče. Ty jsou dlouhé řádově stovky bitů a proto bývají na straně uživatelů typicky uloženy na nějakém tokenu. Poměrně efektivním řešením tohoto problému jsou speciálně navržené autentizační protokoly umožňující namísto klíčů použít data s menší entropií - jako například PINy či hesla - která je schopen si uživatel zapamatovat. Tyto protokoly, někdy označované jako eskalační, jsou založeny na kombinaci symetrické a asymetrické kryptografie. Oproti běžným autentizačním protokolům umožňují použití hesel aniž by je vystavovaly off-line útokům hrubou silou (tj. také slovníkovým útokům). Tyto eskalační protokoly však zatím pronikají do praxe jen pozvolna. Jsou již ale součástí některých nově vytvářených norem a standardů.

Řetězce důvěryhodných autorit

Mnohé v současné době nasazované metody a autentizační protokoly pro ověření autentičnosti dat uložených na tokenu nějakým způsobem využívají prostředků asymetrické kryptografie (kryptosystémy založené na problémech teorie čísel a složitosti). Mezi ně patří např. i systémy pro ověřování nových elektronických (biometrických) pasů, či nových čipových platebních (kreditních i debetních) karet v tzv. EMV platebních systémech.

Aby takovéto řešení mohlo v praxi fungovat, je nutné vytvořit infrastrukturu veřejných klíčů (PKI - Public Key Infrastructure). Ta je budována pomocí řetězce důvěryhodných autorit, kde každá autorita v řetězci ověří a certifikuje veřejný klíč následující autority. Jelikož je veřejný klíč jednoznačně matematicky svázán s příslušným soukromým klíčem, je takto vytvořen efektivní mechanismus pro ověření totožnosti vlastníků soukromých klíčů pomocí "automatické kontroly" certifikátu v řetězci. Tyto důvěryhodné autority se nazývají certifikační autority (CA). CA jsou uspořádány do hierarchické struktury s jasně definovanými vztahy podřízenosti / nadřazenosti. Průchod takovou strukturou vytváří výše zmíněný řetězec autorit s počátkem v kořeni hierarchické struktury.

V praxi je ale často používán pouze jedno- až tří-úrovňový hierarchický stromový model. Certifikát je digitálně podepsán zpráva sestávající ze dvou hlavních informací: jména vlastníka veřejného klíče a samotného veřejného klíče. Hlavním účelem certifikátu je kryptografické spojení veřejného klíče a identitou daného subjektu (za korektnost této vazby ručí CA, která certifikát vydala).

Certifikační autoritu může zřídit libovolná organizace a výstupy používat pro svou interní potřebu (toho využívají některé velké instituce jako banky či univerzity) nebo v rámci účelového sdružení více institucí, které deklarují vzájemnou důvěru k vydaným klíčům a certifikátům. Subjekt stojící mimo sdružení může ale i nemusí takovéto CA důvěřovat.

Akreditované CA jsou certifikačními autoritami, které prošly akreditačním procesem ze strany státních orgánů (u nás např. První certifikační autorita I.CZ, Česká pošta, elidentity). Mají proto postavení kvalifikované instituce s obecně uznávanou důvěryhodností a využitím zejména pro orgány státní

správy, a také bez omezení pro libovolné nestátní subjekty. Toto postavení akreditované CA můžeme přirovnat k funkci notáře - kdy notářem podepsaná písemnost nebo ověřený podpis občana jsou obecně důvěryhodné pro ostatní instituce a není potřebné dále zpětně zkoumat pravost. Těmto CA se také někdy říká kvalifikovaná certifikační autorita.

Šifrování

Kryptografie, neboli šifrování dat, je náuka o transformování dat do podoby, která je čitelná jenom s určitou znalostí. Slovo Kryptografie pochází z řečtiny – kryptós (je skrytý) a gráphein (psát). Často je pojednáván, nebo používán, pro vědu o šifrách – kryptologii.

Kryptografie se vyvíjí již celá staletí, nebo tisíciletí. V historii postupně vznikali lepší a důmyslnější šifry, které často ovlivnili mnohé historické události (zejména, jednalo li se o utajování a vyzrazení informací).

Symetrická a asymetrická kryptografie

Symetrické šifry se často používají společně s asymetrickými. Obvyklé použití je takové, že otevřený text se zašifruje symetrickou šifrou s náhodně vygenerovaným klíčem. Tento symetrický klíč se zašifruje veřejným klíčem asymetrické šifry, takže dešifrovat data může pouze majitel tajného klíče dané asymetrické šifry.

Symetrické šifry se dělí na dva druhy. Proudové a blokové. Proudové šifry zpracovávají otevřený text po jednotlivých bitech. Blokové šifry rozdělí otevřený text na bloky stejné velikosti a doplní vhodným způsobem poslední blok na stejnou velikost. U většiny šifer se používá blok o 64 bitech, AES používá 128 bitů.

Blokové šifry

AES

Blowfish

DES

GOST

IDEA

RC2

RC5

Triple DES

Twofish

Skipjack

Proudové šifry

FISH

RC4

Asymetrická kryptografie (kryptografie s veřejným klíčem) je skupina kryptografických metod, ve kterých se pro šifrování a dešifrování používají odlišné klíče (veřejný a soukromý). To je základní rozdíl oproti symetrické kryptografii, která používá k šifrování i dešifrování jedený klíč.

Kromě očividné možnosti pro utajení komunikace se asymetrická kryptografie používá také pro elektronický podpis, tzn. možnost u dat prokázat jejich autora. Šifrovací klíč pro asymetrickou kryptografii sestává z dvou částí: jedna část se používá pro šifrování zpráv (a příjemce zprávy ani tuto část nemusí znát), druhá pro dešifrování (a odesílatel šifrovaných zpráv ji zpravidla nezná). Je vidět, že ten, kdo šifruje, nemusí s dešifrujícím příjemcem zprávy sdílet žádné tajemství, čímž eliminují potřebu výměny klíčů; tato vlastnost je základní výhodou asymetrické kryptografie.

Nejběžnější verzí asymetrické kryptografie je využívání tzv. veřejného a soukromého klíče: šifrovací klíč je veřejný, majitel klíče ho volně uveřejní, a kdokoli jím může šifrovat jemu určené zprávy; dešifrovací klíč je privátní (tj. soukromý), majitel jej drží v tajnosti a pomocí něj může tyto zprávy dešifrovat (existují i další metody asymetrické kryptografie, ve kterých je třeba i šifrovací klíč udržovat v tajnosti). Je zřejmé, že šifrovací klíč e a dešifrovací klíč d spolu musí být matematicky svázány, avšak nezbytnou podmínkou pro užitečnost šifry je praktická nemožnost ze znalosti šifrovacího klíče spočítat dešifrovací.

Matematicky tedy asymetrická kryptografie postupuje následujícím způsobem:

Šifrování

$$c = f(m, e)$$

Dešifrování

$$m = g(c, d)$$

V principu se mohou šifrovací a dešifrovací funkce lišit, zpravidla jsou však matematicky přinejmenším velmi podobné.

V dnešnej dobe sa prístup utajovania šifrovacieho algoritmu nepokladá za správny. Väčšina šifier je preto založená na tzv. výpočetnej bezpečnosti. Väčšina asymetrických šifier používa prvočísla a práve na princípe faktorizácie prvočísel, na ktorú je potrebný obrovský výpočetný výkon je ich bezpečnosť založená. Pri ich použití predpokladáme, že útočník nebude mať dostatočný výpočetný výkon / neoplatí sa mu investovať do ňho pre zistenie nášho tajomstva. To je zároveň dôvodom, prečo sa po čase menia odporučené dĺžky šifrovacích kľúčov.

Mechanismy funkce

Asymetrická kryptografie je založena na tzv. jednocestných funkcích, což jsou operace, které lze snadno provést pouze v jednom směru: ze vstupu lze snadno spočítat výstup, z výstupu však je velmi obtížné nalézt vstup. Nejběžnějším příkladem je například násobení: je velmi snadné vynásobit dvě i velmi velká čísla, avšak rozklad součinu na činitele (tzv. faktorizace) je velmi obtížný. (Na tomto problému je založen např. algoritmus RSA.) Dalšími podobnými problémy jsou výpočet diskrétního logaritmu či problém batohu.

Bezpečnost

Některé kryptografické metody mohou být označeny za bezpečné na základě složitosti matematického problému jako prvočíselný rozklad násobku dvou velkých prvočísel nebo počítání diskrétního logaritmu. Toto je pouze matematický smysl bezpečnosti a existuje více různých definicí, dle kterých se může kryptografická metoda označit za bezpečnou.

Na rozdíl od Vernamovy šifry se žádná metoda asymetrické kryptografie neukázala jako bezpečná při použití nekonečného výpočetního výkonu. Důkazy bezpečnosti těchto metod tedy počítají s omezeným výpočetním výkonem a říkají například že „metoda je nerozluštitevná pomocí osobního počítače za 1000 let“, nebo „tentot algoritmus je bezpečný pokud nedojde k objevení lepší metody pro faktorizaci“.

Jednou z aplikací asymetrické kryptografie je elektronický podpis. Uživatel, který chce poslat zprávu, tak spočítá její digitální podpis a ten pošle spolu se zprávou příjemci. Digitální podpis může být spočítán jedině pomocí odesílateleova privátního klíče, ale pro ověření stačí pouze veřejný klíč. V některých případech (např. RSA) jsou metody elektronického podpisu velmi podobné kryptografickým metodám. V jiných případech (např. DSA) algoritmus nepřipomíná žádnou kryptografickou metodu.

Pro dosažení ověřitelnosti a utajení odesílatele zprávu podepíše svým privátním klíčem, potom ji zašifruje veřejným klíčem příjemce.

Zástupci

RSA

ElGamal

Kryptografie nad eliptickými křivkami

Kryptografie s Lucasovými funkcemi

Diffie-Hellman

Digital Signature Algorithm

Merkle-Hellman (už zastaralý)

Medzi najmodernejšie, ale ešte nie rozšírené technológie sa považujú napr. zero-knowledge protokoly alebo kvantová kryptografia.

Softwarové inženýrství

Životní cyklus SW a související aktivity. Specifikace požadavků a systémová analýza. Strukturované vs. objektově orientované metody analýzy a návrhu. Klíčové modely strukturované analýzy. Role jazyka UML v podpoře analýzy a návrhu SW.

