

# Operační systémy 1 (KMI/OS1)

---

Přepis přednášek od Mgr. Petr Krajča, Ph.D.

**Martin Hořava**

HISTORIE OPERAČNÍCH SYSTÉMŮ .....	3
TYPY OPERAČNÍ SYSTÉMŮ .....	3
<b>I. PŘEDNÁŠKA.....</b>	<b>4</b>
OPERAČNÍ SYSTÉM .....	4
ARCHITEKTURA POČÍTAČE .....	5
CPU .....	6
<b>II. PŘEDNÁŠKA.....</b>	<b>11</b>
ADRESACE PAMĚTI .....	11
INTEL X86: PŘÍZNAKY .....	15
INTEL X86: BĚH PROGRAMU A PODMÍNĚNÉ SKOKY .....	15
INTEL X86: PODMÍNĚNÉ SKOKY A POROVNÁNÍ.....	17
SMYČKY .....	17
<b>III. PŘEDNÁŠKA.....</b>	<b>18</b>
ZÁSOBNÍK.....	20
ULOŽENÍ/ODEBRÁNÍ HODNOT POMOCÍ OPERACÍ: .....	20
VOLÁNÍ PODPROGRAMŮ/FUNKCÍ .....	20
<b>IV. PŘEDNÁŠKA.....</b>	<b>28</b>
REŽIMY PRÁCE CPU .....	28
SYSTÉMOVÁ VOLÁNÍ .....	29
POZNÁMKA K HISTORII .....	29
REPREZENTACE ČÍSEL S PLOVOUCÍ ŘADOVOU ČÁRKOU .....	30
ZÁSOBNÍKOVÉ CPU .....	31
DALŠÍ ROZŠÍŘENÍ .....	32
AT&T .....	34
SPARC.....	34
<b>V. PŘEDNÁŠKA.....</b>	<b>36</b>
ARM .....	36
SHRNUTÍ KONCEPCÍ .....	37
PŘEKLAD PROGRAMU.....	38
DYNAMICKY LINKOVANÉ KNIHOVNY .....	40
DYNAMICKY NAHRÁVANÉ KNIHOVNY .....	41
VIRTUÁLNÍ STROJE .....	41
<b>VI. PŘEDNÁŠKA.....</b>	<b>42</b>
JIT PŘEDKLAD (JUST IN TIME).....	42
JAVA VIRTUAL MACHINE A JAVA BYTOCODE .....	42
COMMON LANGUAGE RUNTIME .....	43
MACOS X .....	44
ARCHITEKTURA OS.....	45
ARCHITEKTURA JÁDRA.....	46
PROBLÉMY S NÁVRHEM OS.....	47
HISTORIE OS.....	48
HISTORIE UNIXŮ .....	48

ZÁKLADNÍ VLASTNOSTI UNIXŮ .....	50
DALŠÍ UNIXY.....	50
<b>VII. PŘEDNÁŠKA.....</b>	<b>51</b>
XNU/DARWIN.....	51
HISTORIE WINDOWS .....	52
WINDOWS NT .....	53
WINDOWS NT: ARCHITEKTURA.....	55
KOMUNIKACE V OS.....	56
ANDROID & IOS.....	56
PROCESY .....	58
INFORMACE O PROCESU .....	60
PLÁNOVÁNÍ PROCESŮ .....	61
ALGORITMY PRO PLÁNOVÁNÍ PROCESŮ .....	62
<b>VIII. PŘEDNÁŠKA.....</b>	<b>64</b>
ÚLOHY BĚŽÍCÍ V REÁLNÉM ČASE .....	64
VLÁKNA .....	66
IMPLEMENTACE VLÁKEN .....	66
IMPLEMENTAČNÍ ASPEKTY: UNIX.....	68
PLÁNOVÁNÍ PROCESŮ V LINUXU .....	69
PLÁNOVÁNÍ PROCESŮ V LINUXU .....	69
<b>IX. PŘEDNÁŠKA.....</b>	<b>71</b>
IMPLEMENTAČNÍ ASPEKTY: WINDOWS .....	71
MAC OS X .....	74
SYNCHRONIZACE VLÁKEN A PROCESŮ .....	76
ATOMICKÝ PŘÍSTUP DO PAMĚTI.....	76
KRITICKÁ SEKCE (CRITICAL SECTION).....	77
PETersonŮV ALGORITMUS.....	80
<b>X. PŘEDNÁŠKA.....</b>	<b>80</b>
SEMAFOR.....	80
DALŠÍ SYNCHRONIZAČNÍ NÁSTROJE .....	81
SYNCHRONIZAČNÍ PRIMITIVUM VE WINDOWS.....	82
SYNCHRONIZAČNÍ PRIMITIVUM V UNIXECH.....	83
SYNCHRONIZACE PROCESŮ .....	83
DEADLOCK.....	84
ŘEŠENÍ DEADLOCKU .....	84
ŘEŠENÍ DEADLOCKU .....	85
<b>XI. PŘEDNÁŠKA.....</b>	<b>87</b>
MEZIPROCESNÍ KOMUNIKACE .....	89
SdíLENÁ PAMĚŤ .....	89
ROURY.....	90
ZASÍLÁNÍ ZRPÁV .....	91
ZASÍLÁNÍ ZPRÁV V OS .....	92
DALŠÍ MECHANIZMY .....	92

# ÚVOD DO OPERAČNÍCH SYSTÉMŮ

## HISTORIE OPERAČNÍCH SYSTÉMŮ

1. generace (1945 – 1955): relé, elektronky a program „zadrátovaný“ do počítače
2. generace (1955 – 1965): tranzistory, děrné štítky, dávkové zpracování a FORTRAN
3. generace (1965 – 1980): integrované obvody, IBM Systém/360 a minipočítače PDP
  - multitasking
  - timesharing (CTSS – MIT)
  - současná práce více uživatelů, ale pořád prvky dávkového zpracování
  - spooling (sdílení periférií)
  - virtuální paměť, první sítě
4. generace (1980 – současnost): vysoký stupeň integrace, Intel 8080, x86, CP/M, DOS, Windows 95/NT, Unix, GNU/Linux
  - Řada dalších OS, často se specifickým účelem

## TYPY OPERAČNÍCH SYSTÉMŮ

### PODLE URČENÍ

- **Mainframy** - OS/400, zOS
- **Serverové/multiprocesorové** - BSD, AIX, GNU/Linux, HP-UX, Solaris, Windows NT, atd.
- **Desktopové** - BSD, GNU/Linux, Mac OS X, Windows NT
- **Realtime** - VxWorks, QNX // jsou schopné zajistit odpovědi v přesně stanovených intervalech, použití v průmyslovém nasazení, kde si nelze dovolit opoždění ani v řádech ms
- **Distribuované** - Plan B // je možné rozinstalovat mezi několik strojů, úložný prostor a výkon je rozložen mezi všechny počítače
- **Mobilní telefony, tablety** - Android, BlackBerry OS, iPhone OS, Symbian, Windows Phone
- **Experimentální** //výukové - Minix (Unix), Plan 9

## HISTORICKÉ ZÁLEŽITOSTI

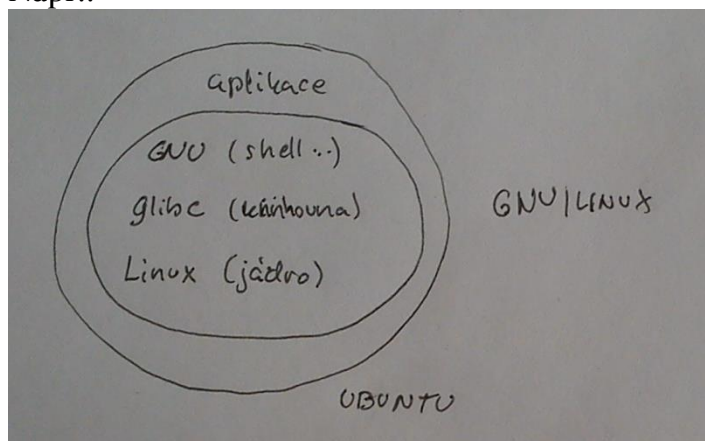
- CP/M, MS-DOS, Windows 9x
- BeOS, Mac OS, OS/2

### I. Přednáška

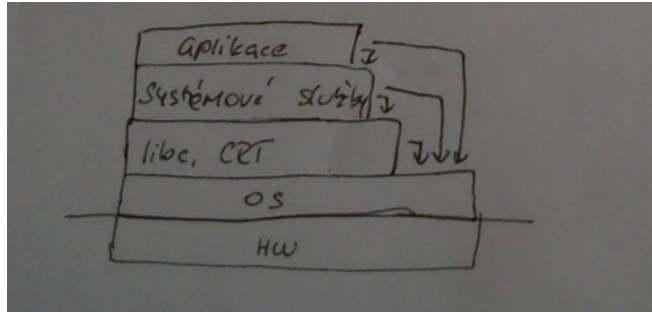
## OPERAČNÍ SYSTÉM

### VRSTVY HW/SW

1. **Hardware** - k HW by měl mít přístup pouze OS
2. **Operační systém**
  - OS běží v privilegovaném režimu (jeho funkce jsou upřednostněné), vyšší vrstvy jsou v nepriviligovaném režimu (aby nemohla poškodit HW)
  - Rozlišují se na USER SPACE a KERNEL SPACE  
Např.:



3. **standartní knihovna** (libc, CRT) - alokace paměti, stará se o přidělování kousků paměti (malloc)
4. **Systémové nástroje** - logování, ls, dir atd
5. **Aplikace**
  - každá vrstva může volat předchozí vrstvu, může přeskočit vrstvu → vzniká problém přenositelnosti

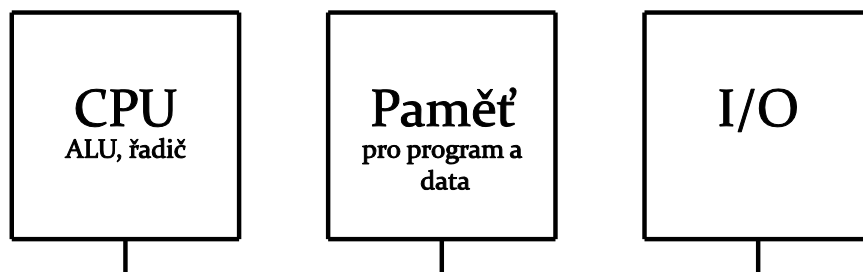


- Jádro OS vs. aplikace
- Hranice mezi vrstvami nemusí být ostré
- Situace se komplikuje - virtualizace, běhové prostředí
- Další funkce: operační systém zajišťuje správu zdrojů - sdílení času (CPU zařízení), místa (paměti, disky)

## ARCHITEKTURA POČÍTAČE

### JOHN VON NEUMANNOVA ARCHITEKTURA

- **CPU** (ALU - aritmeticko-logická jednotka, řadič - stará se o rozložení komunikace po sběrnici, může být reálně umístěn jinde)
- **Paměť** společná pro program i data – jednodušší implementace, horší časová náročnost při práci
- **Vstup/výstup**
- **Sběrnice** (řídící, adresní, datová)
- Instrukce procesoru jsou zpracovávány v řadě za sebou (není-li uvedeno jinak)
- **Výhoda:** umožňuje implementovat systém, kde máme v paměti vše
- **Nevýhoda:** že operační paměť se musí využívat i pro kód programu



## ABSTRAKCE HW

- Vyvíjet software na míru jednoho HW náročné/neefektivní (obvykle): hardware je neuvěřitelně složitý
- Operační systém - rozhraní mezi SW a HW
- Operační systém poskytuje abstrakci nad daným hardwarem (+ jazyky vyšší úrovně)
- V konečném důsledku několik úrovní abstrakce
- Odstiňuje konkrétní HW, jednotné API

## CPU

### OBEČNÁ STRUKTURA CPU

- Je to jednotka, která zpracovává instrukce
- **Aritmeticko-logická jednotka** (ALU) - provádí výpočty
- **Řídící jednotka** - řídí chod CPU
- **Registry**
  - Slouží k uchování právě zpracovávaných dat (násobně rychlejší přístup než do paměti)
  - Speciální registry obsluhující chod CPU:
    - **IP** (instruction pointer) – ukazuje na instrukci, která se má právě provádět
    - **Program status word** (PSW, FLAGS - pokud došlo k přetečení, nastaví se příznak do registru),
    - **IR** (instruction register) - instrukce, která je právě prováděna
    - **SP** (stack pointer) - ukazatel na zásobník, kde jsou uloženy výpočty, argumenty, atd.

### INSTRUKČNÍ SADA (ISA)

- Sada ovládající procesor (specifikovaná pro daný CPU/rodinu CPU)
- Instrukce a jejich operandy jsou reprezentovány jako čísla → strojový kód
- Každá instrukce má obvykle 0 až 3 operandy (může to být registr, konstantu nebo místo v paměti)
- Pro snazší porozumění, se instrukce CPU zapisují v jazyce symbolických adres (též vulgárně označován jako Assembler)

- Instrukce jsou zpracovávány v několika krocích: // z důvodu větší efektivity
  1. načtení instrukce do CPU (Fetch)
  2. dekódování instrukce (decode)
  3. výpočet adres operandů
  4. přesun operandů do CPU
  5. provedení operace (Execute)
  6. uložení výsledku (Write-back)
- **Pipelining** - umožňuje zvýšit efektivitu CPU // díky rozdělení do jednotlivých kroků je možné provádět více instrukcí zároveň

Virtuální procesorový cyklus	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6
1	lhz					
2	or	lhz				
3	stw	or	lhz			
4	lfs	stw	or	lhz		
5	fsub	lfs	stw	or	lhz	
6	stfs	fsub	lfs	stw	or	lhz
7		stfs	fsub	lfs	stw	or
8			stfs	fsub	lfs	stw
9				stfs	fsub	lfs
10					stfs	fsub
11						stfs

- je potřeba zajistit správné pořadí operací
- **Superskalární procesor** - procesor může mít víc jednotek např. pro výpočty (FPU, ALU), musíme se postarat o správnou synchronizaci
- **Problém s podmíněnými skoky (branch prediction)** // jakmile se má provést nějaký podmíněný, nevíme, která instrukce bude následovat → může se stát, že se musí vyprázdnit instrukci z pipeline → zpomalení



## INTEL X86: REGISTRY

- Registry jsou 32 bitové
- Obecně použitelné (i když existují určité konvence, jak by se měli používat)
  - **EAX** (Accumulátor) - střadač pro násobení a dělení (temp), vstupní a výstupní operace
  - **EBX** (Base) - nepřímá adresace paměti (uložen ukazatel)
  - **ECX** (Counter) - počítadlo při cyklech, posuvech a rotacích
  - **EDX** (Data) - všeobecně použitelná data
- Každý registr má svou spodní 16bitovou část reprezentovanou jako registr AX, BX, CX, DX
- Tyto 16 bitové registry lze rozdělit na dvě 8bitové části reprezentované jako AH, AL, BH, BL  
IMAGE 5
- **EDI** (Destination index) - adresa cíle
- **ESI** (Source index) - adresa zdroje
- **EBP** (Base pointer) - adresace parametrů funkcí a lokálních proměnných
- **ESP** (Stack pointer) - ukazatel na vrchol zásobníku (adresa vrcholu zásobníku)
- **EIP** (Instruction pointer) - ukazatel na aktuální místo programu, adresa instrukce následující za právě prováděnou instrukcí, není možné jej přímo měnit (jen patřičnými instrukcemi)
- **EF** (FLAGS) - příznaky nastavené právě proběhlou instrukcí
- Spodních 16 bitů těchto registrů lze adresovat pomocí registrů DI, SI, BP, SP, IP, F (FLAGS), další dělení není možné
- **ESI** a **EDI** jde používat jako obecně použitelné
- změny v registrech EBP, ESP by měly být uvážené

## INTEL X86: OPERACE

- Operandy instrukcí mohou být
  - r - registry
  - m - paměť
  - i - hodnoty
- Paměť lze v jedné instrukci adresovat pouze jednou

```
MOV r/m, r/m/i      ; op1 := op2
ADD r/m, r/m/i      ; op1 := op1 + op2
SUB r/m, r/m/i      ; op1 := op1 - op2
NEG r/m             ; op1 := - op1
MUL r/m             ; EDX:EAX := EAX * op1 // vezme hodnotu
                    ; registru EAX * op1, 2 krát 32 bitů je 64 bitů a proto se výsledek
                    ; ukládá do dvou registrů EDX:EAX | EDX spodních 32, EAX horních 32 bitů;
                    ; neznaménková čísla
IMUL r, r/m         ; op1 := op1 * op2          // znaménková
IMUL r, r/m, i      ; op1 := op1 * op2 * op3 // při přetečení se
                    ; ořeže, i pro znaménková čísla
```

```
OR r/m, r/m, i      ; op1 := op1 | op2
AND r/m, r/m/i      ; op1 := op1 & op2
XOR r/m, r/m/i      ; op1 := op1 ^ op2
NOT r/m             ; op1 = ~op1
```

// nelze sečíst 16 bitové a 32 bitové číslo

```
SHL r/m, i          ; op1 := op1 << op2 (bezznaménková
operace)
SAL r/m, i          ; op1 := op1 << op2 (znaménková operace)
SHR r/m, i          ; op1 := op1 >> op2 (bezznaménková
operace)
SAR r/m, i          ; op1 := op1 >> op2 (znaménková operace)
// bitový posun o 2 doleva = násobení 2 | o 4 = násobení 4. Násobení je pomalé, proto se používají
bitové posuny
// znaménková a bezznaménkové bitové operace - nejvyšší bit udává, zdali je kladné/záporné
```

```
ROL r/m, i          ; rotace bitů doleva
ROR r/m, i          ; rotace bitů doprava
// použitelné pro optimalizaci a u přístupu k HW, C nepodporuje z důvodu, že PC při vývoji C je
nepodporovali
```

**U bitového posunu:** musíme rozlišovat, zdali se jedná o znaménkové/neznaménkové číslo → přesouvá se kopie nejvyššího bitu u bitového posunu doprava (SAR)

### UKÁZKA ROTACE BITŮ DOPRAVA

1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

→

1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---

```
INC r/m      ; op1 := op1 + 1
DEC r/m      ; op1 := op1 - 1
```

## II. Přednáška

### ADRESACE PAMĚTI

- Lineární struktura s pevnou délkou a náhodným přístupem
- **Přímá adresa** - ukazuje na pevně dané místo v paměti
- **Nepřímá adresa** - před přečtením hodnoty se vypočítá z hodnot registrů podle vzorce:

$$adresa = posunutí + báze + index \times faktor$$

- *Posunutí* je konstanta
- *Báze* a *index* jsou registry
- *Faktor* je číslo 1, 2, 4, 8
- Kteroukoliv část vzorce lze vypustit

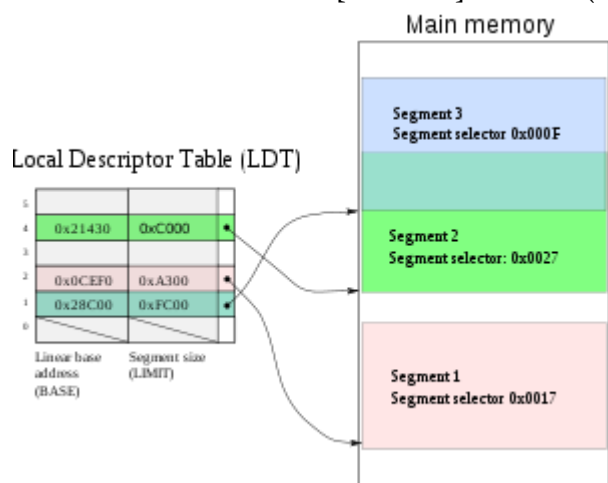
### ADRESACE PAMĚTI V ASSEMBLERU X86

- V assembleru se čtení/zápis do paměti zapisuje ve tvaru:  
velikost PTR [velikost], kde velikost může být (podle velikost): BYTE, WORD, DWORD  
`mov dword ptr [eax], ebx // chceme číst 4 B z registru EBX do EAX`  
`add ax, word ptr [ebx + esi * 2 + 10] // přičtení k AX hodnoty z EBX + ESI * 2 + 10`
- Pokud lze odvodit velikost dat z použitelných registrů, je možné vypustit velikost PTR, např.:  
`mov [eax], ebx`  
`add ax, [ebx + esi * 2 + 10] // překladač umí sám doplnit délku ale ne vždy`  
`mov word ptr [eax + esi * 2 + 100], 42 // zde neumí doplnit délku, neví délku registru → musíme explicitně uvést`
- Při přístupu k proměnným ve VS (jsou adresy doplněny automaticky)

### SEGMENTACE X86

- i386 má ve skutečnosti 48bitové adresy: selector (16 b) + offset (32b)
- **Selector** je určen pomocí segmentových registrů (CS (segment kde je uložený kód), DS (data), SS (zásobník), ES, FS, GS)
- Segmentový registr většinou určen implicitně → pracuje se jen s offsetem

