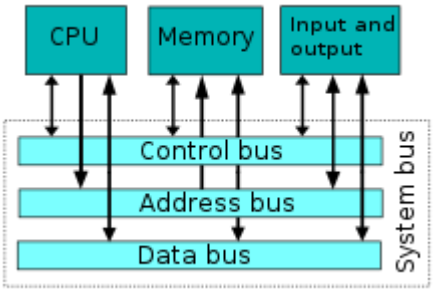
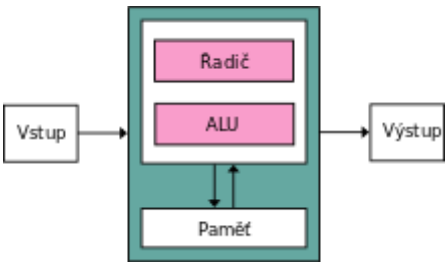
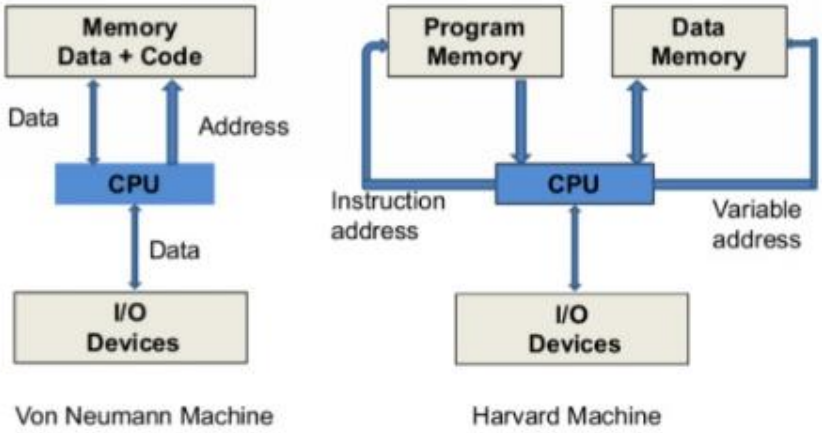
1) Von Neumannův model

Jedná se o základní model počítače. Architektura je popsána jako sekvenční stroj. Program i data uloženy ve stejné paměti. Příkazy jsou vykonávány v řadě za sebou. Schéma se skládá ze sběrnice, paměti, CPU s ALU a řadičem a I/O jednotka. Sběrnice je propojuje a skládá se ze tří sběrnic: řídící, adresové a datové.







2) CPU (central procesing unit)

Centrální procesní jednotka se intuitivně chápe jako nejdůležitější součást počítače. Neformálně se jí říká procesor. V současné době se CPU nejčastěji implementuje **mikroprocesorem**. Úlohou CPU je vykonávat instrukce uložené v operační paměti. Procesor obsahuje ALU, která vykonává aritmetické a logické operace, řídící jednotku (též CU => řídí chod procesoru) a registry, což jsou velice rychlé paměťové buňky v procesoru. Každý procesor má svou vlastní instrukční sadu, což je sada instrukcí, kterým rozumí. Instrukční sada je v podstatě na výrobci daného procesoru, proto bývají mnohdy dost velké odlišnosti v názvech.

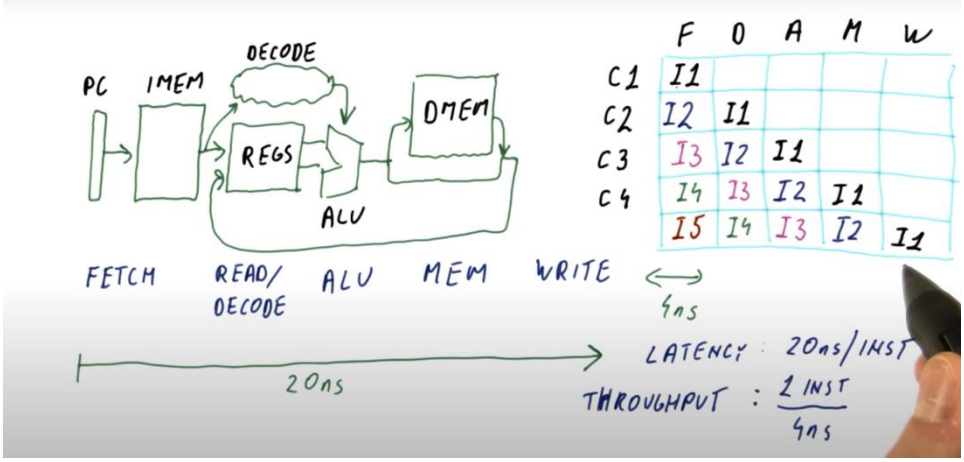
3) Instrukce

Jsou to příkazy v programu uložené v operační paměti. Každý procesor má svou instrukční sadu, což jsou instrukce/příkazy, kterým procesor rozumí. Jedním z typu instrukcí jsou například řídící instrukce, které mohou změnit instruction pointer registr, čímž se provede skok v programu. Instrukce má svůj číselný kód, a také název, který je určen pro lidi. Instrukce se obvykle zapisují na samostatné řádky. Každá instrukce může (ale nemusí mít) nějaké operandy (něco jako parametry při volání funkce), obvykle jeden nebo dva. Operandem může být buď registr, přímá hodnota anebo paměť. Mnohdy je první operand jak vstupní, tak výstupní, tedy že se do něj uloží výsledek instrukce.

4) Vykonávání instrukcí

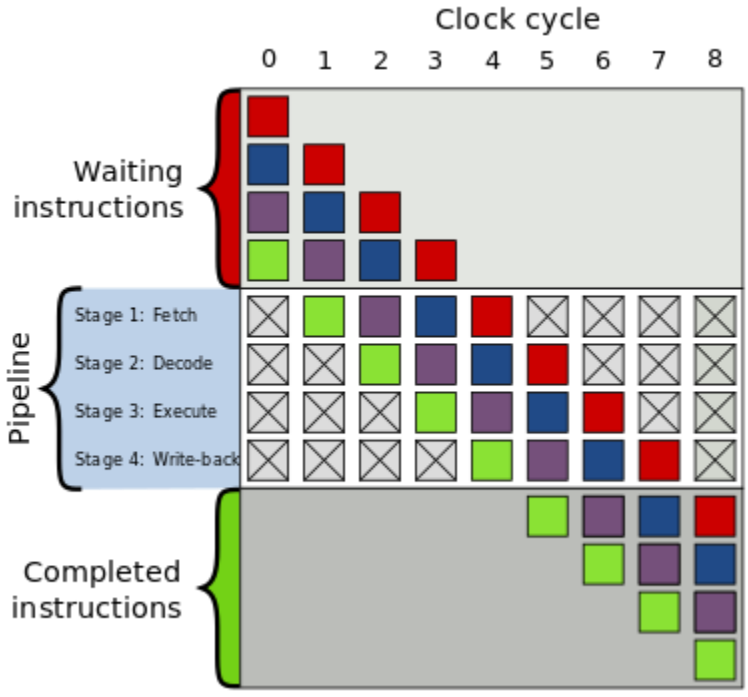
Vykonávání instrukce probíhá v sedmi krocích (obecné schéma)

1) Přesun na další instrukci
2) Načtení instrukce do CPU – načte se kód instrukce z paměti
3) Dekódování instrukce – zjistí se, co je to za instrukci
4) Výpočet adres operandů
5) Přesun operandů do CPU
6) Vlastní provedení operace
7) Uložení výsledku -> zpět k bodu 1.



PC = program counter | IMEM = Instruction Memmory | REGS = Registers | DMEM = Data Memmory | ALU = Arithmetic Logic Unit

Jedná se o model, který říká, že CPU má 7 samostatných částí, kdy každá z nich má zcela jinou úlohu, a CPU dělá vždy 1 z nich a zbylých 6 čeká. Moderní procesory jsou dělány



tak, aby se co nejvíce kroků dělalo paralelně. Kroky 1 – 4 se dělají v předstihu a v ideálním případě trvají 0 Tiků. Moderní procesory také mají více ALU, díky čemuž mohou pracovat současně. V dnešní době probíhá paralelní vykonávání instrukcí. Často tak skutečná rychlost programu nezávisí tolik na tom, jaký má procesor takt, ale na rychlosti paměťového čipu nebo způsobu, jakým překladač programovacího jazyka dokáže seskládat instrukce do řady za sebou, aby mohly být vykonány paralelně. Základní forma paralelizmu je tzv. **pipelining**.

