Materiály na SZZ pro Informační Bezpečnost

Matěj Douša

Červen 2024

Obsah

1 Počítače a systémy

1.1 SP-19 (PA1)

Datové typy v programovacích jazycích. Staticky a dynamicky alokované proměnné, spojové seznamy. Modulární programování, procedury a funkce, vstupní a výstupní parametry. Překladač, linker, debugger.

Datové typy

V programovacích jazycích používáme proměnné, tedy něco, co uchovává datovou hodnotu s nějakou vnitřní strukturou. Proměnné jsou identifikovány svými jmény — identifikátory. Datový typ proměnné definuje vnitřní strukturu/reprezentaci dat a jejich význam. Tím určuje jakých hodnot může proměnná nabývat a také jaké operace lze s proměnnou (její hodnotou) vykonávat.

Jednoduché datové typy:

- celočíselné
  - existují různé délky — short, int, long (byte, long long, ...)
  - signed (znaménkové) — umí uložit i záporné hodnoty, používá se doplňkový kód
  - unsigned (neznaménkové) — ukládá jen kladné hodnoty, přímý kód
- s pohyblivou řádovou čárkou
  - existují různé délky — float, double, long double
  - znaménko (1 bit) + mantisa (velikost=přesnost) + exponent (velikost=rozsah)
- znakové

  Znaky jsou kódovány jako čísla, používá se ASCII / extended ASCII / UNICODE.
- logická hodnota

  Není v C, ale často se v jazycích vyskytuje (boolean — true/false).

Další datové typy:

- ukazatel (pointer)

  Adresy paměti, kde je uložen datový typ pointeru (pointer vždy ukazuje na konkrétní typ/funkci, případně void).
- výčtový typ (enum)
- struktura

  Je složena z dalších datových typů, klidně dalších struktur.
- union

  Ukládá více různých datových typů na stejné místo.
- třída (ve vyšších jazycích)

Statická a dynamická alokace

Staticky alokované proměnné:

- vzniknou běžnou deklarací
- ukládají se na zásobník (lokální proměnné) či do části .BSS (neinicializované globální proměnné) a .DATA (inicializované globální proměnné)
- v případě pole je nutno znát v době kompilace velikost (statická velikost)

Dynamicky alokované proměnné

- vzniknou použitím speciální funkcí/operátorem
- ukládají se na haldě (heap)
- přistupujeme přes pointer
- je možné alokovat paměť podle hodnot spočítaných za běhu programu

Spojové seznamy

- oproti poli nejsou položky seřazeny v paměti, ale každý prvek seznamu obsahuje ukazatel na další prvek.
- podobně jako v dynamicky alokovaném poli lze ukládat předem neznámý objem dat
- nelze jednoduše indexovat, ale lze libovolně přidávat či ubírat prvky z jakékoliv pozice v seznamu

Modulární programování

- složitější programy mohou být rozděleny do modulů
- tyto moduly lze použít v různých dalších částech programu
- modul má svou specifikační část (deklarace poskytovaných prostředků/rozhraní) a implementační část (definice/implementace poskytovaných prostředků)
- v C/C++ typicky hlavičkový soubor (.h/.hpp) a implementační soubor (.c/.cpp)

Procedury, funkce a parametry

- procedura/funkce je posloupnost příkazů uložených v paměti programu
  - procedura — bez návratové hodnoty (typ void)
  - funkce — s návratovou hodnotou
- použijeme ji zavoláním přes její jméno
- deklarace je specifikace jejího rozhraní — parametrů a typu návratové hodnoty
- definice je samotný kód funkce
- vstupní paramtery jsou informace, které využije kód funkce
- výstupní parametry jsou výsledkem běhu funkce — typicky se nějak změní a tím nám dají výsledek

Překladač

- překládá vyšší programovací jazyky do nižších
- ze zdrojového kódu vzniká objektový soubor — modul se strojovým kódem
- front-end přeloží konkrétní jazyk do vnitřní reprezentace (abstrakce nezávislá ani na platformě ani na jazyku)
- back-end přeloží vnitřní reprezentaci do strojového kódu konkrétní platformy

Linker

- spojuje přeložené moduly do výsledného celku — programu
- výstupem je spustitelný soubor

Debugger

- usnadňuje hledání chyb v kódu, také usnadňuje pochopení programu
- je vhodné kompilovat s informacemi pro ladění
- je možné si na nějakém místě běh programu zastavit a např. sledovat obsah proměnných, pouštět každý krok programu postupně...

1.2 SP-30 (SAP)

Kódy pro zobrazení čísel se znaménkem a realizace aritmetických operací (paralelní sčítačka/odčítačka, realizace aritmetických posuvů, dekodér, multiplexor, čítač). Reprezentace čísel v pohyblivé řádové čárce.

Čísla se znaménkem

Existuje několik možností, jak v počítači ukládat celá čísla:

- Přímý kód
  - první bit je znaménkový — určuje tedy, zda je hodnota za ním kladná či záporná
  - ostatní bity představují absolutní hodnotu čísla
  - existuj zde kladná i záporná nula
- Doplňkový kód
  - dle prvního bitu lze poznat znaménko čísla
  - převod kladné $\leftrightarrow$ záporné lze vysvětlit jako inverze bitů a následné přičtení jedničky
  - 0111 (7) $\rightarrow$ 1001 (-7)
  - není zde záporná nula
- Aditivní kód
  - uložené číslo je posunuto o nějakou konstantu, typicky polovina rozsahu
  - pro 4 bity určeme nulu jako 1000 — pak 1111 je 7, 0000 je -8
  - nula není zobrazena jako nula
  - není zde záporná nula

Čísla v pohyblivé řádové čárce

Reprezentace pohyblivé řádové čárky vychází ze zobrazení $A = M * z^e$ používaném např. ve fyzice, kde z je základ soustavy (zde 2), e je exponent jako celé číslo, M je mantisa.

- používá se normalizovaný tvar, tedy mantisa je zapsána tak, že ji nelze "posunout"více doleva.
- v přímém kódu mantisy je vlevo vždy jednička, která se skrývá (zvýšení přesnosti)
- pro mantisu se typicky používá přímý kód, pro exponent aditivní
- float (32b) typicky vypadá jako 1b znaménko, pak 8b exponent a nakonec 23b mantisa (tedy přesnost 24b)

Realizace aritmetických operací

- Paralelní sčítačka

  Tvořena více jednobitovými sčítačkami. Jednobitová sčítačka má 3 vstupy: A, B (sčítané bity) a vstupní přenos (carry — např. ze sčítačky nižšího řádu). Výstupy jsou S (výsledek) a výstupní přenos.
- Aritmetické posuvy

  Posun čísla vlevo/vpravo. Realizuje posuvný registr. Existuje více různých posuvů:
  - logický posuv — doplňuje nuly
  - cyklický posuv — doplňuje co vylezlo na druhé straně
  - aritmetický posuv — doplňuje 1 nebo 0 podle znaménka čísla
- Dekodér

  Kombinační logický obvod, který má méně bitů na vstupu než na výstupu, a podle tabulky převádí. Kodér má opačnou funkci.
- Multiplexor

  Na základě řídícího signálu vybere, který ze vstupů pošle na výstup (má několik vstupů + řídící vstup, a jeden výstup).
- Čítač

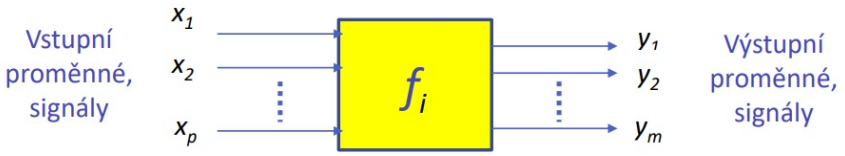
  Registr s funkcí inkrementu/dekrementu, může čítat nahoru a/nebo dolů. Existují úplné čítače (do mocnin 2) či neúplné (do jiných čísel). Typicky čítají v binárním kódu, lze i např. v Grayově kódu.

1.3 SP-28 (SAP)

Kombinační a sekvenční logické obvody (Mealy, Moore), popis a možnosti implementace na úrovni hradel. Minimalizace vyjádření logické funkce s využitím map.

Kombinační obvody

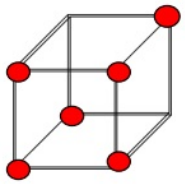
- popsány kombinační funkcí
- hodnoty všech výstupů (výstupních proměnných) jsou v každém časovém okamžiku určeny pouze vstupem (hodnotami vstupních proměnných) ve stejném okamžiku
- mohou být popsány např. Booleovskou (logickou) formulí
  Příklad: $f = x_1.\overline{x_2} + \overline{x_1}.x_2 = (x_1 + x_2).(\overline{x_1} + \overline{x_2})$
- obecně kombinační obvod vypadá následovně:



  Logická funkce $y_k = f(x_1, x_2, x_3, ..., x_p)$ existuje pro každý výstup y.
- možnosti reprezentace logických funkcí:
  - tabulka

ab	f
00	0
01	1
10	1
11	0

  - n-rozměrná krychle



  - Booleovský výraz
    Viz výše
  - mapa (Karnaughova)

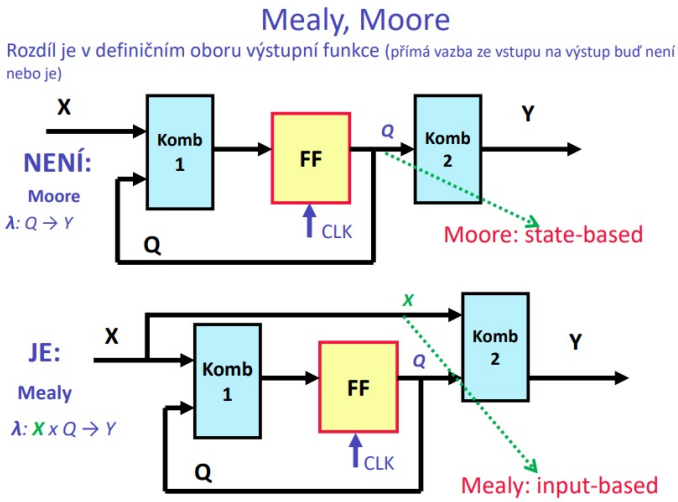
	a		b	
		X	1	
c		1	1	X

- možnosti realizace obvodů:
  - na úrovni hradel
  - mapování na technologii (FPGA, ASIC)
  - popis v jazyku (VHDL, Verilog)

Sekvenční obvody

- výstup závisí na posloupnosti/sekvenci hodnot na vstupu
- zapamatování se realizuje zpětnou vazbou
- popsány konečným stavovým automatem

- typy sekvenčních obvodů:
  - Moore
    Obvod, jehož výstup závisí pouze na vnitřním stavu.
  - Mealy
    Obvod, jehož výstup závisí také na aktuálním vstupu (kromě stavu).

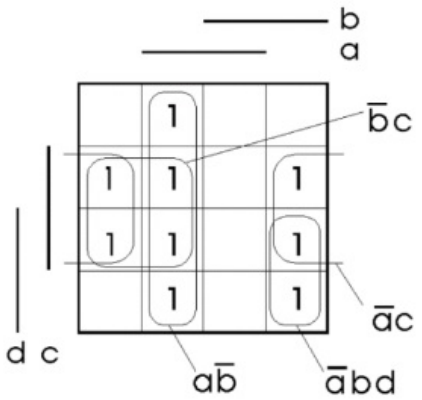


Implementace na úrovni hradel

- nejprve minimalizace logické funkce a zápis výsledku např. v Booleovském výrazu
- následně nakreslení/vytvoření funkce pomocí základních hradel — NOT, AND, OR, NAND, NOR, XOR

Minimalizace logické funkce

- smysl — zjednodušit a zkrátit zápis, snížit potřebný materiál pro výrobu
- minimalizace pomocí map — založenana hledání co největších skupin sousedních stavů.
- postup při minimalizaci:
  - vytvoření Karnaughovy mapy pro funkci
  - nalezení všech přímých implikantů (maximální skupiny jedniček či "dont care")
  - určení všech podstatných implikantů (obsahující jedničku, kterou jiný implikant neobsahuje)
  - pokud nejsou pokryty všechny vrcholy s "1", nutno vybrat další přímé implikanty (takové, kde je nejméně negací)



$a\bar{b} + \bar{b}c + \bar{a}c + \bar{a}bd$

- lze také kroužkovat nuly — pak je ale nutné funkci sestavit jinak.
  $(\bar{a} + \bar{b})(a + c + d)(a + b + c)$

1.4 SP-29 (SAP)

Architektura číslicového počítače, instrukční cyklus počítače, základní třídy souborů instrukcí (ISA). Paměťový subsystém počítače, paměťová hierarchie, skrytá paměť (cache).

Architektura číslicového počítače

- architektura se zabývá strukturou a chováním počítače
- řeší specifikaci různých funkčních modulů jako procesor a paměť a nebo např. instrukční sadu
- podsměry:
  - ISA — Instruction Set Architecture — architektura souboru instrukcí
  - Mikroarchitektura — konkrétní sestavení a složení procesoru
  - Systémový design — řeší další HW komponenty
- každý počítač je složen z následujících částí:
  - datová část procesoru — ALU, registry
  - řadič — řídící jednotka procesoru
  - paměťový subsystém
  - vstupní zařízení
  - výstupní zařízení
- typy architektury:
  - Von Neumannova architektura
    Data i instrukce jsou uložena spolu, nejsou explicitně označena/ny.
  - Harvardská architektura
    Data a instrukce jsou rozdělené.

Instrukční cyklus počítače

- čtení instrukce (IF — Instruction Fetch)
- dekódování instrukce (ID — Instruction Decode)
- načtení operandů (OF — Operand Fetch)
- provedení instrukce (IE — Instruction Execution)
- zapsání/uložení výsledku (WB — Write Back / Result Store)
- přerušení?

Co je instrukce? Obsahuje informace:

- co se má provést
- s čím se to má provést (operandy)
- kam se má uložit výsledek
- kde se má pokračovat

Tyto informace mohou být zadány explicitně, nebo mohou být dány typem instrukce, tedy architekturou počítače — tedy implicitně.

ISA — Architektura souboru instrukcí Co je potřeba určit:

- typy a formáty instrukcí, instrukční soubor
- datové typy, kódování a reprezentace, způsob uložení dat v paměti
- módy adresování paměti a přístup do paměti dat a instrukcí
- mimořádné stavy

Výhody:

- abstrakce — možnost různě implementovat stejnou architekturu instrukcí
- definice rozhraní mezi nízkoúrovňovým AW a HW
- standardizuje instrukce, bitové vzory strojového jazyka

Třídy souborů instrukcí (ISA)

- Střadačově (akumulátorově) orientovaná ISA

  Akumulátor je registr pro mezivýpočty, používá se implicitně jako zdroj pro výpočty i jako cíl pro výsledky. Používají se instrukce s jedním operandem. Nejstarší ISA (1949-60) — vyvinula se z kalkulaček.

  Výhody:

  - jednoduchý HW
  - minimální vnitřní stav procesoru — rychlé přepínání kontextu
  - krátké instrukce
  - jednoduché dekódování instrukcí

  Nevýhody:

  - častá komunikace s pamětí
  - omezený paralelismus mezi instrukcemi

  Populární v 50. — 70. letech, HW byl drahý, paměť byla rychlejší než CPU.

- Zásobníkově orientovaná ISA

  Pracovní registry jsou uspořádány do struktury zásobníku. Přistupuje se k vrcholu tohoto zásobníku. Využití pro vyhodnocení výrazů a vnořená volání podprogramů. Většina instrukcí nemá operand (použije se implicitně např. vrchní 2 registry zásobníku).

  Výhody:

  - jednoduchá a efektivní adresace operandů
  - krátké instrukce
  - krátké programy
  - jednoduché dekódování instrukcí
  - snadno lze napsat neoptimalizující překladač

  Nevýhody:

  - nelze náhodně přistupovat k lokálním datům
  - omezený paralelismus — zásobník je sekvenční
  - přístupy do paměti je těžké minimalizovat

- ISA orientovaná na registry pro všeobecné použití

  Dnes převládá. GPR — General Purpose Registers. Typicky 2 nebo 3 operandy.