- **Lineární adresa** = DT [selector] + offset (kde DT je LDT nebo GDT)



## VZTAH ADRESACE PAMĚTI PROCESORU A JAZYKA C

### 1. Deference

```
mov eax, dword ptr [ebx]    ;; eax := *ebx
// provedeme dereferenci EBX → získáme hodnotu a provedeme přiřazení do EAX
```

### 2. Pole

```
short a[ ] = malloc(sizeof(short) * 10);
_asm{
    mov ebx, a
    mov ax, [ebx + esi * 4] ;; ax := a[esi]
}
```

### # 1

```
// do EBX si uložíme hodnotu pole a,
// z logického hlediska je pointer a pole to samé
```

### 3. Strukturované hodnoty

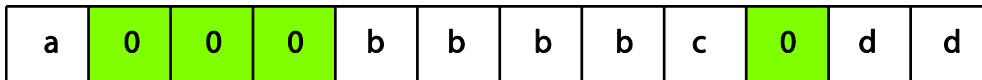
```
struct foo { int x; int y; int z [10]; };
struct foo * a = malloc(sizeof(struct foo));

_asm {
    mov ebx, a
    mov [ebx], ecx    ;; a -> x := ecx
    mov [ebx + 4], ecx ;; a -> y := ecx
    mov [ebx + esi * 4 + 8], ecx ;; a-> z[esi] := ecx
}
```

## ZAROVNÁNÍ HODNOT

- Adresa paměti mem je zarovnaná na n bytů, pokud je mem násobkem n
- Z paměti procesor čte celé slovo (např. 32 bitů) → výhodné, aby čtená hodnota ležela na zarovnané paměti (rychlejší přístup, snazší implementace CPU)
- Některé CPU neumožňují číst data z nezarovnané adresy (RISC), jiné penalizují zpomalení výpočtu
- Hodnoty jsou zarovnány na svou velikost např.:
  - char na 1B
  - short na 2B
  - int na 4B, atd.
- Tzn. hodnoty typu short jsou v paměti vždy na adresách, které jsou násobky 2, hodnoty int na násobcích 4, atd
- Velikost struktur se obvykle zaokrouhluje na 4B nebo na 8B
- Příklad:

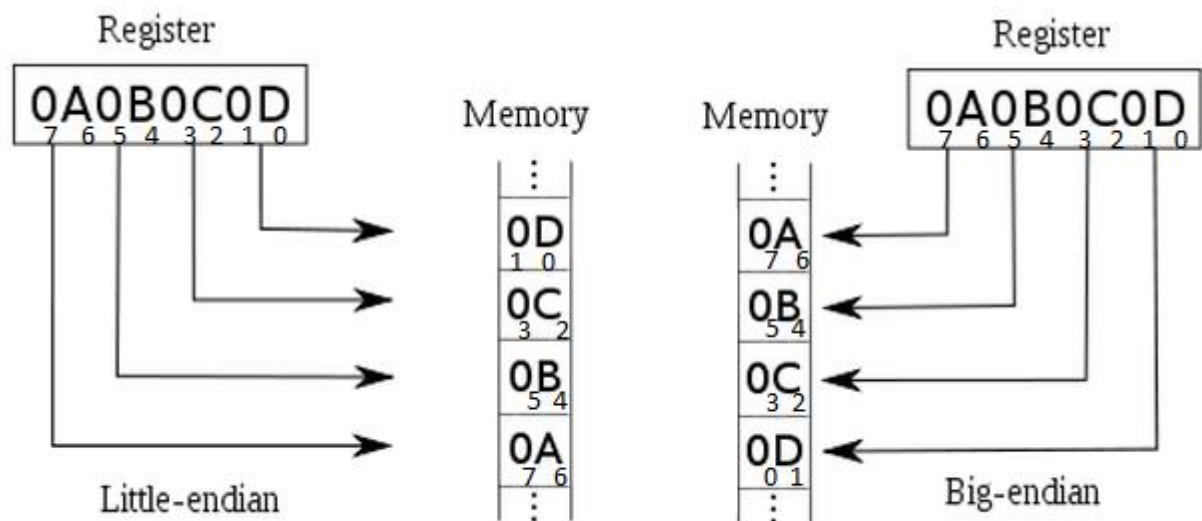
```
struct foo {  
    char a;  
    /* mezera 3B */  
    int b;  
    char c;  
    /* mezera 1B */  
    short d;  
};
```



- Toto je chování překladače, lze jej změnit či potlačit, což může zpomalit čtení nebo skončit chybou

## ULOŽENÍ VÍCEBYTOVÝCH HODNOT: ENDIANITA

- Liší se mezi procesory → potřeba brát v úvahu při návrhu datových formátů a protokolů
- **Little-endian**: hodnoty jsou zapisovány od nejméně významného bytu (x86, Amd64, Alpha, ...)
- **Big-endian**: hodnoty jsou zapisovány od nejvýznamnějšího bytu (SPARC, IBM POWER, ...)
- **Bi-endian**: za určitých okolností lze přepínat (ARM, PowerPC, SparcV9, IA-4, ...)



## REPREZENTACE HODNOT

- Čísla jsou v doplňkovém kódu (zápornou hodnotu dostaneme tak, že provedeme inverzi bitů a přičteme 1) → snadná manipulace
- Znaménková a bezznaménkové typy (unsigned int vs. int) #4
- Pokud se hodnota nevejde do rozsahu typu → přetečení/podtečení

`char a = 127 + 1; // → - 128`

`unsigned char c = 255 + 1; // → 0`

`char b = -10 - 120; // → 126`

//odpadá anomálie – záporná o „nula“, provádění operace s čísly atd.

- #5 

## BCD (BINARY CODED DECIMAL)

- Čísla v desítkové soustavě 4b na cifru
- Výhodné u přístupu k HW, ale je nepraktické (redundantní, málo využité byty)



## INTEL X86: PŘÍZNAKY

- Jednotlivé operace nastavují hodnotu bitů v registru EF // registr příznaků - EFlags
- Záleží na operaci, které příznaky nastavuje
- Příznaky pro řízení výpočtu

// 1 bitové příznaky, EF je 32 bit

- SF (sign flag) – podle toho jestli je kladný nebo záporný

```
mov eax, 42
```

```
sub eax, 43
```

- ZF (zero flag) – výsledek byla nula
- CF (carry flag) – výsledek je větší nebo menší/nejmenší možné číslo

př.:

```
1 1 1 1 1 1 1 1
```

```
0 0 0 0 0 0 0 1
```

```
-----
```

```
0 0 0 0 0 0 0 1
```

1 navíc přenesena do registru CF

- OF (overflow flag) – příznak přetečení znaménkové hodnoty daný rozsah

```
127
```

```
127
```

```
-----
```

254 → nepřetekla v rámci 8 bit ale kvůli znaménkám v rámci 7 bitů

- další příznaky
  - AF (auxiliary carry flag) – nastaven na jedna ze čtvrtého do pátého bitu (BCD čísla)
  - PF (parity flag) – nastaven na jedna při sudé paritě (pouze dolních 8 bitů)
- řídicí znaky
  - TF (trap flag) – slouží ke krokování
  - DF (direction flag) – ovlivňuje chování instrukcí blokování přesunu
  - IOPL (I/O privilege level) – úroveň oprávnění (2 bity, nastavuje pouze jádro)
  - IF (Interrupt enable flag) – možnost zablokovat některá přerušení // do 0:56

## INTEL X86: BĚH PROGRAMU A PODMÍNĚNÉ SKOKY

- Program zpracovává jednu instrukci za druhou (pokud není uvedeno jinak) → skok
- Nepodmíněný skok



- operace `JMP r/m/i` – ekvivalentní `GOTO` (použití při implementaci smyček)  
 // při použití pouze nepodmíněných skoků → vytvářeli bychom pouze nekonečné skoky
- Není přítomná operace ekvivalentní `if`
- Podmíněný skok je operace ve tvaru `Jcc` (operace porovnání), provede skok na místo v programu, pokud jsou nastaveny příslušné příznaky
- Např.: `JZi` (provede skok, pokud výsledek předchozí operace byl nula), dále `JNZ`, `JS`, `JNS`...
- `JS` – provede se skok, pokud byl výsledek záporný
- `JZ` – nulový
- `JWZ` – nenulový
- `JWS` – nezáporný

## POROVNÁNÍ ČÍSEL

- Srovnání čísel jako rozdíl (operace `CMP r/m, r/m/i`, je jako `SUB`, ale neprovádí přiřazení)
- Je skok při rovnosti, `JNE`, při nerovnosti (v podstatě operace `JZ` a `JNZ`)
- a další operace
- př. 1.: `a - b` :

I. `A = B = 0`

II. `A > B => 0`

III. `A < B =< 0`

// nastavuje se příznak do registru `EF`

- př. 2.: `sub eax, ebx` // změni se nám hodnota v `eax`, což může být nežádoucí  
`jz foo` // `foo` – návěští  
 ...  
`cmp eax, ebx` // změni pouze hodnotu v registru `EF` a díky tomu může dojít ke

skoku

# INTEL X86: PODMÍNĚNÉ SKOKY A POROVNÁNÍ

- Příklady použití
- podmíněné skoky po porovnání **bezznaménkových** hodnot

instrukce	Alternativní jméno	příznaky	podmínka
JA	JNBE	(CF or ZF) = 0	A > B
JAE	JNB	CF = 0	A >= B
JB	JNAE	CF = 1	A < B
JBE	JNA	(CF or ZF) = 1	A <= B

- Podmíněné skoky pro porovnávání **znaménkových** hodnot

instrukce	Alternativní jméno	příznaky	Podmínka
JG	JNLE	(SF = OF) & ZF = 0	A > B
JGE	JNL	(SF = OF)	A >= B
JL	JNLE	(SF != OF)	A < B
JLE	JNL	(SF != OF) or ZF = 1	A <= B

## SMYČKY

- Pro snadnější implementaci cyklů byly zavedeny speciální operace
- JECXZ, JCXZ – provede skok, pokud registr ECX/CX je nulový (není potřeba explicitně testovat ECX)
- LOOP – odečte jedničku od ECX a pokud v registru ECX není nula, provede skok

## POZNÁMKY

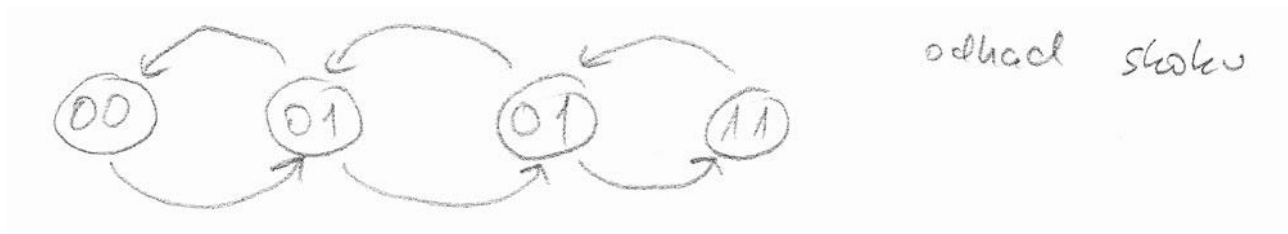
- Uvádí se, že složené operace jsou pomalejší, než jednotlivé kroky // jsou značně pomalejší
- (obecně) podmíněné skoky zpomalují běh programu → zrušení výpočtu v pipeline // procesor má několik pipeline a zpracovává několik instrukcí zároveň, jakmile se má provést skok, musí počkat, než se provedou všechny instrukce v pipeline, poté otestuje, esli se má skok provést nebo ne. Poté se instrukce začnou zpracovávat v řadě za sebou. Uvádí se, že bývá skok každých 6-8 instrukcí.

- Procesory implementují různé heuristiky pro odhad, jestli daný skok bude proveden
  - **statický přístup** (např. u skoků zpět se předpokládá, že budou provedeny)
  - **dynamický přístup** (na základě historie skoků se rozhodne)
  - **nápověda poskytnutá programátorem** (příznak v kódu)

### III. Přednáška

#### ODHAD SKOKŮ

- Procesory používají kombinaci výše zmíněných metod (hlavně dynamický odhad), různé hodnoty
- **Čtyřstavové počítadlo**: při každém průchodu procesor ukládá do Branch Prediction Buffer (2b příznak, jestli byl skok proveden nebo ne) a postupně přechází mezi čtyřmi stavy: // pokud je stav proveden, tak zvýší stav, pokud se neprovede, tak se sníží
  - 11 – strongly taken
  - 10 – weakly taken // chyba v obrázku
  - 01 – weakly not taken
  - 00 – strongly not taken

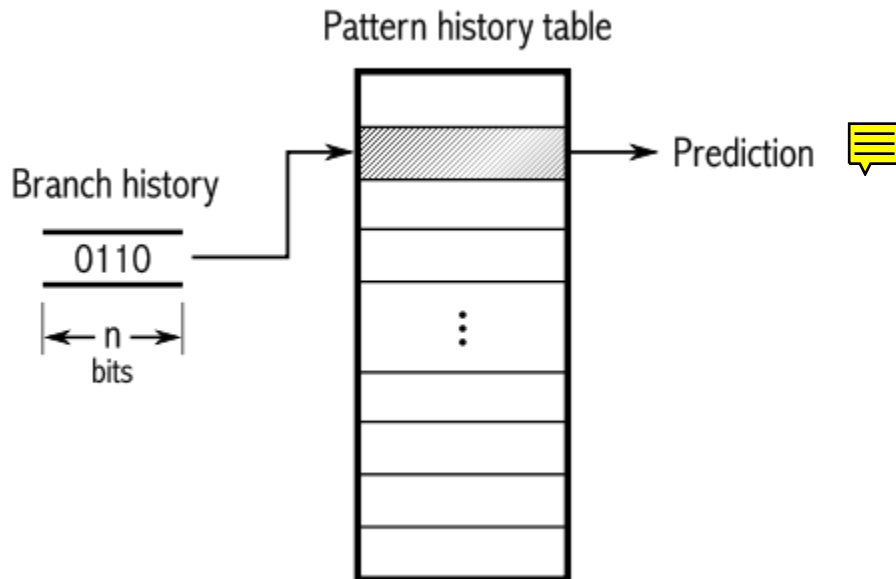


- Až na stav 00 předpokládá, že skok bude proveden
- Velikost BPB a počáteční stav počítadla se mezi procesory liší
- **Problém**: pravidelné střídání úspěšnosti → dvouúrovňový odhad (vzor chování obrázek #21) // Příklad: pokud skok provede, zvýší počítadlo z 01 na 10 a předpokládá skok, poté skok neprovede, sníží počítadlo na 01, očekává skok, skok provede a zvýší na 10 atd... přesnost pouze 50 %, když bude cyklit mezi 00 a 01 tak 0 % přesnost
- Může se zacyklit mezi dvěma stavy, proto dvě vrstvy, viz dále

Může dojít k chybě: pokud jednou skok provede, podruhé ne a tak dále dokola, zacyklí se mezi stavem 01 a 10 -> 50 % chyba, z toho důvodu byla zavedena Two-Level adaptive predictor, který má přesnost zhruba 97%.

## TWO-LEVEL ADAPTIVE PREDICTOR

- Pro každý vzor existuje odhad založený na výše zmíněném přístupu
- Velikost vzoru závisí na procesoru
- Globální vs. Lokální tabulka // lokální má vyšší přesnost, globální zanedbatelně menší, ale je jednodušší pro implementaci



# VOLÁNÍ FUNKCÍ

## ZÁSOBNÍK

- Procesor má vyčleněný úsek paměti pro zásobník (LIFO) → mezi výpočty, návratové adresy, lokální proměnné, ...
- Vyšší programovací jazyky obvykle neumožňují přímou manipulaci se zásobníkem (přesto má zásadní úlohu)
- procesory i386 mají jeden zásobník, který roste shora dolů
- registr ESP ukazuje na vrchol zásobníku (`mov eax, [esp]` načte hodnotu na vrchol zásobníku)

## ULOŽENÍ/ODEBRÁNÍ HODNOT POMOCÍ OPERACÍ:

```
PUSH r/m/i          ;; sub esp 4
                    ;; mov [esp], op1

POP r/m              ;; mov op, [esp]    // načte hodnotu
z vrcholu zásobníku
                    ;; add esp, 4    // posune vrchol zásobníku
směrem nahoru
```

## VOLÁNÍ PODPROGRAMŮ/FUNKCÍ

- K volání podprogramu se používá operace `CALL r/m/i` → uloží na zásobník hodnotu registru IP a provede skok // nelze implementovat návrat z funkce, proto `CALL`
- K návratu z funkce se používá operace `RET` → odebere hodnotu ze zásobníku a provede skok na adresu danou touto hodnotou
- Použití zásobníku umožňuje rekurzi

```
call foo      ────────────────────▶   foo:
...<- sem ukazuje registr EIP          ...
└──────────────────────────────────▶   ret
```

## VOLÁNÍ FUNKCÍ

Dále se při volání funkcí musí vyřešit: (řeší se konvencí)

- Předání parametrů
- Vytvoření lokálních proměnných
- Provedení funkce
- Odstranění informací ze zásobníku
- Návrat z funkce, předání výsledku

## KONVEKCE VOLÁNÍ FUNKCÍ

- Způsob jakým jsou předávány argumenty funkcí, jsou jen konvence (specifické pro překladač, i když často jsou součástí specifikace ABI OS)
- Předávání pomocí registrů (dohodnou se určité registry), případně zbývající argumenty se uloží na zásobník
- Předávání argumentů čistě přes zásobník
- Díky použití konvencí nám umožňuje, aby mohl program používat různá API, např.: API OS pro zápis souboru na disk atd.
- Kdo odstraní předané argumenty ze zásobníku? (volaná funkce nebo volající? Neexistuje jednoznačná odpověď), 3 různé konvence:
  - **Konvence C** (cdecl)
    - Argumenty jsou předané čistě přes zásobník
    - Zprava doleva
    - Argumenty ze zásobníku odstraňuje volající
    - Umožňuje funkce s proměnlivým počtem parametrů
  - **Konvence Pascal** (pascal)
    - Argumenty jsou předané čistě přes zásobník
    - Zleva doprava
    - Argumenty ze zásobníku odstraňuje volaný
    - Neumožňuje funkce s proměnlivým počtem parametrů
  - **Konvence fastcall** (fastcall, msfastcall)
    - První dva parametry jsou předány pomocí ECX, EDX
    - Zbylé argumenty jsou na zásobníku zprava doleva
    - Argumenty ze zásobníku odstraňuje volaný
    - Mírně komplikuje funkce s proměnlivým počtem parametrů
    - Pod tímto jménem mohou existovat různé konvence

- // rychlejší, protože při PUSH argumentů na zásobník, se musí data zapsat do paměti a posunout vrchol zásobníku, což je pomalejší, než přes registry
- Návrátová hodnota se na i386 obvykle předává pomocí registru EAX, příp. EDX:EAX
- Větší hodnoty než je velikost registru, jsou předávány odkazem

## RÁMEC FUNKCE (STACK FRAME)

- Při volání se na zásobníku vytváří tzv. Rámec (stack frame)
- Obsahuje předané argumenty, adresy návratu, případně lokální proměnné
- K přístupu k tomuto rámci se používá registr EBP

## VOLÁNÍ FUNKCE S KONVENCÍ CDECL

### VOLÁNÍ FUNKCE

1. na zásobník jsou uloženy parametry funkce zprava doleva (`push <arg>`)
2. zavolá se funkce (`call <adresa>`), na zásobník se uloží adresa návratu
3. funkce uloží do registru EBP na zásobník (adresa předchozího rámce)
4. funkce uloží do registru EBP obsah ESP (začátek nového rámce)
5. vytvoří se na zásobníku místo pro lokální proměnné
6. na zásobník se uloží registry, které se budou měnit (`push <reg>`)

### PŘÍKLAD VOLÁNÍ FUNKCE:

```
foo(1, 2, 3)           // deklarace volané funkce
    push arg 3          // vložení argumentů na zásobník
    push arg 2
    push arg 1
    call foo             // zavolání funkce foo
3. add esp, 12           // 4 * počet argumentů
```

```
foo: push ebp;           // uložíme na
    // zásobník (pravděpodobně obsahuje adresu
    // rámce volající funkce)
```

```
    mov ebp, esp         // do EBP vložíme
    // adresu na zásobník, za alokaci lokálních
    // proměnných
```

```
    sub esp, [4 * n] - velikost
```

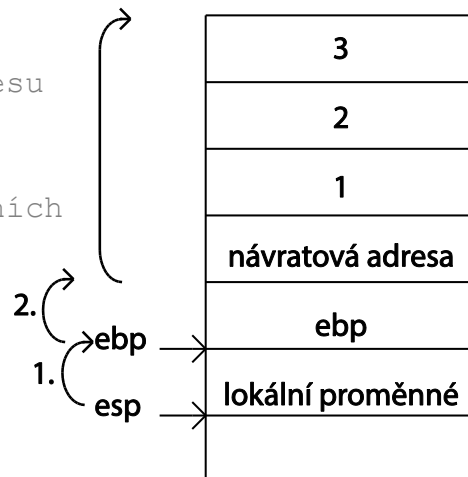
```
    ...
```

```
    ...
```

```
1. mov esp, ebp
```

```
2. pop ebp
```

```
    ret
```