To funguje tak, že každá ze 7 části CPU po vykonáni své práce dané instrukce, okamžitě přechází na další instrukci. V praxi ne každá část musí podporovat pipelining. Pokud ho podporuje tak se můžeme teoreticky dostat do stavu, kdy se jedna intrukce provede v každém tiku hodin. Pipelining je obvykle implementován hardwarově, a tedy navenek pipelining zaveden není. Pipelining, neboli zřetězené zpracování či překrývání strojových instrukcí je způsob zvýšení výkonu procesoru současným prováděním různých částí několika strojových instrukcí. Základní myšlenkou je rozdělení zpracování jedné instrukce mezi různé části procesoru a tím i umožnění zpracovávat více instrukcí najednou. Druhým modelem je **superskalární architektura**. Procesory musí obsahovat více než jednu ALU. Procesor díky tomu může zpracovávat dvě nebo více instrukcí současně. Více ALU umožňuje lépe zvládnout podmíněné skoky. Projeví se to tak, že každá ALU jde svým vlastním směrem, aniž by předem věděla, jestli to nedělá zbytečně, jakmile se zjistí, která podmínka platí, tak výpočet, který není potřebný je anulován. Z toho důvodu může jedna instrukce probíhat méně než 1T.

5) Instrukční sada

Instrukční sada, je soubor instrukcí, kterým procesor rozumí a umí je vykonávat.

Z hlediska instrukční sady rozlišujeme 2 základní skupiny procesorů:

a) **CISC** (complex instruction set computing) – **procesor má úplnou sadu instrukcí**. Složité procesory nabízejí velké množství instrukcí a běžné operace umějí provádět přímo v operační paměti (v RAM paměti). Instrukce v těchto procesorech často přímo podporují operace prováděné vyššími programovacími jazyky (práce se zásobníkem, lokálními proměnnými, funkcemi apod.). **Jedna instrukce obvykle trvá déle než 1T**. A jedná se o řadu **x86 a x64**. Označení complex vyjadřuje skutečnost, že strojové instrukce pokrývají velmi široký okruh funkcí, které by jinak šly naprogramovat pomocí jednodušších již obsažených strojových instrukcí (například násobení je možné nahradit sčítáním a bitovými posuny). Opakem procesorů CISC jsou procesory RISC, které obsahují redukovanou instrukční sadu.

b) **RISC** (reduced instruction set computer) - redukována sada instrukcí. Vychází z teze, že stejného výsledku je možné dosáhnout posloupnosti jednoduchých instrukcí. Procesory tedy mají základní instrukce. Vykonávání jedné instrukce obvykle trvá 1T. Příklady řad jsou ARM a SPARC.

Klasické procesory používaly úplnou sadu instrukcí, ale jejich složitost přiměla výrobce, aby přešli na RISC technologii. Dnešní procesory jsou tak ve skutečnosti CISC, ale rozkládají operace na mikrooperace, takže vypadají jako RISC. Vzhledem k tomu, že dnes jsou paměťové čipy poměrně levné a rychlé, takže na velikosti programu nezáleží.

6) Běh programu

Běh programu probíhá tak, že program provádí jednu instrukcí za druhou, pokud někde není skok. Skoky jsou buď podmíněné nebo nepodmíněné. Není zde přítomna něco jako operace „IF“, ta je implementována pomocí skoků. Skok způsobí to, že program pokračuje na jiném místě, než se aktuálně nachází, ruší se tím tedy návaznost po jednotlivých řádcích.

7) Registry

Procesor velmi často přesouvá data z **operační** pamětí do **registrů**, aby je mohl zpracovat (např. aritmetickými strojovými instrukcemi). Protože je počet **registrů v** procesoru omezen, jsou právě nepotřebná data z **registrů** zapisována zpět do **operační paměti**. Registry jsou paměťové buňky přímo v procesoru. Jsou extrémně rychlé. Procesor jich má obvykle velmi malý počet. Slouží k uchováni právě zpracovaných dat. Jsou zde také speciální registry, ke kterým programátor nemusí mít přímý přístup. Například **Instruction Pointer** (registr, který ukazuje na další instrukci). **FLAGS**, registr příznaků. Jeho jednotlivé bity mají smysl příznaků. Například Overflow Flag, pokud došlo k přetečení aritemtické operace. Zero Flag, pokud je výsledek operace 0. Sign Flag, pokud je výsledek záporný a další… U x86 mají registry 32 bitů. Některé z nich jsou obecně použitelné, ale existují určité konvence, jak by se registry měly používat. Procesory rodiny AMD64 nabízejí následující registry:

64bitové: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` až `r15`,

32bitové: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `r8d` až `r15d`,

16bitové: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `r8w` až `r15w`,

8bitové: `ah`, `al`, `bh`, `bl` `ch`, `cl`, `dh`, `dl`, `r8b` až `r15b`.

x64 assembly code uses sixteen 64-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 32-, 16- or 8-bit registers. The register names are as follows:

8-byte register	Bytes 0-3	Bytes 0-1	Byte 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

8-byte (1B = 8b) 8 * 8 = 64bits
0-3bytes (0, 1, 2, 3) => 4 then 8 * 4 = 32bits
0-1bytes (0, 1) => 2 then 8 * 2 = 16bits
0bytes (0) => then 8 * 1 = 8bits

```
          64
     /          \
    32          32
          /          \
         16          8
```

8) Skok

Společným nástrojem pro implementaci řídících struktur jsou takzvané skoky, které mohou přenést provádění výpočtu na novou adresu. Připomeňme, že prováděný program je uložený v paměti jako data a registr rip ukazuje na adresu instrukce, která se má provádět jako další. Standartně jsou instrukce prováděny v řadě za sebou, avšak

změna hodnoty v registru rip může procesoru určit, jaký kod se bude provádět jako další. Registr rip je řídící a nelze jej měnit přímo, ale pomocí instrukcí skoku to lze. Skoky dělíme na **podmíněné a nepodmíněné**. **Nepodmíněné skoky** vždy převedou řízení výpočtu na zadanou adresu a provádíme instrukcí **jmp a umožňuje nám skok na libovolné místo v programu**. Naprotitomu **podmíněné skoky jsou realizovány pouze**

instrukce	alt. jméno	příznaky	podmínka
`jg`	`jnle`	(SF = OF) & ZF = 0	$A > B$
`jge`	`jnl`	(SF = OF)	$A \geq B$
`jl`	`jnge`	(SF $\neq$ OF)	$A < B$
`jle`	`jng`	(SF $\neq$ OF) nebo ZF = 1	$A \leq B$

v případě, že je splněna podmínka, jinak program pokračuje na následující instrukci. V registru ef, který nám pomáhá při podmíněných skocích jsou tyto čtyři příznaky ZF (zero flag – vysledek byl nula), CF (carry flag – výsledek je větší/menší než největší/nejmenší možné číslo), SF (sign flag – výsledek je nezáporný => 0, nebo záporný => 1), OF (overflow flag – příznak přetečení znaménkové hodnoty mimo daný rozsah). Pro každý z těchto čtyř příznaků existují dvě instrukce, které provedou skok, pokud je příznak nastaven, nebo nenastavcen.

9) Volání podprogramu

Volání funkcí, nebo obecně podprogramů, obnáší celou řadu činností, jak na straně volajícího, tak na straně volaného kodu, tj. kodu funkce. Podprogram jsou v podstatě funkce/metody. Jejich volání je nejčastějším úkolem, který program po procesoru vyžaduje. Od volání podprogramů obvykle očekáváme předání parametrů do podprogramu, předání návratové hodnoty z podprogramu a umožnění rekurze. Předávání parametrů u x86 se dá dělat několika způsoby. Buď pomocí registrů nebo přes zásobník. Na zásobník je konvence C (zprava doleva). C umožňuje proměnlivý počet parametrů. Návratová hodnota vždy na registr eax. Volací konvence na platformě AMD64 a unixových os jsou postaveny na několika málo jednoduchých pravidlech, která si shrneme následovně:

argumenty jsou předávány přes registry (`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`), zbývající argumenty přes zásobník (zprava doleva),

o odstranění hodnot ze zásobníku se stará volající funkce,

registr `al` obsahuje počet argumentů s plovoucí řádovou čárkou,

hodnoty na zásobníku jsou zarovnány na 8 B,

obsah registrů `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11` není při volání funkce zachován (caller-saved registry),

obsah registrů `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15` musí být před a po zavolání funkce stejný (callee-saved registry).