  Výhody:

  - registry (a cache) jsou rychlejší než paměť
  - k registrům lze přistupovat náhodně
  - registry mohou obsahovat mezivýsledky a lokální proměnné
  - méně častý přístup do paměti

  Nevýhody:

  - složitější překladač (optimalizace pro použití registrů)
  - přepnutí kontextu trvá déle

Paměťový subsystém počítače

- cache (skrytá paměť) — rychlá, drahá, umístěna blíž k procesoru
- hlavní paměť — pomalejší, levnější, větší
- vnější paměť — pomalá, velká
- záložní paměť (CD, DVD, flash, magnetické pásky)
- RAM — random access memory (přístup adresou)
- CAM — content adressable memory (přístup klíčem)

Paměťová hierarchie

- registry
- L1 cache (SRAM)
- L2 cache (SRAM)
- L3 cache (SRAM)
- hlavní paměť (DRAM)
- HDD, SSD
- Mass storage (optical disks, tapes)
- Remote storage (cloud)

Cache

Kopie často používaných dat z hlavní paměti

- časová lokalita
  Data, ke kterým bylo právě přistupováno, budou pravděpodobně brzy potřeba znovu.
- prostorová lokalita
  Po přístupu k nějakým datům se pravděpodobně budou používat i vedlejší data.

1.5 OB-4 (APS)

Instrukční cyklus počítače a zřetězené zpracování instrukcí. Mikroarchitektura skalárního procesoru se zřetězeným zpracováním instrukcí, datové a řídicí hazardy při zřetězeném zpracování instrukcí a způsoby jejich ošetření.

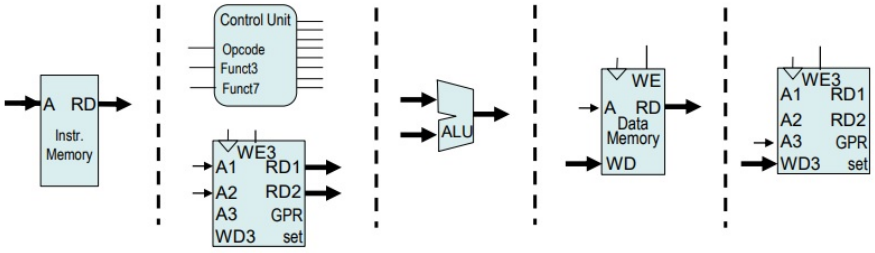
Definice 1: ISA (Instruction Set Architecture) je abstraktní rozhraní mezi HW a nízkoúrovňovým SW, které zahrnuje vše nezbytné pro psaní korektních programů ve strojovém jazyce. Zahrnuje instrukční sadu, registry, organizaci paměti, vstupy a výstupy,...

Definice 2: ISA je kompletní instrukční sada procesoru, včetně adresních módů.

Mikroarchitektura: Mikroarchitektura je detailní interní organizace procesoru, včetně hlavních funkčních jednotek, jejich propojení a řízení.

Instrukční cyklus počítače

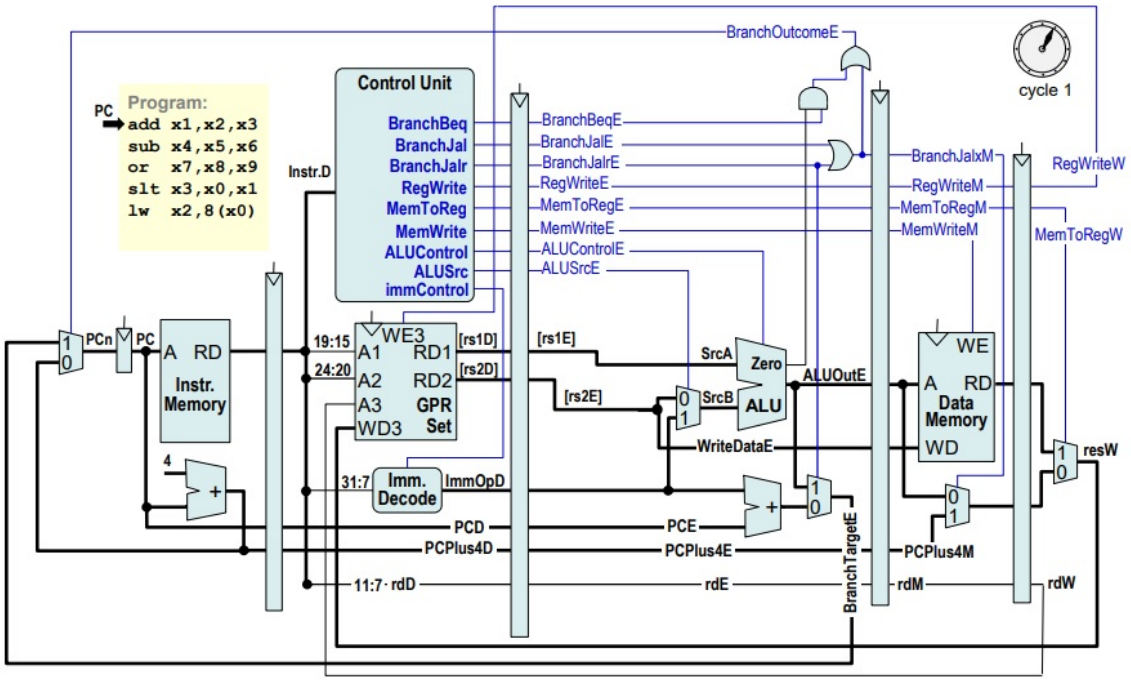
- IF — Instruction Fetch
- ID/OF — Instruction Decode and Operand Fetch
- EX — Execute
- MEM — Memory access
- WB — Write Back



Zřetězené zpracování instrukcí

Instrukce se dle svých fází rozdělí a v procesoru vykonává postupně. Procesor je rozdělen dělícími registry. Instrukce se zpracovává postupně v rozdělených částech procesoru, vykonává se zároveň více instrukcí naráz (v různých fázích).

Mikroarchitektura skalárního procesoru se zřetězeným zpracováním instrukcí



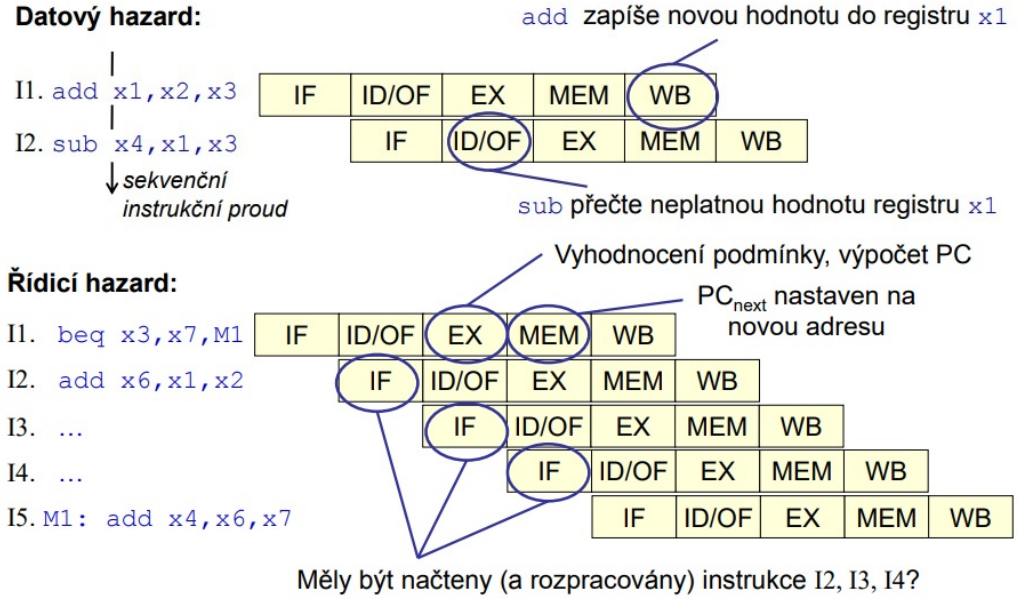
Hazardy

Protože je rozpracováno více instrukcí najednou, mohou vznikat konflikty při přístupu ke sdíleným prostředkům počítače. Tomu se říká hazardy. Sdíleným prostředkem je prostředek, který je opakovaně použit v různých stupních instrukčního zřetězení.

Typy hazardů:

- datové (důsledek datových závislostí: RAW, WAR, WAW)
- řídící (instrukce měnící PC, tedy obsah fronty instrukcí)
- strukturální (počet současných požadavků na daný prostředek převyšuje počet jeho instancí)

Hazardy mohou způsobovat pozastavení instrukčního zřetězení (stall) nebo vyprázdnění (flush).



Možnosti řešení hazardů:

- přeposílání (forwarding)

  Lze použít v případě datového hazardu, kdy výsledek předchozí instrukce požadovaný instrukcí následující vznikne dříve nebo ve stejném cyklu, kdy má být následující instrukcí použit. Výsledek je přeposlán tam, kde je potřeba (část EX).

- pozastavení (stall)

  Lze použít v případě datového hazardu, kdy je výsledek předchozí instrukce potřeba před jeho vznikem. Zpracovávání následujících instrukcí se pozastaví a vyplní se instrukce nop (bublina). V dalších instrukcích se pokračuje, když výsledek existuje — lze jej tedy přeposlat.

  Také může řešit strukturální hazardy.

- vyprázdnění části pipeline (flush)

  Lze použít v řídících hazardech, kdy se PC nastaví na neočekávanou adresu nějakou skokovou instrukcí. Všechny již načtené a zpracovávané instrukce, které následují po skoku, se musí z fronty odstranit, a následuje zpracovávání instrukcí z chtěné adresy (kam skok skočil).

Hazardy řeší nová jednotka v procesoru — HMU (Hazard Management Unit).

1.6 OB-5 (APS)

Paměťová hierarchie se skrytou pamětí (cache memory), principy lokality a fungování skryté paměti. Architektura přímé, částečně asociativní, plně asociativní skryté paměti.

Paměťová hierarchie

Rozdíl mezi rychlostí procesoru a rychlostí odpovědi paměti je velký (procesor vs DRAM — 100x, procesor vs HDD — 10 milionkrát). Tento rozdíl se překlene paměťovou hierarchií.

- L1 cache
  - SRAM
  - nejmenší, nejblíže jádru, díky tomu nejrychlejší
  - velikost v řádu jednotek či desítek KB
  - pro každé jádro zvlášť, bývá rozdělena na instrukční a datovou cache
  - obsahuje právě nejpoužívanější data a instrukce
- L2 cache
  - SRAM
  - větší, blízko jádra, latence větší než L1
  - velikost v řádu stovek KB
  - pro každé jádro zvlášť, společná pro instrukce a data
  - obsahuje vše co je v L1 + druhá nejpoužívanější data a instrukce
- L3 cache
  - SRAM
  - ještě větší, stále poměrně blízko jádrům, latence větší než L2
  - velikost v řádu jednotek MB
  - typicky sdílena více jádry, společná pro instrukce i data
  - obsahuje vše co je v L2 + třetí nejpoužívanější data a instrukce
- Hlavní paměť
  - DRAM
  - velká, mimo procesor, tedy latence zásadně vyšší než u cache
  - velikost typicky v řádu jednotek či desítek GB, může být i větší/menší
  - společná pro celý procesor(y), obsahuje data i instrukce
  - obsahuje vše co je v L3 + naprostou většinu potřebných dat i instrukcí
- Sekundární paměť
  - HDD, SSD
  - největší, nejpomalejší
  - společná pro celý počítač, obsahuje vše

Principy lokality

Programy typicky přistupují v daném okamžiku jen k malé části instrukčního a datového adresního prostoru.

Časová lokalita:

- položky, ke kterým se přistupovalo nedávno, budou brzy zapotřebí znovu
- příklad: opakované procházení dat v cyklu, opakované čtení instrukcí v rekurzivních algoritmech

Prostorová lokalita:

- položky poblíž právě používaných budou brzy zapotřebí také
- příklad: sekvenční přístup k instrukcím programu, sekvenční přístup k datovým polím nebo lokálním proměnným umístěným poblíž sebe

Principy fungování skryté paměti

- Cache block

  Souvislý, nedělitelný úsek hlavní paměti, který lze přenést do cache během jedné paměťové transakce.

- Cache hit

  Úspěšný přístup ke cache block — tedy úspěšný přístup k datům/instrukcím, které jsou obsaženy v cache a jsou platné.

- Cache miss

  Opak cache hit — neúspěšný přístup k datům/instrukcím v cache.

- Hit rate

  Úspěšnost přístupů do cache (poměr).

- Miss rate

  Neúspěšnost přístupů do cache (poměr).

- Hit time

  Latence cache, čas na získání dat z dané úrovně cache.

- Miss penalty

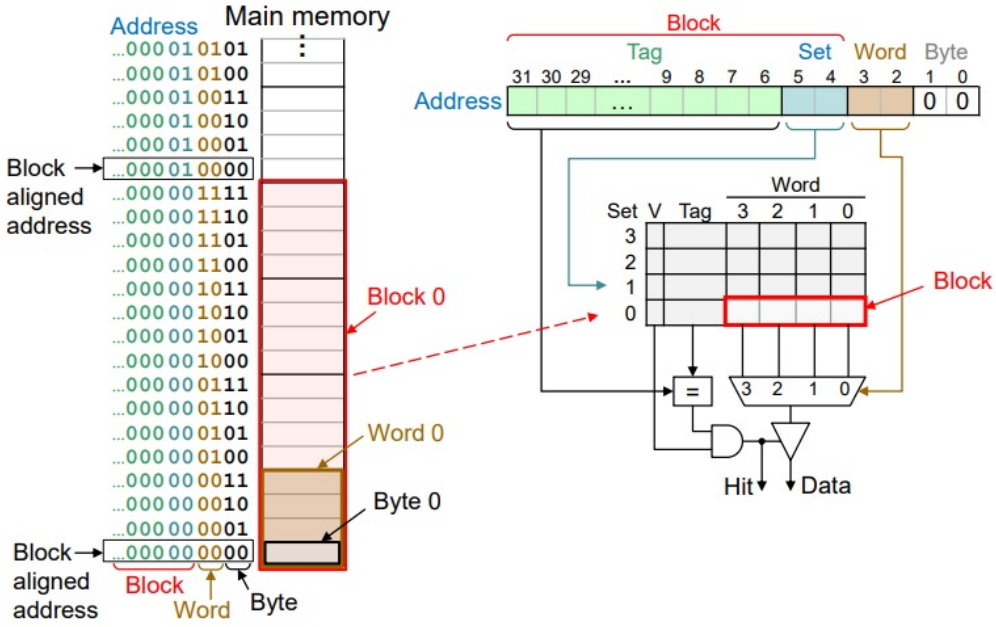
  Celkový čas pro získání dat při výpadku (cache miss) dané úrovně (počítá se od dané úrovně dál).