### NÁVRAT Z FUNKCE // 00:40 ZHRUBA

1. obnovíme hodnoty registrů (které byly umístěny na zásobník (pop <reg>))
2. odstraníme lokální proměnné (lze k tomu použít obsah EBP)
3. obnovíme hodnotu EBP
4. provedeme návrat a odstraníme ze zásobníku jednotlivé argumenty
5. odstraníme argumenty ze zásobníku (lze použít přičtení k ESP - nejrychlejší)
  - první argument leží na adrese [ebp + 8], druhý na [ebp + 12], atd.
  - První lokální proměnná na [ebp - 4], druhá na [ebp - 8], atd.
  - **Nevýhoda konvence Pascal:** na zásobníku první přistoupíme k poslednímu argumentu a nevíme, kolik argumentů máme ještě uloženy. U **konvence CDECL** např.: u funkce printf dostaneme jako první ze zásobníku formátovací řetězec a dle něho odvodíme předpokládaný počet parametrů



## Obsah zásobníku

	...
	Argument n
	...
<b>EBP + 12</b> →	Argument 2
<b>EBP + 8</b> →	Argument 1
	Návratová hodnota
<b>EBP</b> →	Původní EBP
<b>EBP - 4</b> →	Lokální proměnná 1
<b>EBP - 8</b> →	Lokální proměnná 2
	...
<b>ESP</b> →	Lokální proměnná n

```
printf(„%i %i“, 42); // funkce přečte argumenty ze zásobníku, ikdyž je tam  
neuložila, protože předpokládá, že tam jsou uloženy a přečte hodnotu ze zásobníku
```

## UCHOVÁNÍ REGISTRŮ

- uchování všech použitých registrů na začátku každé funkce musí být efektivní  
// přidáváme argumenty zleva doprava, abychom mohli mít proměnný počet argumentů  
→ víme kde je první a k dalším se dopočítáme
- používá se konvence, kdy se registry dělí na:
  - **callee-saved** – o uchování hodnot se stará volaný (EBX, ESI, EDI)
  - **caller-saved** – o uchování hodnot se stará volající (EAX, ECX, EDX)
- po návratu z funkce mohou registry EAX, ECX a EDX obsahovat cokoliv

Nejjednodušší (špatné) řešení vypsání 10 řetězců:

```
mov ecx, 10
```

cyklus:

```
push řetězec
```

```
call printf    // pravděpodobně nám printf změní hodnotu ECX
```

```
add esp, 4
```

```
loop cyklus
```

# PŘERUŠENÍ

- Mechanismus umožňující reagovat na asynchronní události
- Nejčastěji vyvolané vnějším zařízením (např. stisk klávesnice, příchod síťového paketu), které vyžaduje CPU
- Pokud vznikne přerušení (Interrupt Request – IRQ, testuje se po provedení instrukce), činnost procesoru je zastavena a je vyvolána obsluha přerušení
- Po skončení obsluhy přerušení program pokračuje tam, kde byl přerušen → program by vůbec neměl poznat, že byl přerušen
- Obslužné rutiny – velice podobné běžným funkcím
- Procesor ví, kde jsou uloženy obslužné rutiny přerušení → číslo přerušení → vektor přerušení (pole adres)
- Souběh více přerušení je možné řešit těmito způsoby (záleží na implementaci dle OS): // v průběhu provádění kódu přerušení přijde požadavek na další přerušení
  - přerušení je možné přerušit
  - přerušení nelze přerušit (o řazení přerušení se postará řadič přerušení v procesoru)
  - systém priorit (přerušení s nižší prioritou nemůže přerušit, pokud již běží přerušení s vyšší a musí počkat)
    - např.: zvuková karta má vyšší prioritu než např.: diskové operace → čekání zvukové karty by znamenalo přerušování záznamu, praskání atd.
- Maskované a nemaskované přerušení (lze/nelze blokovat)
  - v momentě kdy probíhá přerušení a nelze vyvolat jiné přerušení – tzv. maskované

- Registr IDTR nám říká, čím které přerušení má být obslouženo

**Tabulka přerušení**

- Na x86 256 přerušení (prvních 32 speciální určení pro výjimky)
- Adresa vektoru přerušení (IDT – Interrupt Description Table) uložena v registru IDTR
- Při přerušení se na zásobník uloží aktuální adresa (CS + EIP + EFLAGS)

<b>0</b>
<b>1</b>
<b>...</b>
<b>255</b>

- Obslužná rutina obvykle ukládá i ostatní registry
- Proveďte se obsluha přerušení
- Návrat z obsluhy přerušení je realizovaný operací IRET // vrátí registr EF do původního stavu

## DALŠÍ UŽITÍ SYSTÉMU PŘERUŠENÍ

- Ošetření výjimek (dělení nulou, neplatná operace)
- Debugování (krokování, breakpointy)
- Implementace multitaskingu
- Explicitní vyvolání přerušení operace INT → systémové volání


**Chybí zde akorát tabulka přerušení....**

## I/O ZAŘÍZENÍ

### AKTIVNÍ ČEKÁNÍ

- Procesor se zařízením přímo (instrukce in, out – zápis/čtení hodnoty z portu)
- Výpočetně náročné (obzvlášť přenosy velkých dat), omezené na speciální operace (jen zápis/čtení)

### DMA

- Řadič DMAC dostane požadavek: čtení/zápis + adresu v paměti, CPU dál může pokračovat v činnosti
- Předá požadavek řadič zařízení (např. Disku)
- Zapisuje/čte data z/do paměti
- Dokončení je oznámeno řadiči DMAC
- DMAC vyvolá přerušení 

#34

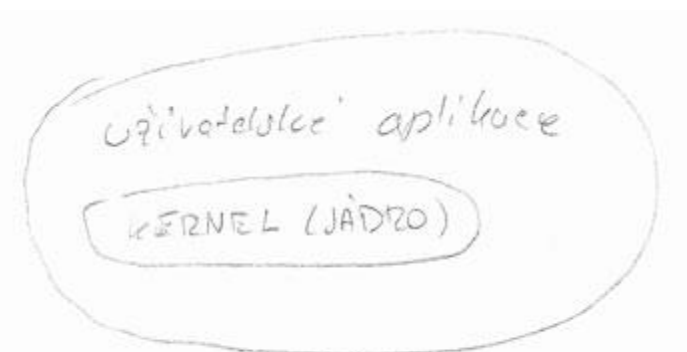
## IV. Přednáška

### SDÍLENÍ PAMĚŤOVÉHO PROSTORU

- zařízení mají přímý přístup k operační paměti

### REŽIMY PRÁCE CPU

- Od operačního systému očekáváme:
  - **správu a sdílení procesoru** (možnost spouštět více procesů současně)
  - **správu paměti** (procesy v paměti odděleny a nemohou se navzájem přepisovat)
  - **komunikace mezi procesy** (IPC)
  - **obsahu zařízení a organizaci dat** (souborový systém, síťové rozhraní, uživatelské rozhraní)
- není žádoucí, aby:
  - každý proces implementoval tuto funkcionalitu po svém
  - každý proces měl přístup ke všem možnostem hardwaru
- → jádro operačního systému → sdílené funkcionality, zajištění bezpečnosti/konzistence systému
- CPU různé režimy práce:
  - **privilegovaný** (kernel mode) – běží v něm jádro OS (umožňuje vše)
  - **neprivilegovaný** (user mode) – běží v něm aplikace (některé funkce jsou omezeny)
- existují i další, moc se nepoužívají
  - x86 má 4 módy označované ring 0-3 // 0 – jádro, 1 a 2 neužívané (s výjimkami), 3 aplikace
  - OS/2 používá tři úrovně oprávnění
  - VMS čtyři – kernel, executive, supervisor a user



# SYSTÉMOVÁ VOLÁNÍ

- Přepnutí režimu na režim jádra je řešeno pomocí: výjimky, přerušení nebo systémového volání // → přesně definované co se má stát, pokud funkce volá funkci z jádra
- **Systémové volání:** komunikace aplikace s jádrem OS pomocí přesně definovaného rozhraní
- Přepnutí do režimu jádra by mělo být co nejrychlejší, různé metody řešení

## SW PŘERUŠENÍ

- OS má definované číslo přerušení obsluhující systémová volání (Linux: 0x80, Windows NT: 0x2e, MS-DOS: 0x21) // vyvolá se pomocí: `int 0x80`
- Je zvolen registr (na i386 typicky EAX), který udává číslo požadavku (např. Otevření souboru atd.)
- Ostatní registry slouží k předání argumentů (případně se použije zásobník) // např.: cesta k souboru, oprávnění atd.
- Je vyvoláno SW přerušení

## SPECIÁLNÍ INSTRUKCE

- Pro zrychlení systémových aplikací volání bývají do ISA začleněny speciální instrukce
- i386: SYSENTER/SYSCALL, SYSEXIT/SYSRET

## VOLACÍ BRÁNY (CALL GATES)

- Volá se specifická funkce, která se postará o přechod z jednoho módu do druhého
- Využívá se mechanismus spojený se segmentací
- Možnost přecházet mezi různými úrovněmi oprávnění
- Používaly jej Windows NT (přesun ke specializovaným instrukcím)

## POZNÁMKA K HISTORII

- Reálný mód (používaný u MS-DOS) – nelze oddělit aplikace a jádro → aplikace mohly přistupovat k HW atd., ale bylo možné systém jednou aplikací odrovnat a je možné ho jednoduše zásadně poškodit
- BIOS – zajišťuje základní operace počítače (u rodiny PC) // obslužné rutiny pro práci s diskem, obrazovkou atd, implementované pomocí přerušení, po zavedení jsou tyto rutiny ignorované

# OPERACE S ČÍSLY S PLOVOUCÍ ŘÁDOVOU ČÁRKOU

## REPREZENTACE ČÍSEL S PLOVOUCÍ ŘÁDOVOU ČÁRKOU

- Standart IEEE 754 

- čísla zakódované ve tvaru:

$$\text{hodnota} = (-1)^{\text{znaménko}} \times \text{mantisa} \times 2^{\text{exponent}}$$

### JEDNODUCHÁ PŘESNOST

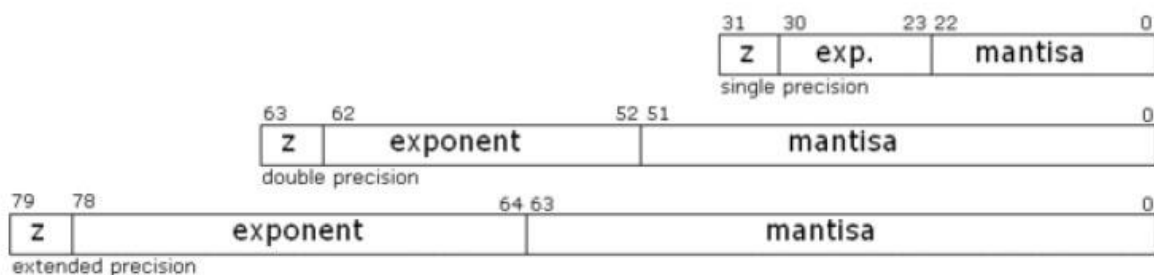
- Single precision, float
- velikost 32 bitů
- 1 bit znaménko, 8 bitů exponent (v doplňkovém kódu), 23 bitů mantisa

### DVOJITÁ PŘESNOST

- Double precision, double
- velikost 64 bitů
- 1 bit znaménko, 11 bitů exponent (v doplňkovém kódu), 52 bitů mantisa

### ROZŠÍŘENÁ PŘESNOST

- Extended precision, long double
- velikost 80 bitů
- 1 znaménko, 15 bitů exponent (v doplňkovém kódu), 64 bitů mantisa




## POZNÁMKY

- Existuje záporná nula – znaménko + nulový exponent a mantisa
- Existují nekonečna – maximální exponent + nulová mantisa
- Existuje NaN (not a number) – maximální exponent + nenulová mantisa
- Při použití v bankovníctví nebo tak → vznikají drobné chyby, lepší použít čísla s pevnou řádovou čárkou

## ZÁSOBNÍKOVÉ CPU

- Dvě koncepce CPU – registrové vs. zásobníkové
- Registrové: operandy uloženy v registrech (načtení/uložení dat z registru)
- Zásobníkové
  - operandy uloženy do zásobníku
  - přidávání/odebírání hodnot přes push/load, pop/store
  - operace pracují s vrcholem zásobníku – add, sub, dup, swap
  - příklad  $a2 - 1$ 

```
load 1
load a
dup      // duplikuj hodnotu na vrcholu zásobníku
mul
sub
```
- obvykle druhý zásobník pro volání funkcí
- výrazně jednodušší instrukční sada 

## FLOATING-POINT UNIT (FPU)

- Řeší výpočty s čísly s plovoucí řádovou čárkou
- Pracuje s 80 bitovými hodnotami (nutné převody)
- Vychází z koprocessoru 80x87 (původně oddělená jednotka)
- → odlišná architektura + omezení
- → zásobníkový procesor, přenášení dat pouze přes paměť
- Zásobník má kapacitu 8 hodnot
- Se zásobníkem jde pracovat jako s registry (označované jako ST(0) – ST(7))
- ST(0) ukazuje vrchol zásobníku // při vložení dat se všechny hodnoty posunou dolů → vrchol bude vždy ST(0)



## OPERACE

- FLD, FST – načtení hodnot na zásobník, odebrání hodnot ze zásobníku (dále FLDZ, FLD1, FLDPI pro uložení konstant)
- FADD, FSUB, ... - numerické operace, jako jeden argument se používá vrchol zásobníku (registr ST(0)), jako druhý je možné použít kteroukoliv hodnotu ze zásobníku (registr ST(1-7)), případně hodnotu v paměti
- větvení kódu řešeno pomocí porovnávání FCOM a podmíněných přiřazení FCMOVx (FCMOVE (pokud jsou čísla si rovny), FCMOVB (přiřazení pokud je číslo menší), ...)
- další operace FSQRT, FSIN, FCOS,...

## VOLÁNÍ FUNKCÍ

- Při volání funkcí jsou hodnoty předávány přes zásobník
- Návrátová hodnota přes ST(0)

## DALŠÍ ROZŠÍŘENÍ

- Podpora „multimédií“
- SIMD (single instruction multiple data)

## MMX

- 64bitové registry mmo – mm7 (shodné s ST(0) – ST(7))
- Možné používat jako vektor 1-, 2-, 4-, 8bytových celých čísel
- Operace se saturací // saturace – při přetečení nastaví na nejvyšší možnou, zvyšuje efektivitu např. při práci s barvami (Red:  $250 + 10 = 4 \rightarrow 255$ )



## SSE

- 128 bitové registry XMM0-XMM7
- Kapacita pro 4 FP hodnoty s jednoduchou přesností
- Základní aritmetika // sečítat vektory, násobit skalárem atd.



## SSE<sub>2</sub>

- Operace pro práci s hodnotami s dvojitou přesností (CAD)
- Možnost používat hodnoty v registrech XMM0-7 jako vektory celých čísel (16 8bitových hodnot, 8 16bitových, atd.), včetně saturace

## AMD64

- 64 bitové rozšíření ISA procesorů x86 (označovaná i jako EM64T, x86\_64, x64)
- Rozšíření velikosti registrů na 64 bitů (rax, rdx, rcx, rbx, rsi, rdi, rsp, rbp)
- Nové 64bitové registry r8-r15
  - spodních 32 bitů jako registry rXd (např. r8d)
  - spodních 16 bitů jako registry (např. r8w)
  - spodních 8 bitů jako registry (např. r8b)
- Nové 128bitové registry xmm8-xmm15
- Nejnovější procesory s AVX (Sandy Bridge, Bulldozer) rozšiřují xmm0-xmm15 na 256 bitů (registry ymm0-ymm15)
- Adekvátní rozšíření operací (prefix REX), omezení délky instrukce na 15B
- V operacích je možné používat jako konstanty maximálně 32bitové hodnoty → výjimkou je operace (`movabs r, i`) // omezení aby jednotlivé instrukce nebyly moc velké
- Větší množství registrů umožňuje efektivnější volání funkcí (podobné fastcall)
- Možnost zakódovat strukturovanou hodnotu do registru
- Na zásobníku se vytváří stínové místo pro uložení argumentů
- Zarovnání zásobníku na 16 B
- Sjednocení volacích konvencí (v rámci platformy)

## LINUX/UNIX

- Prvních 6 argumentů: rdi, rsi, rdx, rcx, r8, r9
- Čísla s plovoucí řadovou čárkou přes xmm0-xmm7
- Zbytek přes zásobník
- Návrátové hodnoty přes rax nebo xmm0

## WINDOWS

- První 4 argumenty: rcx, rdx, r8, r9
- Čísla s plovoucí řadovou čárkou přes xmm0-xmm3
- Zbytek přes zásobník
- Návrátové hodnoty přes rax nebo xmm0
- Rozšíření adresního prostoru
- Fyzicky adresovatelných  $2^{40}$  B paměti (virtuální paměť  $2^{48}$  B)

## REŽIMY PRÁCE

- 64bitová ISA je velice podobná 32bitové → minimální režie
- **Long mode:** dva sub módy (ve kterých jsou k dispozici 64bitové rozšíření)
  - 64 bit mode v OS i aplikace v 64bitovém režimu

- Compatibility mode: umožňuje spouštět 32 bitové aplikace v 64 bitovém OS
- **Legacy mode** – režim pro zajištění zpětné kompatibility (protected, read)
- Pro výpočty s čísly s plovoucí řadovou čárkou se používají operace SSE, SSE2


## AT&T

- Kód v assembleru jde zapsat vícero způsoby
- Dosud diskutovaná syntaxe assembleru se označuje jako Intel
- Často se používá alternativní syntaxe AT&T
- Operace zapisované ve tvaru  
`<jmeno><velikost> zdroj, cil`
- `<jmeno>` označuje název operace `mov`, `add`, `cmp`
- `<velikost>` je písmeno `b`, `w`, `l`, `q` udává velikost operandů
- Registry se zapisují ve tvaru `%reg` (např. `%eax`)
- Konstanty začínají znakem `$` (např.: `$100`)
- Adresace paměti ve tvaru `disp(base, index, scale)`  
(např. `-10(%ecx, %ebx, 2) → [ecx + 2 * ebx - 10]`)

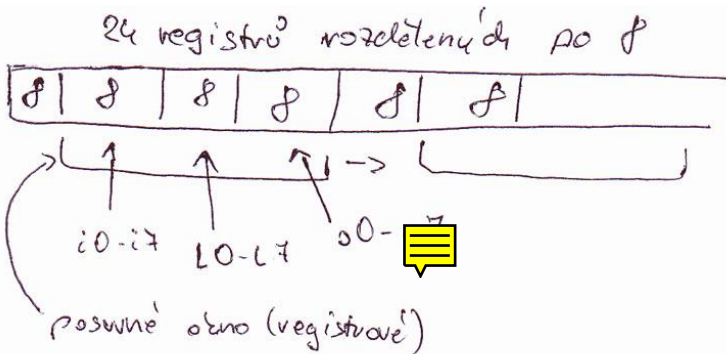
AT&T	Intel
<code>Pushw %ax</code>	<code>Push ax</code>
<code>Movl \$100, %eax</code>	<code>Mov eax, 100</code>
<code>Addl %ebx, %eax</code>	<code>Add eax, ebx</code>
<code>Subl (%eax), %ecx</code>	<code>Sub ecx, [eax]</code>
<code>Subl (%eax, %ebx), %ecx</code>	<code>Sub ecx, [eax + ebx]</code>
<code>Andw \$42, -16(%eax)</code>	<code>And word ptr [eax - 16], 42</code>

## SPARC



- Rodina procesorů (kompletní dokumentace GPL)
- Každá instrukce zabírá v paměti 4B 
- Snaha eliminovat množství operací
- Operace běžně se třemi operandy
- Velké množství registrů (řádově stovky), běžně dostupných 32 registrů
- Globální registry `go-g7` (`go` je vždy nula)
- Registrové okno – 24 registrů  
// mělo by být k dispozici alespoň 2-3 okna
  - `io-i7` – argumenty předané funkci

- lo-l7 – lokální proměnné
- oo-o7 – argumenty předávané další funkci
- Speciální využití některých registrů
  - fp – frame pointer (i6)
  - sp – stack pointer (o6)
  - návratová adresa – i7/o7



#### • Příklady operací

```
add %i0, 1, %l1 ; l1 := i0 + 1
subcc %i1, %i2, %i3 ; i3 := i1 - i2
subcc %i1, %i2, %g0 ; g0 := i1 - i2 (cmp)
or %g0, 123, %l1 ; l1 := g0 | 123 (mask)
```

#### • Malá velikost instrukce

- Operace neumožňují adresovat paměť → specializované operace ld, st  
// je možné pracovat pouze se 32 bity
- Interně se pracuje s celými registry
- Jako konstanty jde běžně používat pouze hodnoty -4096 – 4095
- Přiřazení velkých čísel ve dvou krocích

```
sethi 0x226AF3, %l1 ; nastaví horní bity
or %l1, 0x1EEF, %l1 ; nastaví dolní bity
```

- Jednoduché instrukce
- Potenciálně rychlejší zpracování
- Skoky (podmíněné i nepodmíněné) se neprovádí okamžitě
- K optimálnímu využití pipelingu se přidává delay slot // prve se provede instrukce za skokem a až poté skok
- Ještě je zpracována následující instrukce
- Možnost nastavit annul bit, operace v delay slotu provede jenom, pokud se provede i skok

```
cmp %l1, %l2
bl, a addr
mov %g0, %l3
```

## V. Přednáška

### ARM

- Rodina procesorů typicky využívaná v emebded a přenosných zařízení
- Optimalizace na nízkou spotřebu napájení a paměti
- Není jeden výrobce
- Základní jádro je licencováno výrobcům k výrobě SoC (Qualcom Snapdragon, nVidia Tegra, Apple A4-A6, ...)