10) Přerušení

Důležitý prvek počítače. Týká se hardwaru i softwaru. **Původně sloužilo zejména k reagování na nepředvídatelné události.** Například na **stisk klávesy nebo myši**. Když nějaké I/O (input/output) zařízení chce něco sdělit procesoru, tak vyvolá přerušení.

Činnost procesoru se přeruší a přejde do speciálního podprogramu zvaného obsluha přerušení. Přerušení nastává vždy po vykonání celé instrukce (s výjimkou neošetřitelné chyby). Po skončení přerušení procesor pokračuje tam, kde přestal. Přerušení může být přerušeno jiným přerušením, ale za určitých podmínek. Některé přerušení lze, jiné nelze blokovat. Také je zde systém priorit. Pokud běží přerušení s vyšší prioritou, tak přerušení s nižší nemůže přerušit. Je zde tedy řadič přerušení. Před vstupem do podprogramu přerušení si procesor ukládá důležitá data, aby mohl pokračovat. U x86 je celkem 256 přerušení. Systém přerušení se mimo jiné **používá k ošetření výjimek (dělení nulou, neplatné operace), debugování, krokování**.

11) DMA (direct memory access)

Technika, umožňující I/O zařízením přímo přistupovat do paměti. Musí ho podporovat jak počítač, tak operační systém. Na počítačích to zajišťuje DMA Controller. Jedná se o zařízení, které umí komunikovat jak s I/O zařízeními tak s operační pamětí. Umožňuje to přenos dat, aniž by byl využíván kvůli tomu procesor. Problém je v tom, že tyto data plují po stejné sběrnici, kterou využívá právě procesor, může docházet ke kolizím. Dnes je největší problém v pomalé rychlosti a nemožnosti adresovat více jak 16MB dat. Dnešní sběrnice umožňují přímý přístup do paměti i bez DMA Controlleru. Hodně hardwarového zařízení využívá DMA, například ovladače pro diskovou jednotku, grafické karty, síťové karty, nebo zvukovky.

12) Režimy práce CPU

Dva základní typy. **Privilegovaný a neprivilegovaný**. Privilegovaný (**kernel mode**, master mode, supervisor mode), ve kterém běží jádro operačního systému, **je bez omezení**, tím pádem zde neprobíhají bezpečnostní kontroly. Vše ostatní běží v neprivilegovaném (user mode, **slave mode**), kdy jsou **některé funkce omezeny**, probíhá zde kontrola. Existují i další módy např. x86 má 4 módy označené jako ring 0-3. A u AMD 64 existují ještě dva speciální mody. **Long mode** - umožňuje spouštět 32 bitové aplikace v 64bit OS. **Legacy mode -** režim pro zajištění zpětné kompatibility.

13) Systémová volání

Jedná se o **komunikaci aplikace s jádrem OS pomocí přesně definovaného rozhraní**. **Slouží** to například **k přístupu k hardwarovým zařízením**. Jedná se o vyvolání služby operačního systému, ke které naše aplikace nemá přímý přístup. Přepne se tím v podstatě do režimu jádra přes výjimku. Když se budeme bavit o systémových volání, tak je nutno zmínit „**minimální program**“, což je pojem který nám říká, že každý os poskytuje uživatelským procesům sadu služeb, jako je vytvoření souborů, zápis/čtení, spouštění nového procesu apod. a tyto funkce jsou obvykle poskytovány jadrem os pomocí jednoznačně definovaného rozhraní. A když bychom chtěli implementovat „minimální program“, tak se musíme držet pravidla, že **každý program by měl obsahovat minimálně jedno systémové volání** a tím je volání exit (**sys_exit**), které se postará, že aktuálně běžící proces je ukončen.

14) Instrukční sada (CISC, RISC, registrové, zásobníkové procesory)

CPU má dvě koncepce a to registrové a zásobníkové. Registrové mají operandy uloženy v registrech (probíhá načtení a uložení dat z registru) a zásobníkové mají operandy uloženy na zásobníku, přidávání a odebírání hodnot přes push a pop, operace pracují

s vrcholem zásobníku. Obvykle bývá druhý zásobník pro volání funkcí, výrazně jednodušší instrukční sada.

15) Instrukční sada procesorů x86 (operace, registry, příznaky, předvídání skoků)

Registry jsou 32bitové, obecně použitelné EAX, EBX, ECX, EDX, mají určitou konvenci použití.
EAX – pro násobení a dělení, vstupně výstupní operace
EBX – nepřímá adresace paměti, přístup do paměti
ECX – počítadlo při cyklech posuvech a rotacích
EDX – uložení dat
Každý registr má svou 16 bitovou část reprezentovanou jako AX,BX, CX…, lze rozdělit na 2 8 bitové části AH, AL.
Další registry:
EDI – adresa cíle
ESI – adresa zdroje
EBP – adresace parametrů funkcí a lokálních proměnných
ESP – ukazatel na vrchol zásobníku
EIP – ukazatel na aktuální místo programu
EF – příznakový registr
Operace: operandy instrukcí mohou být registry, paměť a konstanty, přičemž paměť lze v jedné instrukci adresovat pouze jednou.
Příznaky: Jsou uloženy v registru EF, některé instrukce mění tyto příznaky. Programátor je nemůže změnit přímo. Příznaky jsou využívány podmíněnými skoky.
SF – sign flag, nastaven, pokud výsledek záporný (1)
ZF – výsledek je nula
CF – je nastaven, pokud při operaci došlo k přenosu mezi řády
OF – pokud dojde k přetečení
TF – trap flag, slouží ke krokování
DF – direction flag, ovlivňuje chování instrukcí blokového přístupu.

Předvídání skoků:
Procesory implementují různá řešení pro odhad, jestli daný skok bude proveden. Patří mezi ně **statický přístup**, **dynamický přístup** (rozhoduje se na základě historie skoků), nápověda poskytnutá programátorem (příznak v kódu). Procesory používají obvykle kombinace těchto metod, hlavně dynamický odhad. Má čtyřstavové počítadlo. Při každém průchodu procesor ukládá do Branch Prediction bufferu 2b příznak jestli byl skok proveden nebo ne a postupně přechází mezi čtyřmi stavy. Pravidelně se střídá úspěšnost.

16) Instrukční sada procesorů AMD64 (operace, registry, příznaky, předvídání skoků)

Jedná se rozšíření instrukční sady x86, které umožňuje 64bitové výpočty při zachování kompatibility se stávajícím softwarem x86. CPU může pracovat v 64bitovém režimu, kde byla změněna sémantika několika x86 instrukcí. Registry zde mají velikost 64 bitů (rax, rbx, rcx, rsi, rdi, rsb, rbp)
Nové 64 bitové registry r8 – r15

spodních 32 bitů jako registry rXd (r8d)

16 bitů rXw

8 bitů rXb

Nové 128 bitové registry xmm8-xmm15, některé procesory rozšiřují xmm0 – xmm15 na 256 bitů. Délka instrukce omezena na 15 bajtu. V operacích je možné používat jako konstanty maximálně 32 bitové hodnoty a je zde rozšíření adresního prostoru dále je zde fyzicky adresovatelných typicky 2^{36} až 2^{46} B paměti.

17) Instrukční sada ARM

Jedná se o 32 a 64 bitove procesory. Optimalizují se na nízkou spotřebu napájení a paměti. Několik variant instrukčních sad. ARMv1 – v9, dnes především v7 a v8. **Architektura big.Little – jedná se o kombinaci pomalejších úspornějších jader s výkonnými** big, která jsou využívaná podle aktuálního zatížení systému. **ARMv7** má podpora několika různých typů instrukčních sad, přímá podpora až 16 koprocesorů. Jedná se o l**oad/store** architekturu (což je způsob jak přistupovat do paměti => architektura load–store je architektura instrukční sady, která **rozděluje instrukce do dvou kategorií** a to na **přístup do paměti** (načítání a ukládání mezi pamětí a registry) a **operace ALU** (které se vyskytují pouze mezi registry). Má 32 obecně použitelných registrů, ale jen 16 v jeden okamžik. Všechny instrukce mají velikost 32 bitů. Argumenty jsou předávány přes registry a zásobník. **ARMv8** – 64bitový nástupce v7 má instrukce velikosti 32 bitů a módy pro zpětnou kompatibilitu, dále 31 64-bitových registrů.