Průměrný čas přístupu do paměti: Hit time + (Miss rate * Miss penalty)

Adresace skryté paměti

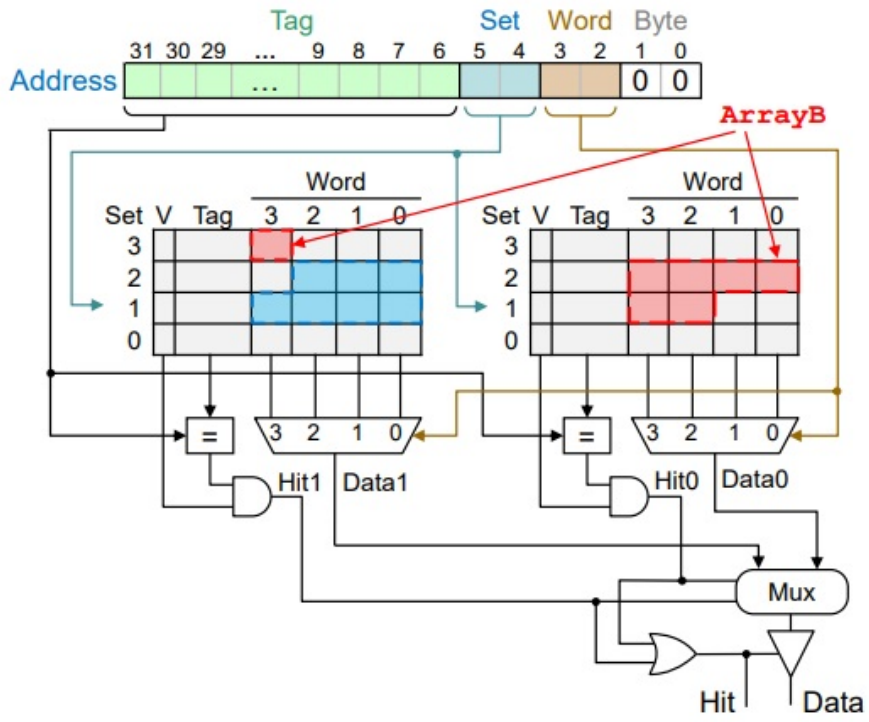
Kusy hlavní paměti se do cache ukládají po blocích. Blok může být různě veliký, typicky obsahuje více bytů (nejmenších adreovatelných kusů paměti).

- Přímo mapovaná cache
  - každý set (řádek) cache obsahuje právě 1 blok.
  - za sebou jdoucí bloky hlavní paměti se mapují do za sebou jdoucích bloků cache
  - řádek = (Adresa v hlavní paměti / velikost bloku) % počet řádků
  - každý blok hlavní paměti má tedy vždy stejný blok cache kam se uloží.
  - na stejný blok cache lze uložit více různých bloků paměti, musíme tedy uložit navíc číslo bloku hlavní paměti (část původní adresy) = Tag



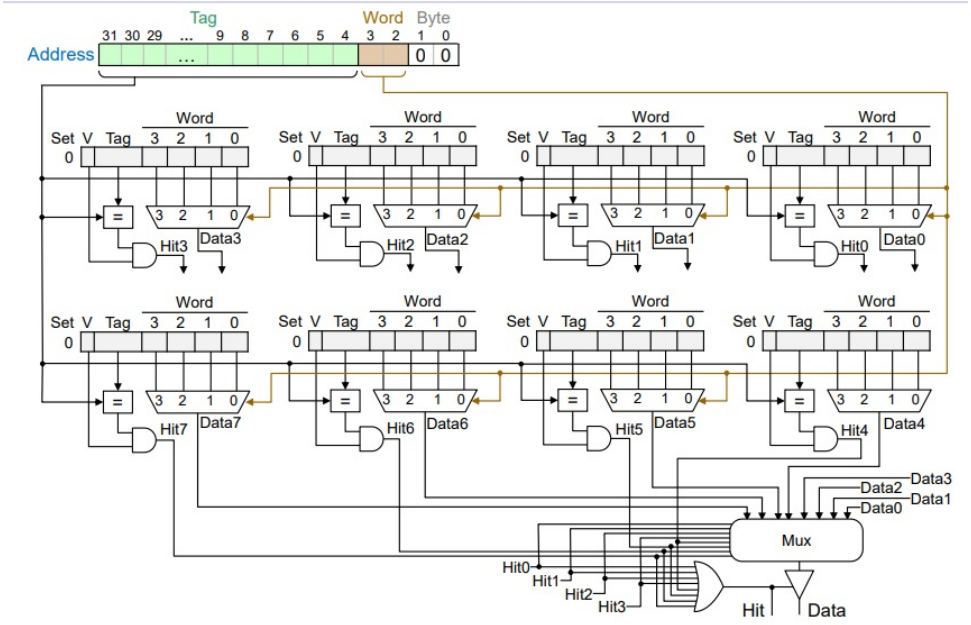
V takovéto cache ale často dochází ke kolizím, když chceme např. přistupovat do více různých částí hlavní paměti. Je ale snazší ji implementovat.

- Cache s částečným stupněm asociativity
  - replikace instancí přímo mapované cache
  - stupeň asociativity je počet takových instancí, neboli také počet cest v cache
  - bloky hlavní paměti lze uložit na více různých míst (přesně na tolik, kolik je stupeň asociativity)



Vyšší složitost implementace, ale zásadně nižší miss rate.

- Plně asociativní cache
  - počet cest je roven počtu bloků cache
  - instance přímo mapované cache mají tedy jen jeden řádek (set)
  - u každého bloku se ukládá celá jeho adresa v hlavní paměti



Nejnáročnější na HW prostředky, ale musí řešit maximální počet kolizí.

1.7 OB-6 (APS)

HW podpora virtualizace hlavní paměti stránkováním, funkce MMU (Memory Management Unit) a překlad virtuálních adres na fyzické adres pomocí TLB (Translation Lookaside Buffer), ošetření výpadku stránky.

Problémy bez virtualizace (motivace pro virtuální paměť):

- Velikost paměti: paměť nemusí být dost velká pro spuštění procesu.
- Fragmetace: Při ukončování procesů vzniknou v hlavní paměti volá místa různých velikostí.
- Dynamická alokace: Jak alokovat další paměť?
- Bezpečnost: Jak zařídit, aby si procesy nemohly číst paměť navzájem?

Jak funguje stránkování?

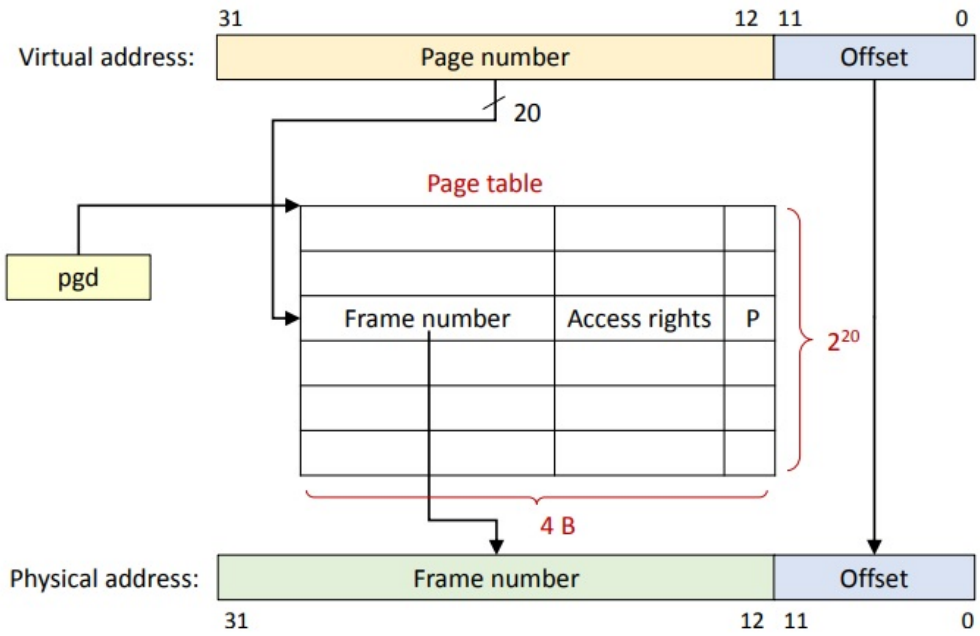
- uživatelským procesům je nabídnut Virtuální Adresní Prostor (VAP)
- každý proces má svůj VAP
- VAP je rozdělen na stejně velké *stránky*, hlavní paměť (HP) je rozdělena nastejně velké *rámce*.
- běžící procesy do rámců HP umisťují momentálně potřebné stránky svého VAP (pracovní množina)
- nepoužívané stránky se při nedostatku paměti odloží na disk
- mapování VAP do HP a přenos stránek mezi HP a diskem zajišťuje OS
- virtuální adresa se skládá z čísla stránky a offsetu
- fyzická adresa se skládá z čísla rámce a offsetu

Možnosti překladu virtuálních adres na fyzické:

- Konvenční stránkovací tabulka
  Každý proces má svou stránkovací tabulku (většinou strom stránkovacích tabulek).
- Inverzní stránkovací tabulka
  Všechny procesy sdílejí jedinou, inverzní stránkovací tabulku.

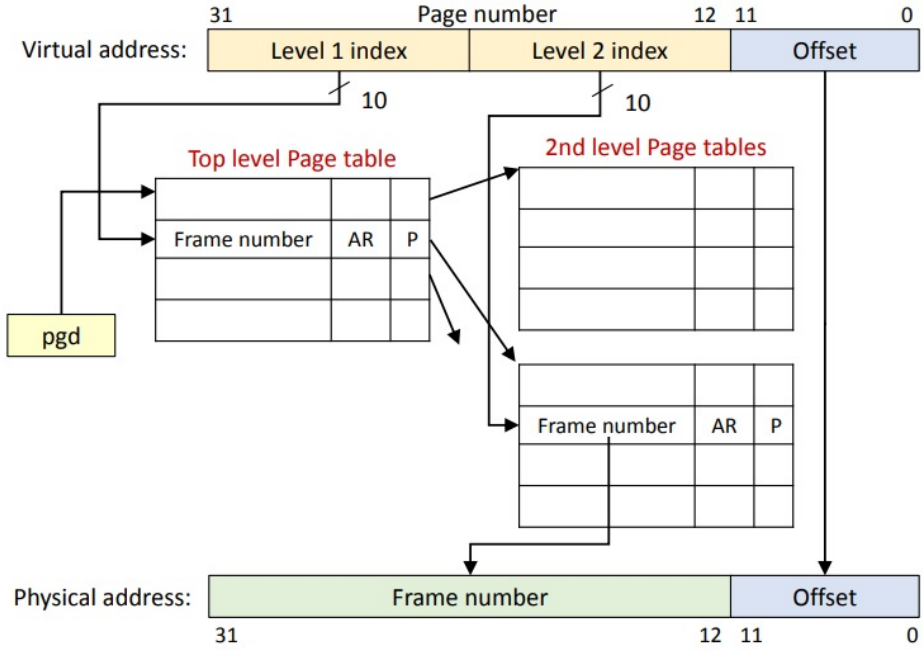
Jednoúrovňová stránkovací tabulka:

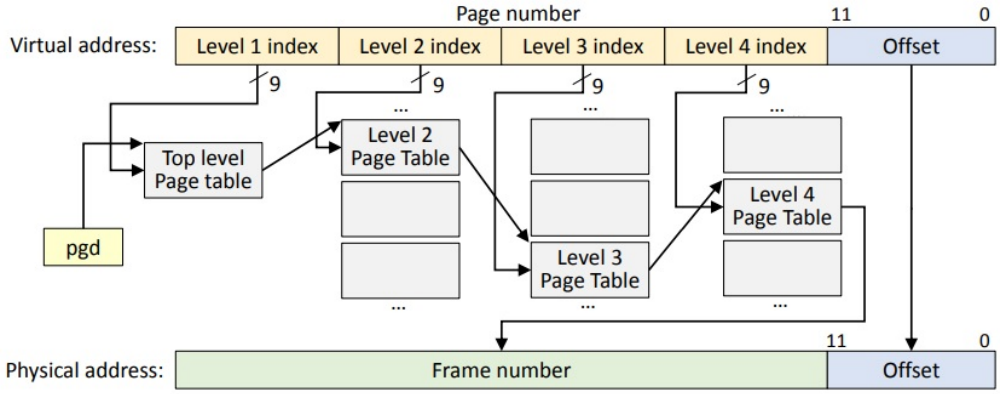
- pro překlad se použije číslo stránky jako index do tabulky, a tam se najde číslo rámce
- jednoúrovňová tabulka je ale i pro 32bitové systémy velká, a když ji má každý proces, zabere to hodně místa



Víceúrovňové stránkovací tabulky:

- pro překlad se použije číslo stránky, které je složené z indexů do různých úrovní stránkovacích tabulek
- jednotlivé tabulky jsou zásadně menší, na počátku stačí jedna tabulka v každé úrovni, další OS může podle potřeby přidat





Při překladu VA na FA se musí procházet několik úrovní tabulek stránek (page walk). Ty mohou být uloženy mimo paměť a může dojít k výpadku stránky (page fault).

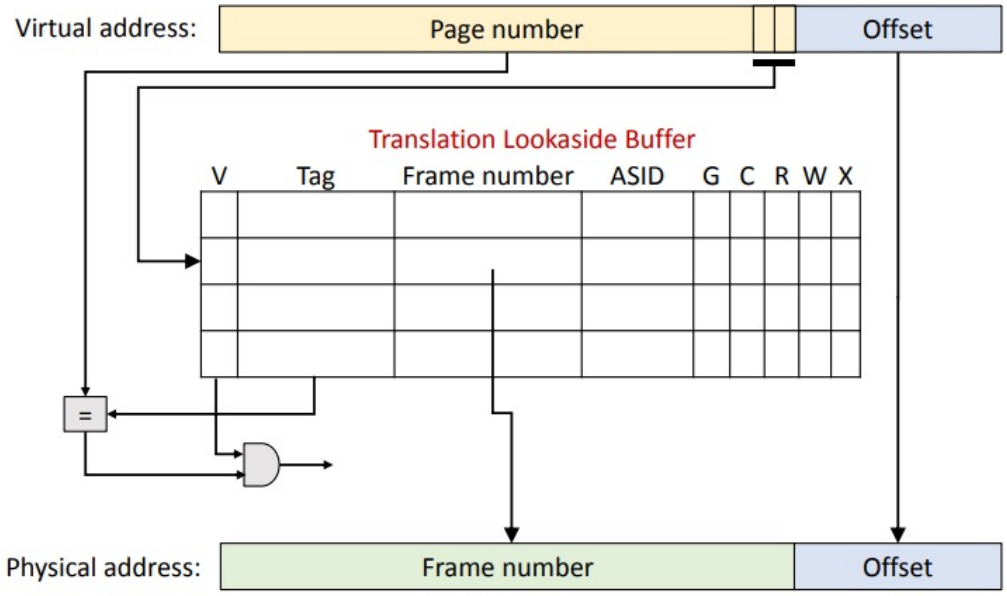
MMU — Memory Management Unit

- vykonává page walk
- dostane adresu stránkovací tabulky první úrovně přes domluvený registr
- pokud nedokáže přeložit adresu, nastává page fault, a procesor generuje výjimku
  - Invalid page fault — adresa není součástí adresního prostoru procesu — obvykle porces zastaven se segmentation fault
  - Valid page fault — adresa je součástí VAP, ale nezle přeložit (nenachází se v MMU, tedy musí page walk vykonat OS / překlad neexistuje — stránka není v HP ale na disku — OS vymění stránky v HP)

TLB — Translation Lookaside Buffer

Vykonávání page walk je časově náročný proces. Aby nebylo nutné pokaždé page walk vykonávat, každá MMU používá speciální HW překladovou tabulku — TLB.