### INSTRUKČNÍ SADA

- Podpora několika různých typů instrukčních sad (+ rozšíření dle modelu)
- Load/store architektura
- 32 registrů z toho jen 16 je v daný okamžik použitelných (R0-R15)
- R13 – stack pointer, R14 – Link registr, R15 – program counter
- Registry > R8 jsou přepínány podle aktuálního režimu procesoru (např.: při ošetření přerušení)

### INSTRUKČNÍ SADA (PŮVODNÍ)

- Všechny instrukce o velikosti 32 b
- Obvykle 2-3 operandy, příznaky nastavují jen programátorem určené instrukce
- Možnost podmíněného vykonávání instrukcí
- Jako přímé hodnoty lze používat jen 8 b čísla

Absolutní hodnota čísla:

```
XOR r1, r1, r1      ; r1 := r1 XOR r1
CMP ro, r1           ; nastav příznaky ro – R1
SUBLT ro, r1, ro     ; ro := r1 – ro
```

### INSTRUKČNÍ SADA THUMB

- Zahuštění kódu (velikost instrukce 16 bitů)
- Zmenšení operandů (podobná ISA x86)
- Bez možnosti podmíněného provádění instrukcí

## INSTRUKČNÍ SADY

- Thumb 2 – podobná Thumb, efektivnější kódování instrukcí (lepší výkon)
- Jazzelle – spouštění Java bytecode

## SHRNUTÍ KONCEPCÍ

### RISC: REDUCED INSTRUCTION SET COMPUTER

- Zjednodušený návrh a implementace CPU
- Rychlejší běh, určitá omezení

### CISC: COMPLETE (COMPLEX) INSTRUCTION SET COMPUTER

- Poskytuje operace blízké vyšším programovacím jazykům
- Snadné pro ruční programování
- Náročné na implementaci CPU (+ již nepoužívané instrukce v ISA)

### REÁLNĚ...

- Procesory typu CISC provádí rozklad operací na mikrooperace -> (vnitřně RISC)
- Další úroveň abstrakce
- Vnitřně dochází ještě k dalším úpravám kódu, např. přejmenování registrů
- **Out-of-order execution** → rozdělení (mikro) operací jednotlivým jednotkám → určitá forma paralelismu
- Plánování out-of-order execution komplikuje návrh CPU

### VLIW: VERY LARGE INSTRUCTION WORD

- Snaha využívat několik funkčních jednotek
- Jedna instrukce může obsahovat několik operací
- Souběžné zpracování
- Spolupráce s překladačem → „CPU nemusí hádat, jak poběží program“  
// paralelismus přesunut z procesoru na překladač
- Složitější návrh dekódovací jednotky

# PŘEKLAD PROGRAMU A KNIHOVNY

## PŘEKLAD PROGRAMU

1. **Preprocesor** – expanduje makra, odstraní nepotřebný kód, načte požadované hlavičkové soubory (např. math.h) – deklarace struktur, deklarace prototypů, atd.
2. **Překladač** – generuje kód v assembleru
3. **Assembler** – vygeneruje objektový kód (foo.c → foo.obj/foo.o)
4. **Linker** – sloučí několik souborů s objektovým kód + knihovny do spustitelného formátu

## UKÁZKA MAKRA C

```
#define foo(x) x + x  
#include „foo.h“  
#include <math.h>
```

```
foo(a+x) -> a++ + a++
```

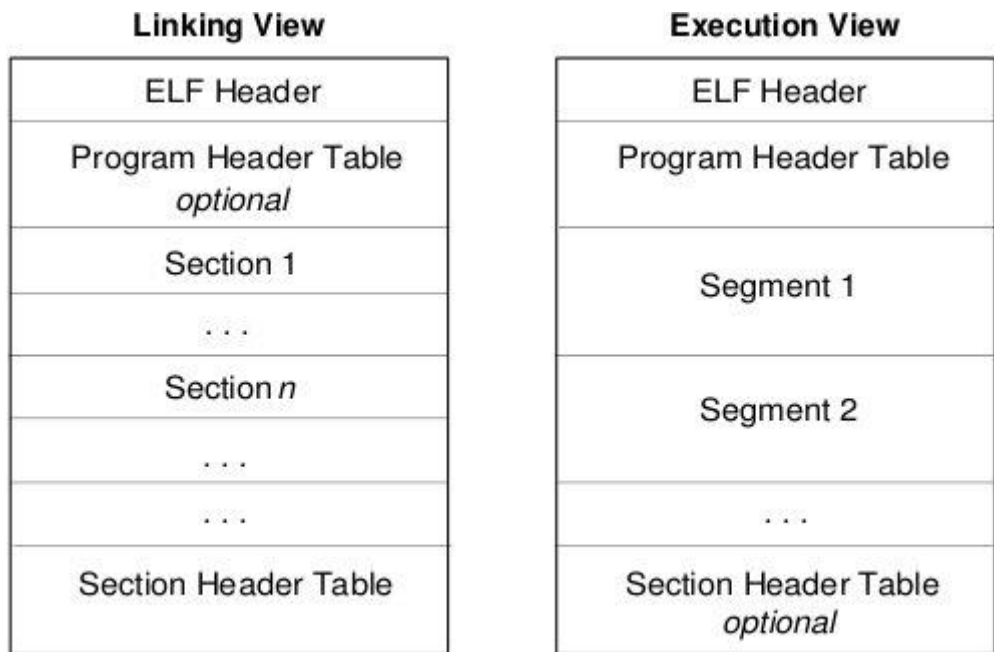
## POZNÁMKY

- Některé kroky mohou být sloučeny nebo vypuštěny
- Některé vyšší programovací jazyky jsou (BÚNO) překládány do nižšího jazyka (např. C)
- Oddělený překlad do objektových souborů a jejich spojení →
  - Možnost kombinovat různé programovací jazyky
  - Komplikuje interprocedurální optimalizace

## OBJEKTOVÝ SOUBOR

- Formát specifický pro každý OS
- Obecně obsahuje
  - **Hlavička** – informace o souboru
  - **Objektový kód** – strojový kód + data
  - **Exportované symboly** – seznam poskytovaných symbolů (např. funkce nedeklarované jako static)
  - **Importované symboly** – seznam symbolů použitých v tomto souboru
  - **Informace pro přemístění** – seznam míst, které je potřeba upravit v případě přesunutí kódu
  - **Debugovací funkce**
- Rozdělení sekce

- **Kód**
- **Data jen pro čtení** (konstanty, const)
- **Inicializovaná data** (globální proměnné, statické proměnné)
- Program v paměti obsahuje navíc informace o neinicializovaných datech atd.
- Možnost sdílet jednotlivé části mezi instancemi programu
- Formát často sdílený i binárními soubory



OSD1980

## LINKOVÁNÍ

- Spojí jednotlivé objektové soubory do spustitelného formátu (sloučí jednotlivé akce)
- Postará se o správné umístění kódu a vyřešení odkazů na chybějící funkce a proměnné
- Připojení knihoven (hlavičkové soubory většinou neobsahují žádný kód)

## STATICKY LINKOVANÉ KNIHOVNY

- Archiv objektových souborů (+ informace o symbolech)
- **Výhody:** jednoduchá implementace, nulová režie při běhu aplikace, žádné závislosti
- **Nevýhody:** velikost následného binárního souboru, aktualizace knihovny -> nutnost rekompile



## DYNAMICKÉ LINKOVÉ KNIHOVNY (DLL)

- Knihovna je načtena až při spuštění programu
- Sdílení kódu
- Nutnost provázat adresy v kódu s knihovnou
- Nutnost spolupráce OS

## DYNAMICKY LINKOVANÉ KNIHOVNY

- **Problém:** umístění knihovny v paměti

### ŘEŠENÍ V UNIXU

- Sdílené knihovny (shared objects, foo.so)
- → position independent code (PIC) – kód, který lze spustit bez ohledu na adresu paměti
- x86 používá často relativní adresování (i tak PIC pomalejší než běžný kód)
- Při spuštění dynamický linker (ld.so) provede přenastavení všech odkazů na vnější knihovny
- Global Offset Table (GOT) – tabulka sloužící k výpočtu absolutních adres (nepřímá adresace)
- Procedure Linked Table (PLT) - tabulka absolutních adres funkcí
  - Na začátku PLT obsahuje volání linkeru
  - Při volání funkce se provede skok do PLT
  - Nastavení se informace o funkci, nastavení záznamu v PLT
  - Linker zavolá funkci
  - Další volání se provádí bez účasti linkeru → adresa v PLT

#51

### ŘEŠENÍ VE WINDOWS

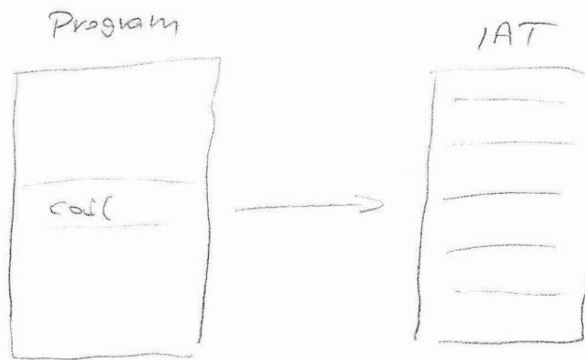
- Dynamic-link library (DLL)
- Windows nepoužívá PIC → každá knihovna má svou adresu v paměti
- V případě kolize nutnost přesunu + přepočítání absolutních adres
- Každý program obsahuje *import address table* (IAT) – tabulka adres volaných funkcí (nepřímá adresace)
- Inicializace při spuštění
- Volání přes `call [adresa] nebo thunk table`  
... kód ...  
`00401002 CALL 00401D82`

```

... thunk table ...
00401D82 JMP DWORD PTR DS:[40204C]
... adresy funkcí ...
40204C > FC 3D 57 7C ; adresa

```

- Vyhledávání funkcí podle čísla nebo jména



- Tyto řešení vyžadují spolupráci s OS

## DYNAMICKY NAHRÁVANÉ KNIHOVNY

- Možnost explicitně nahrát knihovnu za běhu
- Implementace pluginu
- Mechanismus podobný dynamickému linkování
- Unix: dlopen, dlsym (vrátí ukazatel na knihovnu; vyhledávání funkcí podle jména)
- Windows: LoadLibrary, GetProcAddress (vrátí ukazatel na knihovnu, poté můžeme zavolat funkci, kterou chceme)
- Kombinace: zpožděné načítání knihoven

## VIRTUÁLNÍ STROJE

- Virtualizace systému vs. virtualizace procesu
- Program se nepřekládá do strojového kódu cílového procesoru
- **Bytecode**: instrukční sada virtuálního procesoru (virtuálního stroje VM)
- Bytecode → interpretace jednotlivých instrukcí nebo překlad do instrukční sady cílového procesoru běhovým prostředím
- Přenositelný kód nezávislý na konkrétním procesoru
- Možnost lépe kontrolovat běh kódu (oprávnění, přístupy)

- Režie interpretace/překladač
- VM může řešit i komplexnější úlohy než běžný CPU (správa paměti, výjimky, atd.)
- Příklady: Java Virtual Machine (& Java Byte Code), Common Language Runtime (& Common Intermediate Language), UCSD Pascal (p-code), LLVM atd.

## VI. Přednáška

### JIT PŘEDKLAD (JUST IN TIME)

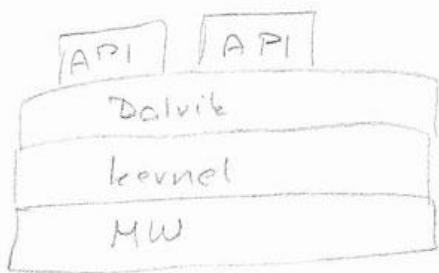
- Běhové prostředí generuje kód dané architektury za běhu (Von Neumanova architektura)
- Možnost optimalizace pro konkrétní typ CPU
- Optimalizace podle aktuální prováděného kódu (profilování)

### ZÁSObNÍKOVÉ VIRTUÁLNÍ STROJE

- Jednoduchá instrukční sada → snadná implementace
- Potřeba více instrukcí, nicméně kratší kód
- JVM, CLR

### REGISTROVÉ VIRTUÁLNÍ STROJE

- Efektivní překlad do instrukční sady (pipelined) procesorů
- Odolnější proti chybám
- Dalvik – Android, minimální spotřeba paměti; Parrot – Perl 6, LLVM (low level virtual machine) – optimalizace přes Single Static Assignem



- umožňuje přeměnit aplikace  
mezi mobility, pokud obsahují  
přeložený kód pro určitý procesor

### JAVA VIRTUAL MACHINE A JAVA BYTOCODE

- 1995: SUN programovací jazyk Java 1.0
- Překlad Java → Java Bytecode (JBC)

- JBC vykonává pomocí Java Virtual Machine (JVM)
- Implementace JVM není definovaná (pouze specifikuje chování), JBC lze:
  - Interpretovat
  - Přeložit do strojového kódu daného stroje (JIT i AOT)
  - Provést pomocí konkrétního CPU
- JVM – virtuální zásobníkový stroj
- Malý počet instrukcí (< 256)
- Zásobník obsahuje rámce (rámec je vytvořen při zavolání funkce)
  - Lokální proměnné, mezi výpočty
  - Operand stack – slouží k provádění výpočtů
- Heap s automatickou správou paměti
- Jednoduché i velmi komplexní operace (volání funkcí, výjimky)
- Základní aritmetika se primitivními datovými typy (hodnoty menší, než int převedeny na int)
- Speciální operace pro práci s prvními argumenty, lokálními prostředím, jedničkou, nulou
- Pouze relativní skoky

## PŘÍKLAD

```
Public static void foo (int a, int b) {
    System.out.println(a + b);
}
```

Vygenerovaný Byte Code:

```
0. Getstatic          #21; //Field java/lang/System.out:Ljava...
3. iload_0
4. iload_1
5. iadd
6. Invokevirtual      #27; //Method java/io/PrintStream...
9: return
```

## COMMON LANGUAGE RUNTIME

- Microsoft .NET implementuje obdobný přístup
- Common Language Runtime (CLR) + Common Intermediate Language (CIL) = běhové prostředí + bytecode
- Koncepčně velice podobné JVM a JBC
- Od začátku navržen s podporou více jazyků

- Při prvním zavolání metody → překlad do strojového kódu CPU

## OPUŠTĚNÍ BĚHOVÉ PROSTŘEDÍ

// občas je potřeba interagovat se systémem, na kterém je spuštění. K tomu slouží dále uvedené rozhraní. Nevýhodou je, že využití tohoto API znemožňuje přenositelnost na jinou platformu než Windows

- Java: Java Native Interface – rozhraní pro spolupráci s C++
- .NET: platform Invocation Services (P/Invoke) – umožňuje spouštět kód z DLL  
// prakticky nemožná přenositelnost kvůli volání => praktická nemožnost implementovat jinde jak na Windows

## MACOS X

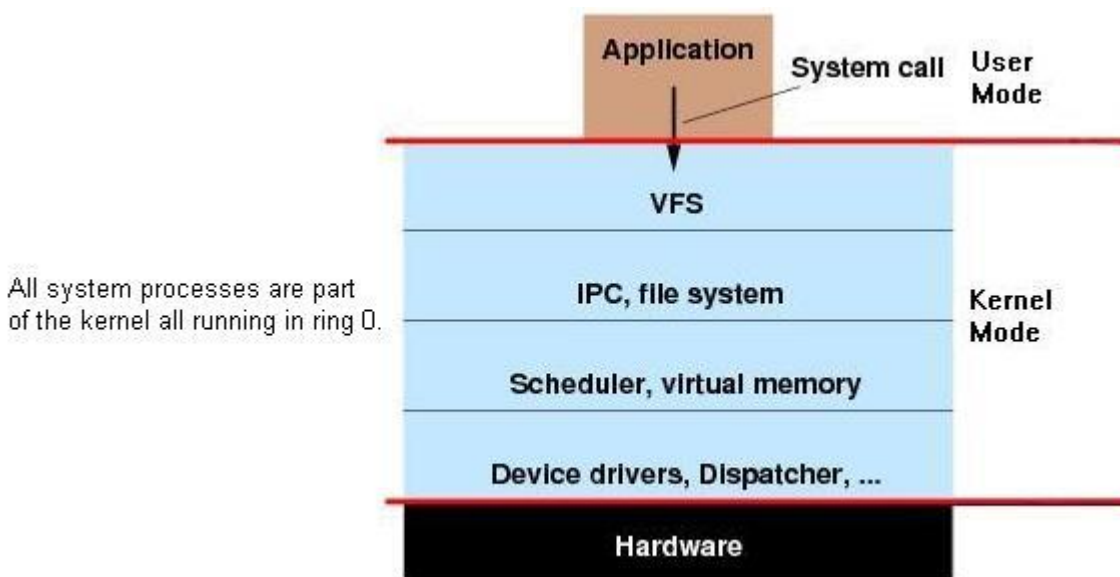
### MULTI-ARCHITECTURE BINARIES

- Součástí jednoho binárního souboru je kód pro víc architektur (např.: Motorola 68k + PowerPC)
- **Universal binaries** – možnost více platforem (nejen i386/PowerPC, ale i 32/64 bitů)
- Pro přechod od procesorů PowerPC k i386 → technologie *Rosetta* → překlad kód  
// umožňuje spouštět kód pro PowerPC na i386

# ARCHITEKTURA A HISTORIE OPERAČNÍCH SYSTÉMŮ

## ARCHITEKTURA OS

- Od operačního systému očekáváme:
  - **Správu a sdílení procesoru** (možnost spouštět více procesů současně)
  - **Správu paměti** (procesy jsou v paměti odděleny)
  - **Komunikaci mezi procesy** (IPC)
  - **Obsluhu zařízení a organizaci dat** (souborový systém, síťové rozhraní, uživatelské rozhraní)
- Není žádoucí, aby:
  - Každý proces implementoval tuto funkcionalitu po svém
  - Každý proces měl přístup ke všem možnostem hardwaru
- → jádro operačního systému
- CPU různé režimy práce
  - **Privilegovaný** (kernel mode)
  - **Neprivilegovaný** (user mode)
- Přejechod mezi režimy pomocí systémových volání (SW přerušení, speciální instrukce, speciální volání)



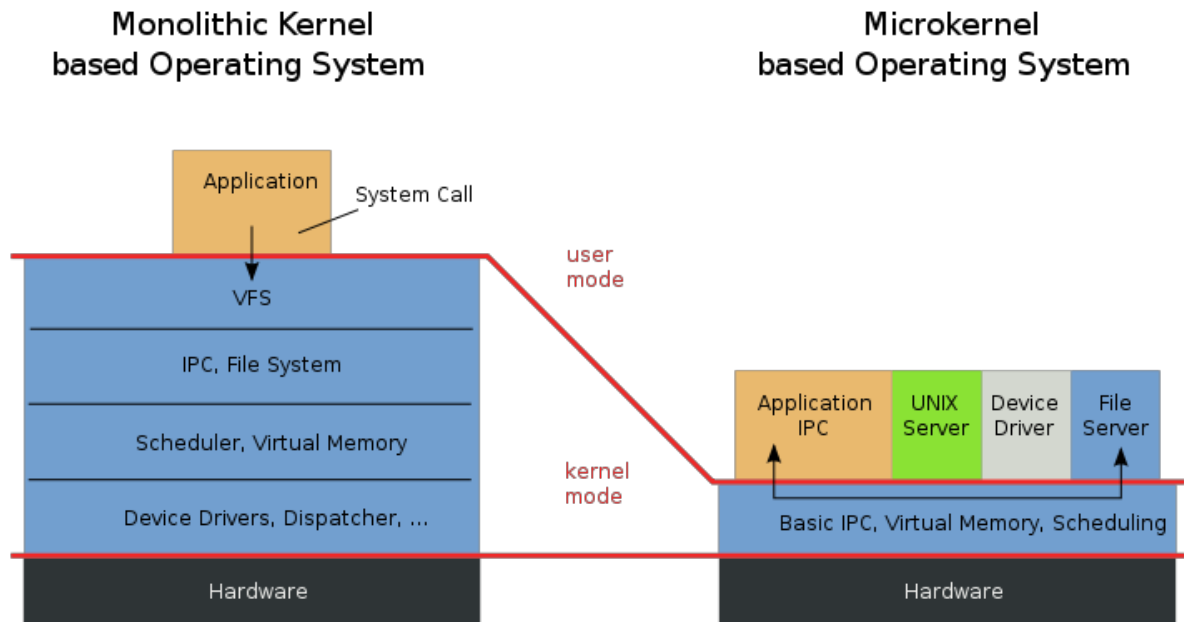
# ARCHITEKTURA JÁDRA

## MONOLITICKÉ JÁDRO

- Vrstvená architektura
- Moduly
- Všechny služby hromadě → lepší výkon
- Problém s chybnými ovladači
- Linux, \*BSD
- **Nevýhoda:** pokud se objeví chyba ve vrstvách jaderného režimu, může poškodit systém, musí se zajistit bezpečnost celého systému jako celku