18) Instrukční sada SPARC

Každá instrukce zabírá 32 bitů. Je zde snaha eliminovat množství operací přičemž operace mají 3 až operandy. Velké množství registrů (stovky), běžně dostupných 32. Stejně jako ARM i toto je load/store architektura. Jednoduché instrukce -> rychlejší zpracování. Skoky se neprovádí okamžitě. Zpracovává se i následující instrukce za operací skoku.

19) Předávání parametrů podprogramu

U archiektury x86 existují 3 konvence předávání parametrů. Obvykle se předává pomocí registrů, zbývající uložíme na zásobník. Také se dá předávat čistě přes zásobník.

Konvence C (cdecl)

Argumenty předávané čistě přes zásobník zprava doleva a argumenty ze zásobníku odstraňuje volající. Umožňuje funkce s proměnlivým počtem parametrů

Jak to vypadá v ASM:

```
push ebp;                   //na zásobník uložíme obsah ebp
mov ebp, esp;               //do ebp uložíme aktuální vrchol
mov eax, [ebp + 8];         //načteni prvního argumentu (2. arg +12)
inc eax;
pop ebp;                    //obnoveni ebp do původního stavu
ret;                        //návrat z funkce
```

Konvence Pascal

Také přes zásobník zleva doprava ale argumenty odstraňuje volaný. Neumožňuje proměnlivý počet parametrů.

Konvence fastcall (fastcall, msfastcall)

První dva parametry pomocí ECX, EDX, zbylé zprava doleva přes zásobník. Argumenty odstraňuje volaný. Mírně komplikuje proměnlivý počet parametrů. Návratová hodnota se předává obvykle přes EAX.

20) Struktura zásobníku z pohledu programu

Zásobník umožňuje snadno používat rekurzi. Budeme-li se bavit o x86 procesorech, tak ty mají jeden zásobník typu LIFO. Zásobník má podobu pole **4 bajtových** čísel a nic jiného než 4 bajtové hodnoty zde ukládat nemůžeme. Pokud například chceme uložit 64 bitové číslo, tak to vlastně znamená, že na něj uložíme adresu, která má vždy 4 bajty. Používají se operace push a pop, pro uložení a vyzvednutí ze zásobníku. Implementace pomocí pole je jednoduchá. Procesor používá registr ESP, který funguje jako ukazatel na vrchol zásobníku. Hodnota ESP je tedy adresa posledně uloženého prvku.

21) Operační paměť

Počítač si v operační paměti uchovává data, někdy i program. Operační paměť spadá do kategorie vnitřní, oproti hard-disku, který je vnější. Za operační paměť obvykle považujeme takové paměti, které se dají číst i zapisovat. Je to lineární struktura s pevnou délkou a náhodným přístupem. Procesor může číst a zapisovat libovolné paměťové buňky. Operační paměť neboli RAM patří mezi nejdůležitější parametry každého počítače, notebooku, serveru či mobilního zařízení. Větší kapacita RAM znamená plynulejší chod programů. Operační paměť RAM (Random Acces Memory), lidově též „ramka" je rychle přístupná elektronická paměť sloužící procesoru pro často využívaná data právě spuštěných aplikací. Na rozdíl od pevného disku nebo SSD operační paměť data neukládá, a tak se při vypnutí zařízení kompletně vymaže.

22) Reprezentace hodnot (celá čísla, s plovoucí řádovou čárkou, řetězce)

Při práci s **celými** čísly procesory nejčastěji používají doplňkový kód, který umožňuje snadno pracovat jak s kladnými, tak zápornými čísly. Změna znaménka probíhá tak, že se převrátí všechny bity a k poslednímu se přičte jednička (hezké video na vysvětlení => https://www.youtube.com/watch?v=4qH4unVtJkE). Může nastat **přetečení** nebo **podtečení**. Přetečení znamená, že výsledek je větší než maximální hodnota uložitelná do paměti. Výsledkem je tedy nejmenší možná hodnota. Podtečení je podobné. Pokud dojde k přetečení tak se do Carry Flag příznaku uloží 1. Pokud jde o vícebajtová čísla, tak je více způsobu, jak je uložit:

Big endian, ukládá čísla od nejvyššího řádu po nejmenší (stejně jako to zapisujeme na papír). Takto se to zapisuje u SPARC.

Little endian ukládání od nejnižšího řádu po nejvyšší. Výhoda tohoto typu je, že můžeme číslo zvětšovat bez změn adresy. Takto se to zapisuje u x86. (endianita => https://www.youtube.com/watch?v=T1C9Kj_78ek)

Bi-endian procesory podporující oba způsoby.

Binary coded decimal – málo používaná forma čísel, kdy byly mikroprocesory určené především pro kalkulačky. Každé 4 bity uchovávají jednu desítkovou cifru. (neefektivní). **Při práci s čísly s plovoucí řádovou čárkou označována též jako FP je nutno kodovat na tři části**. Čísla zabírají 8 bajtů. **Znaménko, mantisa a exponent** (hodnota = znaménko * mantisa * základ na exponent) (video pro vysvětlení => https://www.youtube.com/watch?v=Zzx7HN4wo3A). Základ má hodnotu 2. Toto kódování je jednotné pro všechny typy procesorů. Nevýhodou, některá čísla nejdou zcela přesně uložit při základu 2. Když má 8 bajtů je to dvojitá přesnost, 4 – jednoduchá přesnost, 10 – rozšířená přesnost. Existuje kladná a záporná nula, nekonečna a not a number. Při práci s řetězci je uložení **znaků** jednodušší než u čísel. Nejvíce rozšířené je kódování **ASCII**, která používá 7 bitů (velikost americké abecedy 2 ** 7 = 126). Osmibitová (2 ** 8 = 256) kódování obsahují i znaky národních abeced (není to jednotné). **Unicode** řeší tuto nevýhodu. Jedná se o světovou znakovou sadu společnou pro všechny národní abecedy. Definuje vazbu *číslo x znak*. Každý znak má 4 bajty. Dělí se na tzv. roviny (každá rovina obsahuje jiné znaky – základní obsahuje naše). Různé typy kódování:
UCS universal character set. Má pevně danou velikost. Dělí se na UCS-2 a UCS-4. UCS-2 pojme základní rovinu Unicode a UCS-4 všechny.
UTF-8 kódování s proměnlivou délkou, zpětně kompatibilní s ASCII.
UTF-16 proměnlivá délka, rozšiřuje UCS-2.

23) Adresování paměti *

Adresace paměti je určení, se kterou buňkou chceme pracovat. Každá buňka má svou vlastní adresu, která má velikost od 0 do velikosti paměti – Tento způsob se nazývá **lineární adresování paměti**. Současné procesory rozdělují lineární adresu na 2 složky a to báze a posunutí. U architektury x86 je tento princip realizován pomocí tzv. **segmentového adresování**, adresa se vypočítá jako segment + offset. Novější procesory mají další adresovací režimy, kdy se dá adresovat i 4 GB paměti. Zde se používají opět dvě složky a to selektor + offset. Selektor má 16 bitů a ukazuje do tabulky deskriptorů. Deskriptor má záznam v tabulce, který popisuje segment. Tabulky deskriptorů nemůže program ovlivnit je to v kompetenci procesoru. Procesor má jednu globální tabulku GDT, na kterou ukazuje registr GDTR.
(adresování => https://www.youtube.com/watch?v=TGcjn8zMhfM)

24) Typy adres

U architektury x86 je možno určit adresu 2 způsoby.

Přímá adresa – ukazuje na pevné místo v paměti. Používá se zejména pro určení adres globálních proměnných a podprogramů, které jsou vždy na stejném místě v paměti.