Softwarové inženýrství

- podobor systémového inžinierstva; zaoberá sa metodikami a nástrojmi pre vývoj SW
- kľúčový pojem = vývoj kvalitného SW pri obmedzenom rozpočte (**cost-effective development**)
- výsledkom procesu sú 2 druhy produktov:
 - **krabicové (generic) produkty**
 - standalone SW bez špecifických požiadavkov
 - široká škála užívateľov
 - napr.: grafické programy, OS, CAD (computer-aided design)
 - **produkty na zákazku**
 - špecifické požiadavky dané zákazníkom
 - napr.: embedded systémy, systém pre monitoring dopravy
- princípy SWING aplikovateľné na akýkoľvek SW:
 - systémy by sa mali využívať s využitím **organizovaného a zrozumiteľného vývojového procesu**
 - klásť dôraz na **spoľahlivosť a výkon**
 - porozumenie a správa SW **špecifikácie a požiadavkov**
 - používať skôr už vyvinutý SW (adaptovať ho), než vytvárať úplne nový – **reused SW**

1) Životný cyklus SW a súvisiace aktivity

- životný cyklus (SW lifecycle) zahrňuje fázy:
 1. **Špecifikácia** (Specification)
 - užívatelia a inžinieri definujú SW a obmedzenia na jeho operácie
 2. **Analýza a návrh** (Analysis and design)
 - požiadavky zo špecifikácie sú zapracované do systémového návrhu
 3. **Implementácia** (Implementation)
 - SW je implementovaný do podoby spustiteľného systému
 - dôraz je okrem programovania kladený na implementačné problémy:
 - reuse – maximalizácia využitia už napísaného kódu
 - configuration management – proces manažovania meniaceho sa SW systému (držať sa v obreze v súvislosti s odlišnými verziami SW komponent); cieľom je podporiť proces integrácie systému
 - host-target development – otázka vývojovej a exekučnej platformy; SW je často vyvíjaný na inom PC (host) než na ktorom je následne používaný (target)
 - využitie IDE (Integrated Development Environment) ako nástroja na implementáciu
 - opensource vývoj
 - zdrojový kód je verejný a modifikovateľný širokou verejnosťou
 4. **Validácia a verifikácia** (Validation & Verification)
 - **verifikácia:**
 - overenie toho, že produkt poskytuje funkcionality danú špecifikáciou
 - Building the thing right
 - prebieha obvykle pred validáciou
 - využitie dokumentácií, plánov, kódov, špecifikácií, požiadavkov
 - **validácia:**

- overenie toho, že poskytovaná funkcia pracuje podľa očakávaní
- Building the right thing
- využitie samotného produktu

▪ **neformálne metódy:**

- **simulácia** – imitácia chovania vyvádzaného produktu a pozorovanie behu
- **testovanie** – pozorovanie chovania nad vybranou množinou vstupných dát
- **runtime analýza** – pozorovanie chovania systému v dobe skutočného behu

▪ **formálne metódy:**

- **model checking** – systém abstrahovaný do modelu s konečne stavovou sémantikou, pre ktorý je algoritmicky dokázaná platnosť danej vlastnosti
- **dokazovanie** – matematické preukázanie vlastnosti algoritmov

▪ **poloformálne metódy:**

- **statická analýza programu** – zistenie vlastností programu na základe ich popisu v danom programovacom jazyku (tzn. bez nutnosti skutočného spúšťania programu); typické použitie – prekladač (detekcia mŕtveho kódu, použitia nedefinovaných premenných)

▪ fázy testovania:

1. **development testing** (unit testing, component testing, system testing) – testovanie systému počas vývoja
2. **release testing** – separátny testovací tím testuje celú aplikáciu
3. **user testing** – testovanie systému užívateľmi

5. Provoz a údržba (Evolution)

▪ nevyhnutnosť vývoja SW produktu spôsobená:

- nové požiadavky užívateľov
- oprava chýb
- zmena HW možností (dostupné výkonnejšie stroje)

▪ fázy SW produktu:

1. **evolúcia (evolution)** – implementácia nových požiadavkov
2. **obsluha (servicing)** – oprava chýb, nová funkcia nepridávaná
3. **phase-out** – SW použiteľný, ale nepridávané žiadne zmeny

- podoba SW lifecycle záleží aj od modelu vývoja SW:

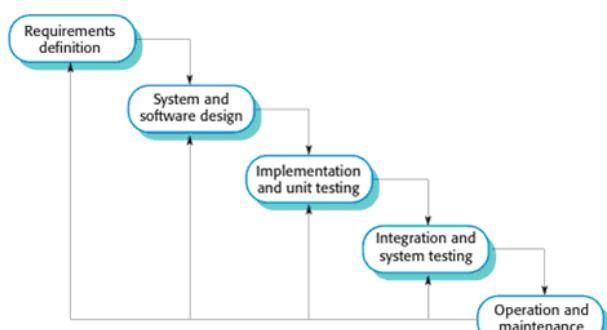
1. waterfall model

- model založený na presne definovanom postupe jednotlivých procesov
- **plan-driven model** – presná definícia jednotlivých procesov a artefaktov, detailný procesný monitoring, kontrola, validácia a verifikácia, silný dôraz na dokumentáciu
- **výhody:**

- vhodný pre veľké/malé systémy

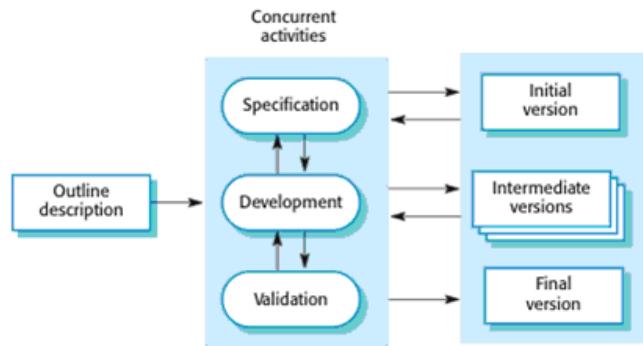
- **nevýhody:**

- nie je flexibilný – nereaguje rýchlo na zmeny v užívateľských požiadavkách (a ak áno, tak za vysokú cenu)
- dôraz na dokumentáciu každého procesu



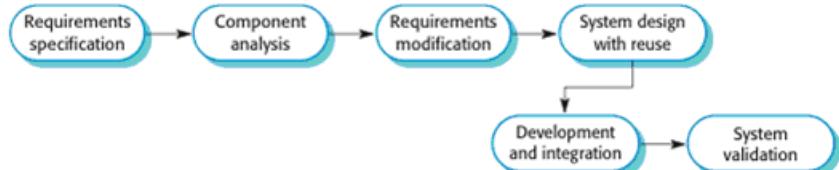
2. inkrementálny (incremental) model

- založený na postupnom pridávaní funkcionality (nabáľovanie komponent) v podobe tzv. inkrementu
- **plan-driven/agile**
- **výhody:**
 - klesá cena implementácie
 - zmeny požiadaviek od zákazníka (prerobenie menšieho množstva dokumentácie a analýzy)
 - získanie zákazníkovho feedbacku počas implementácie funkcionality
 - zákazník získava funkčný SW skôr, než pri modeli vodopádu
- **nevýhody:**
 - proces nie je viditeľný – potreba neustáleho releasu znamená rýchly a kontinuálny vývojový proces, je teda neefektívne dokumentovať každú zmenu
 - štruktúra systému má tendenciu degradovať s ďalším inkrementom – potreba neustáleho refactoringu



3. reuse-oriented model

- založený na systematickom použití už vyvinutých komponent alebo COTS systémov (Commercial Off The Shelf)
- **plan-driven/agile**
- **výhody:**
 - ľahkosť
 - vymeniteľnosť komponenty
 - závisí od integrovanej komponenty
- **nevýhody:**
 - závisí od integrovanej komponenty



2) Špecifikácia požiadavkov a systémová analýza

- proces, v ktorom zákazník definuje **požadované služby** a **obmedzenia**, za ktorých má systém fungovať a byť vyvýjaný
- **požiadavky** – popis systémových služieb a obmedzení, od veľmi abstraktných až po požiadavky špecifikované matematickou funkciou
- prvý krok pri vývoji SW, niekedy nezbytná podmienka pre získanie kontraktu
- zaoberá sa ňou tzv. Requirements engineering
- požiadavky kladené na špecifikáciu (v praxi obtiažne udržateľné):
 - precíznosť – jednoznačná interpretácia
 - kompletnosť – obsahuje popis všetkých požadovaných možností
 - konzistencia – nemali by vznikať kontradikcie v popisoch

- delenie požiadavkov:
 - **funkčné**
 - služby, ktoré má systém poskytovať
 - spôsob reakcie systému na určité vstupy a v daných situáciách
 - môžu uvádzať aj to, čo má systém robiť
 - **nefunkčné**
 - vlastnosti a obmedzenia systému vzťahujúce sa na služby (spoločnosť, bezpečnosť, robustnosť, obmedzenie na platformy)
 - často vplyv na systém ako celok (než na jednotlivé časti)
 - nefunkčný požiadavok môže vygenerovať niekoľko súvisiacich funkčných
 - musia byť verifikovateľné skrz metriku
 - systém musí byť ľahko použiteľný = zamestnanci budú schopní po 4hodinovom školení so systémom pracovať
- **systémová analýza:**
 - kreatívna aktivita, v ktorej identifikujeme procesy, entity (objekty) a ich vzťahy
 - závislá od schopností a skúseností analytika (neexistuje univerzálna formula)
 - iteratívny proces (prvýkrát sa spravidla nedarí úplná identifikácia)
 - výsledok = návrh (nie však detailný) systému, jeho dekompozícia na časti, znázornenie komunikácie s prostredím

3) Štruktúrované vs. OO metódy analýzy a návrhu

- rozlišujú sa nasledovné pohľady:
 - **funkčne orientovaný** (function oriented)
 - systém ako množina interagujúcich funkcií
 - funkčná transformácia založená na procesoch prepojených s dátami
 - **dátovo orientovaný** (data oriented)
 - definuje základné dátové štruktúry systému
 - funkčný aspekt (transformácia dát) je menej podstatný
 - **objektovo orientovaný** (object oriented)
 - systém ako množina interagujúcich objektov zahŕňajúcich dátá (ich štruktúru) a operácie na nich vykonávané
- **OO analýza a návrh:**
 - využíva OO pohľad na systém
 - systém = skupina interagujúcich objektov, každý charakterizovaný triedou, stavom a chovaním
 - metódy:
 - Rational Unified Process – **RUP**
 - Unified Process – **UP**
 - použité diagramy (+ slovníček pojmov):
 - **use-case diagram** – actors, požiadavky na systém, flow of events
 - **activity diagram** – nodes, flows, connectors, edges
 - **class diagram** – triedy, vzťahy a väzby, kardinalita, dedičnosť, generalizácia
 - **object diagram** – instance tried (objekty), linky
 - **sequence diagram** – actors, messages, lifelines, objects