## MIKROJÁDRO

- Poskytuje správu adresního prostoru a procesů, IPC //IPC – meziprocenší komunikace
- Oddělení serverů (služeb systému); běžné procesy se speciálními právy → bezpečnost
- Možnost restartu serverů
- Zavedení systému
- MINIX, QNX
- **Výhoda:** lze rozdělit na menší subsystémy (na procesy se speciálními právy -> lepší bezpečnost), při chybě stačí restartovat subsystém a ne celý systém, malý systém -> jednodušší na pochopení
- **Nevýhoda:** meziprocenší komunikace je pomalá



## HYBRIDNÍ JÁDRO

- Kombinují prvky obou přístupů
- Část funkcionality v jádře, část mimo
- Windows NT

## EXOKERNEL

// mikrokernel zahnaný do extrémů

- Řeší jen to nejnútnejší → přidělování HW zdrojů
- Neposkytuje HW abstrakci → knihovny v uživatelském prostoru

## PROBLÉMY S NÁVRHEM OS

- Je potřeba mít vizi a cíl // obtížné definovat cíl: abstrakce nad nějakým hardwarem? Jaké aplikace by tam měli běžet? Všechny?
- Definovat primitiva, datové abstrakce, izolace
- Způsob provádění programů:
  - Algoritmicky
  - Událostmi řízené
- Různé přístup OS k datům:
  - Vše je **páska** (původní FORTRAN)
  - Vše je soubor **soubor** (UNIX) // proc, sys, /sys/power/state – soubor pro např.: uspání



- Vše je **objekt** (Windows)
  - Vše je **dokument** (Web)
- Souběžný přístup
- Použitelnost SW/designu OS vs. vývoj HW
- Správa HW (nespočetné množství)
- Přenositelnost, specifikované možnosti
- → KISS, jen to nejnútnější → omezení počtu volání (exec vs. CreateProcess)

## HISTORIE OS

### PRAVĚK OS PRO PC

#### CP/M

- Pro procesory Intel 8080, 8085, Zilog 80
- Basic input/output systém, basic disk operating system (rezidentní)
- Console command processor // něco jako jednoduchý Shell
- Jednoduchý přístup k diskům (disketám) // jeho v podstatě jediná činnost

#### MS-DOS

- Byl navržen pro procesory Intel 8080, 8086
- Jednouchátratský, jednouchátratský
- Omezené možnosti práce s pamětí (problém s ovladači)
- Více paměti přes rozšíření → DOS Protected Mode Interface

## HISTORIE UNIXŮ

- MULTICS (MULTIplexed Information and Computing Service): MIT + Bell Labs + GE
  1. „Výpočetní výkon ze zásuvky“ // hlavní myšlenka – poskytnuti výpočetního výkonu pro více uživatelů přes terminál
  2. Současná práce pro více uživatelů // současné ukládání více souborů do paměti atd...
  3. Jednotná paměť: výlučné mapování souborů do paměti, paměť procesoru součástí FS // je jedno zdali se zapisovalo do paměti nebo na disk
  4. Segmentace a stránkování
  5. Dynamické linkování

- Kernighan → UNICS (UNiplexed Information Computing Service) – obklesžený MULTICS pro PDP-7
- + Ritchie, Thmoson → port na počítače PDP-11
- Snažší psaní a portování → vznik C // aby nemuseli vše kódovat v Assembleru
- Počítače PDP-11 populární na univerzitách UNIX
- ARPA + DARPA + UCB → First Berkeley Software Distribution
  1. Dále vyvíjené → 3BSD, 4BSD
  2. Stránkování, FS s dlouhými názvy, TCP/IP
  3. Řada nástrojů vi, csh, překladače
- BSD základ pro další unixy → SUN
- Současně s BSD, vydává v polovině 80. Let AT&T → System III a System V
- Vznikla nová řada implementace unixu – HP-UX, AIX, SunOS, atd.
- Nekompatibilita mezi různými verzemi unixů
- Snaha o jednotnost
  1. **Systém V Interface Definition**
  2. **POSIX** (Portable Operating System Interface for Unix; IEEE 1003.1) – nepopisuje jádro, ale funkce standartní knihovny a funkcí (průnik funkcionality) // pokud program splňuje POSIX, není problém přenést mezi verzemi unixu, ale nemůžeme používat specifické vlastnosti jednotlivých systémů
  3. **Open Software Foundation** (OSF) – snaha specifikovat i zbylé části systému (X11), neujal se
- Jazyk C standardizován jako ISO a ANSI // přenositelnost mezi Unixovými systémy je velká
- 1987 Tanenbaum vydává MINIX – výukový OS založený na mikrokernelu; kompatibilní s POSIX
- Začátek 90. let → 386BSD → NetBSD, FreeBSD, OpenBSD
- 1984: Richard Stallman → GNU is Not Unix
  - Pokus o svobodnou reimplentaci Unixu
  - General Public License (GPL) // lze vzít kód, upravit ho a dále distribuovat za podmínky, že dále bude kód volně distribuován
  - Vývoj základních nástrojů včetně editoru Emacs, GNU C Compileru // skládá se z mnoha víceúčelových programů pouze
  - Dlouho chybělo jádro s definovanými rozhraními
  - Pokus použít 4.4 BSD nebo vyvinout vlastní (Hurd)
- 1991: Linus Torvalds → Jádro Linux 0.01
  - Původně vyvíjeno na MINIXu s GCC z projektu GNU
- Spojení GNU + Linux → GNU/Linux // nástroje GNU + jádro Linux
- Možnost použít i jiné kombinace GNU/Hurd, GNU/FreeBSD

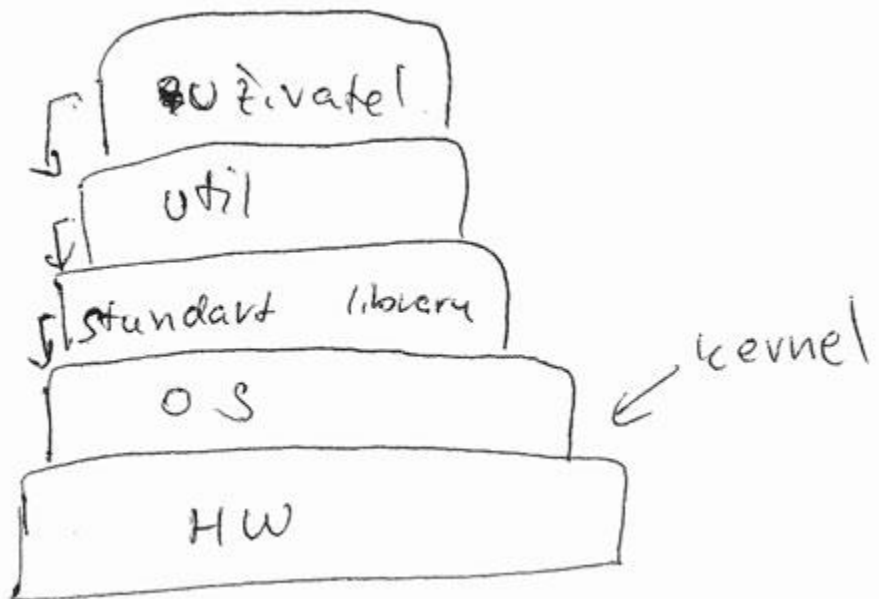
- Řada aplikací portovaná na GNU/Linux → de facto standart

## ZÁKLADNÍ VLASTNOSTI UNIXŮ

- Od začátku počítá s víceuživatelským přístupem
- Počítá se spolupracujícími uživateli
- Počítá se zkušeným uživatelem (nejlépe programátorem)
- Snaha být jednoduchý, elegantní důsledný → např. všechno je soubor (textový soubor → protokoly)
- Snaha omezit redundanci
- Možnost komponovat věci do větších celků
- Transparentnost → debugování
- Není jeden způsob, jak dělat věci správně
- Eric S. Raymond:  
The Art of Unix  
Programming

## ROZHRANÍ V UNIXU

- Vrstvená architektura a pojící prvky
- Systémové volání
- Volání knihoven
- Uživatelské aplikace (utility – práce se soubory, filtry, vývojové nástroje, administrace)
- → schopnost přežít 40 let



## DALŠÍ UNIXY

### BSD

- Volnější licence
- Monolitické jádro

- Základní nástroje vyvíjené společně, adopce GNU nástrojů
- FreeBSD
- NetBSD – podporuje 57 platforem
- OpenBSD – odvozeno z NetBSD, zaměřeno na bezpečnost
- Oddělený vývoj – nejedná se o distribuce

## VII. Přednáška

### (GNU) MACH

- Unixový mikrokernél
- Základní jednotka je task skládající se z vláken
- Komunikace přes porty (fronty zpráv) – tasky získávají oprávnění k jednotlivým portům
- Podpora paralelismu
- Problémy s výkonem (přepínání kontextu, validace zpráv)

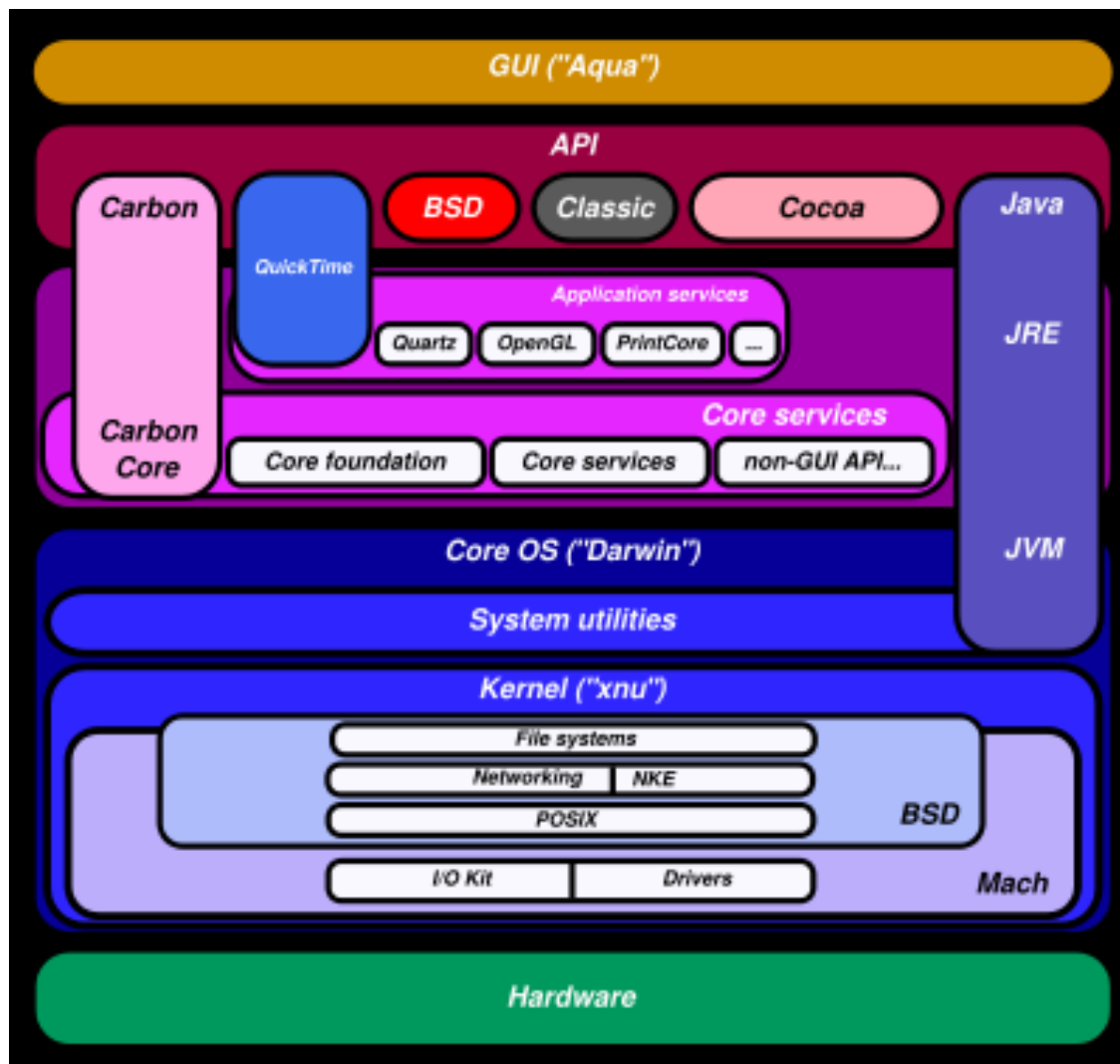
### GNU HURD

- Původně zamýšleno jako jádro pro GNU
- K Mach přidává servery (sloužící jako ovladače, autorizace, spouštění aplikací, implementace FS, atd.)
- Pokus přeportovat na jiný typ jádra – L4, Coyotos // Coyotos – psaný ve svém vlastním jazyce BitC (Scheme a C), je u něj formálně dokázaná správnost

### XNU/DARWIN

- X is Not Unix
- Část Darwinu, část Mac OS X
- Hybridní kernel
- Slučuje jádro Mach a Free BSD
- Z Mach si bere převážně správu procesoru, paměť, IPC
- Z BSD bere POSIX API, síťování, souborový systém
- Mac OS X certifikovaný jako Unix
- Rozhraní nad jádrem (Framework, kity)
  - Cocoa (Objective-C)
  - Carbon (zpětná kompatibilita)
  - Quartz 2D, OpenGL

## Architektura MacOS X



## HISTORIE WINDOWS

### WINDOWS 1.0, 2.X

- Nástavba nad MS-DOS
- Kooperativní multitasking // pokud proces běží a sekne se -> musí se restartovat celý systém
- Softwarová virtuální paměť založená na segmentaci

### WINDOWS 3.X

- Přidávají lepší práci s pamětí

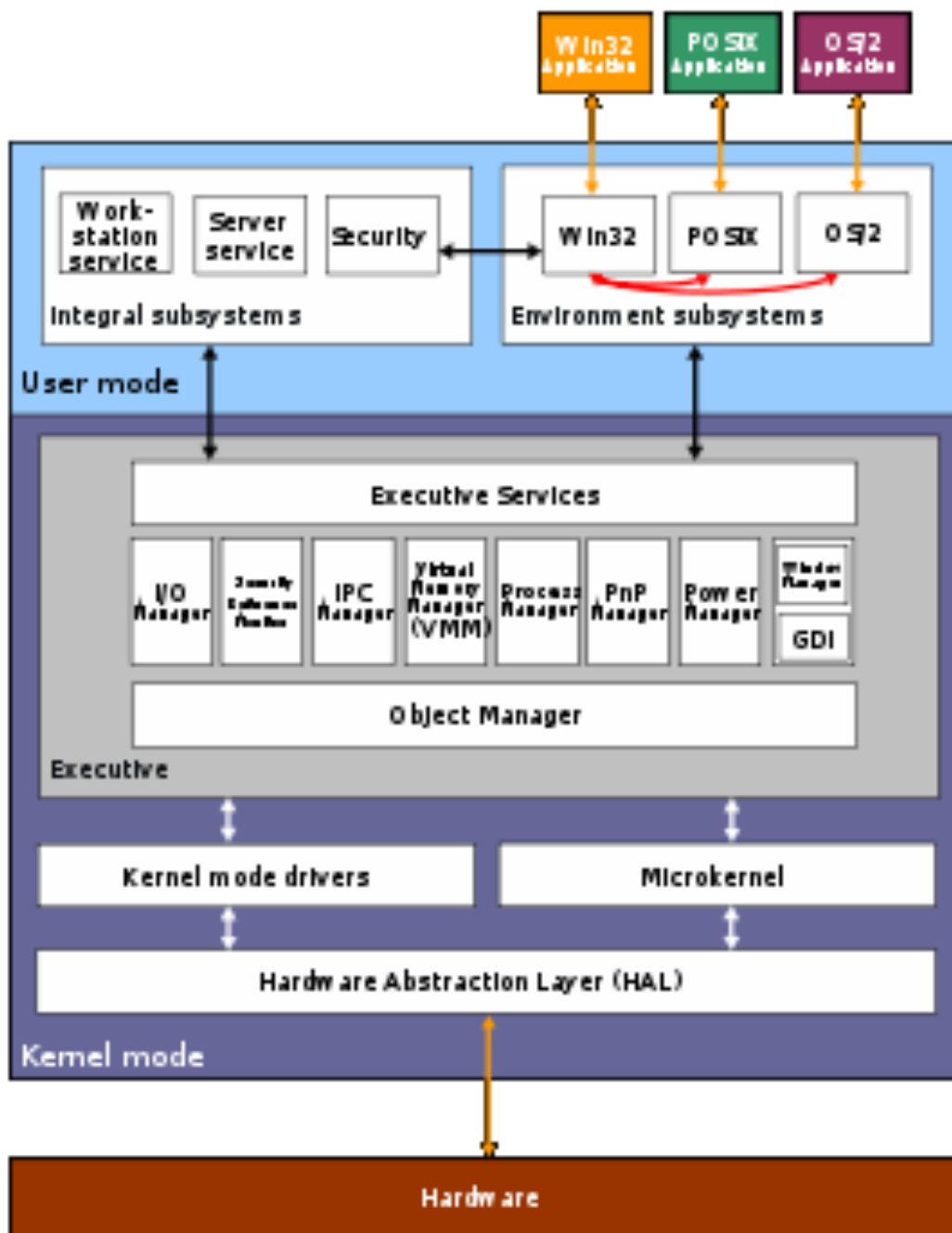
- Lepší ovladače a další funkcionalita: práce s fonty, video, síťování (bez TCP/IP), SMB

## WINDOWS 9X

- Integrace MS-DOSu + GUI
- Paměť a přístup k zařízení si řeší po svém (32 bitů)
- Preemptivní multitasking // pokud proces běží určitou danou časovou periodu, tak může být procesoru odebrán a může běžet jiný proces -> pokud aplikace odmítá spolupráci, nevytuhne celý OS
- Zpětná kompatibilita // několik kompromisů a díky ní několik chyb -> modrá obrazovka smrti
- Jedno-uživatelský OS

## WINDOWS NT

- Vychází z OS/2
- Kompatibilita s verzemi (W9x, atd.)
- Několik obecných principů:
  - Bezpečnost (certifikace pro armádu)
  - Spolehlivost (interní testování)
  - Kompatibilita s ostatními systémy (OS/2, POSIX)
  - Přenositelnost (HAL // Hardware Abstraction Layer, dříve možnost běžet např.: na PowerPc atd., dnes už jen: x86, amd64, Itania, Windows RT: ARM)
  - Rozšiřitelnost (s ohledem na vývoj HW)
  - Výkon
- Objektový přístup
- Implementovaný v C/C++
- Hybridní architektura
  - Oddělené procesy pro subsystémy (mikrokernel)
  - Spousta funkcionality v jaderném prostoru (monolitický kernel) // kvůli rychlosti



#### POPISY OBRÁZKU

- Windows Executive – poskytuje funkce jádra
- Object manager spravuje všechny objekty, které vznikly v systému -> díky jednomu správci se dají řešit oprávnění, vytváření nových objektů do jednoho modulu
- Kvůli výkonu je v jádru OS je i uživatelské rozhraní

# WINDOWS NT: ARCHITEKTURA

## WINDOWS EXECUTIVE

- Klíčová část OS: přes Ntdll.dll poskytuje funkce do uživatelského prostoru
- Obsahuje jednotlivá čísla jádra
  - Configuration manager (registry)
  - Process thread manager
  - I/O manager
  - Security reference manager
  - PnP manager
  - Cache manager
  - Memory manager
  - Object manager
  - A další (mj. Windows, GDI, USER)

#71

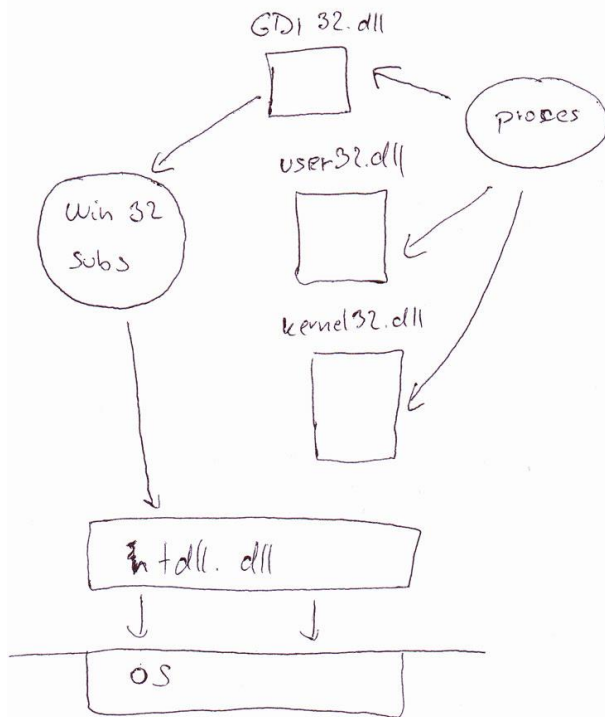
## SUBSYSTÉMY

- „pohled“ na funkce poskytované Windows executive (Tan. 794)
- Jeden subsystém Windows (csrss.exe), další pro POSIX, OS/2, ...
- Další systémové procesy (Session Manager – smss.exe, atd.)