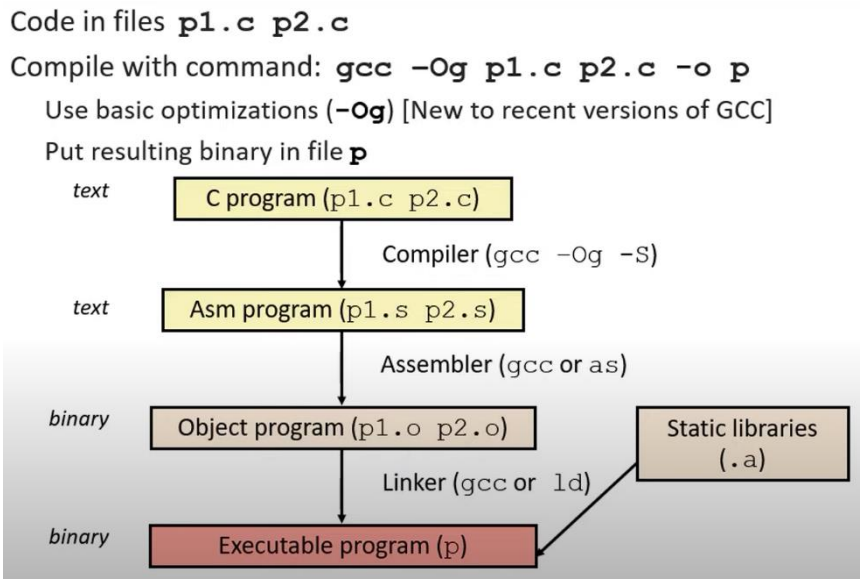
Nepřímá adresa – ukazuje na místo v paměti nepřímo. Adresa se vypočítá z hodnot registrů a přičtením posunutí. Adresování je pomalejší ale flexibilní. 32-bitové procesory mají tento vzorec: *adresa = posunutí + báze + index * faktor*. Jednotlivé složky jsou nepovinné. Báze a index jsou registry. Posunutí je přímá hodnota a faktor číslo 1, 2, 4 nebo 8
(adresování => https://www.youtube.com/watch?v=TGcjn8zMhfM)

25) Překlad programu

1) Preprocesor expanduje makra (makro představuje skupinu běžně používaných příkazů ve zdrojovém programovacím jazyce, dále makro procesor nahradí každou makro instrukci odpovídající skupinou příkazů zdrojového jazyka a toto je známé jako expanze maker), odstraní nepotřebný kód a načte požadované hlavičkové soubory (např math.h), proběhne deklarace struktur, prototypů atd..
2) Překladač vygeneruje kód v assembleru.
3) Assembler vygeneruje objektový kód.

```
        00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   01234
000000  CA FE BA BE 00 00 00 32 00 2C 07 00 02 01 00 09   .....
000010  54 65 73 74 41 72 72 61 79 07 00 04 01 00 10 6A   TestA
000020  61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 01   ava/l
000030  00 06 3C 69 6E 69 74 3E 01 00 03 28 29 56 01 00   ..<in
000040  04 43 6F 64 65 0A 00 03 00 09 0C 00 05 00 06 01   .Code
000050  00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C   ..Lin
000060  65 01 00 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C   e...l
000070  65 54 61 62 6C 65 01 00 04 74 68 69 73 01 00 0B   eTabl
000080  4C 54 65 73 74 41 72 72 61 79 3B 01 00 04 6D 61   LTest
000090  69 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E   in...
0000A0  67 2F 53 74 72 69 6E 67 3B 29 56 07 00 11 01 00   g/Str
0000B0  02 5B 49 09 00 13 00 15 07 00 14 01 00 10 6A 61   .[I..
0000C0  76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 0C 00   va/la
0000D0  16 00 17 01 00 03 6F 75 74 01 00 15 4C 6A 61 76   .....
```

4) Linker sloučí několik souborů s objektovým kódem + knihovny do spustitelného formátu.



Některé kroky mohou být vynechány, některé vyšší jazyky dělají nejdříve překlad do nižšího jazyka např do C. **Objektový soubor** je specifický pro každý OS. Obsahuje hlavičku, objektový kód, exportované symboly, importované symboly, informace pro přemístění, debugovací informace. Dělí se na 3 sekce a to jsou kód, data jen pro čtení a inicializovaná data. **Linkování** spojuje jednotlivé objektové soubory do spustitelného formátu
stará se o správné umístění kódu a vyřešení odkazů na chybějící funkce a proměnné.
připojení knihoven (hlavičkové soubory většinou neobsahují žádný kód)

26) **Knihovny (statické, dynamické linkování)**

Staticky linkované knihovny slouží jako archiv objektových souborů. Výhodou je jednoduchá implementace, nulová režie při běhu aplikace, žádné závislosti a nevýhody jsou velikost výsledného binárního souboru, aktualizace knihovny. Obecně statická knihovna, případně staticky linkovaná knihovna, je pojem, kterým se označuje knihovna, jejíž kopie je linkerem zahrnuta přímo do výsledného spustitelného souboru na pevně danou adresu. Tím se odlišují od dynamických knihoven, které jsou ve vnější paměti uloženy odděleně a do operační paměti jsou nahrávány až dynamickým linkerem při startu programu. Výhodou staticky linkovaných knihoven je, že je program má v sobě a nemusíme spoléhat na to, že je v dané instanci os k dispozici jejich vhodná verze, nebo nemusíme myslet na to, že je potřeba jejich doinstalace. Program pak z hlediska knihoven na ničem nezáleží a nedochází tak k DLL HELL (závislostnímu peklu). Nadruhou stranu neumožňují aktualizace nezávislé na programu. Při potřebě opravit chybu/y v knihovně/nách je potřeba udělat to pro každý program. V běžném stolním počítači jsou obvykle široce využívány knihovny linkované dynamicky, zatímco staticky bývají linkované jen knihovny, které dodává k programu sám jeho tvůrce. Naopak v prostředí vestavěných systémů, kde je obvykle aktualizován celý operační systém včetně programů jako celek, nepředpokládá se instalace množství programů uživateli a režie při nahrávání všech knihoven jako dynamických by při startu programů měla příliš velký vliv na výkon systémů, je statické linkování běžnější.

Dynamicky linkované knihovny jsou načteny až při spuštění programu. Mají dobrou vlastnost, ktrá spočívá ve sdílení kódu mezi programy. Nutnost provázat adresy v kódu s knihovnou a nutná spolupráce s OS. Problém s DLL HELL => DLL peklo (anglicky DLL hell) je označení pro komplikace, které v operačním systému Microsoft Windows způsobuje používání dynamických knihoven (DLL) – nemožnost spuštění některých programů kvůli chybějícím knihovnám, chybné fungování programů kvůli nekompatibilním verzím knihoven nebo hromadění nevyužívaných knihoven a jejich verzí. Problémy byly závažné zejména u starší 16bitové verze, ve které všechny aplikace používají stejný paměťový prostor. DLL peklo je konkrétní forma obecného problému závislostního pekla (anglicky dependency hell).

27) Běhová prostředí (JVM, CLR, aj.)

Běhové prostředí (anglicky run-time environment) je v informatice skupina software, určená na podporu realizace počítačových programů napsaných v některém z programovacích jazyků. Toto běhové prostředí poskytuje softwarové služby jako podprogramy a knihovny pro společné operace, provádění příkazů programovacího jazyka, typové kontroly, ladění a dokonce i generování a optimalizace kódu. (vysvětlení => https://www.youtube.com/watch?v=gQdZq_SRvE8)

Java Virtual Machine (JVM)

Je to sada programů a datových struktur. Probíhá překlad Javy na Java bytecode (JBC). JBC vykonáván pomocí JVM. Implementace JVM není definovaná, pouze se specifikuje chování. JBC lze přeložit do strojového kódu nebo provést pomocí konkrétního CPU. JVM – virtuální zásobníkový procesor, který má malý počet instrukcí. Zásobník obsahuje rámce, který je vytvořen při zavolání funkce. Obsahuje lokální proměnné, mezivýpočty a operand stack- k provádění výpočtů, dále obsahuje heap s automatickou správou paměti. Hodnoty menší, než int se převádí na int. Existují zde pouze relativní skoky.

Common Language Runtime (CLR – běhové prostředí)

Je to velice podobné Microsoft .NET, kde .NET nepředepisuje použití žádného programovacího jazyka - vždy přeloží do mezijazyka CIL (Common Intermediate Language). Zdrojový kód -> Common Intermediate Language -> bytecode -> strojový kód. Je navržen s podporou více jazyků a při prvním zavolání metody se provede překlad do strojového kódu CPU.

28) Architektura jednotlivých OS

Operační systém by měl umět spravovat a sdílet procesor, spravovat paměť, umožnit komunikaci mezi procesy, obsluhovat zařízení a organizovat data. Rozhodně nechceme aby si každý proces implementoval tuto funkcionalitu po svém a dále není žádoucí aby každý proces měl přístup ke všem druhům a možnostem hardwaru.