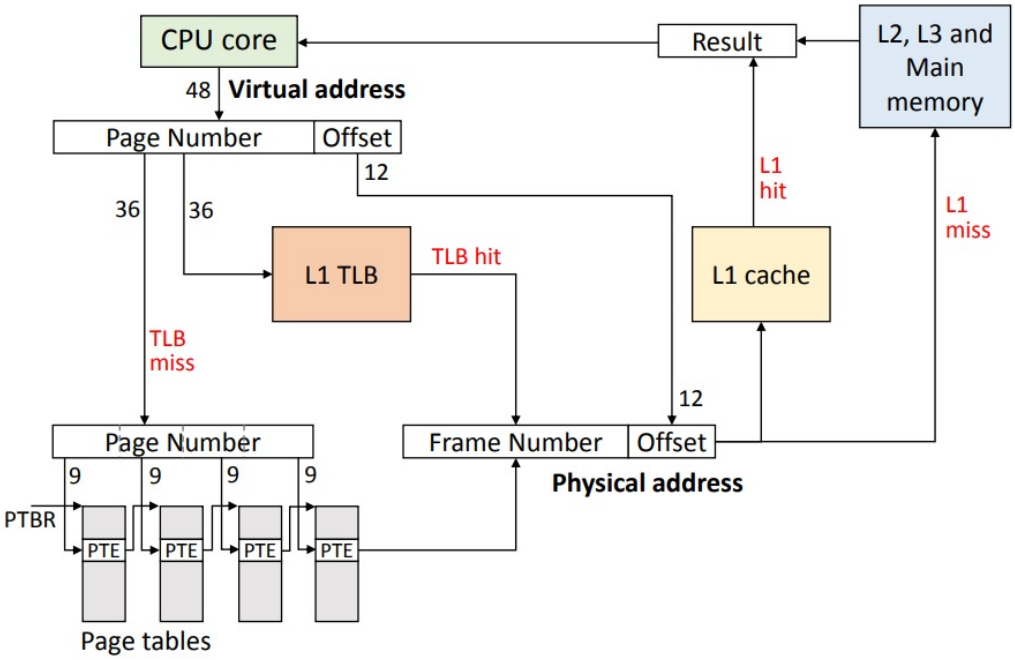
TLB jako přímo mapovaná cache:



Typicky se používá stupeň asociativity 4-64. Podobně jako u cache se používá SRAM.
Každá položka TLB typicky obsahuje:

- V: validity bit
- Tag: číslo stránky (část)
- Frame number: číslo rámce
- ASID: Identifikátor adresního prostoru (pro oddělení procesů)
- G: global flag (pokud G = 1, ASID se ignoruje)
- C: cache policy (informace, jestli jsou data na adrese "cachovatelná"— pro I/O zařízení, kam se změny dat musí posílat rovnou)
- Access permissions: R, W, X

Sumarizace HW podpory:



1.8 SP-18 (OSY)

Virtualizace hlavní paměti stránkováním, principy překladu virtuálních adres na fyzické, struktura tabulek stránek, algoritmy pro nahrazování stránek.

Princip virtuální paměti se stránkováním:

- Proces používá virtuální/logické adresy, ty adresují virtuální adresní prostor
- VAS (virtual adress space) je rozdělen na stejně velké stránky — typicky 4KB nebo 8KB
- na stejně velké úseky (rámce) je rozdělena fyzická paměť
- aktuálně používané stránky musí být aktuálně v hlavní paměti
- virtuální adresa = číslo stránky + offset

Možnosti překladu adres:

- jednoúrovňová tabulka stránek
- víceúrovňová tabulka stránek
- invertovaná tabulka stránek

Překlad adres zajišťuje MMU s TLB (viz 1.7).

Jednoúrovňová TS:

- pro každou stránku VAS daného procesu obsahuje jeden řádek obsahující číslo rámce a kontrolní bity (Present bit (P) — je stránka v hlavní paměti?, Reference bit (R) — přistupovalo se ke stránce?, Modify bit (M) — byl obsah modifikován?, Přístupová práva, Cache disabled/enabled, R/W, User/Supervisor (U/S) - lze přistupovat v uživatelském módu?)
- číslo stránky = index do této tabulky
- pro každý proces jedna tabulka

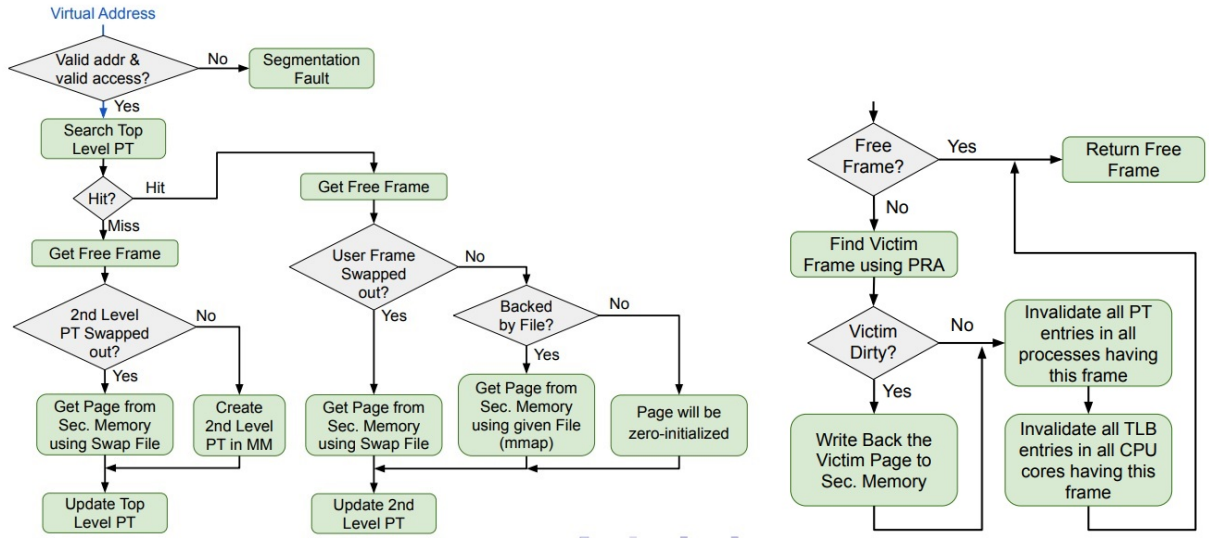
Víceúrovňová TS:

- virtuální adresa se skládá z n indexů, které ukazují do tabulek jednotlivých úrovní + offset
- tabulky stránek úrovní 1,...,$n-1$ obsahují číslo rámce kde je následující tabulka + present bit
- tabulka úrovně n obsahuje present bit + číslo rámce hledané stránky
- hlavní/první tabulka je v paměti vždy

Invertovaná TS:

- obsahuje pro každý rámec fyzické paměti jeden řádek, kde je uloženo: číslo stránky nahrané do rámce, číslo procesu, kterému stránka patří, kontrolní bity, index zřetězení (stejně velký jako index do tabulky)
- existuje 1 tabulka pro celý systém
- číslo stránky se hashovací funkcí převede na index do tabulky
- více stránek se může namapovat na stejný rámec — proto index zřetězení

Řešení výpadku stránek:



Algoritmy pro náhradu stránek

V okamžiku kdy většina/všechny rámce fyzické (hlavní) paměti jsou obsazené, je úkolem OS najít vhodný rámec, jehož obsah (stránka) se uvolní. K tomu slouží algoritmy pro náhradu stránek.

Co je od takových algoritmů požadováno?

- minimalizace počtu výpadků stránek
- rychlost
- jednoduchá implementace

Tyto algoritmy využívají proncipů prostorové a časové lokality.

Optimální algoritmus:

- nahradí se stránka, která má čas příštího přístupu nejdelší
- generuje minimální počet výpadků stránek
- sice nelze udělat, ale slouží pro porovnání kvality reálných algoritmů

NRU (Not Recently Used)

- pro každou stránku se pamatuje reference bit (R) a modified bit (M) (viz výše)
- reference bit se periodicky nastavuje na 0
- stránky jsou rozděleny do 4 tříd (RM == 00, RM == 01, RM == 10, RM == 11)
- nahradí se nějaká stránka z co nejnižší třídy (tedy 00 → 01 → 10 → 11)

Jednoduchý na pochopení i implementaci, poměrně nízký počet výpadků stránek.

FIFO (First In First Out)

- je udržován seznam stránek nahraných v paměti
- nově nahraná stránka je zaznamenána na konec seznamu
- nahrazena je první stránka ze seznamu

Jednoduchý na pochopení i implementaci, ale generuje poměrně vysoký počet výpadků stránek.

Clock algoritmus

- modifikovaný FIFO algoritmus
- seznam stránek jako kruhová fronta
- na počátku ručička ukazuje na první položku seznamu

- pro každou položku je zaznamenán reference bit, který je nastaven na 1 při pridání stránky do seznamu a při přístupu k ní
- při potřebě náhrady stránky ručička u položky, na kterou ukazuje, zjistí stav R bitu — pokud 1, vynuluje a jde na další položku — pokud 0, tato stránka se nahradí a ručička se posune

Jednoduchý na implementaci, generuje poměrně nízký počet výpadků stránek. Existují varianty s více ručičkami, kde podle rychlosti posunu ručiček a jejich rozevření je definováno časové okno, dle kterého zjišťujeme, zda byla stránka nedávno použita.

LRU (Least Recently Used)

- vybere se stránka, která je nejdelší dobu bez přístupu
- pro každou položku je navíc zapamatován čas použití, který se aktualizuje při každém použití (existuje globální čítač, který se zvýší při každém přístupu do paměti, jeho hodnota je pak zanesena k právě použité stránce)
- kandidát je taková stránka, která má nejnižší čas posledního přístupu (nutno porovnat všechny)

Generuje poměrně nízký počet výpadků stránek, dobrá aproximace optimálního algoritmu. Složitější implementace (čítač s časem a porovnání všech stránek)

Aging algoritmus

- simulace LRU lgoritmu
- pro každou stránku je zapamatováno: R bit (nastaví se na 1 při každém přístupu), n-bitový čítač C, který má po načtení stránky do paměti všechny bity na 1
- periodicky se pro každou stránku C posune o 1 doprava, jeho nejvýznamější bit se nastaví na R, a R se nastaví na 0
- vhodným kandidátem je stránka s nejnižší hodnotou C

Menší režie než LRU, ale není tak přesný (nepamatuje se přesný čas, ale jen interval, kdy se naposledy přistupovalo — omezená historie).

1.9 SP-17 (OSY)

Procesy a vlákna, jejich implementace, nástroje pro synchronizaci vláken. Klasické synchronizační úlohy. Uváznutí (deadlock) vláken (alokace prostředků, Coffmanovy podmínky, strategie pro řešení uváznutí).

- **Program:**

  Program je v systému reprezentován spustitelným binárním programem, který je uložený v sekundární paměti (např. disk).

- **Proces:**

  Instance spuštěného programu/aplikace. Entita, v rámci které jsou alokovány prostředky (paměť, vlákna, otevřené soubory, zámky, semafory, sokety,...).

- **Vlákno:**

  Výpočetní entita (proud instrukcí), které je přidělováno jádro CPU. Vlákna vytvořená v rámci procesu sdílí většinu prostředků alokovaných v tomto procesu.

Vytvoření procesu:
Nový proces lze vytvořit jako kopii/klon původního procesu, či jako úplně nový proces. V Unixu fork() exec(), ve Windows CreateProcessA().

- **fork():**

  Vytvoří nový proces, který je kopií toho procesu, ze kterého byla tato funkce zavolána. V případě chyby vrací -1, v potomkovi vrací 0, v rodiči vrací PID potomka.

- **exec():**

  Adresový prostor aktuálního procesu je přepsán obsahem souboru, který se začne vykonávat od začátku.

- **wait():**

  Zablokuje rodičovský proces, ve kterém je zavolána, dokud se konkrétní/jeden potomek neukončí.

Ukončení procesu:

- jádro se pokusí předat návratový kód rodiči
- ukončí se všechna vlákna pod procesem
- uvolní se adresový prostor procesu a příslušné struktury OS
- proces se může ukončit sám (buď normální konec programu jako *return*, nebo chyba, kvůli které se sám ukončí), nebo může být ukončen jádrem (fatální chyba nebo signál od jiného procesu)

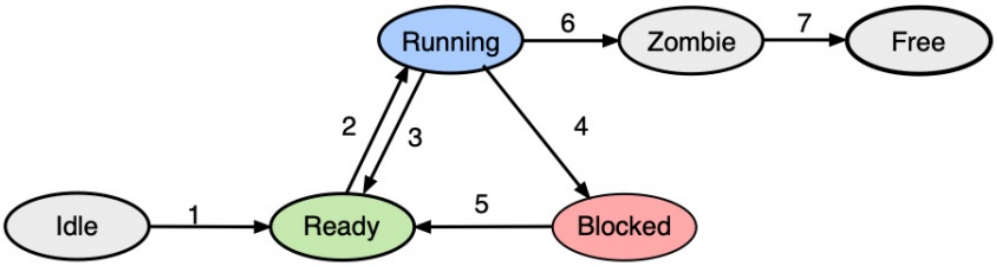
Vlákna:

- proces se implicitně vytváří s jedním "main"vláknem
- další vlákna lze vytvořit z hlavního (volání OS)

Plánování:

- vláken je typicky zásadně více než logických jader procesoru
- jedno vlákno je zpracováváno max 1 logickým jádrem
- aby se vlákna na jádrech vystřídala, používá se typicky preemptivní plánování
  - vlákno je na základě plánovacích kritérií vybráno, a je mu přiděleno volné jádro CPU
  - vláknu je přiděleno množství času na CPU
  - vláknu je jádro odebráno, pokud uplyne přidělený čas, vlákno provede systémové volání nebo dojde k přerušení
- přepínání kontextu (vystřídání vláken na jádře CPU)
  - kontext = všechny nezbytné informace pro pozdější spuštění přerušeného vlákna od okamžiku přerušení
  - kontext se uloží do paměti, naplánuje se další vlákno a jeho kontext se nahraje do CPU jádra

Stavy vláken:



- **Časově závislé chyby:**

  Situace, kdy více vláken používá společné sdílené prostředky a výsledek deterministického algoritmu je závislý na rychlosti jednotlivých vláken, které používají tyto prostředky. Špatně se detekují — lze předcházet správným návrhem paralelního algoritmu.

- **Kritická sekce:**

  Část programu, kde vlákna používají sdílené prostředky.

- **Sdružené kritické sekce:**

  Kritické sekce více vláken, které se týkají stejného sdíleného prostředku.

- **Vzájemné vyloučení:**

  Vláknům není dovoleno sdílet stejný prostředek ve stejném čase, tedy se nenachází ve stejné sdružené sekci současně.

- **Korektní paralelní program:**

  Nesmí klást předpoklady na rychlost vláken a počet jader. Musí zajistit výlučný přístup ke sdíleným prostředkům. Mimo kritické sekce by vlákno nemělo být zpomalováno ostatními vlákny.

Problémy s použitím synchronizace vláken:

- deadlock (x vláken čeká na událost, kterou může vyvolat jen jedno z čekajících vláken)
- livelock (několik vláken vykonává neužitečný výpočet, ale nemohou dokončit)
- starvation (vlákno ve stavu "ready"předbíháno a dlouho se nedostane na řadu)

Zamykání kritických sekcí:

- zámek značí, zda je ve sdružené kritické sekci už jiné vlákno — musí se k němu přistupovat atomicky (např. TSL — Test and Set Lock)
- aktivní vs blokující čekání
- **Zámek (mutex):**

  Pamatuje si svůj stav (zamčený/odemčený) a množinu vláken blokovaných na něm. Jsou nad ním definovány atomické operace lock() a unlock().

- **Podmíněná proměnná (conditional variable):**

  Pamatuje si, která vlákna jsou na ní blokována. jsou definovány operace cond_wait(mutex) a cond_signal(). cond_wait(mutex) — mutex musí být zamčen volajícím vláknem, funkce odblokuje mutex a zablokuje vlákno dokud nepřijde cond_signal().

- **Semafor:**

  Obsahuje čítač a seznam blokovaných vláken. Jsou definovány operace sem_init(int) (nastaví čítač na 0), sem_wait() (vstup do sekce — pokud čítač je větší než 0 tak vlákno vstoupí a dekrementuje čítač, jinak se zablokuje) a sem_post() (uvolní nějaké čekající vlákno, nebo inkrementuje čítač).