- **component diagram** – komponenty, konektory
- **deployment diagram** – HW komponenty (nodes), SW komponenty (artifacts)
- **state (machine) diagram**
- **package diagram** – package, trieda, interface, objekt
- priebeh:
 1. požiadavky – modelovanie hraníc, aktérov a požiadavkov (Use-case diagram)
 2. analýza – identifikácia analytických tried, vzťahov, dedičnosti, polymorfizmu (Class diagram); možný prevod Use-case na Activity/Interaction diagram
 3. návrh – z komponent sa zostavuje finálny Class diagram a Component diagram; detailný rozbor Use-case diagramu pomocou State a Interaction diagramu
- **štruktúrovaná analýza a návrh:**
 - využíva funkčne orientovaný pohľad na systém v spojení s dátovo orientovaným
 - rozdelenie systému na malé, dobre definované aktivity
 - určenie poradia aktivít a ich interakcie
 - pomocou hierarchických grafických techník konštrukcia detailnej štruktúrovanej špecifikácie zrozumiteľnej pre užívateľov aj vývojárov
 - metódy:
 - Yourdon: Modern Structured Analysis – **YMSA**
 - Structured System Analysis and Design Method – **SSADM**
 - použité diagramy:
 - **context diagram** – modeluje hranice systému a prostredie
 - **dataflow diagram** – systém ako sieť procesov sprostredkovávajúcich funkcie systému a manipuláciu s dátami
 - **Entity Relationship Diagram** – modeluje systémové dátá
 - príklad krokov štruktúrovanej analýzy:
 1. definovať systémový kontext a vytvoriť prvotné DFD
 2. načrtnúť iniciálny dátový model – ERD
 3. analyzovať dátové entity a vzťahy a vytvoriť finálny ERD
 4. upraviť DFD podľa ERD (vytvoriť tzv. logical process model)
 5. rozložiť logický model na procedurálne elementy
 6. špecifikovať detaily každého elementu

4) Kľúčové modely štruktúrovanej analýzy

- **Context diagram**
 - diagram modelujúci najvyšší pohľad na vyvíjaný systém
 - použitie počas počiatočných fáz vývoja s cieľom určiť hranice a rozsah aplikácie
 - znázorňuje jeden proces reprezentujúci celý systém
 - zdôrazňované elementy:
 - terminátori – ľudia a systémy komunikujúce so systémom
 - dátá na spracovanie – prijímané z okolitého prostredia
 - dátá zo systému – produkovaná systémom a vyslané zo systému von
 - dátá uložené – systémom a terminátormi (Asta la vista, baby!)
 - hranice systému
 - zahrňuje aj tzv. Event list – zoznam podnetov z okolia, na ktoré musí systém reagovať

- **Dataflow diagram**

- grafická reprezentácia systému ako siete procesov, ktoré naplňujú systémové funkcie a komunikujú skrz systém dát
- zobrazenie toho, aká informácie bude vstupom/výstupom, odkiaľ/kam dátá putujú, ako budú dátá uložené
- použitie počas počiatokných fáz vývoja
- typy dát:
 - procesy – transformujú vstupy na výstupy
 - toky dát – prenos dát z jednej časti systému do inej
 - úložište dát – modeluje statickú kolekciu dát zdieľanú procesmi v čase
 - terminátory – externé entity komunikujúce so systémom

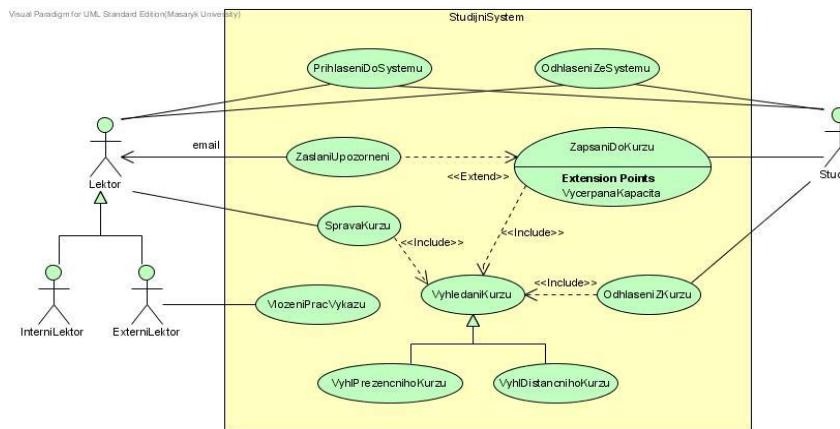
- **Entity-relationship diagram – ERD**

- abstraktný spôsob popisu databáze
- identifikuje systémové entity – abstraktné (napr. výuka) aj konkrétné (napr. študent)
- skúma:
 - entity a ich typy
 - vzťahy a ich typy
 - atribúty a ich domény
- entita:
 - jednoznačne identifikovateľná (využitie kľúčov – kandidátny, primárny)
 - potrebná – dôležitá rola v systéme
 - opísateľná atribútmi
- pre každý vzťah medzi entitami určujeme kardinalitu (1:1, 1:n, m:n, ...), medzi entitami môže byť viac vzťahov

5) Role jazyka UML v podpore analýzy a návrhu SW

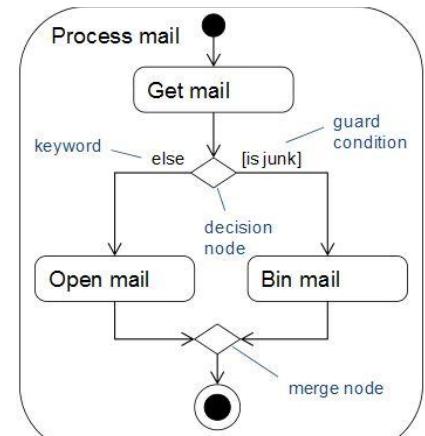
• Use-case diagram

- využitie pri špecifikácii požiadavkov
- definuje:
 - hranice systému
 - use-cases (prípady užitia)
 - funkcionálita
 - požadovaná aktérom
 - actors (aktérov) –
 - externé entity k systému,
 - špecifikujú role
 - a interagujú so systémom
- rozšírené modelovanie:
 - generalizácia/špecializácií a use-cases/aktérov
 - inkluzia(<<include>> stereotype)/rozširovanie(<<extends>> stereotype)



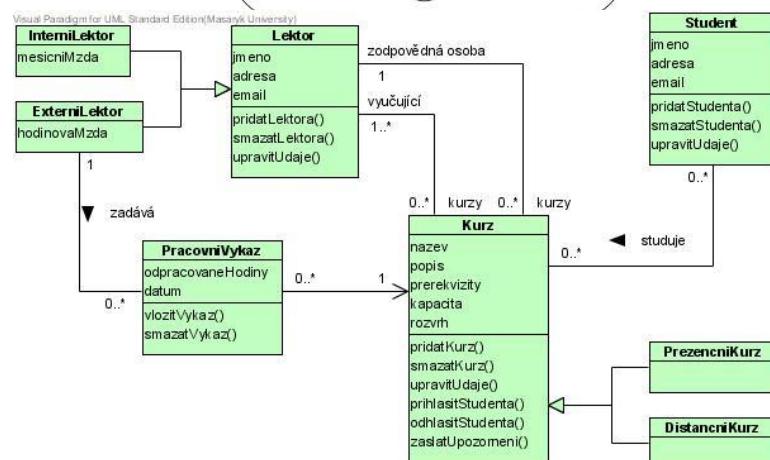
• Activity diagram

- modelovanie procesov ako kolekciu uzlov a hrán medzi nimi
- modelovanie správania use-casees, tried, rozhraní, komponent, metód,...
- aktivita = siet uzlov spojených hranami
- možné modelovať paralelné behy aktivít



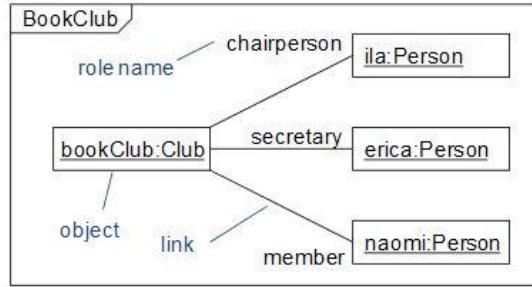
• Class diagram

- využitie:
 - analýza – jednoduchšia podoba, nie tak obsiahľa
 - implementácia – zložitejší, možno podľa neho programovať
- základná jednotka diagramu sú triedy, ktorých instance sú objekty
- trieda – atribúty špecifické pre všetky jej instance
- vzťahy – potrebné obsiahnuť kardinalitu, smer (navigability), dedičnosť (inheritance), závislosti (dependencies)



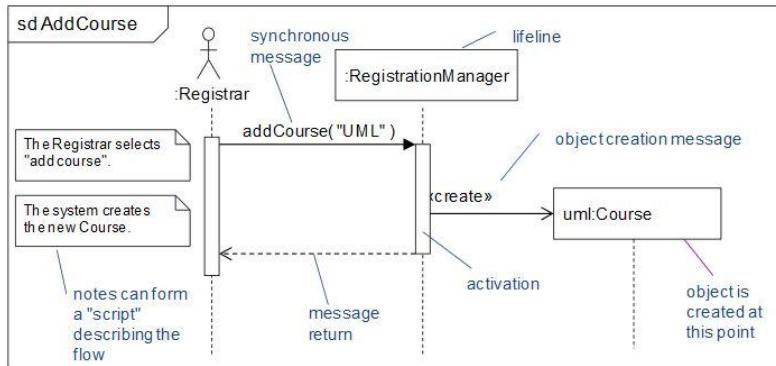
• Object diagram

- zobrazuje kompletný/čiastočný pohľad na štruktúru systému v špecifickú dobu
- pojmy:
 - objekty (instancie) a ich atribúty
 - spojenia (links) medzi objektami