# KOMUNIKACE V OS

Tu má být #71

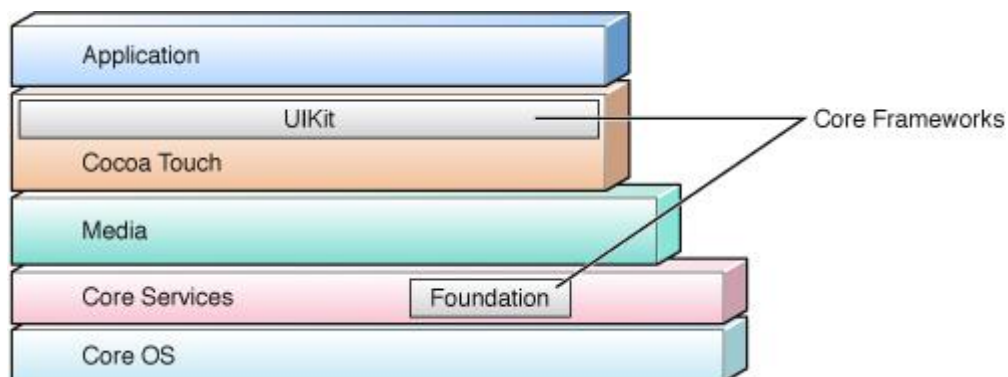


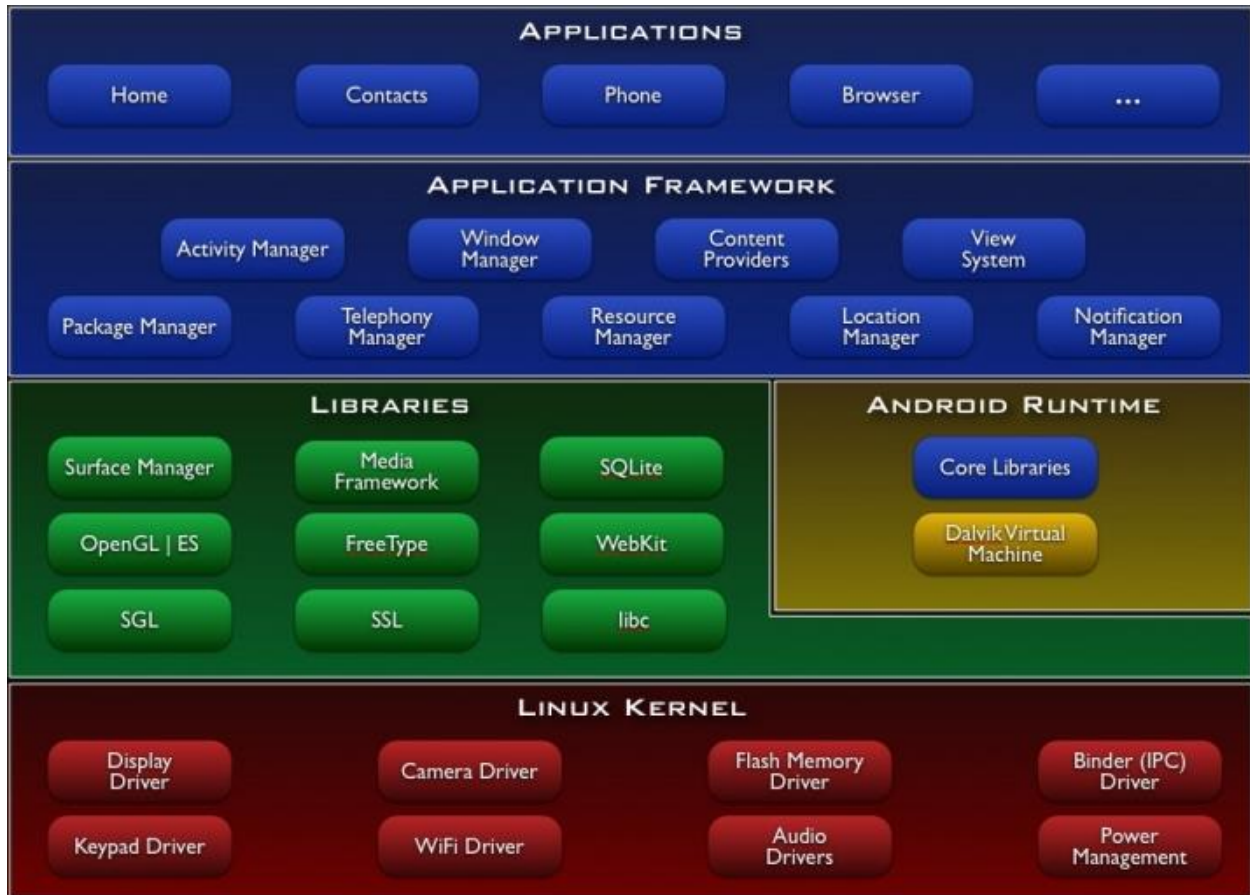
Proces -> konkrétní knihovna (např.: kernel32.dll) -> ntdll.dll -> OS

- Tento přístup dovoluje efektivně měnit OS, protože jsou požadavky abstrahovány
- pomocí knihoven

## ANDROID & IOS

- jejich jádra vychází z existujících systémů (Linux, resp. Darwin)
- jiný userland





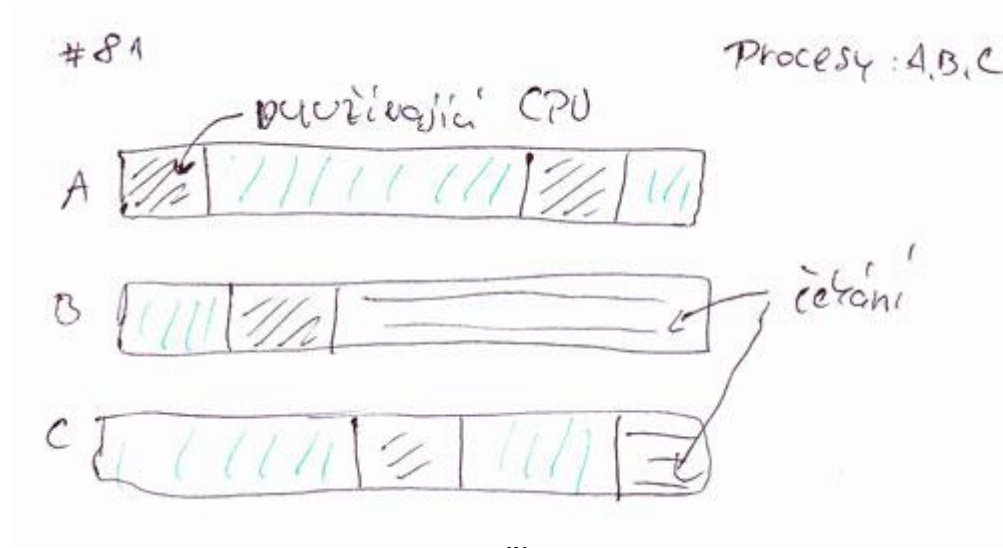
# PROCESY

## PROCESY

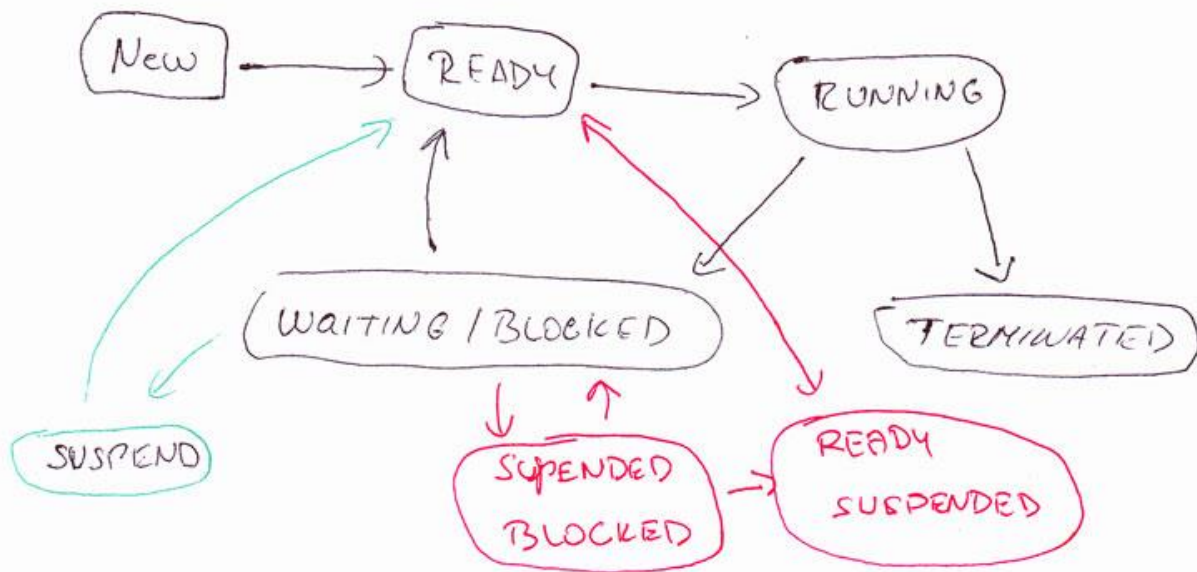
- **Neformálně:** proces = běžící program (vykonává činnost)
- Proces charakterizuje:
  - Kód programu
  - **Paměťový prostor** (!!!)
  - Data – statická a dynamická (halda – jsou tam uložené data, které se alokují při běhu procesu, typicky objekty alokované malloc nebo volané přes new)
  - Zásobník
  - Registry
- Operační systém: organizace sekvenčních procesů
- Oddělení jednotlivých úloh (abstrakce)
- Multiprogramování: (zdánlivě) souběžný běh více procesů
- Efektivní využití prostředků CPU (čekání na I/O)
- Komunikace mezi procesy, sdílení zdrojů → synchronizace

## OBECNÝ ŽIVOTNÍ CYKLUS PROCESU

- Nový (new) – proces byl vytvořen
- Připravený (ready) – proces čeká, až mu bude přidělen CPU
- Běžící (running) – CPU byl přidělen a právě provádí činnost
- Čekající (waiting/blocked) – proces čeká na vnější událost (např. na vyřízení I/O požadavku, synchronizační primitiva)
- Ukončený (terminated) – proces skončil svou činnost (dobrovolně × nedobrovolně)

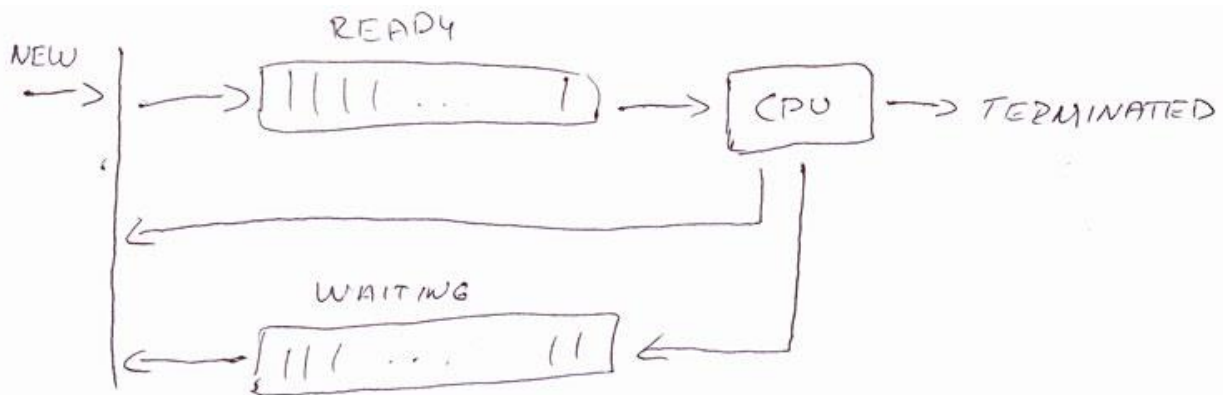


- Čas kdy proces nevyužívá CPU – pokud by běželi sekvenčně, tak by byl CPU nevyužitý v ty chvíle



// černý je „základní mode“, jsou i další modely, tedy některé systémy přidávají např.: SUSPEND (chybí šipka z READY do SUSPEND: pokud je moc procesů READY, odsunutí pryč)

- *New* – proces byl vytvořen
- *Ready* – vytvořený a čeká na přiřazení CPU
- *Running* – procesu byl přidělen procesor a právě provádí činnost
- *Waiting/blocked* – čeká, až mu bude přiřazení CPU nebo na např.: I/O, proces nepřechází znova do *running*, protože už může běžet jiný proces a docházelo by k chybě
- *Terminated* – proces běží ještě nějakou dobu po ukončení kvůli návratové hodnotě atd.



### ROZŠÍŘENÍ:

- *Supsended* – proces byl odsunut do sekundární paměti
- *Ready/suspended + block/supsended* – vylepšení přechozího mechanismu
- Fronty pro přechod mezi stavy
  - Front je ve skutečnosti několik, rozdělené dle událostí, na které se čeká

## INFORMACE O PROCESU

- Tabulka procesů → PCB: Proces Control Block – informace o procesu

### INFORMACE IDENTIFIKUJÍCÍ PROCES

- Identifikátor procesu, uživatele, rodičovského procesu

### STAVOVÉ INFORMACE

- Stav uživatelských registrů
- Stav řídicích registrů (IP, PSW)
- Vrchol zásobníku/ů

### ŘÍDÍCÍ INFORMACE

- Informace sloužící k plánování (stav procesu, priorita, odkazy na čekající události)
- Informace o přidělené paměti
- Informace o používaných I/O zařízeních, otevřených souborech atd.
- Oprávnění atd.

### PŘEPÍNÁNÍ PROCESŮ (CONTEXT SWITCH)

1. Uložení stavu CPU (kontextu, tj. registrů IP, SP) do PCB aktuálního procesu
2. Aktualizace PCB (změna stavu, atd.)
3. Zařazení procesu do příslušné fronty
4. Volba nového procesu
5. Aktualizace datových struktur pro nový proces (nastavení paměti, atd.)
6. Načtení kontextu z PCB nového procesu

- → jde řešit softwarově nebo s podporou HW (různá náročnost na čas CPU)
- Kooperativní (// proces sám nejlépe ví, kdy může být přepnut, nevýhodou je, že pokud je špatně napsaný, tak může využívat procesor na maximum a nejde ho vypnout)

vs. Preemptivní přepínání (// k možnosti rozhodnutí procesu k přepnutí je také přidělení časového kvanta, po jehož uplynutí může být proces přesunut do stavu READY )

## DŮVODY K PŘEPÍNÁNÍ

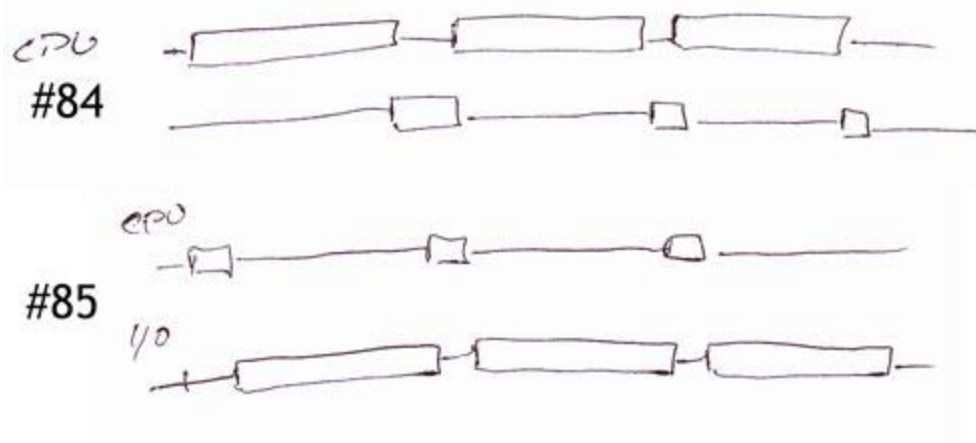
- Vypršení časového kvanta (nutná podpora HW)
- Přerušování I/O (aktuální proces může pokračovat nebo čekající proces může začít běžet)
- Výpad paměťové stránky, vyvolání výjimky (např.: dělení nulou)

## PLÁNOVÁNÍ PROCESŮ

- Potřeba efektivně plánovat procesorový čas
- Časové kvantum: maximální čas přidělený procesu
- Samotné přepnutí procesu má režii (uložení kontextu, vyprázdnění cache) → latence
- Jak zvolit velikost? → interaktivita nebo odvedená práce
- CPU-I/O burst cycle: pravidelné střídání požadavků na CPU a I/O
- → procesy náročné na CPU nebo I/O

## TYPY PLÁNOVÁNÍ

- **Dlouhodobé** – rozhoduje, zda bude přijat k běhu (změna stavu z NEW na READY)
- **Střednědobé** – načtení/odložení procesu do sekundární paměti
- **Krátkodobé** – rozhoduje dostupné procesy ke spuštění na CPU



- I/O – rozhoduje jednotlivé požadavky I/O

#84 schéma využití CPU a I/O pro intenzivně využívané CPU

#85 schéma pro více užívané I/O (např. práce se soubory atd)

## RŮZNÉ TYPY ÚLOH/SYSTÉMŮ

- Interaktivní
- Dávkové zpracování
- Pracují v reálném čase

## OBECNÉ POŽADAVKY NA PLÁNOVÁNÍ PROCESŮ

- **Spravedlnost** – každému procesu by v rozumné době měl být přidělen CPU
- **Vyváženost** – celý systém běží
- **Efektivita** – maximální využití CPU
- **Maximalizace odvedené práce** (throughput)
- **Minimalizace doby odezvy**
- **Minimalizace doby průchodu systémem** (turnaround) – od spuštění do konce procesu, aby byl co nejmenší časový úsek

## ALGORITMY PRO PLÁNOVÁNÍ PROCESŮ

### VHODNÉ PRO DÁVKOVÉ ZPRACOVÁNÍ

#### FIRST-COME-FIRST-SERVED

- První proces získává procesor
- Nové procesy čekají ve frontě
- Proces po skončení čekání, zařazen na konec fronty
- Nepreemptivní // nebere v úvahu délku běhu jednotlivých procesů
- Jednoduchý, neefektivní

#### SHORTEST JOB FIRST

- Vybere se takový proces, který poběží nejkratší dobu
- Nepreemptivní
- Zlepšuje celkovou průchodnost systémem
- Je potřeba znát (odhadnout) čas, který proces potřebuje
- U interaktivních systémů lze použít informace o využití CPU
- Použití.: např. u bankovního systému, když známe čas ke zpracování faktury, tak n faktur je n krát delší

### SHORTEST REMAINING TIME NEXT

- Pokud aktivní proces potřebuje k dokončení činnosti méně času než aktuální, je spuštěn
- Preemptivní
- Negativum na posledních dvou: musíme znát nebo aspoň odhadnout čas zpracování