- **Bariéry:**

  Obsahuje čítač (síla bariéry — kolik vláken musí čekat, aby byla odblokována) a blokovaná vlákna. Operace: barrier_init(int) (nastaví sílu bariéry) a barrier_wait() (pokud čítač je více než 1, vlákno čeká a čítač je dekrementován, jinak jsou všechna vlákna probuzena)

Synchronizační úlohy:

- Večeřící filosofové
  - N filosofů u kulatého stolu
  - každý má před sebou jídlo a mezi sousedními talíři je vždy 1 vidlička (celkem tedy N vidliček)
  - pokud chce filosof jíst, musí získat obě vidličky vedle jeho talíře
  - stavy filosofa: přemýšlí (nechce a nemá vidličky), má hlad (pokouší se získat obě vidličky), jí (má obě vidličky)
  - optimální řešení: může jíst až $\lfloor N/2 \rfloor$ filosofů, nevznikají časově závislé chyby ani synchronizační problémy
  - řešení: pokud mam hlad zamknu mutex, kouknu jestli jsou volny vidlicky, pokud ne spim (odemykam mutex), pokud jo, beru, odemykam mutex a jim. Az dojim, zamknu mutex, vratim vidlicky, probudim sousedy a odemknu mutex.
- Čtenáři — písaři
  - v systému je 1 sdílený prostředek
  - písaři mohou modifikovat, čtenáři pouze číst
  - chceme, aby pokud není modifikováno (nepřistupuje písař) mohlo číst více čtenářů
  - zároveň by nikdo neměl být předbíhán
  - řešení: písaři i čtenáři se řadí do fronty, ale po skupinách — pokud přijdu na konec fronty a je tam už stejný typ, přidám se do skupiny — na začátku fronty je probuzena celá skupina a buď písaři postupně zapíšou, nebo čtenáři společně přečtou
- Spící holiči
  - v holičství je N holičů a křesel k holení, a M křesel k čekání
  - pokud nejsou zákazníci, holič sedne do holícího křesla a usne
  - pokud přijde zákazník, buď probudí holiče (pokud je volný), nebo si sedne do čekárny (pokud je místo) jinak odejde

Obecně alokace prostředku:

- vlákno žádá o prostředek pomocí alokační funkce
- pokud je prostředek volný, je přidělen
- pokud je již alokovaný, vlákno může být blokováno (v závislosti na alokační funkci — mutex_lock blokuje, mutex_try_lock blokuje jen na určitý čas, fork() a malloc() neblokují)
- v případu 2 a 3 se vlákno pak samo rozhodne jak pokračovat

Coffmanovy podmínky
Uváznutí (deadlock) nastane pouze pokud jsou splněny všechny následující podmínky:

- Vzájemné vyloučení — každý prostředek nemůže být sdílen více vlákny
- Podmínka neodnímatelnosti — již přidělený prostředek nemůže být odebrán násilím
- Podmínka "drž a čekej"— vlákno s již přiděleným prostředkem může žádat o další
- Podmínka kruhového čekání — musí existovat smyčka více vláken, ve které každé vlákno čeká na prostředek držený dalším vláknem ve smyčce

Řešení uváznutí:

- Pštrosí strategie — ignorování
- Prevence uváznutí — nesplnění alespoň jedné z Coffamnových podmínek
- Předcházení vzniku uváznutí — pečlivá alokace prostředků
- Detekce uváznutí a zotavení — uváznutí je detekováno a odstraněno

Implementace procesů:

- jádro OS si udržuje zřetězený seznam struktur — tabulku procesů
- jedna položka tabulky obsahuje vše nezbytné, co si OS musí o procesu pamatovat (PCB — Process Control Block)
  - identifikace procesu (id procesu, id rodiče, číslo úlohy/seance/projektu, jméno procesu...)
  - identita/bezpečnost (vlastník, skupiny, práva procesu)
  - informace o alokovaných prostředcích (paměť, soubory, prostředky pro meziprocesovou komunikaci)

Implementace vláken:

- Thread Control Block (TCB) — identifikace vlákna, info o přepínání kontextu (registry), informace pro plánování vláken
- Implementace v uživatelském prostoru (zastaralé)
  - OS přistupuje k procesům jako by měly jedno vlákno
  - proces si svá vlákna spravuje sám
  - kooperativní plánování pro vlákna v procesu
- Implementace v jádře OS
  - OS plánuje samotná vlákna, ne procesy
  - OS udržuje jede PCB pro každý proces, jeden TCB pro každé vlákno
  - preemptivní plánování pro vlákna v procesu

Plánování v dnešních OS: Prioritní Round Robin
X front s různou prioritou. Na základě předchozího běhu vlákna se buď zvýší časové kvantum a sníží priorita (pokud vlákno využilo všechen čas) nebo zvýší priorita a sníží čas (pokud nevyužilo celé časové kvantum).

1.10 OB-3 (ADU)

Procesy a systémové služby v unixových operačních systémech: hierarchie a vzájemné vazby, limity, zapínání a vypínání systému, logování aktivit systému.

Procesy:

- vykonávané programy
- mají svoje ID (PID), id uživatele (UID), id rodiče (PPID) ...
- každý proces někdo vytvořil, tedy každý proces má nějaké PPID — init proces je první proces spuštěn při bootu, má PID = 1 a PPID = 0
- rodič typicky čeká, až mu dítě předá návratový kód
- daemon = systémový proces (běží na pozadí bez nutnosti vstupu ... jako u windows služby)
- orphan = proces, jehož rodič už neexistuje — adoptuje se pod proces init
- zombie = proces, který již skončil, ale ještě si rodič nepřevzal návratovou hodnotu
- procesy si navzájem mohou posílat signály

Limity:

- soft limit — limit mezi 0 a hard limitem, lze uživatelsky přenastavit
- hard limit — může změnit jen root, nelze překročit
- lze nastavit max počet procesů pro uživatele a další omezení využití zdrojů (paměť, velikost souborů, počet file descriptorů...)
- příkaz ulimit

BOOT

- Firmware fáze
  - POST (Power On Self Test)
  - inicializace HW a driverů
  - výběr bootovacího zařízení
  - načtení a spuštění bootovacího kódu
- Boot-loader fáze
  - nalezení a načtení boot programu
  - boot program se nahrává do pevně daných adres v paměti
  - kontrola řízení se předá programu
- GRUB (Grand Unified Boot Loader) fáze
  - načte se GRUB konfigurace
  - vybere se odkud/co bootovat
  - kernel se načte a spustí
- Kernel fáze
  - inicializace datových struktur jádra
  - načtení driverů, inicializace HW
  - mount kořenového filesystému
  - načtení konfigurace jádra
  - načtení modulů
  - vytvoření init procesu

- Init fáze
  - dříve spouštění start/stop scriptů dle úrovní běhu systému (run levels / milestones)
    - 0 — systém vypnutý
    - 1, s, S — single user mode
    - 2 — multi user mode
    - 3 — multi user mode + síť
    - 4 — nepoužívaný (občas GUI)
    - 5 — vypnutí
    - 6 — restart
  - v dnešní době spouštění služeb
  - konfigurace procesu init — /etc/inittab
  - init se přes fork() a exec() naklonuje a spouští další procesy

Vypínání systému:

- shutdown *level timeout -y* (přívětivý jak k uživatelům a aplikacím, tak k systému)
- init *level* (přívětivý k aplikacím a systému)
- poweroff, restart (přívětivé k systému)
- vypnout napájení (fuj)

Logování:

- servrová logovací služba s dlouhou historií
- umožňuje oddělit SW generující zprávy, SW který je ukládá a SW který je analyzuje a nahlašuje
- konfigurace: /etc/syslog.conf
- logovací složky: /var/log a /var/adm
- mnoho variant
- záznamy v logu mají info o službě a závažnosti logu (alert, critical, error, warning, notice, info, debug)

1.11 OB-2 (ADU)

Správa disků a souborových systémů (zařízení, souborové systémy UFS (EXT) a ZFS, disková pole RAID, diskové kvóty), síťové souborové systémy (NFS, CIFS), swap v unixových operačních systémech.

Zařízení:

- popsána speciálními soubory v adresáři /dev
- znakové/blokové
- zápis/čtení souboru znamená stejnou operaci nad zařízením
- pseudozařízení — /dev/null, /dev/random, /dev/zero,...
- vše potřebné uloženo v i-node (jako odkaz na driver)
- většinou může operace nad těmito soubory provádět jen root
- lze vytvořit přes mknod

Disky:

- rozdělení disku závisí na HW i na OS
- na x86: partitions (partition tabulka)
- na solaris: slices (pokud na x86, jsou v rámci partitions)
- Partitions:
  - rozdějují fyzický disk na více logických celků
  - popsány partition tabulkou
  - MBR — Master Boot Record — v prvním sektoru disku, obsahuje kód zavaděče (GRUB) a partition tabulku se 4 záznamy (max 4 partitions)
  - GPT — GUID Partition Table — partitions mají globálně unikátní ID, je součástí UEFI (nástupce BIOS)

Filesystem:

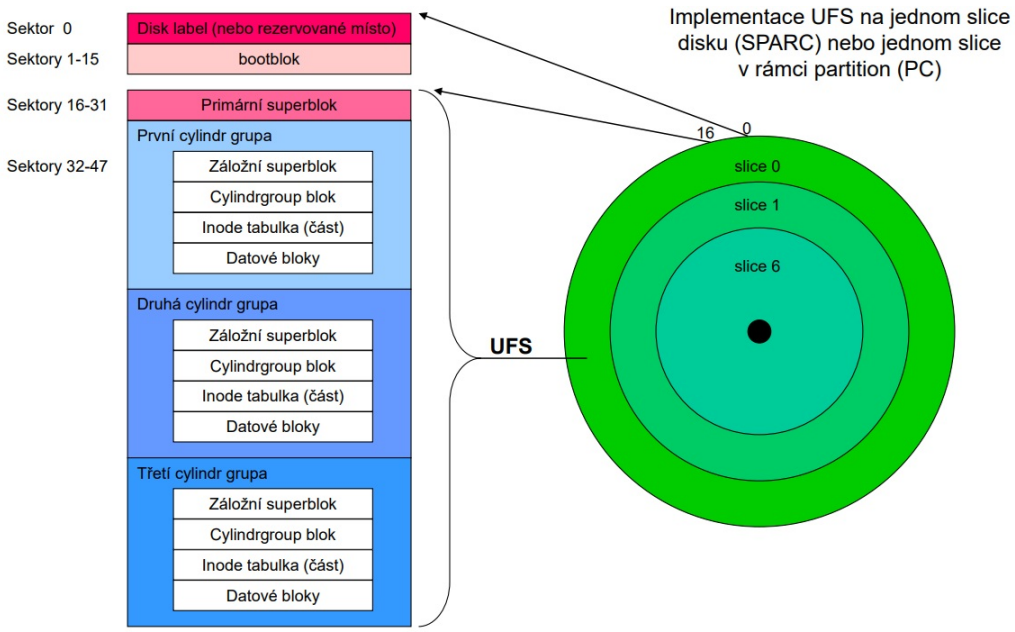
- logický — adresářový strom
- fyzický — filesystém na disku, připojuje se k mountovacímu bodu
- diskový, síťový, ...

SWAP:

- odkládací prostor na disku pro nepoužívané stránky paměti
- většinou se řeší buď pomocí swap partition, nebo swapovacími soubory

UFS

- efektivní pro menší soubory (max desítky/stovky MB)
- nevzniká fragmentace
- bootblock (na začátku disku), superblock primární (po bootblocku) a záložní (na začátku každé cylindr grupy)
- i-node
  - struktura obsahující metadata
  - typ, přístupová práva, vlastník, skupina, velikost, čas vytvoření, přístupu a modifikace, čítač linků
  - odkazy na datové bloky: 12 přímých, 3 nepřímé (první, druhé a třetí úrovně)
- příkazy: mkfs (vytvoření filesystému), mount (připojení FS k mountovacímu bodu), fsck (kontrola a oprava disku)
- snapshot — zapamatování stavu FS, používá se k archivaci



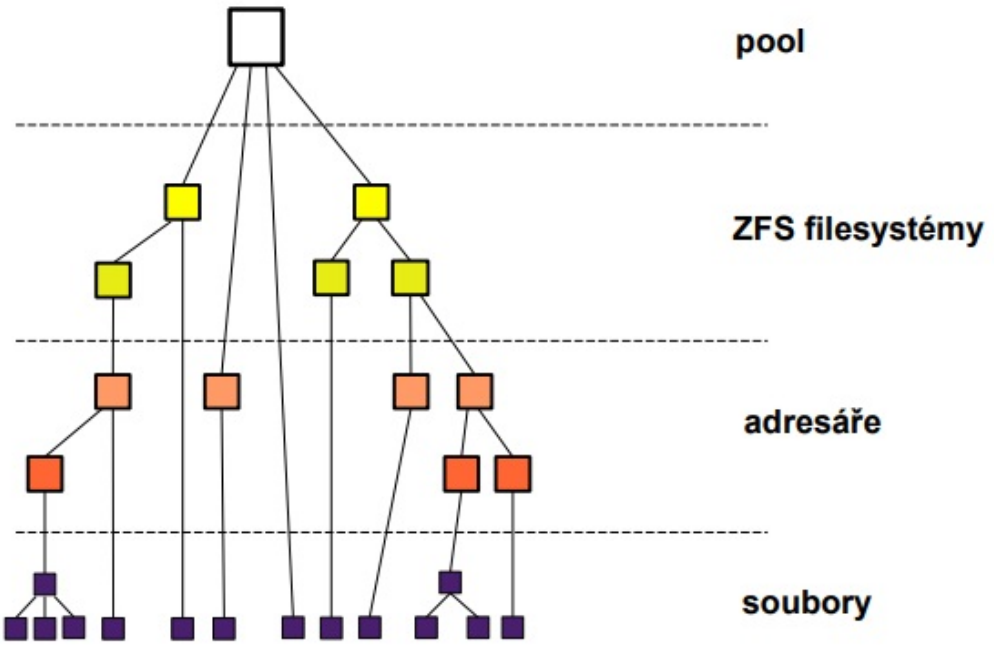
RAID

- Redundant Array of Independent Disks
- možné řešit jak HW tak SW
- složení jednoho logického disku z více fyzických
- může zvýšit kapacitu, zvýšit výkon a zvýšit bezpečnost (obrana proti výpadku)
- RAID 0 - zřetězení (JBOD - just a bunch of disks)
  - data se ukládají postupně na disky za sebe — když se zaplní jeden, začne se ukládat na druhý
  - redundance 0 %
  - výpadek jednoho disku způsobí ztrátu dat
  - výkon se nemění
- RAID 0 - prokládání
  - data jsou ukládána cyklicky po blocích na jednotlivé fyzické disky
  - redundance 0 %
  - výpadek jednoho disku způsobí ztrátu všech dat
  - výkon zvýší m krát (m je počet disků)
- RAID 1 - zrcadlení
  - stejná data jsou uložena na všech discích
  - redundance je $100 \times (m-1)/m$ %
  - data přežijí výpadek $m-1$ disků a výkon nebude degradován
  - write operace stejně rychlé, read se může zrychlit až m krát
- RAID 1+0 / 0+1
  - nejvyšší logický disk je rozdělen podle RAID 0/1, nižší logické disky jsou opačně (RAID 1/0)
  - redundance je podle počtu zrcadlení v RAID 1 části — při 2 kopiích dat je to 50 %
  - data přežijí výpadek od 1 do $m/2$ disků (př 2 kopiích)
  - write operace zrychleny až $m/2$ krát, read až m krát
- RAID 2, 3, 4 se nepoužívají (2: prokládání po bitech + hammingův kód, 3: prokládání po bytech a zabezpečení pomocí uložení parity uložené na jednom fyzickém disku, 4: prokládání po blocích, uládání parity na jednom fyzickém disku)

- RAID 5
  - prokládání po blocích na m fyzických discích + ukládání parity cyklicky ukládané na jednotlivých discích
  - redundance je $100/m$
  - data přežijí výpadek jednoho disku, ale bude degradován výkon
  - read se zrychlí, write pomalejší
- RAID 6
  - prokládání po blocích na m fyzických discích + ukládání dvojí parity cyklicky ukládané na jednotlivých discích
  - redundance je $200/m$
  - data přežijí výpadek dvou disků, ale bude degradován výkon
  - read se zrychlí, write pomalejší

ZFS

- snaží se řešit problémy stávajících FS
  - nekonzistence při nekorektním vypnutí
  - nemožnost/komplikované zvětšování kapacity FS
  - RAID, snapshot, zálohování jsou řešené mimo vlastní FS
  - FS nejsou hierarchické
  - komplikovaná administrace
  - chybějící klonování, cache, kryptování, deduplikace...
- metadata řešena podobně jako v UFS
- datový prostor tvoří virtuální datová oblast — pool
- pool je tvořen zdroji dat — disky, partitions, soubory (speciální — zařízení)
- z poolu se alokují datové bloky — různá velikost, tvořené nad strukturami RAID
- transakční systém "copy on write"— data se nepřepisují, pouze se zkopírují, změní a pak je změna přijata či zamítnuta
- pool sám je (ZFS) FS
- každý blok má kontrolní součet



NFS

- daemons na serveru:
  - mountd — vyřizuje požadavky na mount, vrací file handle
  - nfsd — vyřizuje požadavky na operace se soubory, vrací data
  - statd — spolupracuje s lockd, znovunastavuje spojení po výpadku
  - lockd — zamykání NFS souborů, požadavek se posílá z klienta na server
  - nfslogd — logování přístupů
- daemons na klientu — statd, lockd
- NFS 2/3 nestavové, NFS 4 stavový
- NFS 4 — sjednocený daemon, well known port 2049, delegace cache, možnost mountovat pseudo filesystém (vyšší adresář, ale s přístupem jen do nižšího)

CIFS/SMB (Samba) je novější síťový FS, snadnější vazba unix-windows.