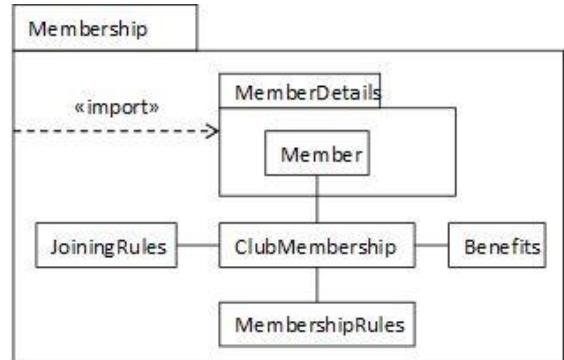
• Sequence diagram

- diagram interakcie, ktorý zobrazuje vzájomné chovanie procesov a ich poradie
- využíva konkrétnu časovú sekvenciu
- využíva:
 - lifelines – paralelné vertikálne čiary pre procesy vykonávané simultánne alebo pre simultánne existujúce objekty
 - messages – horizontálne správy, ktoré sú medzi procesmi/objektami vymieňané



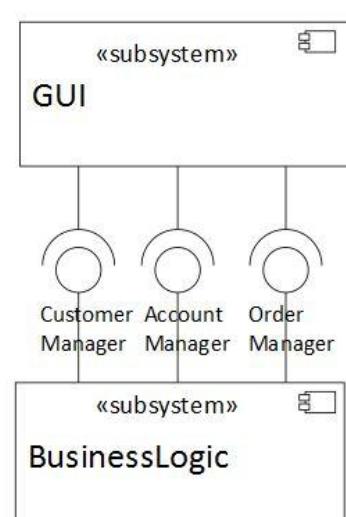
• Package diagram

- package – (logický) mechanizmus organizovania modelovaných elementov do sémanticky definovaných skupín
- možné vzťahy medzi balíčkami:
 - dependencies
 - generalisation (inheritance of elements from parent package)



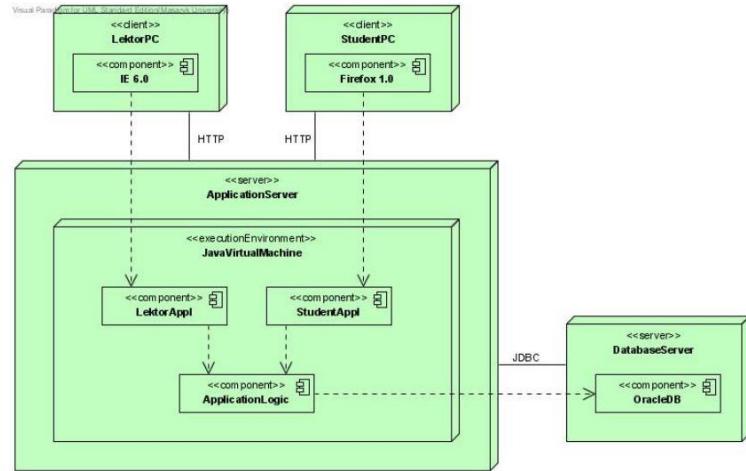
• Component diagram

- zobrazenie fyzického zoskupovania a prepojenia komponent v systéme
- komponenta:
 - modulárna časť systému, ktoré zapuzdruje svoj obsah a ktorá je nahraditeľná v rámci daného prostredia
 - fyzická/logická
- využíva:
 - konektory (connectors) – spájanie komponent



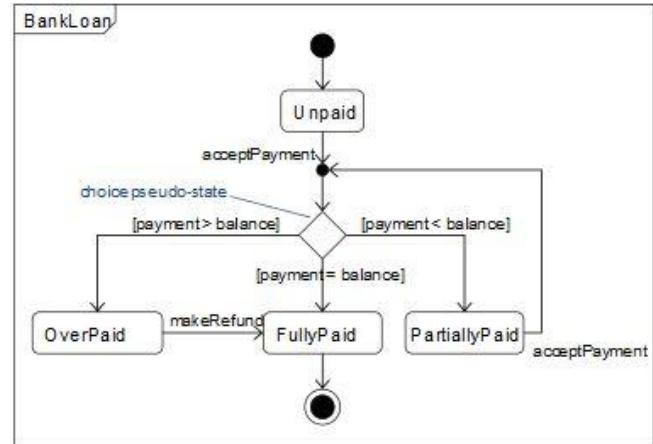
• Deployment diagram

- diagram popisujúci spôsob, akým je funkcia distribuovaná naprieč fyzickými uzlami
- modeluje mapovanie medzi SW architektúrou a fyzickou architektúrou systému
- fázy SW lifecycle: design (návrh diagramu) + implementácia (finálna verzia diagramu)



• State diagram

- modelovanie života objektu
- zobrazovanie stavov, v ktorých sa môže element nachádzať
 - stav môže mať akciu, ktorá je vykonaná pri vstupe/výstupe do tohto stavu
 - môžu byť aj podmienky pre vstup do iného stavu
 - vždy existuje iniciálny stav, pokial' nejde o nekonečný cyklus, tak aj koncový stav



Datové modelování

Návrh datových struktur, grafické vyjádření, převod do relačního modelu. ER diagram (entity, atributy, vztahy), UML diagram tříd a jejich srovnání.

Návrh dátových štruktúr

- Cílem datového modelování je navrhnut kvalitní datovou strukturu pro konkrétní aplikaci a databázový systém, který bude tato aplikace využívat k uložení dat.
- **Datový model** definuje neměnné atributy a strukturu dat a slouží pro návrh datové struktury.
- **Konceptuální datový model** je zobecněním konkrétní implementace datové struktury v relační databázi – lze jej přenášet do různých implementačních prostředí.

ER diagramy

- **Entitně relační model** je konceptuální model, slouží k popisu reálného světa, odvozuje se z něj relační schéma databáze

Entity

- **Entita (entity)** je objekt, který existuje, je odlišitelný od ostatních objektů, je potřebný (signifikantná rola v dizajnovanom systéme) a uchováváme o něm informace (např. osoba, firma, strom)
- Entita je popsána svým názvem a množinou **atributů**
- **Množina entit** (entity set) je skupina entit stejného typu, které sdílejí stejné vlastnosti (atributy) (např. skupina všech osob, firem, stromů)

Atributy

- **Atribut (attribute)** je popisná vlastnost (všech členů) entitní množiny nebo vztahu, jejíž hodnotu chceme uchovat a používat v systému; každý atribut má přiřazen i datový typ
- **Doména atributu (attribute domain)** je množina povolených hodnot pro každý atribut
- Typy atributů:
 - **jednoduché** atributy (např. jméno) a **složené** atributy (např. datum). Zložené atributy sa v ER modeloch môžu zobraziť ako entita.¹ Rozhodujúce sú odpovede na otázky:
 - **chceme o koncepte udržiavať nejaké informácie okrem názvu?**
 - **Má jednoduchú hodnotu?**
 - atributy s **jednoduchou** hodnotou (např. jméno) a s **násobnou** (multivalued) hodnotou (např. telefonní čísla)
 - **nulové** atributy (např. nemá telefon) (null)
 - **odvozené** atributy (např. věk)

¹ PB007/06/18-19 (podzim 2012) (predmet/týždeň/slajd)

Klíče

- **Klíč (key)** je podmnožina atributů
- **Superklíč (superkey)** množiny entit je množina jednoho nebo více atributů, jejichž hodnoty jednoznačně určují entitu
- **Kandidátní klíč (candidate key)** je minimální superklíč;
- **Primární klíč (primary key)** je jeden zvolený kandidátní klíč

Vztahy

- **Vztah (relationship)** je spojení mezi několika entitními množinami, které evidujeme a o němž uchováváme informace
- **Vztahová množina (relationship set)** je množina vztahů stejného druhu, také může mít atributy (např. množina vztahů *vkladatel* mezi množinami entit *základník* a *účet* může mít atribut *(poslední) datum přístupu*)
- **Stupeň vztahu** ukazuje počet množin entit, které jsou součástí množiny vztahů (nejčastěji binární)
- **Role** je vztah na jedné množině entit (např. když zaměstnanec je nadřízený jiného zaměstnance)
- **Četnost vztahů (multiplicity - násobnost)** označuje počet entit, se kterými mohou být ostatní entity propojeny pomocí vztahů (1:1, 1:N, M:N)
- **Existenční závislost – existence** (napr. V ERD jako identifikačná väzba)
- Entity x závisí na existenci ent. y (y je dominantní, x podřízená), jakmile je entita y (např. půjčka) smazána, pak musí být smazány všechny s ní spojené entity x (např. splátky). Dominantní množina entit se nazývá **silná**, podřízená množina entit se nazývá **slabá (weak)** – ta nemá primární klíč, protože závisí na existenci silné množiny entit, musí být spojena vztahem 1:N, primární klíč slabé množiny je tvořen **primárním klíčem** silné množiny a **parciálním (partial) klíčem** slabé množiny.
- **Parciální klíč** (též deskriptor) je množina atributů slabé entitní množiny, které společně s primárním klíčem silné entitní množiny tvoří primární klíč slabé ent. množiny.

Specializace

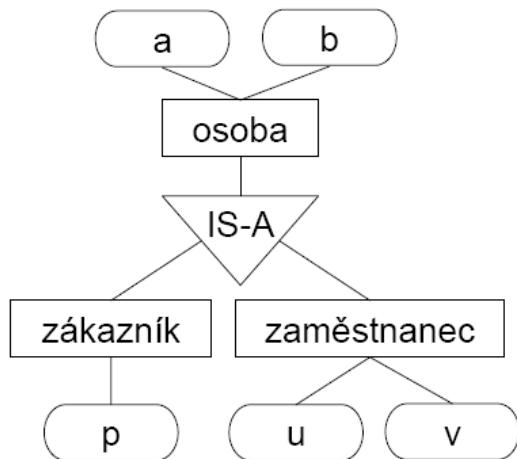
- Tvoříme podskupiny v množině entit, které jsou různé od ostatních entit a mají vlastní atributy
- **Úplná specializace (total specialization)²** – (každá entita z vyšší třídy **musí** patřit do jedné z entitních množin na nižší úrovni) **×** **částečná specializace** (entita z vyšší třídy **nemusí** patřit do jedné z entitních množin na nižší úrovni)

² however, total generalization is more common

- Disjunktívna špecializácia (disjoint/exclusive specialization) – entita generalizovanej entitnej množiny môže byť v jednej a iba v jednej špecializovanej entitnej množine.