Jádro typy:

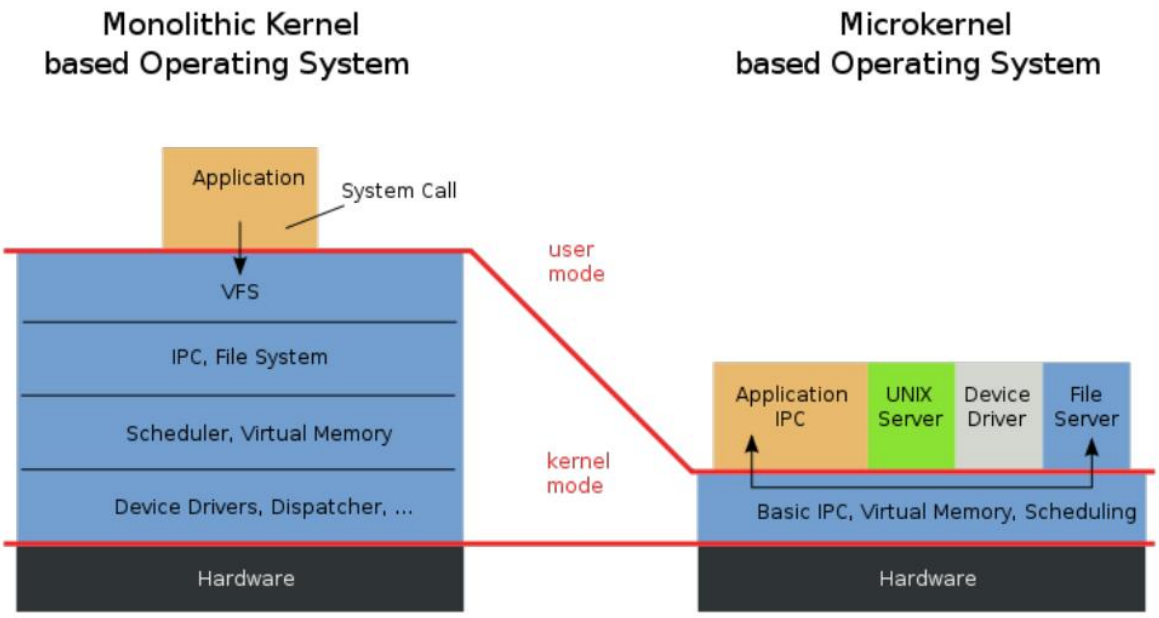
Monolitické jádro je druh jádra operačního systému, jehož veškerý kód (nebo jeho převážná většina) běží ve stejném (jaderném) paměťovém prostoru, který se anglicky označuje jako kernel space. Tím se liší od tzv. mikrojádra, které většinu tradičních činností monolitického jádra, jako je třeba správa souborových systémů, implementuje v procesech, které běží v uživatelském paměťovém prostoru.

Přestože jsou monolitická jádra psána tak, aby byla činnost jednotlivých subsystémů oddělená, jsou jednotlivé části silně provázány. A navíc, protože sdílejí stejný paměťový prostor, může chyba v jednom subsystému zablokovat jiný, nebo dokonce shodit celé jádro. Na druhou stranu, pokud je dbáno na správnou implementaci jednotlivých částí, je monolitické jádro velice efektivní.

Rozhraní mezi operačním systémem a procesy zajišťují v monolitickém jádře tzv. systémová volání. Pomocí systémových volání mohou procesy využívat služby nabízené jádrem operačního systému, je-li jim to povoleno.

Architektura je zde vrstvená, máme zde moduly a všechny služby jsou pohromadě. **Typickým zástupcem je Linux**

Mikrojádro poskytuje správu adresního prostoru, procesů, IPC (meziprocesní komunikace). Pro oddělení serverů (služeb systému) -> běžné procesy se speciálními právy, což zvyšuje bezpečnost. Možnost restartu serverů. Meziprocesní komunikace je pomalá. Typickými zástupci jsou MINIX, QNX.



Hybridní jádro je kombinací obou přístupů. Část funkcionality v jádře, část mimo něj a **typickým zástupcem je Windows NT**.

Exokernel řeší jen to nejnutnější -> přidělování HW zdrojů. Neposkytuje HW abstrakci -> knihovny v uživatelském prostoru

Historie OS

UNIX začal os, který se jmenoval **MULTICS (MULTI**plexed **I**nformation and **C**omputing **S**ervice). Od začátku byl kladen velký důraz na současnou práci více uživatelů a jednotnou paměť. Je zde přítomná segmentace a stránkování společně s dynamickým linkováním.

základní vlastnosti:

- počítá s víceuživatelským přístupem
- zkušený uživatel, nejlépe programátor
- snaha o jednoduchost (všechno je soubor)
- omezení redundance
- možnost komponovat věci do větších celků
- transparentnost

rozhraní v UNIXU:

- vrstvená architektura a pojící prvky
- systémová volání
- volání knihoven
- uživatelské aplikace

WINDOWS původně vznikaly jako nadstavby na MS-DOSem. Kooperativní multitasking, softwarová virtuální paměť založená na segmentaci. Zlepšovala se práce s pamětí. Do příchodu Windows NT byl jednouživatelský

Windows NT má hybridní jádro, je zpětně kompatibilní s některými staršími verzemi.

obecné principy:

bezpečnost

spolehlivost

kompatibilita s ostatními systémy

přenositelnost

rozšiřitelnost

výkon

objektový přístup

implementovaný v C/C++

používá hybridní architekturu => oddělené procesy pro subsystémy + spousta funkcionality v jaderném prostoru

windows executive – část OS, která poskytuje funkce do uživatelského prostoru

Android a iOS jádra vychází z existujících OS (Linux).

29) Procesy

Proces (anglicky process) je v informatice název pro spuštěnou instanci počítačového programu. Je to obecný termín pro označení běžící program. Jakmile se spustí program (uložený například na disku), tak se vytvoří nový proces. Spuštěním více programů vznikne více procesů. Opakovaným spuštěním téhož programů také vznikne více procesů.

Proces je umístěn v operační paměti počítače v podobě sledu strojových instrukcí vykonávaných procesorem. Obsahuje nejen kód vykonávaného programu, ale i dynamicky měnící se data, která proces zpracovává. Program může na jednom, ale i více počítačích běžet v podobě jednoho či více procesů s různými daty.

Proces charakterizuje kód programu, paměťový prostor, data, zásobník, registry. Informace o procesu se nachází v tabulce procesů => PCB: **proces control block**

Procesy byly původně základní entitou vykonávající činnost. Procesy tvoří hierarchii a jednotlivé procesy můžeme indentifikovat pomocí PID.

30) Vlákna

Vlákno (též vlákno řízení, anglicky thread) je prvek reprezentující vykonávání kódu procesu. Mluvíme o nich právě v souvislosti s možností vytvořit více vláken v jednom procesu, což se nazývá multithreading. Jednotlivá vlákna mohou vrámci procesu vykonávat různé, nebo stejné činnsti nad různými, nebo stejnými daty. U počítačů s více procesory nebo na vícejádrových procesorech mohou tyto činnosti probíhat současně (paralelně) v ostatních případech kvaziparalelně (rychlým přepínáním mezi vlákny se vytváří zdání paralelismu). Jak už jsem zmínil je možno mít více vláken v rámci jednoho procesu. Vlákno můžeme považovat za odlehčený proces – zatímco běžné procesy jsou

navzájem striktně odděleny, vlákna jednoho procesu sdílí paměťový prostor a další systémové prostředky. Každé vlákno má své registry, zásobník, IP a stav. Jinak jsou zdroje sdílené. Vlákna sdílí stejné globální proměnné (data) => žádná ochrana => potřeba synchronizace. Využití se najde při rozdělování běhu na popředí a na pozadí, asynchronní zpracování dat a víceprocesorové stroje.