1.12 OB-1 (ADU)

Identita uživatelů v unixových operačních systémech (identita, práva administrátora, sudo, su, PAM moduly, role, privilegia, identita a přístupová práva, ACL, suid programy).

Administrace uživatelů

- konfigurační soubory
- uživatelé a jejich primární skupiny v souboru /etc/passwd
- hesla uživatelů v /etc/shadow
- (sekundární) skupiny uživatelů v /etc/group
- přihlašování na jiného uživatele (změna identit): příkaz su [-] [username [argument]] ("-"rozhoduje zda se použijí přihlašovací skripty cílového uživatele, tedy jestli se změní prostředí)

Identita procesů

- vnější identita: username, groupname
- vnitřní identita: UID, GID
- 3 druhy vnitřní identity:
  - Reálná (RUID, RGID) — využívána při akceptaci signálů
  - Efektivní (EUID, EGID) — využívána při práci se soubory (vlastnictví, práva,...)
  - Uložená/saved (SUID, SGID) — uchování identity při změně EUID/EGID
- potomci procesů dědí identity od rodiče, s výjimkou SUID/SGID programů. kdy potomek získá EUID a SUID od vlastníka programu (respektive EGID a SGID) — např. "su"

Administrátor

- nazýván root
- UID = 0
- může všechno — měnit runlevel systému, vypínat/zapínat systémové služby, používat/přidávat zařízení...
- v bezpečnějších systémech pouze jako role — nelze se přihlásit přímo

sudo

- **s**uper **u**ser **do**
- konfigurace v /etc/sudoers (může měnit jen root)
- spuštění konkrétního příkazu jako jiný uživatel (typicky root)
- lze nakonfigurovat konkrétní práva pro konkrétní uživatele, lze logovat použití
- náchylné k chybám při konfiguraci
- běžný účet může díky právům něco zničit
- záznam v konfiguraci: who where=(as who) what
- možné distribuovat na další zařízení, tedy řídit více systémů
- použití většinou vyžaduje heslo aktuálního uživatele (ne toho za koho je příkaz prováděn)

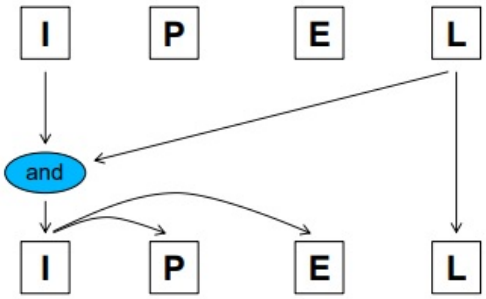
ACL

- klasická oprávnění (rwx pro vlastníka, skupinu a ostatní) nebyla dostatečná
- ACL (Access Control List) jsou dodatečné informace o právech přístupu k souborům (pouze rozšíření, ne náhrada)
- umožňují specifikovat rwx práva pro každého konkrétního uživatele a skupinu
- lze použít masky

- příkaz setfacl a getfacl
- není standardem, některé FS/OS neimplementují

Práva a role

- klasický UNIXový systém řeší práva přes UID (kontroluje, zda UID == 0)
- administrátorská práva jsou nedělitelná, problém v případě více administrátorů
- systém, který podporuje práva:
  - cca 80 práv pokrývajících všechny operace
  - UID nemá efekt na práva, procesy si s sebou přenáší množinu práv
  - root má typicky všechna práva, běžný uživatel podmnožinu
  - 4 množiny práv v procesu: Effective, Inheritable, Permitted, Limit



  - systém kontroluje, zda je odpovídající právo obsaženo v množině E
- RBAC:
  - Role Based Access Control
  - uživatelé mají přiřazené role
  - role mají vlastnosti jako uživatelé, ale nelze se přihlásit přímo, pouze přes "su"— např role root
  - uživatel se pro použití práv role musí přihlásit (musí tedy mít roli přiřazenou, a zároveň znát její heslo)
  - role mají přiřazené profily
  - profily mají přiřazené množiny příkazů, spustitelné pod určitými identitami a se specifickými právy
- PAM
  - Pluggable Authentication Modules
  - moduly pro autentizaci, které se dají připojit ke každému programu zvlášť
  - možnost vytvářet programy nezávisle na konkrétních autentizačních postupech
  - konfigurace pro konkrétní program: typ modulu — typ použití — modul

/etc/pam.d/login

```
auth       required   pam_securetty.so
auth       required   pam_shells.so
auth       required   pam_nologin.so
auth       include    system-auth
account    include    system-auth
password   include    system-auth
session    required   pam_env.so
session    include    system-auth
```

2 Šifrování a sítě

2.1 SP-7 (DML)

Výroková logika: splnitelnost formulí, logická ekvivalence a důsledek, universální systém logických spojek, disjuntivní a konjunktivní normální tvary, úplné normální tvary.

- **Prvotní výrok:** jednoduchá oznamovací věta, u které má smysl se ptát zda je či není pravdivá. Prvotní výroky označujeme velkými písmeny, říkáme jim **prvotní formule**.
- **Pravdivostní ohodnocení:** ohodnocení množiny prvotních výroků je přiřazení v, které každé prvotní formuli přiřadí 0 nebo 1.
  - Je-li $v(A) = 1$, říkáme že A je pravdivý při ohodnocení v.
  - Je-li $v(A) = 0$, říkáme že A je nepravdivý při ohodnocení v.
- **Negace:** $\neg A$ výroku A je pravdivá pro všechna ohodnocení, při kterých je A nepravdivý. Pro ostatní je nepravdivá.
- **Konjunkce:** $A \wedge B$ výroků A a B je pravdivá pro všechna ohodnocení, při kterých jsou A i B současně pravdivé. Pro ostatní ohodnocení je nepravdivá.
- **Disjunkce:** $A \vee B$ výroků A a B je pravdivá pro všechna ohodnocení, při kterých je alespoň jeden z výroků A a B pravdivý. Pro ostatní ohodnocení je nepravdivá.
- **Implikace:** $A \Rightarrow B$ mezi výroky A a B je nepravdivá pro všechna ohodnocení, kdy **předpoklad** A platí a **závěr** B neplatí. Pro ostatní ohodnocení je pravdivá.
- **Ekvivalence:** $A \Leftrightarrow B$ mezi výroky A a B je pravdivá pro všechna ohodnocení, při kterých mají výroky A a B stejnou pravdivostní hodnotu. Pro ostatní je nepravdivá.
- **Tautologie ($\top$):** Formule, která je pro každé ohodnocení pravdivá.
- **Kontradikce ($\bot$):** Formule, která je pro každé ohodnocení nepravdivá.
- **Splnitelná formule:** Formule, která je alespoň pro jedno ohodnocení pravdivá.
- Nechť E a F jsou výroky. Pokud platí $E \Rightarrow F$, pak E je **postačující podmínka** pro F. Na druhou stranu, F je **nutná podmínka** pro E. Pokud platí $E \Leftrightarrow F$, pak je E **nutná a postačující podmínka** pro F a obráceně.
- Nechť E a F jsou výrokové formule. E a F jsou logicky ekvivalentní, právě když pro každé ohodnocení v je $v(E) = v(F)$. Píšeme $E \models\!\models F$.
- Nechť E a F jsou výrokové formule. F je logickým důsledkem E, právě když pro každé ohodnocení v, pro které $v(E) = 1$, je i $v(F) = 1$. Píšeme $E \models F$.
- **Základní principy logiky:**
  - Zákon vyloučení sporu: $A \wedge \neg A \models\!\models \bot$
  - Zákon vyloučení třetího: $A \vee \neg A \models\!\models \top$
  - Zákon dvojí negace: $\neg\neg A \Leftrightarrow A \models\!\models \top$
- **Obměněná implikace:** $(E \Rightarrow F) \models\!\models (\neg F \Rightarrow \neg E)$
- Množina logických spojek tvoří **universální systém**, právě když ke každé formuli existuje logicky ekvivalentní formule, která obsahuje pouze tyto spojky.
- Např. dvouprvkové systémy: $\{\neg, \vee\}$, $\{\neg, \wedge\}$, $\{\neg, \Rightarrow\}$
- Existují i jednoprvkové systémy pouze z NAND ($\uparrow$) či NOR ($\downarrow$)
- **Literál:** Výroková formule, která je prvotní formulí, nebo negací prvotní formule.
- **Implikant:** Literál, či konjunkce několika literálů.
- **Výroková formule v disjunktivním normálním tvaru (DNT)**, pokud je implikantem, či disjunkcí několika implikantů.

- **Klausule:** Literál, či disjunkce několika literálů.
- **Výroková formule v konjunktivním normálním tvaru (KNT)**, pokud je klausulí, či konjunkcí několika klausulí.
- Každá výroková formule lze převést do logicky ekvivalentního KNT i DNT.
- **Minterm:** minterm formule F je takový její implikant, který obsahuje všechny prvotní formule vyskytující se v F a každou právě jednou.
- **Výroková formule v úplném disjunktivním normálním tvaru (ÚDNT)**, je-li mintermem nebo disjunkcí různých (logicky neekvivalentních) mintermů.
- **Maxterm:** minterm formule F je taková její klausule, která obsahuje všechny prvotní formule vyskytující se v F a každou právě jednou.
- **Výroková formule v úplném konjunktivním normálním tvaru (ÚDNT)**, je-li maxtermem nebo konjunkcí různých (logicky neekvivalentních) maxtermů.
- Každá výroková formule lze převést do logicky ekvivalentního ÚKNT i ÚDNT.

2.2 SP-8 (DML)

Základy teorie čísel: dělitelnost, REA a diofantické rovnice, prvočísla, modulární aritmetika, Malá Fermatova a Eulerova věta, lineární kongruence, Čínská věta o zbytcích.

- **Dělitelnost:** Nechť $a, b \in \mathbb{Z}$. Řekneme, že a dělí b, značíme $a \mid b$, jestliže existuje $k \in \mathbb{Z}$ takové, že $a \cdot k = b$. V takovém případě říkáme, že a je (celočíselný) dělitel b a b je (celočíselný) násobek a, případně také, že b je dělitelné a. Pokud a nedělí b, píšeme $a \nmid b$. Samotné $\mid$ nazýváme relací dělitelnosti.
- **Dělení se zbytkem:** Nechť $a \in \mathbb{Z}, d \in \mathbb{N}$. Pak existují jednoznačně určená čísla $q, r \in \mathbb{Z}$ taková, že $a = qd + r \wedge 0 \leq r < d$. Číslo q nazýváme celočíselný podíl (po dělení a číslem d). Číslo $r \in \{0, 1, ..., d-1\}$ nazveme zbytkem po (celočíselném) dělení a číslem d a značíme jej $r = a \bmod d$.
- **Společný dělitel:** Nechť $a, b \in \mathbb{Z}$. Číslo $n \in \mathbb{N}_0$ je společný dělitel čísel a, b, jestliže $n \mid a \wedge n \mid b$.
- **Největší společný dělitel:** Nechť $a, b \in \mathbb{Z}$. Číslo $n \in \mathbb{N}_0$ je největší společný dělitel čísel a, b (značíme $n = \gcd(a, b)$), pokud je jejich společný dělitel a současně je (celočíselným) násobkem každého jejich dalšího společného dělitele, tedy pokud $(n \mid a \wedge n \mid b \wedge (\forall d \in \mathbb{N}_0)((d \mid a \wedge d \mid b) \Rightarrow d \mid n))$.
- **Soudělnost:** Nechť $a, b \in \mathbb{Z}$. Nazveme je nesoudělná, pokud $\gcd(a, b) = 1$. Pokud $\gcd(a, b) > 1$, nazveme tato čísla soudělná.
- **Společný násobek:** Nechť $a, b \in \mathbb{Z}$. Číslo $n \in \mathbb{N}_0$ je společný násobek čísel a, b, jestliže $a \mid n \wedge b \mid n$.
- **Nejmenší společný násobek:** Nechť $a, b \in \mathbb{Z}$. Číslo $n \in \mathbb{N}_0$ je nejmenší společný násobek čísel a, b (značíme $n = \operatorname{lcm}(a, b)$), pokud je jejich společný násobek a současně dělí každý další jejich společný násobek, tedy pokud $(a \mid n \wedge b \mid n \wedge (\forall m \in \mathbb{N}_0)((a \mid m \wedge b \mid m) \Rightarrow n \mid m))$.
- **REA — Rozšířený Euklidův Algoritmus**

$$
\begin{array}{rrrl}
\mathbf{254} = & 1 \cdot 254 + & 0 \cdot 158 & \\
\mathbf{158} = & 0 \cdot 254 + & 1 \cdot 158 & \left(\left\lfloor \frac{254}{158} \right\rfloor = 1\right) \\
\hline
254 - 1 \cdot 158 = \mathbf{96} = & 1 \cdot 254 + & (-1) \cdot 158 & \left(\left\lfloor \frac{158}{96} \right\rfloor = 1\right) \\
158 - 1 \cdot 96 = \mathbf{62} = & (-1) \cdot 254 + & 2 \cdot 158 & \left(\left\lfloor \frac{96}{62} \right\rfloor = 1\right) \\
96 - 1 \cdot 62 = \mathbf{34} = & 2 \cdot 254 + & (-3) \cdot 158 & \left(\left\lfloor \frac{62}{34} \right\rfloor = 1\right) \\
62 - 1 \cdot 34 = \mathbf{28} = & (-3) \cdot 254 + & 5 \cdot 158 & \left(\left\lfloor \frac{34}{28} \right\rfloor = 1\right) \\
34 - 1 \cdot 28 = \mathbf{6} = & 5 \cdot 254 + & (-8) \cdot 158 & \left(\left\lfloor \frac{28}{6} \right\rfloor = 4\right) \\
28 - 4 \cdot 6 = \mathbf{4} = & (-23) \cdot 254 + & 37 \cdot 158 & \left(\left\lfloor \frac{6}{4} \right\rfloor = 1\right) \\
6 - 1 \cdot 4 = \mathbf{2} = & 28 \cdot 254 + & (-45) \cdot 158 & \left(\left\lfloor \frac{4}{2} \right\rfloor = 2\right) \\
\hline
4 - 2 \cdot 2 = \mathbf{0} & & &
\end{array}
$$

- **LDR — Lineární diofantická rovnice:** libovolná rovnice typu $ax + by = c$, kde $a, b, c \in \mathbb{Z}$ pro 2 neznámé $x, y \in \mathbb{Z}$.
  - LDR $ax + by = c$ má alespoň jedno řešení právě tehdy když $\gcd(a, b) | c$.