Generalizace

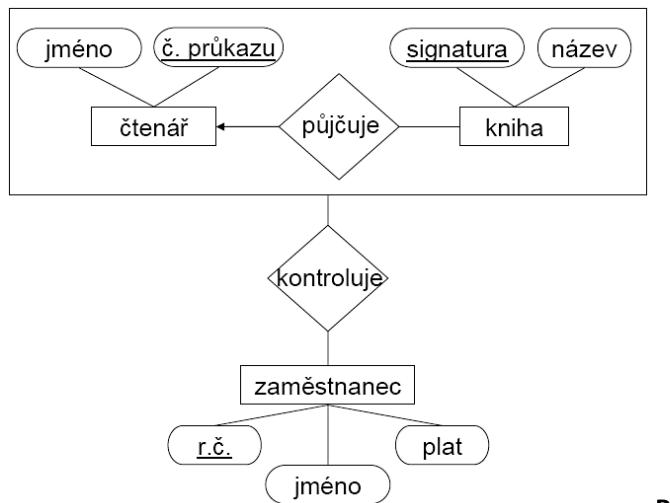
- Kombinujeme několik množin entit, které sdílejí stejné rysy, do množiny entit vyšší úrovně – specializace a generalizace jsou vzájemně inverzní, na ER diagramu se znázorňují stejně.
- Entita nižší úrovně dědí všechny atributy a účasti ve vztazích z množiny entit vyšší úrovně



Obrázek 1

Agregace

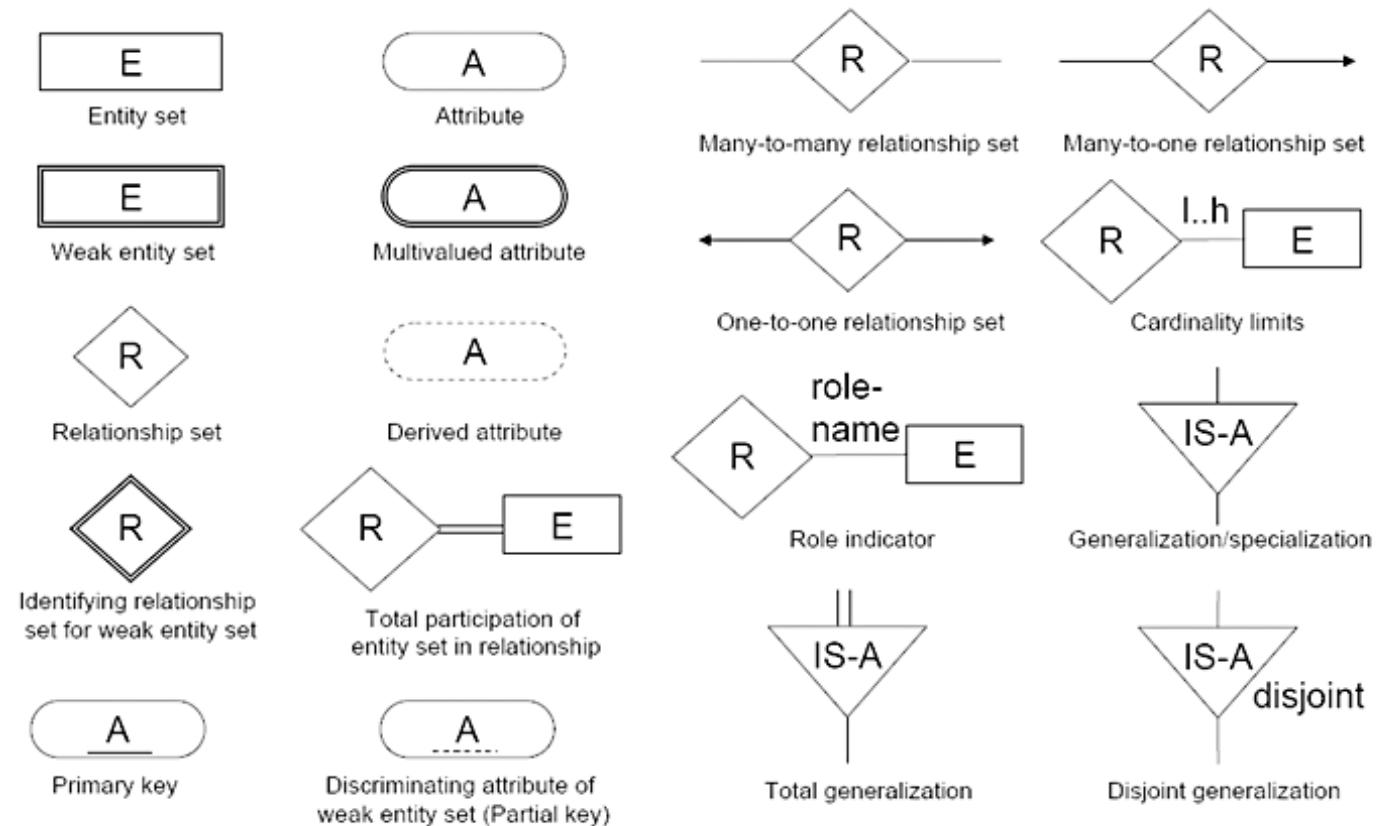
- umožňuje vytvářet vztahy mezi vztahy, se vztahem zacházíme jako s abstraktní entitou



Příklad agregace³

³ převzato z prezentace č. 2 do cvičení z predmetu PB154 Základy databázových systémů

Grafické vyjadrenie



- Entity set – silná množina entít
- Weak entity set – slabá množina entit // – (viď Vzťahy – Existenčná závislosť)
- Atributte – atribút
- Multivalued attribute – vícehodnotový atribut
- Derived attribute – odvozený atribut
- Primary key (attribute of) - atribut primárного klíče
- Partial key (discriminating attribute of weak entity set) – atribut parciálного klíče
- Relationship set – množina vzťahu
- Generalizácia, špecializácia
- Aggregácia – zahrnutí vzťahu i s entitními množinami do obdélníku (pozri vyššie)
- Total participation – každá entita, ktorá podlieha totálnej participácii (účasti) je vo vzťahu.

Podnety, co lze zmíniť dále

- určitej je vhodné nakresliť nějaký malý E-R diagram, abyste ukázali, že to zvládnete a že tématu rozumíte

Prevod do relačného modelu

- Zadanie tejto otázky môžeme chápať niekoľkými spôsobmi
 - Prevod class diagramu na ERD
 - Prevod jednoduchého tabuľkového zápisu databázových štruktúr v ER model (a potom v ERD diagram)
 - Vytvoriť ERD na základe notácie (notácia zložená z grafických vyjádrení na str. 3)
 - Prevod ER modelu v ERD
 - A internet je priam presýtený **transfer ERD to relational model** (čo mi teda pripadá ako najpravdepodobnejšie, že chcú, ale mohli to napísť explicitnejšie. Hlavne preto si myslím, že je to to, lebo je to priamo prevod do relačného modelu, zatiaľ čo ostatné možnosti sú prevod do entitného relačného diagramu, čo nie je to isté)

Do entitného relačného

- to sa asi nechce, ale tak nebudem to mazať, potom zväzte, či to je potrebná informácia
- vykonáva sa z niekoľkých dôvodov⁴
 - ER model obsahuje ER diagram, ktorý je veľmi užitočný na získanie rýchleho prehľadu o modely
 - ER model je dobре rozšírený modelovací nástroj
 - Často sa využíva v neskorších fázach softwarového dizajnu pre poňatie logických a fyzických data modelov

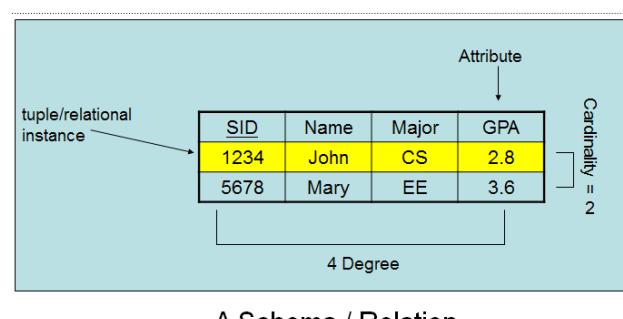
Prevod ERD do relačného modelu

Motivácia

- Relačný model sa dá priamo implementovať do databázy

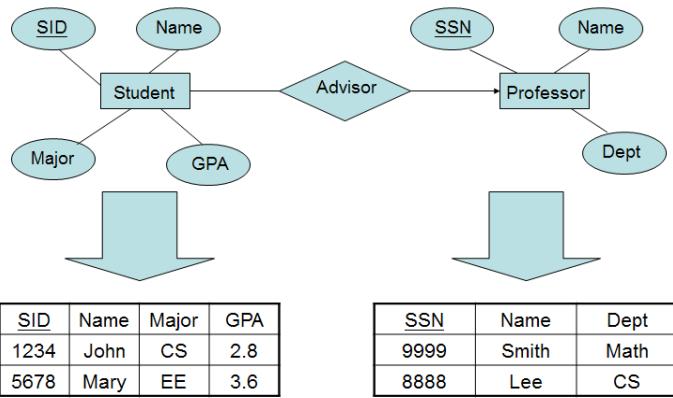
Podoba relačného modelu

- Relačný model je zložený z tabuľiek⁵
 - riadok tabuľky – relačná inštancia / n-tica
 - stĺpec tabuľky – atribút ($A_1, A_2 \dots A_n$)
 - tabuľka – relačná schéma $R = (A_1, A_2 \dots A_n)$
 - relácia – $r(R)$ – názov relačnej schémy R
 - kardinalita – počet riadkov
 - stupeň – počet stĺpcov
 - Majme množiny $A_1, A_2 \dots A_n$, relácia r je podmnožina kartézského súčinu týchto množín (teda je množinou n-tíc)



⁴ Preložené z HIT tutorialu PB114 https://is.muni.cz/auth/el/1433/jaro2013/PB114/um/HIT_tutorial_actual.pdf