Různé typy vztahů (proces-vlákno):

1:1 – jsou to systémy kde proces = vlákno

1:N – systémy kde proces může mít více vláken (nejčastější řešení)

N:1/M:N – více procesů pracuje s jedním vláknem (hypotetické řešení)

Implementace vláken:

jako knihovna v uživatelském prostoru

součást jádra OS

kombinované řešení

green threads

V uživatelském prostoru:

proces se stará sám o přepínání vláken
má vlastní tabulku vláken
problém s plánováním v rámci OS

V jádře:

jádro spravuje pro každé vlákno struktury podobně jako procesy
řeší problémy s blokujícími voláními
vytvoření vlákna je pomalejší
přepínání mezi vlákny jednoho procesu je rychlejší než mezi procesy

31) Životní cyklus procesu

obecný

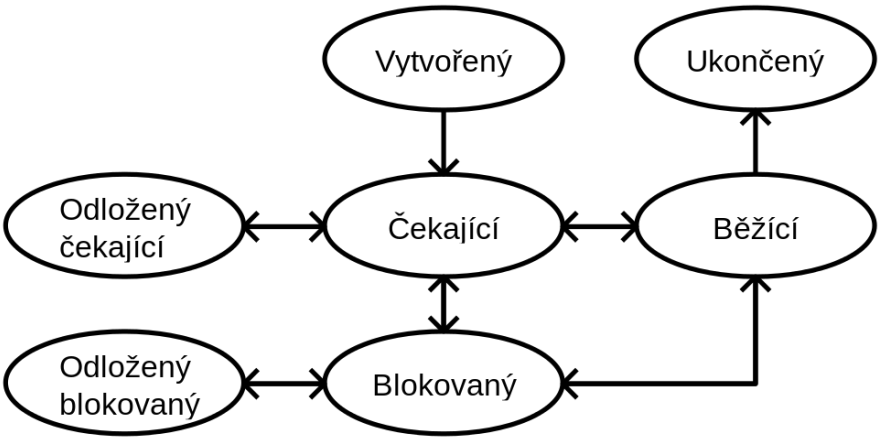
nový – proces byl vytvořen

připravený – proces čeká až mu bude přidělen CPU

běžící – procesu přidělen procesor a provádí se činnost

čekající – proces čeká na vnější událost

ukončený – proces skončil svou činnost (dobrovolně, nedobrovolně)



rozšíření

suspend – proces byl odsunut do sekundární paměti

ready/suspend + block/suspend – vylepšení předchozího mechanizmu

Pokud proces po nějaké době nepřejde do stavu waiting nebo se neukončí, tak jej za nějakou dobu operační systém sám odebere. Po skončení procesu ještě nějakou dobu OS uchovává jeho informace

32) Přepínání procesů

uložení stavu CPU do tabulky procesů -> PCB: **proces control block**

aktualizace PCB

zařazení procesu do příslušné fronty

volba nového procesu

aktualizace datových struktur pro nový proces (nastavení paměti atd)

načtení kontextu z PCB nového procesu

Lze řešit softwarově nebo s podporou HW. Rozlišujeme kooperativní x preemptivní přepínání. **Důvody k přepínání =>** vypršení časového kvanta, přerušení I/O, vyvolání výjimky.

33) Strategie přidělování CPU procesu

Plánování procesů (anglicky scheduling) je úkol jádra os, ve kterém je spuštěno více procesů najednou. Plánování procesů řeší výběr, kterému následujícímu procesu bude přidělen procesor a proces, tak poběží, přičemž výběr je závislý na prioritách jednotlivých procesů a algoritmů, kterým výběr proběhne. Běžné os pro desktopové pc vyžadují, aby byla při přidělování procesoru jednotlivým procesům zachována míra spravedlnosti. OS, které jsou v realném čase vyždují aby byly splněny dodatečné podmínky jako například preciznost, determinističnost a práce se zárukou.

Přidělování CPU procesům má na starosti plánovač úloh, jehož cíle jsou SHRNUTÍ

férovost ke všem procesům – každý proces dostane spravedlivý díl času procesoru

efektivita využití CPU – udržovat maximální vytížení CPU

minimalizace odezvy – uživatel počítače by měl mít přednost před úlohami na pozadí

minimalizace doby průchodu systémem – u každého krátce existujícího procesu je žádoucí minimalizovat čas od spuštění po jeho ukončení (doba obrátky)

maximalizace odvedené práce (průchodnost)

Počítače jsou obecně používány k různým věcem, například u serverových počítačů není tak důležitá odezva, jako odvedená práce, naopak u běžných uživatelských PC je odezva nejdůležitější.

K plánování procesoru dochází v následujících situacích:

pokud některý běžící přejde do stavu blokovaný

pokud některý proces skončí

pokud je běžící proces převeden do stavu připravený

pokud je některý proces převeden ze stavu blokovaný do stavu připravený

Pokud k plánování procesoru dochází jen v prvních dvou výše uvedených případech, označujeme takový os, respektive plánovač jako nepreemptivní. Jinak se jedná o preemptivní plánování procesoru. Jedná-li se o preemptivní případ, tak os je schopen kdykoliv procesu odebrat CPU a to znamená, že drží absolutní kontrolu nad pc a všemi prostředky. Dochází k němu zpravidla po uplynutí časového kvanta určeného pro běh procesu a je vyvoláno přerušením. Do této kategorie spadají například všechny 32bitové systémy pro IBM PC kompatibilní počítače (Windows NT, Linux, Mac OS). Pro systémy, kde je méně procesorů, než je procesů, které musí zároveň běžet se používá přepnutí kontextu takzvaný pseudoparalelismus, který umožňuje mít zdánlivě spuštěno více procesů. Přepíná se velmi rychle a potom vzniká dojem, že procesy běží současně.

To, jak přidělit některému z připravených procesů procesor záleží na tom, jaké kritéria jsou pro nás podstatná (spravedlnost, efektivita, čas odezvy, doba obrátky, průchodnost). Podle toho, které z těchto kritérií je pro nás podstatné se používají různé strategie jako například:

typy:
First come first served
Completely fair scheduler
Earliest deadline first
Last in first out
Priority scheduling
Round robin scheduling
Shortest job first
Shortest remaining time next

Cyklická obsluha – round robin

Klade důraz na spravedlivost. Každý proces dostane pevně dané kvantum času a střídají se. Problém je v tom, že pokud bude kvantum příliš malé, tak se zvyšuje režie, pokud velké, bude špatná odezva.

Pro realtime systémy nevhodné, je 100% spravedlivý, ale neumožňuje efektivní využití zdrojů.

Prioritní fronta

Každý proces má definovanou prioritu a hraje zde roli statické x dynamické nastavení priority. Systém eviduje pro každou prioritu frontu čekajících procesů. Riziko, že některé procesy se vůbec nedostanou k vykonání. Existuje rozšíření => nastavení různých velikostí kvant pro jednotlivé priority. V praxi se kombinují tyto dvě strategie, díky čemuž se vyváží klady a zápory jednotlivých strategií.

34) CPU-I/O burst cyklus

Pravidelné střídání požadavků na CPU a I/O

35) Správa procesů v unixech *

Obecně v unixech rozlišujeme 9 stavů.

a) created
b) ready to run in memory
c) ready to run, swapped out – proces je připraven k běhu, ale není v paměti
d) sleeping in memory – proces čeká na prostředek
e) sleeping swapped out – proces čeká a navíc není v paměti
f) kernel running
g) user running – pouze zde se vykonává kód programu
h) pre-empted – běh procesu násilně přerušen
i) zombie – proces skončil, ale ještě se vedou záznamy o jeho existenci

Procesy tvoří hierarchií, každý je identifikován pomocí PID. Systém při inicializaci spustí první proces (init) a nový proces (potomek) se vytvoří voláním fork() – vytvoří se kopie aktuálního procesu. Sdílí se některé informace v rámci rodiče a potomka. Může nastat to, že rodičkovský proces skončí dřív než jeho děti potom takové procesy, které neskončili, označujeme jako sirotky.

(vysvětlení => https://sites.google.com/site/casestudyonlinuxandwindow/home/linux)

Plánování procesů:

Původně prioritní fronty, každá fronta odpovídá jiné prioritě, procesy jsou obsluhovány dle priority a na stejné úrovni cyklicky.