Nechť $a, b \in \mathbb{Z} \setminus \{0\}$ a dvojice $(x_0, y_0) \in \mathbb{Z}^2$ je řešením rovnice $ax + by = c$. Potom množina všech celočíselných řešení této rovnice je

$$\left\{ \left(x_0 + \frac{b}{\gcd(a,b)} \cdot k,\ y_0 - \frac{a}{\gcd(a,b)} \cdot k \right) \,\middle|\, k \in \mathbb{Z} \right\},$$

což lze ekvivalentně zapsat také jako

$$\left\{ (x_0, y_0) + k \cdot \left(\frac{b}{\gcd(a,b)}, \frac{-a}{\gcd(a,b)} \right) \,\middle|\, k \in \mathbb{Z} \right\}.$$

- Přirozená čísla $\mathbb{N}$ dělíme podle počtu dělitelů do následujících 3 kategorií:
  - číslo 1 (1 dělitel — 1)
  - **prvočísla** — 2 dělitelé, samo číslo a 1
  - **složená čísla** — 3 a více dělitelů

- **Kongruence modulo m:** Nechť $a, b \in \mathbb{Z}, m \in \mathbb{N}$. Pokud $m \mid (a - b)$, říkáme že a je kongruentní s b modulo m a píšeme $a \equiv b \pmod{m}$. V opačném případě a není kongruentní s b modulo m a píšeme $a \not\equiv b \pmod{m}$.
- **Množina zbytků:** Nechť $m \geq 2$. Jako $\mathbb{Z}_m$ označíme množinu všech zbytků modulo m, $\mathbb{Z}_m = \{0, 1, 2, ..., m - 1\}$. Operace sčítání: součet a modulo. Násobení totéž.
- **Inverze v $\mathbb{Z}_m$:** Aditivní inverze (opačný prvek) — součet modulo je 0. Multiplikativní inverze (inverzní prvek) — násobek modulo je 1.
- **Existence multiplikativní inverze:** Nechť $m \geq 2$ a $a \in \mathbb{Z}_m$. V $\mathbb{Z}_m$ existuje multiplikativní inverze k a právě tehdy, když $\gcd(a, m) = 1$. Pokud existuje, je jediná.
- **Krácení v modulu:** Nechť $a, b, c \in \mathbb{Z}$ a $m \in \mathbb{N}, m \geq 2$, označme $d = \gcd(m, c)$. Pak platí ekvivalence: $ac \equiv bc \pmod{m} \Leftrightarrow a \equiv b \pmod{\frac{m}{d}}$
- **Malá Fermatova věta:** Buď p prvočíslo a $a \in \mathbb{N}$ takové přirozené číslo, které není násobkem p (tedy $\gcd(a, p) = 1$). Potom platí kongruence $a^{p-1} \equiv 1 \pmod{p}$.
- Pokud je p prvočíslo a $a \in \mathbb{Z}_p \wedge a \neq 0$, pak a^{p-2} je multiplikativní inverzí čísla a mod p.
- **Eulerova funkce:** $\varphi : \mathbb{N} \to \mathbb{N}$ je zobrazení, které každému $n \in \mathbb{N}$ přiřafí počet přirozených čísel menších nebo rovných n, která jsou s n nesoudělná. Tedy $(\forall n \in \mathbb{N}(\varphi(n) := |\{k \in \mathbb{N} \mid k \leq n \wedge \gcd(k, n) = 1\}|)$.
- **Eulerova věta:** Nechť $m \in \mathbb{N}, m \geq 2$ a $a \in \mathbb{N}$ je číslo nesoudělné s m. potom platí kongruence $a^{\varphi(m)} \equiv 1 \pmod{m}$.
  - p je prvočíslo, tedy $\varphi(p) = p - 1$
  - p je prvočíslo, $\alpha \in \mathbb{N}$, pak $\varphi(p^\alpha) = p^\alpha - p^{\alpha-1}$
  - $m, n \in \mathbb{N}$, $\gcd(m, n) = 1$. Potom $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$.
- Nechť $m \in \mathbb{N}, m \geq 2$ a $a \in \mathbb{Z}_>$ je číslo nesoudělné s m. potom $a^{\varphi(m)-1}$ je multiplikativní inverzí čísla a mod m.
- **Lineární kongruence:** řešením lineární kongruence rozumíme nalezení všech celých čísel x splňujících kongruenci $ax \equiv b \pmod{m}$, kde $a, b, m \in \mathbb{Z}$ a $m \geq 2$.
- **ČVOZ — Čínská věta o zbytcích**

Uvažujme soustavu lineárních kongruencí

$$\begin{aligned} x &\equiv a_1 \pmod{m_1}, \\ x &\equiv a_2 \pmod{m_2}, \\ &\vdots \\ x &\equiv a_N \pmod{m_N}, \end{aligned}$$

kde $m_1, m_2, \ldots, m_N \geq 2$ jsou navzájem nesoudělná, tedy $\gcd(m_i, m_j) = 1$ pro každá $i \neq j$.

Řešení této soustavy vždy existuje a všechna řešení jsou kongruentní modulo M (tedy v $\mathbb{Z}_M$ je řešení určeno jednoznačně), kde

$$M = \prod_{i=1}^{N} m_i.$$

- Výsledek je: $x = a_1 \cdot M_1 \cdot X_1 + ... + a_N \cdot M_N \cdot X_N \pmod{M}$

Vyřešíme soustavu

$$\begin{aligned} x &\equiv 1 \pmod 2, \\ x &\equiv 2 \pmod 3, \\ x &\equiv 3 \pmod 5. \end{aligned}$$

$M = 2 \cdot 3 \cdot 5 = 30$. Předpoklady ČVOZ jsou splněny, modula jsou dokonce různá prvočísla. Tedy řešení soustavy existuje a to jednoznačné v $\mathbb{Z}_{30}$.

Dále určíme

$$M_1 = 3 \cdot 5 = 15, \quad M_2 = 2 \cdot 5 = 10, \quad M_3 = 2 \cdot 3 = 6.$$

Nalezneme „inverze“ k M_i:

- $M_1 X_1 = 15 X_1 \equiv 1 \pmod 2$ má za řešení $X_1 = 1$,
- $M_2 X_2 = 10 X_2 \equiv 1 \pmod 3$ má za řešení $X_2 = 1$,
- $M_3 X_3 = 6 X_3 \equiv 1 \pmod 5$ má za řešení $X_3 = 1$.

Konečně,

$$x = \underbrace{1 \cdot 1 \cdot 15}_{a_1 X_1 M_1} + \underbrace{2 \cdot 1 \cdot 10}_{a_2 X_2 M_2} + \underbrace{3 \cdot 1 \cdot 6}_{a_3 X_3 M_3} = 53 \equiv 23 \pmod{30}.$$

- **ZČVOZ — Zobecněná čínská věta o zbytcích**

Uvažujme soustavu lineárních kongruencí

$$\begin{aligned} x &\equiv a_1 \pmod{m_1}, \\ x &\equiv a_2 \pmod{m_2}, \\ &\vdots \\ x &\equiv a_N \pmod{m_N}, \end{aligned}$$

kde $m_1, m_2, \ldots, m_N \geq 2$. Tato soustava má řešení právě tehdy, když $\gcd(m_i, m_j)$ dělí $a_i - a_j$ pro všechna $i \neq j$. Pokud řešení existuje, je určeno jednoznačně modulo $\operatorname{lcm}(m_1, m_2, \ldots, m_N)$.

2.3 SP-9 (KAB)

Asymetrické kryptosystémy (šifra RSA, Diffie-Hellman, RSA digitální podpis), hešovací funkce (SHA-2, HMAC). Digitální podpis. Certifikáty, certifikační autority.

Asymetrické šifry

- pro šifrování a dešifrování se používají různé klíče
- pomocí privátního klíče šifrujeme, promocí veřejného klíče dešifrujeme
- privátní klíč nelze odvodit z veřejného klíče v rozumném čase
- každý subjekt má svůj vlastní pár veřejný-privátní klíč

princip RSA

- šifrovací systém založený na modulárním umocňování
- dvojice (e, n) je veřejný klíč (e — veřejný exponent, n — modul)
- n je součinem dvou velkých prvočísel p a q, tedy $n = pq$ a $\gcd(e, \varphi(n)) = 1$
- zašifrováni plaintextu — písmena se převedou na numerické ekvivalenty, vytvoří se bloky s největší možnou velikostí — $E(m) = c = |m^e|_n$, $0 < c < n$ (m je vytvořený blok plaintextu, c je výsledný blok ciphertextu)
- k dešifrování je nutná znalost inverze d čísla e modulo $\varphi(n)$ — $\gcd(e, \varphi(n)) = 1$, inverze tedy existuje — $D(c) = |c^d|_n = |m^{ed}|_n$, e a d jsou inverzní modulo $\varphi(n)$, tedy $ed \equiv 1 \pmod{\varphi(n)}$, tedy $ed = 1 + k \cdot \varphi(n)$, z toho $\Rightarrow |m^{ed}|_n = |m^{1+k\cdot\varphi(n)}|_n = |m \cdot (m^{\varphi(n)})^k|_n$, a dle Eulerovy věty $m^{\varphi(n)} \equiv 1 \pmod{n}$, tedy celkově $D(c) = |m|_n$
- dvojice (d, n) tvoří soukromý klíč
- existuje možnost, že m a n jsou soudělné, ale je extrémně malá

Generování RSA klíčů

- jak hledat p a q?
- subjekt nalezne 2 velká náhodná čísla p a q s 340 dekadickými číslicemi
- z věty o prvočíslech plyne, že pravděpodobnost toho, že takto vybraná čísla jsou prvočísla, je cca $2/\log(10^3 40)$
- pro nalezení prvočísla je potřeba v průměru $1 \,/\, (2/\log(10^3 40)) \approx 400$ testů takových čísel
- exponent e by následně měl být zvolen jako číslo větší než p a q — 2^e by mělo být větší než n, aby šifrování i dešifrování muselo používat redukci modulo n a nešlo pouze odmocnit

Bezpečnost RSA

- modulární umocňování potřebné k šifrování zprávy s použitím RSA může být provedeno při velikosti modulu a bloku cca 680 dekadických číslic v řádech milisekund
- dešifrovací exponent d nelze z dvojice (e, n) snadno odvodit, protože je potřeba znát hodnota $\varphi(n)$ — pro rychlé spočítání je třeba znát faktorizaci $n = pq$
- i v případě cca 100-číslicových p a q (tedy 200-číslicového modulu n) trvá nejrychlejším známým algoritmům faktorizace kolem 250 let
- náročnost prolomení je tím větší, čím větší je modul
- ochrana proti speciálním rychlým technikám faktorizace n: např. obě hodnoty $p - 1$ a $q - 1$ by měly mít velký prvočíselný faktor, tedy malé $\gcd(p - 1, q - 1)$ a rozdíl $p - q$ by měl být dostatečně velký

Digitální podpis s použitím RSA

- uvažujme 2 subjekty, každý má svůj pár privátní-veřejný klíč (PK_1, VK_1), (PK_2, VK_2)
- subjekt 1 chce poslat podepsanou zprávu

- subjekt 1 ji podepíše: $S = D_{PK_1}(m) = |m^{d_1}|_{n_1}$
- subjekt 1 zašifruje pro subjekt 2: $c = E_{VK_2}(S) = |S^{e_2}|_{n_2}$ (pokud $n_2 > n_1$, jinak je nutné rozdělit S do bloků o velikosti menší než n_2 před šifrováním)
- subjekt 2 nejprve dešifruje svým privátním klíčem, následně použije šifrovací transformaci veřejným klíčem subjektu 1 pro odkrytí původního obsahu
- subjekt 2 si je tedy jistý, že zpráva přišla od subjektu 1, a zároveň subjekt 1 nemůže popřít, že danou zprávu poslal

Urychlení RSA výpočtů

- urychlení šifrování — doporučená množina exponentů e s nízkou Hammingovou váhou
- urychlení dešifrování — použití Čínské věty o zbytcích (RSA-CRT) (rozklad čísel, počítá se s čísly poloviční délky)

Diffie-Hellmann

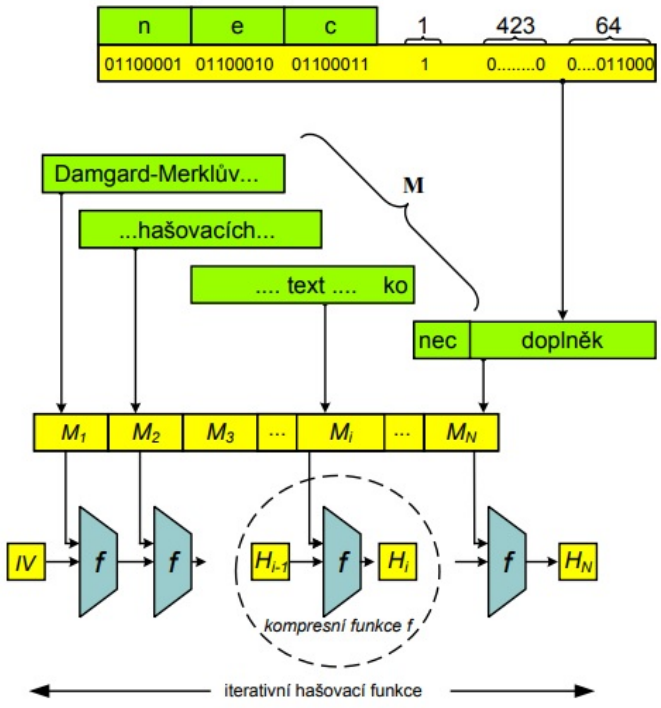
- algoritmus pro ustanovení společného klíče
- subjekt A a B si veřejně dohodnou prvočíslo m a bázi a $(1 < a < m)$, přesněji grupu řádu $m - 1$
- subjekt A si náhodně zvolí číslo k_A takové, že $0 < k_A < m$ a $\gcd(k_A, m-1) = 1$, spočítá $y_a = |a^{k_A}|_m$ a odešle ho B
- subjekt B si náhodně zvolí k_B takové, že $0 < k_n < m$ a $\gcd(k_B, m-1) = 1$, spočítá $y_B = |a^{k_B}|_m$ a odešle ho A
- A i B spočítají sdílený klíč $K = |{y_B}^{k_A}|_m = |(a^{k_B})^{k_A}|_m = |a^{k_A \cdot k_B}|_m = |(a^{k_A})^{k_B}|_m = |{y_A}^{k_B}|_m$
- útočník nemůže z $|a^{k_A}|_m$ ani z $|a^{k_B}|_m$ spočítat $K = |a^{k_A \cdot k_B}|_m$ — tzv. Diffie-Hellmanův problém, DHP
- není složitější než DLP (problém diskrétního logaritmu), a zdá se že ale není ani jednodušší

El Gamal

- vzniká úpravou Diffie-Hellman
- A zvolí číslo g a prvočíslo m, $1 < g < m$, a g je generátor odpovídající grupy řádu $m - 1$
- A si náhodně zvolí soukromý klíč k_A tak, že $0 < k_A < m$, spočítá $y_A = |g^{k_A}|_m$
- A zveřejní uspořádanou trojici (m, q, y_A) jako veřejný klíč, k_A je soukromým klíčem
- B chce odeslat A zprávu p
- B si náhodně zvolí číslo k_B $(0 < k_B < m)$, spočítá $y_B = |g_B^k|_m$
- B spočítá sdílený klíč $K = |{y_A}^{k_B}|_m = |g^{k_A \cdot k_B}|_m$
- B zašifruje zprávu p pomocí vztahu $c = |p \cdot K|_m$
- B odešle A uspořádanou dvojici (y_B, c)
- A si spočítá $K = |{y_B}^{k_A}|_m = |g^{k_A \cdot k_B}|_m$
- A si spočítá $|k^{-1}|_m$ (přes REA)
- A dešifruje zprávu: $p = |c \cdot K^{-1}|_m = |p \cdot K \cdot K^{-1}|_m = |p|_m$