⁵ Prezentácia zo zdroja <http://goo.gl/lWvqg>



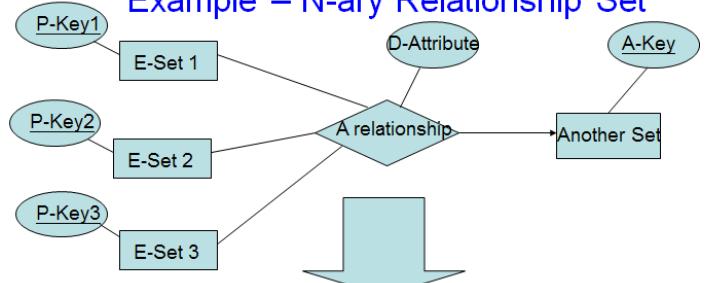
Prevod

- Pre každú entitnú množinu vytvorte schému
- Ak potrebné, vytvorte schému aj pre vzťahovú množinu (relationship set)
- Vytvorte stípec v schéme pre každý atribút z entitnej množiny
- Pravidlo nedeliteľnosti a pravidlo poradia
- Primárny kľúč
- Slabé entitné množiny sa vytvoria rovnako ako silné, ale pridá sa primárny kľúč z dominantnej entitnej množiny (už popísané vyššie)

Prevádzanie vzťahov (zjednodušené)

- Unárny / binárny vzťah
 - bez totálnej participácie
 - nová tabuľka s primárnymi kľúčmi z oboch E. množín
 - jedna množina s totálnou participáciou
 - Pridať jeden stípec na tabuľku množiny s úplnou účasťou (slabá množina). Nie je potrebná samostatná tabuľka na vyjadrenie vzťahu
- N-árny vzťah ($n > 2$)
 - počet stĺpcov je počet atribútov primárnych kľúčov E. množín zúčastnených vo vzťahu + deskriptívne atribúty (dátum a čas transakcie napr.)
 - Primárny kľúč je zjednotenie primárnych kľúčov E. množín, ktoré sú na strane „veľa“ ($1:N - N$ je veľa)
 - (obrázok vpravo)
 - A-key nie je súčasťou primárneho kľúča vzťahu, pretože ku danej entitnej množine ide šípka – vyjadrenie asociačnej entity (entita existuje iba ako výsledok vzniknutého vzťahu medzi ostatnými entitnými množinami)
- Pri slabej množine nie je potrebné vytvárať tabuľku na vyjadrenie vzťahu (identifying relationship). Redukuje sa tak redundancia

Example – N-ary Relationship Set

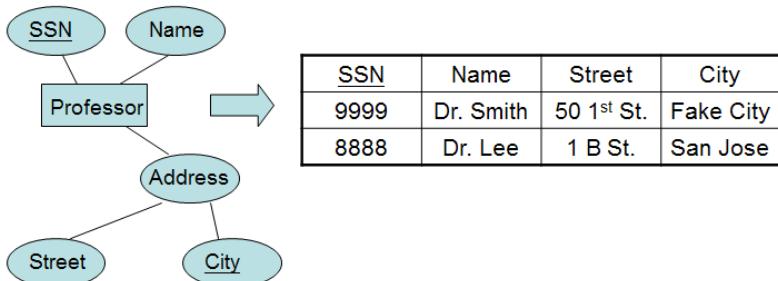


P-Key1	P-Key2	P-Key3	A-Key	D-Attribute
9999	8888	7777	6666	Yes
1234	5678	9012	3456	No

* Primary key of this table is P-Key1 + P-Key2 + P-Key3

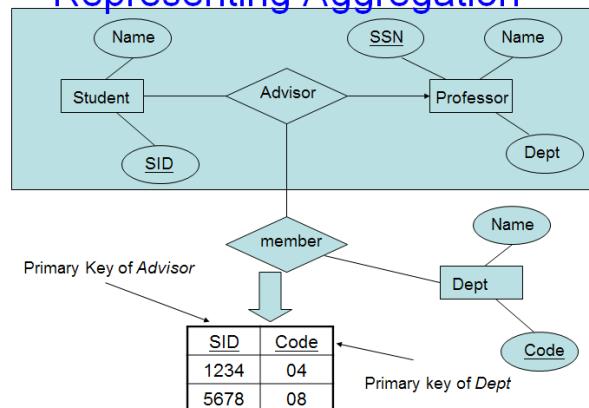
vzťahu, pretože ku danej entitnej množine ide šípka – vyjadrenie asociačnej entity (entita existuje iba ako výsledok vzniknutého vzťahu medzi ostatnými entitnými množinami)

Iné



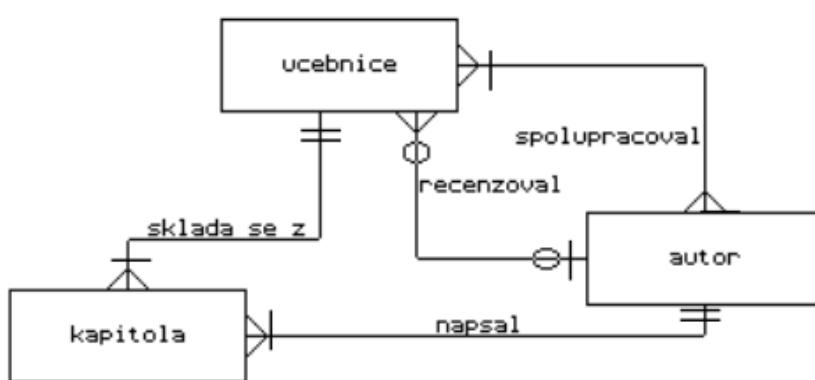
- **Zložený atribút** – nemá samostatný stĺpec v tabuľke. Iba jednotlivé komponenty zloženého atribútu (adresa: ulica, mesto – do tabuľky značíme iba ulicu a mesto).
- **Viacnásobný atribút**
 - nová relačná schéma s dvoma atribútmi:
 - **primárny kľúč** entitnej množiny s viacnásobným atribútom.
 - **jedna hodnota** viacnásobného atribútu (každý ďalší je ďalší riadok v tabuľke)
 - **primárny kľúč tejto schémy** je zjednotenie všetkých atribútov
- **Agregácia**
 - samostatná tabuľka obsahujúca primárny kľúč agregovaného vzťahu s kľúčom nového vzťahu

Representing Aggregation



ER diagram (entity, atribúty, vzťahy)⁶

- definuje nemenné atributy a strukturu dat. Datový model vyjadruje vzťahy, ktoré nejsou zachyceny v procesných modelech. Komponentami datového modelu jsou:
 - Entity (Entitní množiny)
 - Relace mezi entitami.
 - Atribúty

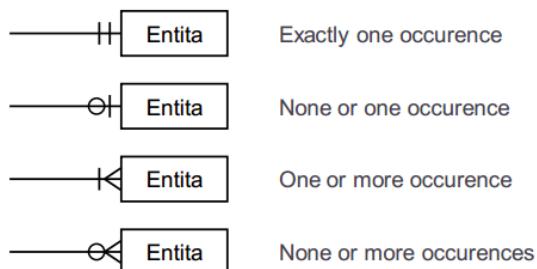


Jednoduchý príklad zo slajdov PB007 2013. Pre ilustráciu.

Z obrázku sa dá vyčítať, že **entitné množiny** sa značia ako **obdĺžnik** s názvom a tie sú prepájané **čiarami**, ktoré zobrazujú (rôzne ostupňované) **vzťahy**.

⁶ Väčšina dát prekladaná zo slajdov Softwarové Inženírství I.

Stupeň vzťahov:

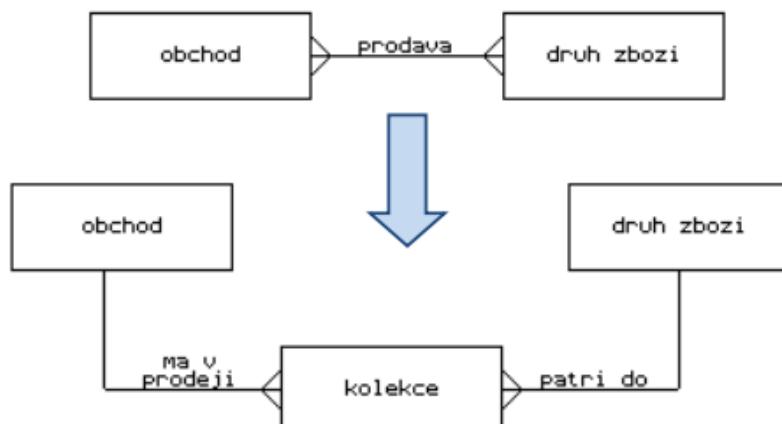


Odhora nadol:

- Jeden výskyt
- Žiadny alebo jeden výskyt
- Jeden alebo viac výskytov
- Žiadny až mnoho výskytov

Kardinalita

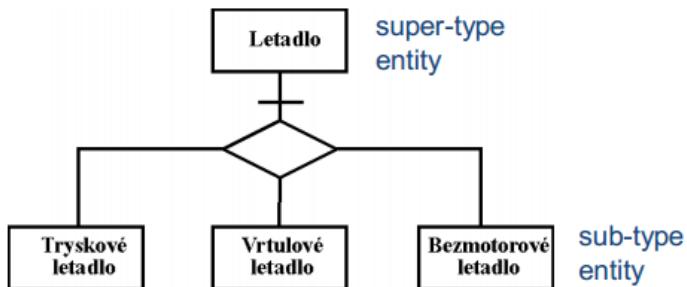
- Popisuje množstvo entít, ktoré sa môžu zúčastniť vzťahu
- Typy: 1:1, 1:N, M:N
- Pri M:N kardinalite sa vytvára stredná entita, ku ktorej majú predošlé dve entity vzťah 1:N.