Princip zařazování byl takový, že nový proces => vysoká priorita, postupně se priorita snižuje

Novější verze 3 třídy priorit

reálný čas (kritické procesy)

systémové procesy

uživatelské procesy

není možné přecházet mezi těmito třídami

36) Správa vláken ve windows NT

Ve Windows NT je celkem 7 stavů, které rozeznáváme:

a) initialized
b) ready – z těchto vláken vybírá plánovač
c) standby – vlákno připraveno k běhu na konkrétním procesoru. Pouze jedno vlákno může být v tomto stavu na jednom procesoru
d) running
e) waiting – čeká na nějaký objekt
f) transition – vlákno má zásobník mimo fyzickou paměť. Jakmile se zásobník dostane zpátky do paměti, vlákno přechází do stavu ready
g) terminated – ukončeno. Vlákno lze znovu oživit do stavu initializied

Vlákna se obvykle přepínají v době, kdy dojde k přerušení časovače. Dosud běžící vlákno je zařazeno na konec své fronty a pro běh se vybere jiné vlákno. Když se objeví vlákno ve stavu ready s vyšší prioritou než v running, okamžitě dojde k přepnutí, říká se tomu preempce. Preempce vrací vlákno na **začátek fronty** nikoliv na konec. Na konec pouze pokud dokončil celé časové kvantum, které mu bylo přiděleno. **Ve Windows NT se plánují vlákna nikoliv procesy**. Z tohoto důvodů, může jeden proces vytvořit 9 vláken a jiný proces jen jedno, a ten první může mít přidělen 90% CPU, za předpokladu že mají stejnou prioritu. Plánovač dynamicky mění priority vláken, aby se docílilo co nejlepšího chodu systému. Priorit pro vlákna je 0-31, celkem tedy 32. Těchto 32 priorit se dělí do 3 kategorií.
Idle úroveň (0) – je určeno pro vlákno, které **se stará o nulování paměti**. Pokud procesor není vytížen, tak tohle vlákno nuluje paměť, kde je to potřeba. Pokud se nedostane ke slovu, tak se paměť nuluje až v případě alokace paměti.

Dynamická úroveň (1-15) běžné uživatelské aplikace

Real-time úroveň (16-31) – pouze pro kritické části OS, není však zaručeno přidělení CPU v pevném čase => NT není systém reálného času.

Kvantum procesy na popředí mají 3x větší kvantum a velikost závisí na verzi OS. PC - 6 jednotek a server - 36 jednotek. Velikost lze měnit

U dynamické kategorie je možné dočasné zvýšení priority. Po uplynutí kvanta se priorita snižuje o jedna až na základní hodnotu. Vlákno co dlouho neběželo dostane prioritu 15 + 2x větší časové kvantum.

37) Vlákna v Linuxu

Vlákno a proces interně pracuje stejně v linuxu oba se účastní plánování a každé vlákno je tedy zveřejňované jako samostatný proces. Ačkoliv jsou rozdíly mezi NT a Linuxem, tak chování je stejné, a oba systémy jsou v tomto hledisku rovnocenné. Vlákno se vytváří příkazem **clone**, což je obdoba fork, ale clone umožňuje nastavit struktury, které chceme sdílet s rodičem. **Clone spouští nový proces na libovolné funkci, kdežto u fork pokračují na stejném místě.** Kontexty, které sdílí nebo klonuje

adresový prostor (základní předpoklad abychom mohli procesu říkat vlákno)

otevřené soubory

odkaz na rodičovský proces

Stavy úloh:

běžící
připravené k běhu
uspané-přerušitelné
uspané-nepřerušitelné
zastavené
skončené

V linuxu mohou vlákna pracovat jako plnohodnotné procesy s vlastní PID nebo ve skupinách pod stejnou PID. Implementace buď pomocí systémových vláken anebo pomocí vláknové knihovny Pthreads.

38) Synchronizace vláken a procesů

Synchronizace vláken a procesů je nutná k tomu, aby při paralelním vykonávaní programů nedocházelo k nepředvídatelným výsledkům. Zejména je nutné, aby byl zajištěn atomický přístup do paměti. Vzhledem k většímu počtu procesorů se využívá cachovaná paměť, někdy ale potřebujeme hodnotu přímo z paměti, jelikož mohla být změněná hodnota, ale v cache je stále stará hodnota.

39) Atomické operace

Atomická operace je nepřerušitelná, a bez ohledu na počet procesorů proběhne vždy celá a bez kolize s jiným vláknem. Existuje několik základních modelů atomické operace:

Test and set => nastav proměnnou a vrať její původní hodnotu

Swap => atomicky prohodí dvě hodnoty

Compare and swap (CAS) => ověří, jestli se daná hodnota rovná požadované a pokud ano, přiřadí ji novou hodnotu

Fetch and add => vrátí hodnotu místa v paměti a zvýší jeho hodnotu o jedna

Load-link => načte hodnotu a pokud během čtení nebyla změněna uloží do ní novou hodnotu

40) Mutex (mutual exclusion)

Zjednodušeně se jedná o binární semafor. Potřebujeme vzájemné vyloučení. Vykonávaný kód tedy může dělat jen jeden proces/vlákno. Rozdíl oproti binárnímu semaforu je v tom, že to vlákno, které si zabere mutex, tak ho musí také vrátit. Vlákno také může opakovaně vstoupit do kritické sekce, ale musí ho také tolikrát i opustit

41) Semafor

Chráněná proměnná obsahující počítadlo s nezápornými celými čísly. Obvykle máme k dispozici funkce wait a signal (atomické).

Wait => pokud je počítadlo větší než 0, snížíme ho o jedna a jdeme dovnitř, jinak čekáme

Signal => přičte k počítadlu 1

Zajímavou vlastností je, že jedno vlákno může posílat pouze signal a druhé pouze wait, využívá se to například k řešení problému producent-konzument. Semafor buď obecný (řídí přístup ke zdrojům) nebo binární (muže nabývat pouze hodnot 0 a 1)

42) Kritická sekce

Je část kódu, kdy program pracuje se sdílenými zdroji (např. paměť). V kritické sekci může být pouze 1 proces. Každý proces před vstupem do KS žádá o povolení vstoupit
Požadavky na KS:

vzájemné vyloučení – maximálně jeden proces je v daný okamžik v KS

absence zbytečného čekání – není-li žádný proces v KS a proces do ní chce vstoupit, tak mu není bráněno

zaručený vstup – pokud chce proces vstoupit do KS, tak do ní v konečném čase vstoupí

43) Problém uváznutí a jeho řešení

Uváznutí nebo-li deadlock je stav, kdy všechna vlákna čekají, ale žádné z nich neprovádí žádný kód.

Podmínky vzniku:

Mutex – alespoň jeden prostředek je využíván jedním procesem

Hold & Wait – proces vlastní jeden prostředek a čeká na další

No preemption – prostředek nelze násilně odebrat

Circular wait – cyklické čekání (proces drží prostředek a chce ten, co si drží ten druhý)

Řešení:

Ignorace (neřešení)

Detekce a zotavení => řešíme až nastane. Detekujeme-li ho, tak zrušíme jeden ze zúčastněných procesů. K detekci se používá alokační graf prostředků a graf čekání a potom deadlock vznikne, pokud je v grafu čekání cyklus

Zamezení vzniku => deadlock řešíme preventivně a snažíme se zajistit, aby jedna z podmínek nebyla splněna. Zamezuje se držení a čekání (procesy si žádají o prostředky na začátku). Zavádí se možnost odebrat prostředek

Vyhýbání se uváznutí => procesy si žádají o prostředky libovolně. Systém vyhoví jen těm žádostem o prostředky, které nemohou vést k deadlocku. Existuje **bezpečný stav** což je pořadí procesů, ve kterém jejich požadavky budou vyřízeny bez vzniku deadlocku. Odmítne se přidělit prostředky, pokud by se přešlo do nebezpečného stavu

Bankéřův algoritmus

Zotavení z deadlocku:

Buď násilně odstřelíme nebo odebereme jeden, či více zúčastněných prostředků, nebo se to dělá tak, že se vybere kandidát, odstřelí se a pak se kontroluje, jestli nedošlo k vyřešení deadlocku.

44) Bankéřův algoritmus

Používá se při vyhýbání deadlocku. Vhodný, když je větší počet prostředků daného typu. Na začátku musí každý proces oznámit kolik bude potřebovat prostředků jakého typu maximálně. Při žádosti o prostředky systém ověří, jestli se nedostane do nebezpečného stavu a pokud nelze vyhovět, je proces pozdržen.

TYPY

```
ow_loop:
        mov byte [rdi + rcx], 46
        add rcx, 1
        add r11, 1
        cmp r10, r11
        jne row_loop
        mov byte [rdi + rcx], 42
```