Hash funkce

- základní pojmy: jednosměrnost, bezkoliznost
- **Jednosměrnost:** $f : X \to Y$, je snadné z jakékoliv hodnoty $x \in X$ vypočítat $y = \mathrm{f}(x)$, ale je výpočetně nemožné pro náhodný obraz $y \in \mathrm{f}(X)$ najít vzor $x \in X$, aby $y = \mathrm{f}(x)$
- **Hashovací funkce** je jednosměrná (1. typu — neexistují zadní vrátka pro zpětný výpočet) a bezkolizní, zároveň pro různé vstupy vrací stejně dlouhý výstup (hash)
- **Orákulum:** libovolná stroj, který na základě vstupu odpoví nějakým výstupem, na stejný vstup odpovídá stejným výstupem
- **Náhodné orákulum:** na nový vstup odpovídá náhodným výběrem z množiny vstupů
- Hashovací funkce se má chovat jako náhodné orákulum (bezpečnostní vlastnosti)
- **Bezkoliznost 1. řádu:** Hash fce h je odolná proti kolizím 1. řádu, jestliže je výpočetně nezvládnutelné nalezení libovolných dvou různých (byť nesmyslných) zpráv M a M' tak, že $h(M) = h(M')$
- bezkoliznost se využívá k digitálním podpisům
- nepodepisuje se zpráva (moc dlouhá) ale její hash
- bezkoliznost zaručuje, že je složité nalézt 2 dokumenty se stejným hashem — proto lze podepisovat hash
- **Bezkoliznost 2. řádu:** Hash funkce h je odolná proti kolizi 2. řádu, jestliže pro jakýkoliv vzor x je výpočetně nezvládnutelné nalézt 2. vzor $y \neq x$ tak, že $h(x) = h(y)$
- **Odolnost pro nalezení kolize 1. řádu:** pokud se hash funkce s hashem délky n bude chovat jako náhodné orákulum, složitost nalezení kolize s 50 % pravděpodobností je $\approx 2^{\frac{n}{2}}$ (narozeninový paradox)
- **Odolnost proti nalezení kolize 2. řádu:** pokud se hash funkce s hashem délky n bude chovat jako náhodné orákulum, složitost nalezení 2. vzoru je $\approx 2^n$
- pokud pro konkrétní hash funkci lze nalézat vzory/kolize rycheji, hovoříme o prolomení hashovací funkce
- při hashování se typicky zpráva rozdělí na bloky, poslední blok se doplní (např. jednou 1 a pak samé 0, aby otisky zpráv co by na konci měly např. jen víc nul byly jiné)
- pro konkrétní bloky se využívá kompresní funkce f, která z předchozího kontextu H_{i-1} a bloku M_i vytvoří kontext H_i
- při konstrukci hash funkce dle následujícího obrázku bezkoliznost kompresní funkce zaručuje bezkoliznost celé hashovací funkce



- jako kompresní funkce se typicky používá bloková šifra — kontext H_{i-1} je vstup, blok M_i je klíč
- dle Davies-Meyerovy konstrukce se ještě po zašifrování přičte (XOR) původní kontext

HMAC

- nepadělatelný integritní kód zprávy
- pro zprávu se spočítá za použití tajného klíče K
- detekuje chyby při přenosu, zabraňuje neoprávněné změně zprávy

Algoritmus HMAC

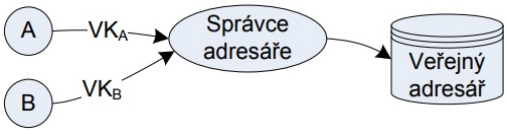
- Definujeme konstantní řetězce *ipad* jako řetězec $b/8$ bajtů s hodnotou `0x36 (0011 0110)` a *opad* jako řetězec $b/8$ bajtů s hodnotou `0x5C (0101 1100)`, kde b je velikost bloku v bitech.
- Klíč K v případě, že $\log_2 K < b$, doplníme bity 0 vlevo od MSB bitu klíče do délky b-bitů a označíme ho K^+.
- Definujeme hodnotu $HMAC_K(M)$ jako

$$HMAC_K(M) = H\Big((K^+ \oplus opad) \| H\left((K^+ \oplus ipad) \| M\right)\Big),$$

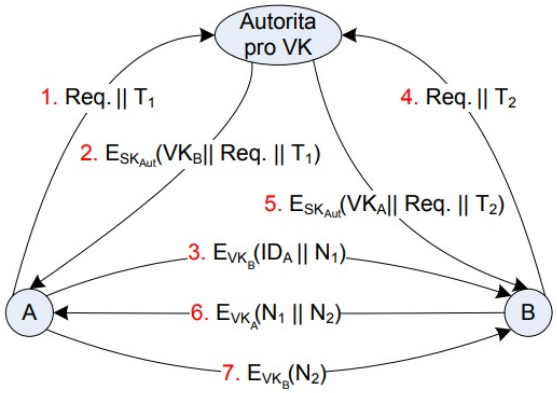
kde $\|$ označuje zřetězení.

Public Key Infrastructure

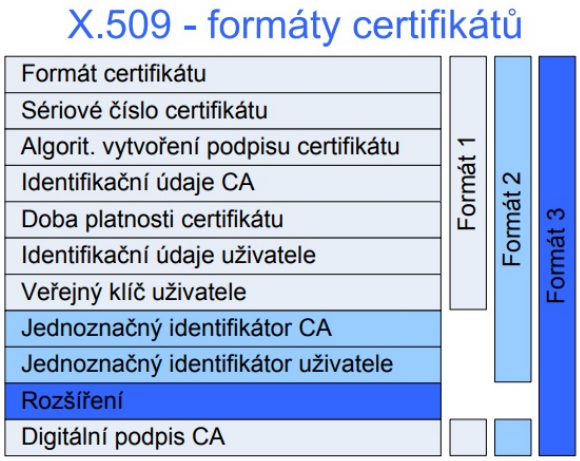
- jak distribuovat tajné symetrické klíče? distribuujeme veřejné klíče, následně si tajné klíče předáme za použití asymetrického šifrování
- jak distribuovat veřejné klíče? jak zabránit možnosti podvrhnutí?
- **Zveřejnění veřejných klíčů**
  - zasílání veřejných klíčů přímo
  - rychlé a jednoduché
  - není odolné proti podvržení
- **Veřejně dostupný adresář**
  - vyšší úroveň bezpečnosti
  - distribuci zajišťuje důvěryhodná autorita/správce



- **Autorita pro veřejné klíče**
  - přísnější dohled na distribuci veřejného klíče z adresáře
  - autorita má svůj pár veřejný-privátní klíč, každý účastník musí znát její veřejný klíč



- **Certifikace veřejných klíčů**
  - distribuce veřejného klíče bez kontaktu se třetí stranou
  - Certifikát: struktura obsahující veřejný klíč držitele, identifikační údaje držitele, dobu platnosti, další údaje vytvořené CA a zejména podpis CA
  - Certifikační Autorita (CA): důvěryhodná třeí strana, která na základě žádosti vydává a aktualizuje certifikáty — certifikáty vytvořené CA lze ověřit jejím veřejným klíčem



- certifikáty mohou být podepsané ve stromové struktuře — certifikát držitele podepsaný CA_1, její certifikát podepsaný CA_2...
- kořenové certifikáty musí být distribuovány jinak (typicky např. s operačním systémem)

2.4 SP-10 (KAB)

Symetrické šifry blokové a proudové, základní parametry, operační módy blokových šifer, jejich základní popis a slabiny.

Symetrické šifry používají pro operaci šifrování a dešifrování stejný klíč (případně snadno převeditelný).

Proudové šifry

- proudová šifra zpracovává každý znak zvlášť
- na každý znak je použita jiná šifrovací transformace E_{h_i}, která je posunem v abecedě o h_i znaků
- posloupnosti $h_1, h_2, ...$ se říká key-stream, a je vygenerována z klíče
- synchronní šifry: key-stream je závislý pouze na klíči (vypadne znak — zbytek textu už nerozšifrujeme)
- asynchronní šifry: key-stream je závislý jak na klíči, tak na předchozích zpracovaných datech (takže na OT a/nebo ŠT) (vypadne znak — po n znacích jsme schopni zase dešifrovat správně)
- příklad šifer: RC4, Salsa20, ChaCha, A5/1

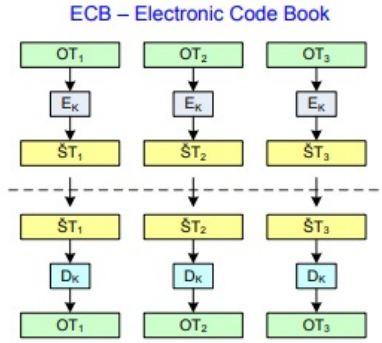
Blokové šifry

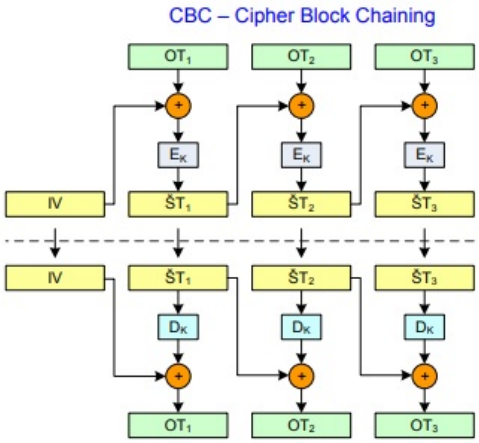
- šifrují t znaků najednou (bloky délky t)
- všechny bloky jsou šifrovány tou samou transformací
- DES, 3DES — 64b blok, DES 56b klíč, 3DES 112b/168b klíč
- AES — 128b blok, 128b/192b/256b klíč
- šifra Feistelova typu — postupná aplikace relativně jednoduchých operací, vytvoří poměrně složitý algoritmus
- iterativní bloková šifra — základem je jednoduchá funkce provádějící šifrování jednoho bloku, která je několikrát zopakována na tom samém bloku — jedna iterace se jmenuje 1 runda
- DES — 16 rund, z 56b klíče se expandují rundovní klíče, v praxi nevýhoda krátkého klíče
- 3DES — kombinace 3 operací DES za sebou, s použitím 2 nebo 3 různých DES klíčů (E_{K_1}, D_{K_2}, $E_{K_{1/3}}$)
- AES — 10/12/14 rund v závislosti na délce klíče — není Feistelova typu, nemá slabé klíče, odolná proti různým útokům
  - stav = 1 blok (16B) v matici 4x4
  - Expanze klíče — z klíče se odvodí rundovní klíče
  - AddRoundKey (xor rundovního klíče se stavem)
  - Iterace/runda: SubBytes (náhrada bytů dle tabulky), ShiftRows (posunuti bytů cyklicky v řádcích), MixColumns (vynásobení matice maticí), AddRoundKey
  - v poslední rundě je vynechána MixColumns

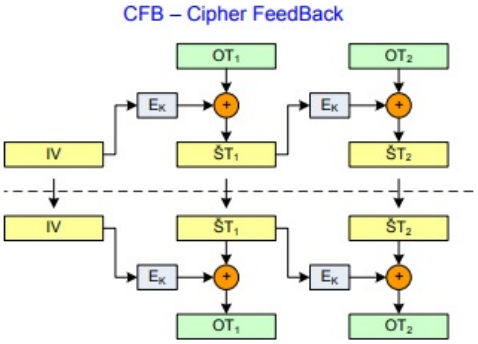
Operační módy

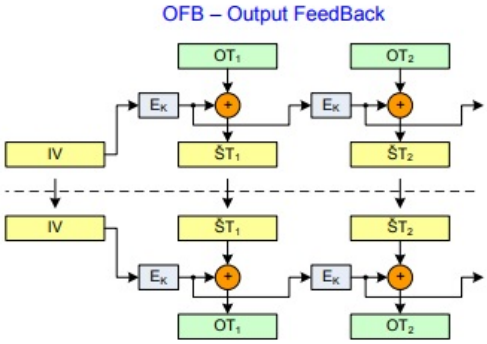
- operační módy blokových šifer nám mohou zlepšit vlastnosti blokových šifer
- ECB (Electronic Codebook)
  - bloky jsou šifrovány i dešifrovány nezávisle na sobě
  - stejný blok OT má stejný obraz v ŠT
  - lze zasahovat do dat — odebírat bloky a přehazovat
  - 1 bit chyby v ŠT poškodí celý blok OT
- CBC (Cipher Block Chaining)
  - je potřeba IV
  - každý blok OT se nejprve sečte (xor) s předchozím blokem ŠT (první IV) a následně zašifruje

  - 1 bit chyby v ŠT poškodí celý aktuální blok OT a 1 bit z následujícího
- CFB (Cipher Feedback)
  - je potřeba IV
  - dělá z blokové šifry proudovou (asynchronní/samosynchronní)
  - každý blok OT se sečte (xor) s přechozím blokem ŠT, který byl nejprve zašifrován klíčem (IV pro první blok)
  - 1 bit chyby v ŠT poškodí celý příští blok OT a 1 bit z aktuálního
- OFB (Output Feedback)
  - je potřeba IV
  - dělá z blokové šifry proudovou (synchronní)
  - každý blok OT se sečte (xor) s aktuálním heslem z key-streamu, key-stream je nezávislý na OT i ŠT, postupně se generuje z IV operací šifrování
  - 1 bit chyby v ŠT poškodí 1 bit v aktuálním bloku OT



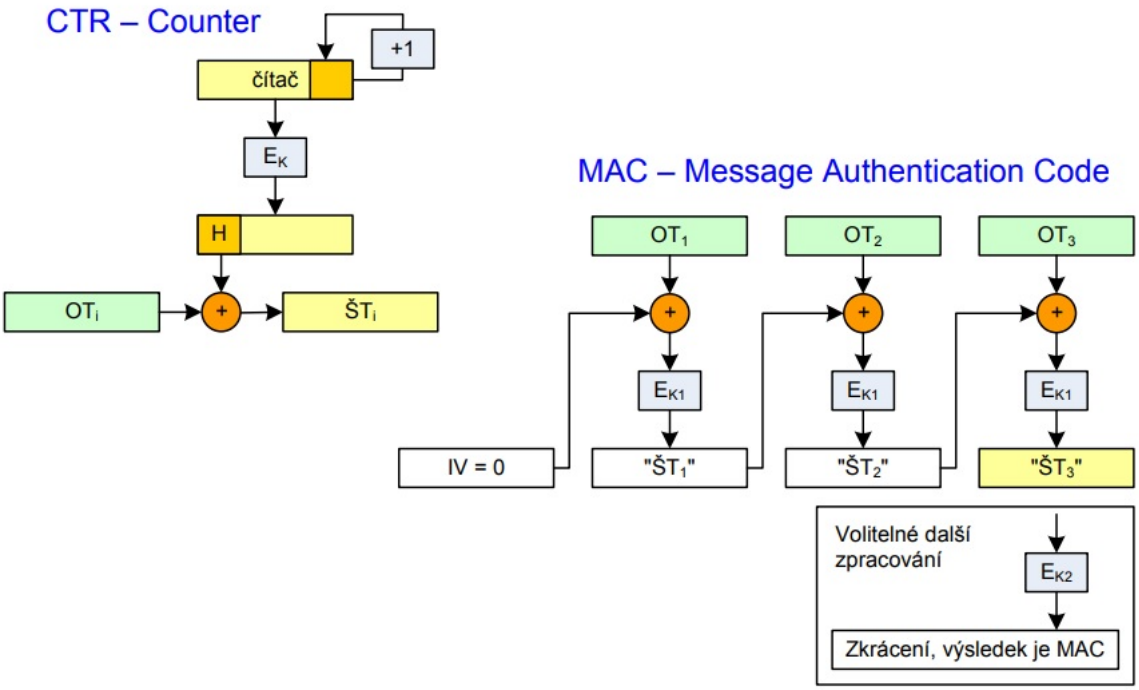






- CTR (čítačový mód)
  - dělá z blokové šifry proudovou (synchronní)
  - každý blok OT se sečte (xor) s aktuálním heslem z key-streamu, key-stream je nezávislý na OT i ŠT, postupně se generuje použitím čítače (načte se IV, poté se něco přičítá)
  - 1 bit chyby v ŠT poškodí 1 bit v aktuálním bloku OT
  - má zaručit maximální periodu hesla
  - v žádných zprávách šifrovaných tímtéž klíčem nesmí dojít k vygenerování stejného bloku hesla vícekrát — obsah čítače nesmí být stejný
- MAC (message authentication code)
  - proudové i blokové šifry zajišťují důvěrnost, ne integritu zpráv
  - MAC zajišťuje integritu a původ zprávy

– použije se jiný klíč než k šifrování
– funguje jako CBC s nulovým IV, průběžný ŠT se neodesílá
– MAC je tvořen posledním blokem ŠT_n



3 Obecná bezpečnostní teorie

4 Matematika

5 Programování