Vid' príklad:

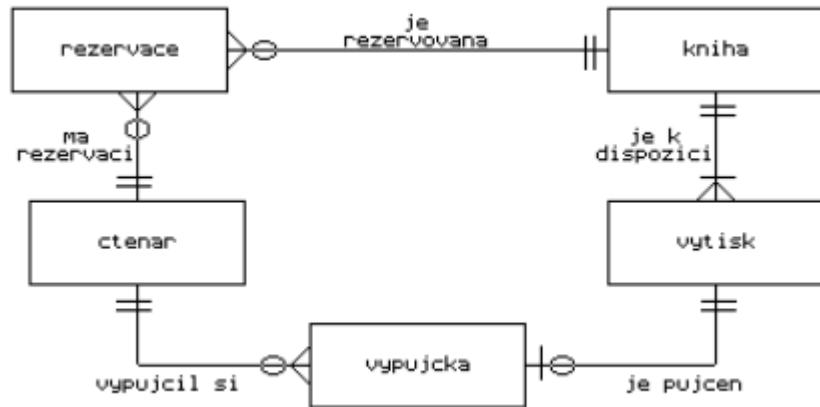
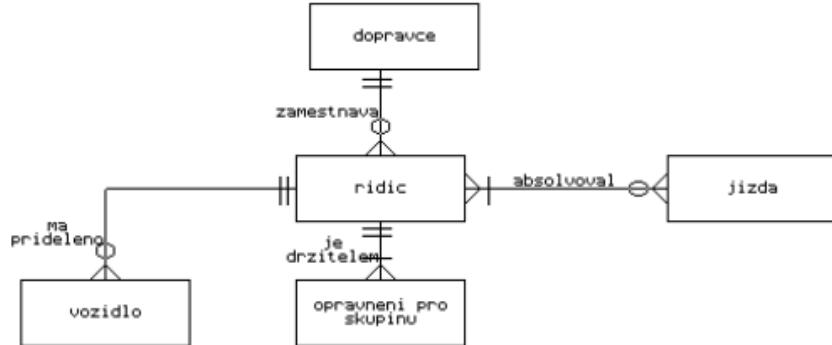
Vzťahy

- Môžu byť **viacnásobné**
 - príklad:
 - výrobok nabízí dodavatel (má atribúty platové podmienky, termíny)
 - dodavatel dodal výrobok (atribúty sú údaje z dodacieho listu)
- Typovo:
 - **Povinné** (mandatory) napr. 1:N maliar namaľoval obraz
 - **Voliteľné** (optional) napr. M:N pracovník rieši projekt
 - **Rekurzívne** napr. 1:N modul sa skladá z modulu(ov).
- V rozšírených ERD modeloch môže byť **dedičstvo** (inheritance). Entity v takomto vzťahu voláme **super-type (nadtyp)** entity a **sub-type (podtyp)** entity. Je to vzťah typu špecializácia - generalizácia



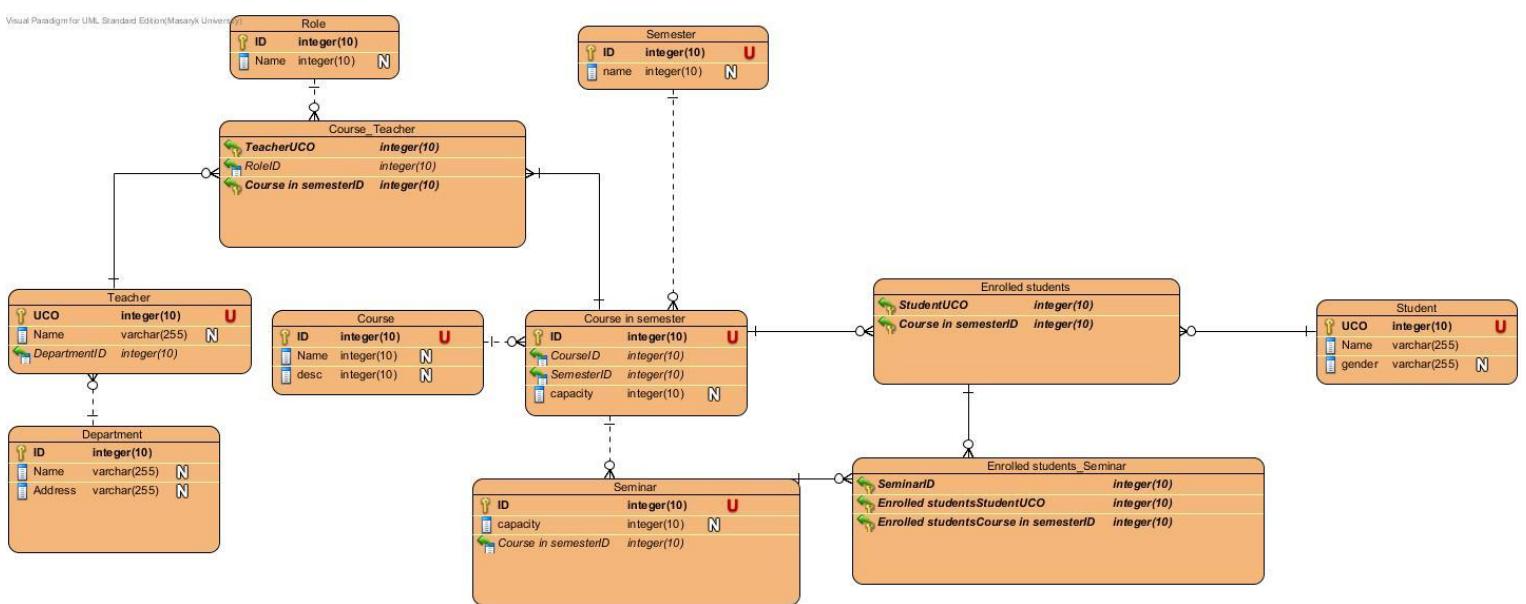
Iné

- Na slajdoch 32-43 z PB007/06 sú dobré príklady toho, ako odstraňovať nepotrebné entity, či nepotrebné vzťahy. To je už od otázky trochu bokom.
- Príklady:



- Lepší príklad ERD modelu z Visual Paradigm.

<https://is.muni.cz/auth/el/1433/podzim2012/PB007/um/35424437/35424448/ERD.jpg?studentId=638553>

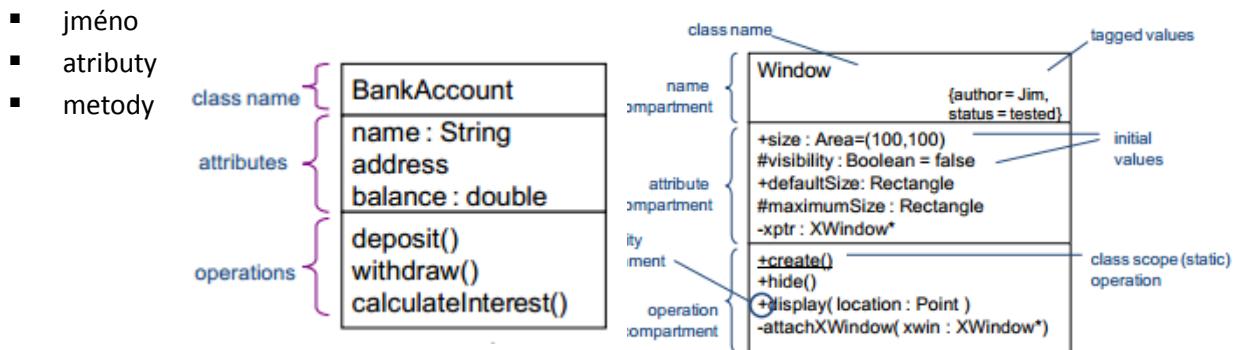


UML diagram tried

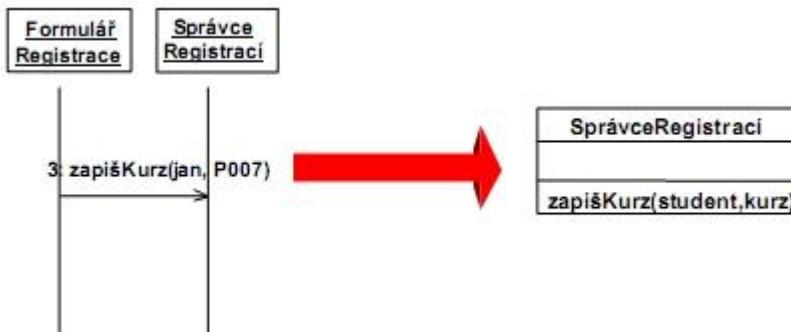
- ukazuje existenci tříd a jejich vztahů v logickém pohledu na systém.
- UML modelovací prvky používané v diagramech tříd:
- **třídy**, jejich struktura a chování
- **vztahy** - asociace, agregace, závislosti a vztahy dědění.
- **násobnost**
- **navigace**
- **mená rolí**

Třída

- skupina objektů se shodnou strukturou, shodným chováním, shodnými vztahy a shodnou sémantikou.
- Třída se znázorňuje obdélníkem, který obsahuje tři části:



- Třídy a jejich operace mohou být nalezeny přezkoumáním diagramů interakcí (posloupnos

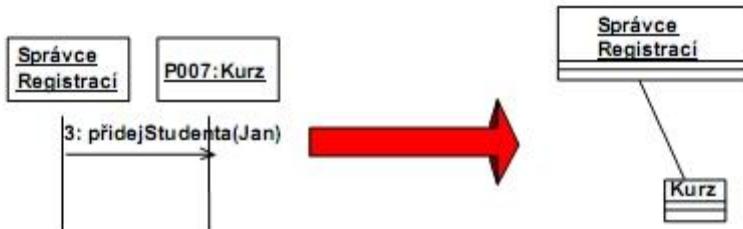


- Prichádzame na ne taktiež skúmaním špecifikácie navrhovaného systému (slovesá sú kandidátmi na zodpovednosť (operácie/metódy), podstatné mená môžu byť atribúty alebo triedy. Preskúmaním požiadavkov na problém a použitím znalostí z predmetnej oblasti môžeme prísť na ďalšie detaily. Pozor na synonymá / homonymá, „skryté“ triedy. Je vhodné zozbierať dokument požiadaviek, use cases, brainstorm. Ďalšie zdroje môžu byť fyzické objekty, formuláre, známe rozhrania s vonkajším svetom...



Vztahy mezi třídami

- Poskytují cestu pro komunikaci mezi objekty
- bývají odhaleny po přezkoumání diagramů interakcí – pokud dva objekty spolu hovoří, musí existovat komunikační cesta.

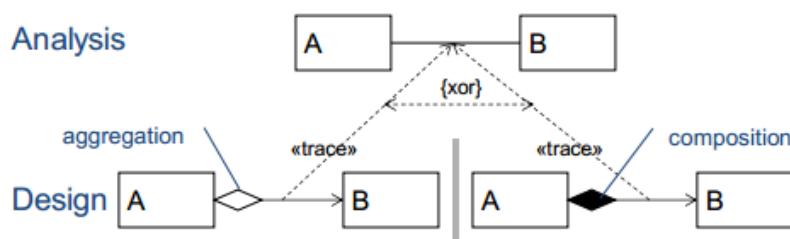


Vztahy:

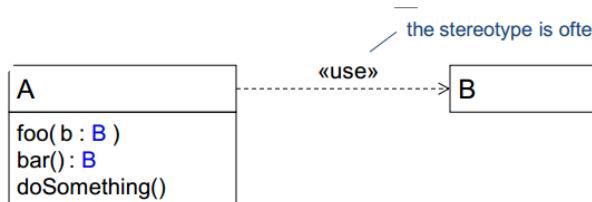
- **Asociace** – obousměrné propojení mezi třídami.
 - Znázorňuje se jako čára propojující vztahené třídy.
- **Agregace** – je silnější forma vztahu, jedná se o vztah mezi celkem a jeho částmi.
 - Znázorňuje se jako čára propojující vztahené třídy, značka diamant je umístěna u třídy, která představuje celek. Agregace se značí prázdným diamantem. Její silnejšia forma – **kompozice** (pri ktorej časť nemá zmysel bez celku (!)) – se značí vyplňeným diamantem.