### VHODNÉ PRO INTERAKTIVNÍ SYSTÉMY

#### ROUND ROBIN (CHODÍ PEŠEK OKOLO)

- Každý proces má pevně stanovené kvantum
- Velikost kvanta? (→ mírně větší, než typicky je potřeba)
- Připravené procesy jsou zařazeny ve frontě a postupně dostávají CPU
- Vhodné pro obecné použití (relativně spravedlivý)
- Protěžuje CPU pro náročné procesy (→ přidána další fronta pro procesy pro zpracování I/O, Sta 406) // ???
- **Nevýhoda:** přiděluje stejný čas jak systémovým, tak uživatelským procesům // skutečně?
- #86 – ilustrativní příklad:  
kvantum: 50 ms  
context switch: 10 ms  
proc. kv. 60 ms
- #87  
pokud potřebuje něco s I/O tak jde do fronty I/O a jinak jde přímo do READY, problém že procesy s I/O s upozadovány
- #88  
ošetřené upozadování procesů s I/O, jakmile jsou odbaveny procesy pracující s I/O, tak jdou s vyšší prioritou do fronty čekající na CPU

#### VÍCEÚROŇOVÁ FRONTA

- Každý proces má definovanou prioritu
- Statické x dynamické nastavení priority (např. vyšší priorita pro I/O)
- Systém eviduje pro každou frontu (čekající procesy)
- Riziko vyhladovění procesů s nízkou prioritou (nedostanou se ke zpracování)
  - Rozšíření: nastavení různých velikostí kvant pro jednotlivé priority (přesun mezi prioritami, nižší priorita → delší kvantum)
- #72  
úlohy brány podle priorit (P<sub>1</sub> je nejvyšší), vyžaduje jasně definovanou prioritu



(statická priorita nebo dynamická – řeší problém proti přetěžování od jednotlivých procesů

- Používán Windows NT, staršími Linuxovými jádry

## VIII. Přednáška

### SHORTEST PROCESS NEXT

- Vhodný pro interaktivní systémy (krátká doba činnosti + čekání)
- Používá se odhad, podle předchozí aktivity procesu

### GUARANTEED SCHEDULING

- Reálně přiděluje rovný čas CPU
- Máme-li  $n$  procesů, každý proces má získat  $1/n$  CPU
- Určí se poměr času kolik získal a kolik má získat ( $< 1$  – procesor měl méně času)
- Volí se proces s nejmenším poměrem // proces s nejméně časem je vybrán aby běžel
- Používají novější Linuxové jádra

### LOTTERY SCHEDULING

- Proces dostane přiděl „losů“
- Procesy voleny náhodně (proporcionální přidělování)
- Možnost vzájemné výměny losů mezi procesy

### FAIR-SHARE SCHEDULING

- Plánování podle skupin procesů (např. podle uživatelů) // např.: u linuxu rozdělení dle uživatelů

## ÚLOHY BĚŽÍCÍ V REÁLNÉM ČASE

- Nutné, aby systém zareagoval na požadavek v požadovaném intervalu
- Dva typy úloh:
  - **Hard real-time** – požadavek je potřeba vyřešit do určité přesně dané doby (intervalu) (např. měření čidel...)
  - **Soft real-time** – zpoždění vyřešené úlohy je tolerované
- **Periodické** (typicky měření) x **neperiodické** úkoly (reakce na vnější vlivy)
- Systém nemusí být schopen všem požadavkům vyhovět

### VARIANTY PLÁNOVÁNÍ

- **Statickou tabulkou** – obsluha periodických úkolů je dána předem
- **Statické definice priorit** – jednotlivým úlohám jsou nastaveny priority, aby byla splněna zadaná kritéria

- **Dynamická plánování** – proces je spuštěn, pokud je možné splnit jeho požadavky
- **Dynamické nejlepší snaha** – žádná omezení, pokud by nebylo možné splnit všechny požadavky v systému, proces je odstraněn

# VLÁKNA

## VLÁKNA

- **Proces** = sekvence vykonávaných instrukcí v jednom paměťovém prostoru
- Procesy jsou od sebe izolovány → nemusí být žádoucí
- Obecný přístup → proces = správa zdrojů (data, kód), vlákno = vykonávaný kód
- Možnost více vláken v rámci jednoho procesu
- Každé vlákno má své registry, zásobník, IP, stav (stejně jako proces), jinak jsou zdroje sdílené
- Vlákna sdílí stejné globální proměnné (data) → žádná ochrana (předpokládá se, že není třeba → potřeba synchronizace)
- Využití vláken:
  - Rozdělení běhu na popředí a na pozadí (CPU x I/O)
  - Asynchronní zpracování dat
  - Víceprocesorové stroje
  - Modulární architektura

## VZTAH PROCES-VLÁKNO

- 1:1 – 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 (clustery, hypotetické řešení, žádný OS asi nepoužívá)

## IMPLEMENTACE VLÁKEN

- Jako knihovna v uživatelském prostoru (první Linuxové systémy)
- Součást jádra operačního systému (Windows a další Linuxové systémy)
- Kombinované řešení
- Green threads // u jazyků vyšší úrovně, plánování je řešeno na úrovni běhového prostředí (běhové prostředí rozhoduje, kolik bude procesu přiděleného procesorového času)

## IMPLEMENTACE VLÁKEN

### V UŽIVATELSKÉM PROSTORU

- Proces se stará o správu a přepínání vláken
- Vlastní tabulka vláken

- Nejde použít preempce → kooperativní přepínání (rychlé, protože není potřeba využívat systémového volání – přepínat se do jaderného režimu)
- Možnost použít plánovací algoritmus dle potřeby
- Problém s plánováním v rámci systému
- Problém s blokujícími systémovými voláními
- Výhodou je, že můžeme zavést i v systému, kde nejsou vlákna podporována

## V JÁDŘE

- Jádro spravuje pro každého vlákno struktury podobně jako pro procesy (registry, stav, ...)
- Řeší problém s blokujícími voláními
- Vytvoření vlákna pomalejší (recyklace → pooly)
- Přepínání mezi vlákny jednoho procesu rychlejší (než mezi procesy, ale pomalejší než u vláken v uživatelském prostoru)
- Preemptivita

## HYBRIDNÍ

- Proces má M vláken v jádře, které má každé  $N_i$  vláken v uživatelském prostoru // OS Solaris

# IMPLEMENTACE PROCESŮ A VLÁKEN

## IMPLEMENTAČNÍ ASPEKTY: UNIX

- Procesy – původně základní entita vykonávající činnost
- Procesy tvoří hierarchii, každý proces identifikován pomocí PID
- Systém při inicializaci spustí první proces (`init`)
- Nový proces (potomek) vytvořen voláním `fork()` – vytvoří kopii aktuálního procesu
- Používá se společně s voláním `exec` – nahraje do paměti kód ze souboru a začne jej provádět
- V rámci vztahu rodič-potomek jsou sdílené některé zdroje (např. popisovače souborů)
- **Sirotci** – pokud je rodičovský proces skončí dřív, přešel do `init`
- **Zombie** – proces již skončil, ale existuje v systému
- **Priorita** – nice (40 hodnot)
- Vlákna přidána do Unixů později (dříve i ve formě knihoven)
- → není zcela konzistentní s původní koncepcí
- Jak se má zachovat `fork()`?
- Oddělené mechanismy pro synchronizaci vláken a procesů
- Vlákno – běžná procedura s jedním argumentem vracující jednu hodnotu

```
void * foo (void * arg) {  
    /* kod vlakna */  
    return (void *) 42;  
}
```

```
pthread_t thr;  
void * result;
```

```
pthread_create (&thr, NULL, foo, (void *) 123);  
/* kod provadeny hlavnim vlaknem */  
pthread_join (thr, &result);
```

## PLÁNOVÁNÍ PROCESŮ V LINUXU

- Interně jádro pracuje s vlákny i procesy stejně
- Proces/vlákno → task (účastní se plánování)
- Systémové volání `clone` – zobecněný `fork` (umožňuje definovat, které struktury se mají sdílet):
  - Paměťový prostor
  - Otevřené soubory
  - I/O
  - id rodiče
  - ...
- **Stavy úloh:** běžící, zastavený, pozastavený (čeká se na nějakou podmínku), zombie...
- Stavy vláken: běžící, připravené k běhu, uspané-přerušitelné – čeká na nějakou podmínku, uspané-nepřerušitelné (čeká na nějakou kritickou HW operaci), zastavené, pozastavené, skončené
- Různé strategie plánování procesů – dávkové úlohy, FIFO nebo RR (pro práci v reálném čase), obecná strategie
- Idle vlákno

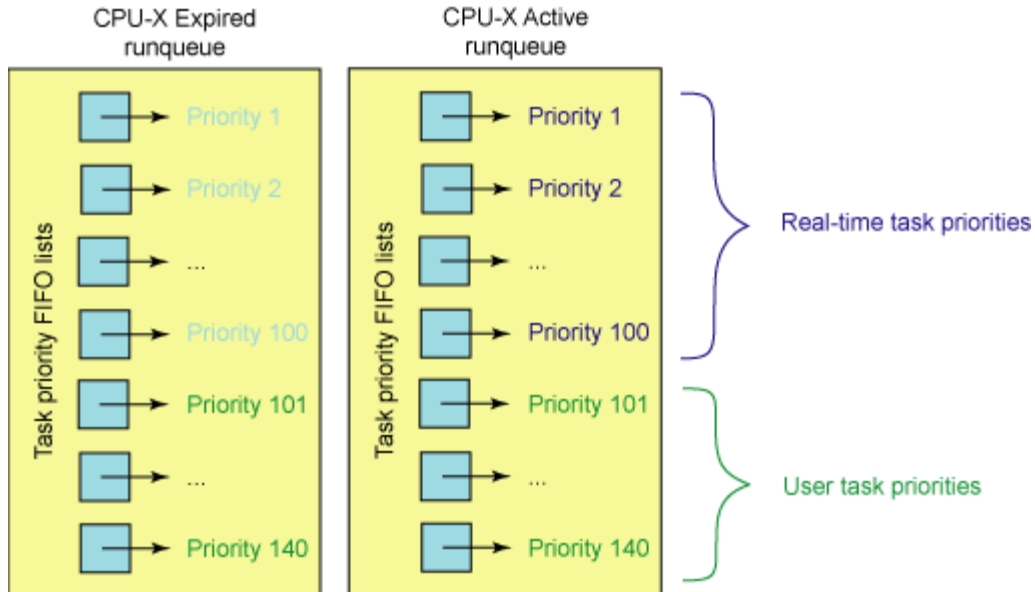
## PLÁNOVÁNÍ PROCESŮ V LINUXU

### DŘÍVE

- Úplně nejdříve round-robin
- 2.4: plánování rozděleno do epoch; každý proces může použít jedno kvantum v rámci epochy; pokud jej nespotřeboval (polovina přesunuta do další epochy) → prohledává všechny procesy →  $O(N)$

## O(1) PLÁNOVAČ

- 140 front priorit – prvních 100 RT úlohy, zbytek pro běžné procesy
- Čekající procesy v active runqueue, po vyčerpání přiděleného procesorového času active queue ...



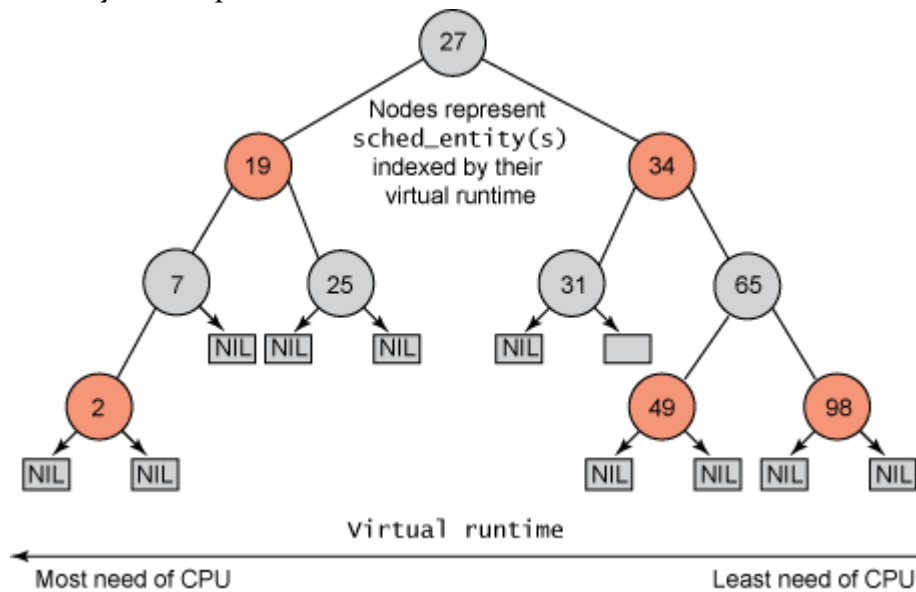
má konstantní složitost – udržujeme si první bit jako příznak, zdali je v nějaké frontě task, který čeká na zpracování

- Heuristiky určují, jestli daný proces zatěžuje CPU, I/O nebo je interaktivní → dočasná úprava priorit (I/O může dostat až pět úrovní navíc)
- SMP

## COMPLETELY FAIR SCHELUDE

- Varianta Guaranteed scheduleru
- Procesy organizovány v RB-stromu (podle toho, kolik dostaly času)
- Logaritmická časová složitost

- Priority řešení pomocí koeficientů



je samovyvažující se, list v nejlevějším listu (2) je kandidát pro běžení, vpravo dostanou méně (např.: 98)

poté co dostal nějaký čas, tak se vrátí do stromu a strom se vyváží  
priorita řešena tak, že se vynásobí určitým koeficientem a dostane se více doleva, procesy vlevo dostaly nejméně, vpravo nejvíce procesorového času

## SMP

- Samotné plánování pro každý procesor
- Nový proces umístěn náhodně
- Každých 200 ms se zkontroluje vytížení a provede se vyvážení výkonu
- Rozdělení procesorů do skupin
- Možnost nastavit afinitu – určit procesor/skupinu procesů, kde má úloha běžet

## IX. Přednáška

## IMPLEMENTAČNÍ ASPEKTY: WINDOWS

- **Vlákno** – základní jednotka vykonávající činnost (účastní se plánování)
- **Proces** - obsahuje jedno a více vláken (společné zdroje a paměť)
- **Job** – slučuje několik procesů dohromady (společná správa, nastavení kvót, atd.)
- **Fiber** – „odlehčená vlákna“ implementované v kontextu vláknu (kooperativní multitasking)



## VZNIK PROCESU

- **Není vyžadován vztah rodič-potomek**
- **CreateProcess** – funkce vytvářející nový proces (10+ argumentů); příprava ve spolupráci s daným subsystém, více verzí jednoho programu v jednom souboru
- **CreateThread** – funkce vytvářející nové vlákno v principu podobná `pthread_create`
- **Plánování se účastní vlákna**

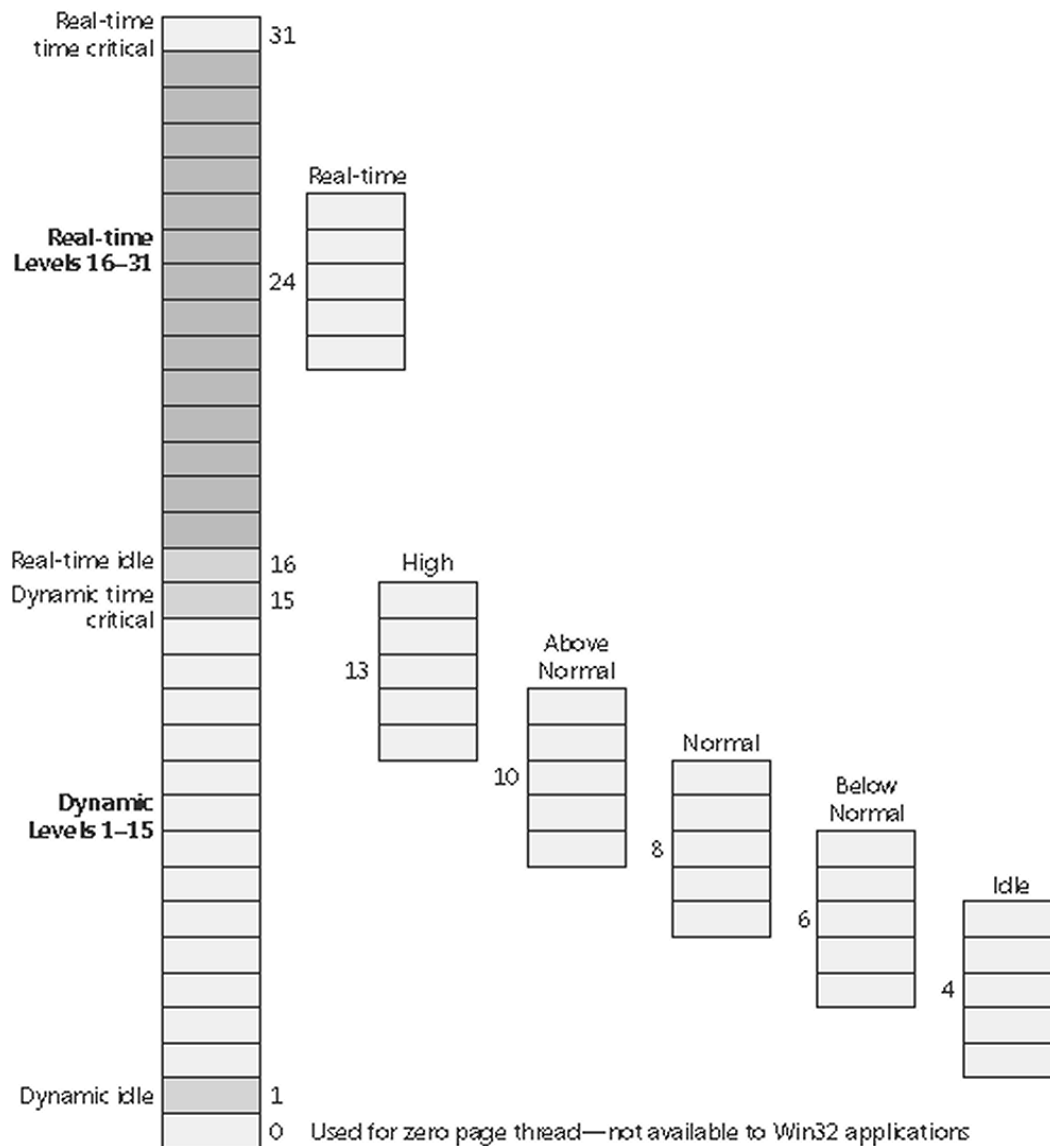
## STAVY VLÁKEN

- **Initialized** – během inicializace vlákna
- **Ready** – čekající na běh (z těchto vláken vybírá sheduler další pro běh)
- **Standby** – vlákno připraveno k běhu na konkrétním CPU
  - Přechod do running
  - Přechod do ready (pokud vlákno s vyšší prioritou přešlo do režimu standby)
- **Running** – vlákno běží; možné přechody
  - Vlákno s vyšší prioritou získalo CPU (návrat do standby nebo ready)
  - Po vypršení kvanta → ready
  - Čekaná na událost → waiting
  - Ukončení vlákna
- **Waiting** – čeká na nějakou událost; přechod o ready, standby či running (v případě s úloh s vysokou prioritou)
- **Transition** – zásobník je mimo fyzickou paměť (přechod do ready)
- **Terminated** – vlákno je ukončeno (lze změnit na initialized)
- Pokud se objeví vlákno s vyšší prioritou ve stavu ready, než má vlákno ready, dostane CPU vlákno s vyšší prioritou a aktuálně běžící vlákno je přesunuto na začátek příslušné fronty

## PRIORITY

- **Priority procesu** – hodnotu 0-31 přiřazena procesu (32 úroňová fronta)
- **Třídy priority** – vlastnost procesu udávající základní prioritu vláken
  - Real-time (24)
  - High (13)
  - Above normal (10)
  - Normal(8)
  - Below normal (6)
- **Priority vláken** – Time critical, highest, above normal, normal, below normal, lowest, idle
- **Priorita vlákna** je dána relativně k prioritě procesu + další úpravy

- Kategorie priorit
  - Idle(o) – zero page thread
  - Dynamické úrovně (1-15) – běžné procesy



## VELIKOST KVANTA

- Závisí na verzi OS
  - Workstation – 6 jednotek (2 tiky přerušovaného časovače)
  - Server – 36 jednotek (12 tiků)
- Velikost jde měnit (v nastavení nebo dočasně)
- Při čekání, přepnutí, atd. se velikost kvanta mírně snižuje
- Proces na popředí – všechna jeho vlákna mají 3x větší kvantum

## DOČASNÉ ZVÝŠENÍ PRIORITY (PRIORITY BOOST)

- U procesů s dynamickou úrovní
- Po dokončení I/O zvýšena priorita o:
  - +1 – disk, CD-ROM, grafická karta
  - +2 – síťová karta, ser. Port
  - +6 – klávesnice, myš
  - +8 – zvuková karta
  - Po uplynutí kvanta se priorita snižuje o jedna až na základní hodnotu

Př. Zvuková karta čeká na něco a poté je potřeba ho rychle např.: transformovat a proto dostane vyšší prioritu, aby nemělo velkou latency