Pozor:

znázornenie vzťahu môže zostať aj len ako asociace, teda nemusí to byť v každom prípade agregace, alebo kompozice.

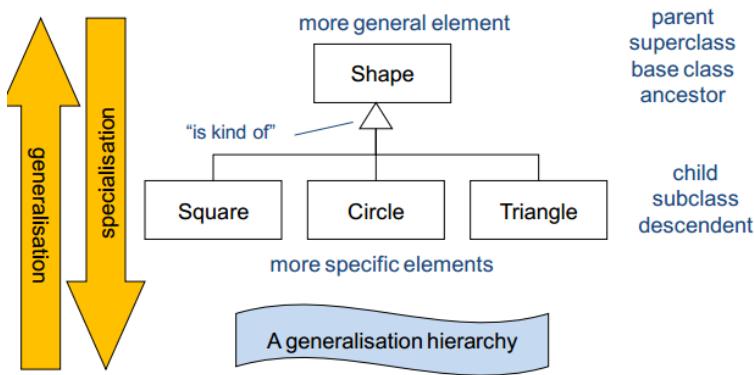


- **Závislost** – je slabší forma vztahu mezi klientem a poskytovatelem, kde klient nemá žádnou sémantickou znalost o poskytovateli. Je ukázána jako čárkovana čára se šipkou ukazující od

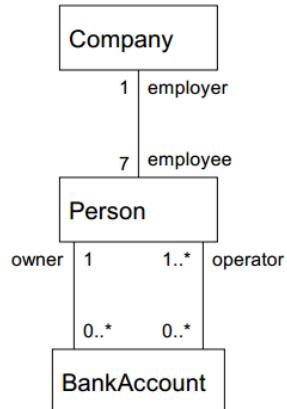


klienta k poskytovateli.

- **Dědičnost** – je vztah mezi nadtírdou a jejími podtírdami. Existují dvě cesty jak ji nalézt – zobecnění a specializace.
 - Dědičnost se znázorňuje šipkou směrem od podtírny k nadtírde. Společné atributy a vztahy jsou zobrazeny na nejvyšší úrovni hierarchie.

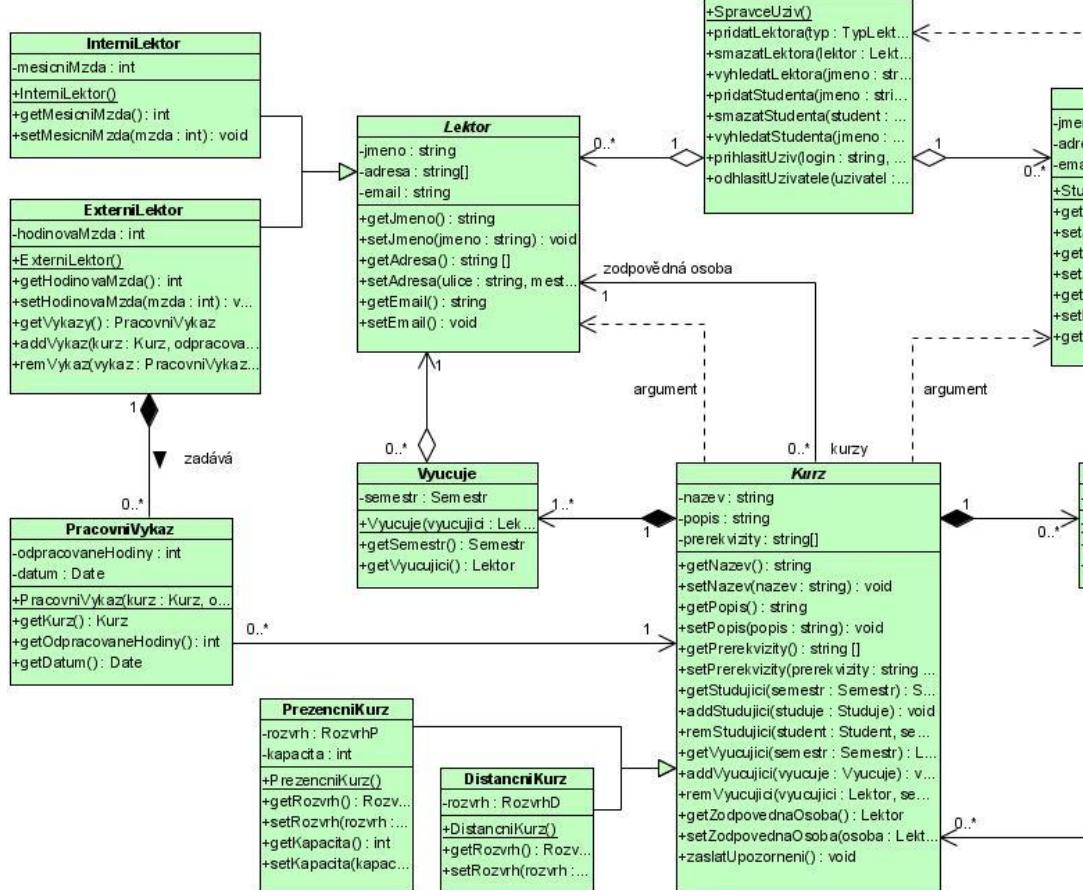


- Násobnosť** – definuje, kolik objektov se účastní vztahů. Násobnosť je počet instancí jedné třídy vztažené k jedné instancii druhé třídy. Pro každou asociaci a agregaci musí být nalezeny dvě násobnosti, pro každý konec vztahu.

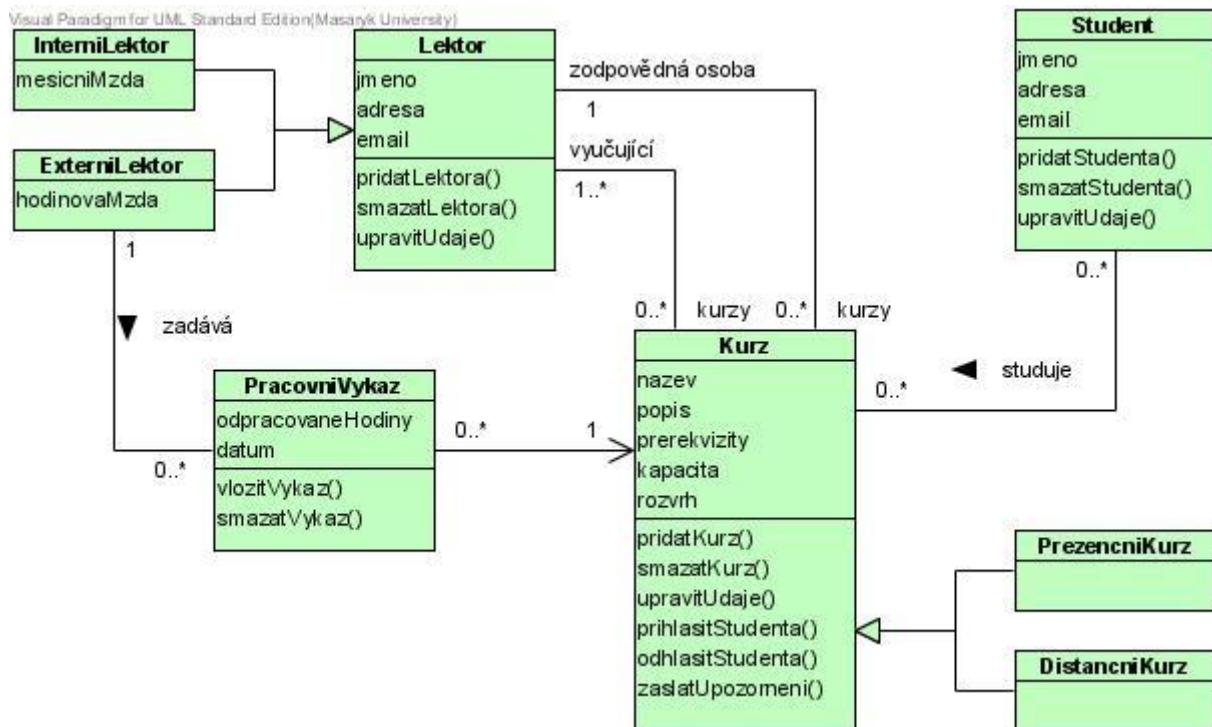


Príklady UML diagramov

Visual Paradigm for UML Standard Edition/Masaryk University



https://is.muni.cz/auth/el/1433/podzim2012/PB007/um/36675960/10_Studium_ClassNavrh.jpg?studium=638553



Zrovnanie

Class diagramy

- Modeluje štruktúru aj chovanie systému (atribúty, operácie)
- Obsahuje mnoho rôznych druhov vzťahov (asociácie, agregácie, kompozície, závislosti, generalizácie)
- Viac pripomínajú objekty z reálneho sveta

ER modely

- Modelujú iba štruktúrovaný pohľad na dátu s nízkou variabilitou vzťahov (jednoduché vzťahy či zriedkavé generalizácie)
- Viac pripomínajú databázové tabuľky (opakované záznamy)
- Umožňujú nám dizajnovať primárne a cudzie kľúče a bývajú normalizované pre uľahčenie dátovej manipulácie

Zhrnutie

- Môže sa stať, že mapovanie medzi ERD a class diagramom je 1:1, ale často je
 - jedna trieda mapovaná na viac než jednu entitu, či
 - viac tried mapované ako jedna entita
- Nie všetky triedy musia byť trvalé (zapamätané) a teda nemusia byť reflektované do ERD modelu, ktorý je hnaný databázovým dizajnovaním
- ERD je **dátovo orientované a persistent-specific** //jednoducho že dátá sú stále
- Class Diagram adresuje aj **operácie** a je **persistence independent**.