- Po čekání na událost nebo synchronizaci s jiným vláknem
  - Na dobu jednoho kvanta zvýšena priorita o 1
  - Při synchronizaci – vlákno může získat prioritu o jedna vyšší než mělo vlákno na které se čekalo
- Vlákno na popředí po dokončení čekající operace → priorita + 2
- Aktivita v GUI → priorita + 4
- Vlákno už dlouho neběželo (řádově sekundy) → priorita 15 + 2x delší kvantum

## SMP

- Proces i vlákno má nastavenou masku affinity – seznam povolených CPU, kde může běžet
- Každé vlákno má ještě dvě hodnoty – ideální procesor a minulý procesor
- Procesor pro vlákno je vybírán následovně:
  1. Nečinný CPU
  2. Ideální CPU
  3. Minulý CPU
  4. Aktuální CPU
- Každý proces má svůj vlastní plánovač → lepší škálování

## MAC OS X

- Koncepčně vychází z Unixu → pojetí procesů, vláken
- K plánování se používá víceúrovňová fronta (podobné jako v Linuxu)

## GRAND CENTRAL DISPATCH

- Rozšíření OS a programovacích jazyků
- Umožňuje je snadno provádět bloky kódu ve vláknech

- Existuje několik front, kam se jednotlivé úlohy řadí (každá má thread-pool)
- Globální fonty se umožňují přizpůsobit konkrétnímu HW
- Soukromé fonty (úkoly zpracovávány sekvenčně), ale v jiném vlákně
- Rozšíření C → blok kódu:
 

```

x = ^{ printf ("FOO!\n"); }
y = ^(int a) {return * a 10; }
x(); y(20);

```
- příklad použití:
 

```

dispatch_async(dispatch_get_main_queue(), ^{...});

```

# SYNCHRONIZACE PROCESŮ

## SYNCHRONIZACE VLÁKEN A PROCESŮ

- procesy a vlákna přistupují ke sdíleným zdrojům (paměť, souborový systém)
- příklad: současné zvýšení hodnoty proměnné o 1
  - 1) A: načte hodnotu proměnné X z paměti do registru ( $X = 1$ )
  - 2) A: zvýšení hodnotu v registru o jedna
  - 3) B: načte hodnotu proměnné X z paměti do registru ( $X = 1$ )
  - 4) A: uloží hodnotu zpět do paměti ( $X = 2$ )
  - 5) B: zvýší hodnotu v registru o jedna
  - 6) B: uloží hodnotu zpět do paměti ( $X = 2$ )díky preemptivnímu něčemu nevíme, kdy přesně se proces ukončí a přepne se
- Chyba souběhu (race-condition) → náročné na debuggování
- Nejznámější chyba: Therac-25 // systém na ozařování rentgenu pacientů, 2 režimy: slabé na určité místo nebo silné na široký prostor, dalo se pomocí 3 kláves dostat do stavu silného ozáření na určité místo
- Řešení → atomické operace a kritické sekce

## ATOMICKÝ PŘÍSTUP DO PAMĚTI

- Obecné přístupy do paměti nemusí být atomické (záležitosti CPU, překladače)
- → více vláknové aplikace (přerušení); víceprocesorové počítače (cache)
- Lze vynutit určité chování → klíčové slovo `volatile` – často záleží na překladači
- Memory barriers umožňují vynutit si synchronizaci (záležitost CPU)

## ATOMICKÉ OPERACE

- **Test-and-Set (TAS):** nastav proměnnou a vrať její původní hodnotu
- **Swat:** atomicky prohodí dvě hodnoty
- **Compare-and-Swap (CAS):** ověří, jestli se daná hodnota rovná požadované a pokud ano, přiřadí ji novou (CMPXCHG)
- **Fetch-and-Add:** vrátí hodnotu místa v paměti a zvýší jeho hodnotu o jedna (XADD)
- **Load-link/Store-Conditional (LL/SS):** načte hodnotu a pokud během čtení nebyla změněna, uloží do ní novou hodnotu

## KRITICKÁ SEKCE (CRITICAL SECTION)

- Obecně třeba zajistit, aby se sdílenými zdroji pracoval jen jeden proces
- → vzájemné vyloučení (mutual exclusion)
- → problém kritické sekce
- Část kódu, kdy program pracuje se sdílenými zdroji (např. paměť)
- Pokud jeden proces je v kritické sekci, další proces nesmí vstoupit do své kritické sekce
- Každý proces před vstupem žádá o povolení vstoupit do kritické sekce
- Ukázka kódu:

```
do {  
    // vstupní protokol KS  
    ... práce se sdílenými daty ...  
    // výstupní protokol KS  
    ... zbylý kód  
} while (1);
```
- #91

## POŽADAVKY NA KRITICKOU SEKCI

- Vzájemné vyloučení – maximálně jeden proces je v daný okamžik v KS
- Absence zbytečného čekání – není-li žádný proces v kritické sekci a proces do ní chce vstoupit, není mu bráněno
- Zaručený vstup – proces snažící se vstoupit do KS, do ní v konečném čase vstoupí

## V KONTEXTU OS

- Potřeba synchronizovat činnost uživatelských procesů/vláken
- V kontextu jádra řada souběžných činností
  - Npreemptivní jádro OS (Linux < 2.6, Windows 2000, XP)
  - Preemptivní jádro (Linux ≥ 2.6, Solaris, IRIX)

## ŘEŠENÍ

- Zablokování přerušení (použitelné v rámci jádra OS); více CPU → neefektivní

## AKTIVNÍ ČEKÁNÍ

- **Spinlocks** – testujeme pořád dokola jednu proměnou

### ŘEŠENÍ Č. 1

V C:

```
while (lock);    // čekej
lock = 1
// kritická sekce
lock = 0;
```

V C → ASM:

```
mov eax, [lock]
test eax, eax
jnz začátek
mov [lock] 1
```

- // pokud je lock 0, tak může vstoupit do kritické sekce, nastaví na 1 a tím je zamčeno použitelné v OS se zablokováním přerušení → nedojde k přepnutí
- Vstup do kritické sekce a její zničení není provedeno atomicky
- **Race-condition** – chyba souběhu, 2 procesy se dostanou do kritické sekce

### ŘEŠENÍ Č. 2

Uvažujeme následující atomickou operaci

```
bool test_and_set(bool * target) {
    bool rv = *target;
    *target = true;
    return rv;
}
```

A kód

```
while (test_and_set(&lock) == true);
// kritická sekce
lock = false
```

### ŘEŠENÍ Č. 3

Uvažujeme následující atomickou operaci, která prohodí dvě hodnoty

```
void swap(bool *a, bool *b) {
    bool tmp = *a;
    *a = *b;
    *b = tmp;
}
```

A kód

```
key = true;
while(key == true)
    swap(&lock, &key);
// kritická sekce
lock = false;
```

# PETRSONŮV ALGORITMUS

- Řešení vzájemného vyloučení bez použití atomických operací

## PROCES A

```
lockA = true;
turn = B;
while (lockB && (turn == B)) { }
...
lockA = false;
```

## PROCES B

```
lockB = true;
turn = A;
while (lockA && (turn == A)) { }
...
lockB = false;
```

- Vyžaduje férové plánování
- Počítá s tím, že jednotlivé operace jsou provedeny atomicky

## X. Přednáška

# SEMAFOR

- Chráněná proměnná obsahující počítadlo s nezápornými celými čísly
- Operace P (proberem – zkusit): pokud je hodnota čísla nenulová sníží hodnotu o jedna, jinak čeká, až bude hodnota zvýšena (operace někdy označena i jako wait)  

```
void P (Semaphore s) {
    while (s <= 0) { }
    s--;
```
- Operace V (verhogen – zvýšit): zvýší hodnotu o jedna (operace označování jako signal, post)  

```
void V(Semaphore s) {
    s++;
```
- Operace P a V se provádí atomicky // operace jsou na začátku a na konci zamykané pomocí spinlocku (protože je kód relativně krátký, tak se nečeká moc dlouho na čekání)
- Binární semafor – může nabývat hodnot 0, 1 (mutex, implementace kritické sekce)
- Obecný semafor – slouží k řízení přístupu ke zdrojům, kterých je končené množství



- Implementace s pomocí aktivního čekání nebo OS (→ pasivní čekání)

```
struct sem {
    int value;
    struct process * list;
};

void P (struct sem * s) {
    s -> value--;
    if (s -> value < 0) {
        // přidej proces s->list;
        block(); // uspi aktuální proces
    }
}

void V(struct sem * s) {
    s-> value++;
    if (s -> value <= 0) {
        // odeber proces P z s s-> list
        wakeup(P);
    }
}
```

- Operace musí být provedeny atomicky
- Seznam by mělo být jako FIFO
- Spolupráce wakeup s plánovačem
- Všimněte si záporné hodnoty s→value → počet čekajících procesů

## DALŠÍ SYNCHRONIZAČNÍ NÁSTROJE

### BARIÉRY

- Synchronizačních metod vyžadujících, aby se proces zastavil v daném bodě, dokud všechny procesy nedosáhnou daného bodu

### READ-WRITE ZÁMKY

- Vhodné pro situace, které čtou i zapisují do sdíleného prostředku
- Čtecí a zapisovací režim zámku
- Vhodný pokud jde rozlišit čtenáře a páse (písařů je víc)

### PODMÍNĚNÁ PROMĚNNÁ

- Čekání na změnu proměnné – neefektivní čekání
- Operace **wait**, **signal**

- Kombinace se zamykáním

## MONITOR

- Modul nebo projekt
- V jeden okamžik může kteroukoliv metodu používat pouze jeden proces/vlákn
- Nutná podpora programovacího jazyka
- Java (synchronized), .NET (lock)
- Rozšíření o podporu čekání (Wait, Pulse, PulseAll) → možnost odemknout zámek společně s čekání

### JAVA EXAMPLE :

```
class FOO {
    synchronized bar();
    synchronized buz();
}

Foo foo
V1 - foo.bar();
V2 - foo.buz();      // vlákno 2 musí počkat, než doběhne vlákno 1
```

### .NET EXAMPLE

```
lock(foo) {
    ... uzamčený kód ...
}
```

## SYNCHRONIZAČNÍ PRIMITIVUM VE WINDOWS

- Obecný mechanismus – synchronizační objekty se nacházejí ve dvou stavech (signalizovaný vs. nesignalizovaný)
- Signalizovaný objekt je dostupný (mutex, semaphore, event., thread, etc.)
- (univerzální) čekací funkce (WaitForSingleObject, WaitForMultipleObject) – čeká, dokud se objekt nebo objekty nedostanou do signalizovaného stavu
- Čekací funkce slouží také k manipulaci s mutexy, semaforey,...
- CreateMutex, CreateSemaphore. ... (možnost vytvořit pojmenované objekty)
- ReleaseMutex, ReleaseSemaphore, SetEvent
- SignalObjectAndWait – kombinuje přechodí operace do jedné atomické

### DALŠÍ SYNCHRONIZAČNÍ METODY

- Interlocked API (atomické operace), spinlocks (jádro)
- Kritická sekce (EnterCriticalSection, LeaveCriticalSection)

# SYNCHRONIZAČNÍ PRIMITIVUM V UNIXECH

## SIGNÁLY

- Mechanismus podobný přerušení (asynchronní volání)
- Základní forma komunikace
- Proces může definovat vlastní handlersy těmto signálům
- Některé speciální určení (případně nastavené implicitní handlersy)
  - SIGINT – ukončení procesu (Ctl + C)
  - SIGQUIT – ukončení procesu (Ctl + /) + Core dump
  - SIGSTOP – pozastavení procesu (Ctl + Z)
  - SIGCONT – pokračování pozastaveného procesu
  - SIGHLD – změna stavu potomka
  - SIGKILL – nezablokovaný signál ukončující proces
  - SIGFPE, SIGBUS – oznamování systémových chyb
  - SIGUSR1, SIGUSR2 – uživatelské signály
  - SIGALARM – alarm
  - SIGPIPE – přerušená roura
- Race-conditions
- Nelze získat složitější zprávy

## SYNCHRONIZACE PROCESŮ

- Systém v IPC
- Sdílená paměť, semaforey, zasílání zpráv
- Práce se semaforey (skupina semaforů) – semget, semctl, semop (mj. společné rozhraní pro operace typu P a V)...
- Sdílené všecmi procesy → správa oprávnění

## SYNCHRONIZACE VLÁKEN

- Libthread – mutexy, semaforey, rw-zámky, bariéry  
(pthread\_mutex\_lock, pthread\_mutex\_trylock...)
- Futexy

## AMOTICKÉ OPERACE

- Chybí obecné rozhraní v uživatelském prostoru
- glib, lib\_atomic\_ops

- Jádro používá vlastní sadu operací (`atomic_read`, `atomic_set`,...)

## DEADLOCK

- Uváznutí – systém se dostal do stavu, kdy nemůže dál pokračovat
- *U množiny procesů došlo k uváznutí (deadlocku), pokud každý proces z této množiny čeká na událost, kterou pouze proces z této množiny může vyvolat.*

## UŽÍVÁNÍ PROSTŘEDKŮ

- **Request** – požadavek na prostředek, není-li k dispozici, proces čeká
- **Use** – vlákno s prostředkem pracuje
- **Release** – uvolnění prostředku pro další použití

## PODMÍNKY VZNIKU

- **Mutual exclusion** – alespoň jeden prostředek je výlučně užíván jedním procesem
- **Hold & wait** – proces vlastní alespoň jeden prostředek a čeká na další
- **No preemption** – prostředek nelze násilně odebrat
- **Circular wait** – cyklické čekání (proces A vlastní prostředek 1, chce prostředek 2, který drží B a současně žádá o 1)

## ŘEŠENÍ DEADLOCKU

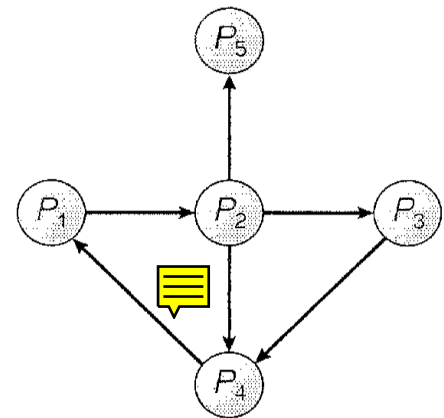
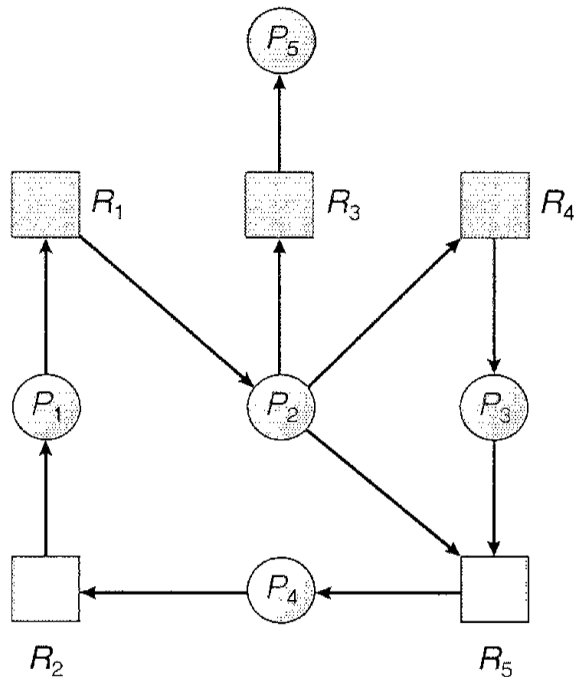
### IGNORACE

- „neřešení“, v praxi často používané  
tzv. pštrosí algoritmus – strčí se hlava do písku a dělá se, že žádný problém není

### DETEKCE

- Pokud vznikne deadlock, je detekován a některý proces odstraněn
- K detekci se používá alokační graf prostředků a graf čekání
- **Alokační graf:**
  - Orientovaný graf
  - Dva typy uzlů – prostředek, proces
  - Hrana proces-prostředek – proces čeká na prostředek
  - Hrana prostředek-proces – prostředek je vlastněn procesem
- **Graf čekání** vznikne vynecháním uzlů prostředků a přidáním hran  
 $P_n \rightarrow P_m$  pokud existovaly hrany  $P_n \rightarrow R$  a  $R \rightarrow P_m$ , kde  $P_n$  a  $P_m$  jsou procesy a  $R$  je prostředek

- Ukázka alokačního grafu



- Deadlock vznikne, pokud je v grafu čekání na cyklus
- → odebrání prostředků, odstranění procesu (jak vybrat oběť?), opakované zpracování (rollback)
- Kdy má smysl provádět detekci?

## ŘEŠENÍ DEADLOCKU

### ZAMEZENÍ VZNIKU (PREVENTENCE)

- Snažíme se zajistit, že některá z podmínek není splněna
- Zamezení výlučnému vlastnění prostředků (často nelze z povahy zařízení)
- Zamezení držení a čekání
  - Proces zažádá o všechny prostředky hned na začátku
  - Problém s odhadem
  - Plýtvání a hladovění
  - Množství prostředků nemusí být známe předem
  - Jde použít i v průběhu procesu (ale proces se musí vzdát všech prostředků)
- Zavedení možnosti odejmout prostředek – vhodné tam, kde nelze odejmout prostředky tak, aby nešlo poznat, že byly odebrány
- Zamezení cyklickému čekání – zavedení globálního číslování prostředků a možnost žádat prostředky jen v daném pořadí

## VYHÝBÁNÍ SE UVÁZNUTÍ

- Procesy žádají prostředky libovolně
- Systém se snaží vyhovět těm požadavkům, které nemohou vést k uváznutí
- Je potřeba znát předem, kolik prostředků bude vyžádáno
- Tomu je přizpůsobeno plánování procesů
- Bezpečný stav – existuje pořadí procesů, ve kterém jejich požadavky budou vyřízeny bez vzniku deadlocku
- Systém, který není v bezpečném stavu, nemusí být v deadlocku
- Systém odmítne přidělení prostředků, pokud by to znamenalo přechod do bezpečného stavu (proces musí čekat)

## ALGORITMUS NA BÁZI ALOKAČNÍHO GRAFU

- Vhodný, pokud existuje jen jedna instance každého prostředku
- Do alokačního grafu přidáme hrany (proces-prostředek) označující potenciální žádosti procesu a prostředky
- Žádosti a prostředek se vyhoví pouze tehdy, pokud konverze hrany na hranu typu (prostředek-je vlastněn-procesem) nepovede ke vzniku cyklu

## BANKÉŘŮV ALGORITMUS

- Vhodný tam, kde je větší počet prostředků daného typu
- Na začátku každý proces oznámí, kolik prostředků jakého typu bude maximálně potřebovat
- Při žádosti o prostředky systém ověří, jestli se nedostane do nebezpečného stavu
- Pokud nelze vyhovět, je proces pozdržen
- Porovnávají se volné prostředky, s aktuálně přidělenými a maximálními
- Uvažujeme  $m$  prostředků a  $n$  procesů
- Matice  $m \times n$ 
  - **Max** – počet prostředků, které bude každý proces žádat
  - **Assigned** – počet přiřazených prostředků jednotlivým procesům
  - **Needed** – počet prostředků, které bude proces ještě potřebovat (evidentně  $needed = max - assigned$ )
- Vektory velikosti  $m$ 
  - **E** – počet existujících prostředků
  - **P** – počet aktuálně držných prostředků
  - **A** – počet zdrojů

Assigned

	K	L	M	N
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Needed

	K	L	M	N
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

E = <6, 3, 4, 2>

P = <5, 3, 2, 2>

A = <1, 0, 2, 0>

- Podmínku splňuje proces D → odebrán a A <- <2, 1, 2, 1>
- Podmínku splňuje proces A → odebrán a A <- <5, 1, 3, 2>
- Podmínku splňuje proces B → odebrán a A <- <5; 2; 3; 2>
- Podmínku splňuje proces C → odebrán a A <- <6, 3, 4, 2>
- Podmínku splňuje proces E → odebrán a A <- <6, 3, 4, 2>

#### ALGORITMUS

1. Najdi řádek i v needed takový, že needed[i] <= A, pokud takový není, systém není v bezpečném stavu
2. Předpokládej, že proces skončil a uvolnil své zdroje  
A <- A + assigned[i] a odstraň řádný i ze všech matic
3. Opakuj body 1 a 2 dokud nejsou odstraněny všechny procesy nebo není jasné, že systém není v bezpečném stavu

## XI. Přednáška

### TRANSAKČNÍ PAMĚŤ

// DB jsou ACID – atomic consistent isolated (durability – změny v transakci jsou trvalé)

- Atomické operace umožňují implementovat bezzámkové (lock-free) datové struktury -> netriviální
- Souběžný přístup k paměti -> vývoj vícevláknových aplikací je komplikovaný
- Vlákna vs. Procesy
- Problémy se synchronizací (zámky nejsou reentrantní)
- Ale: databázové systémy zvládají pracovat s daty bez vážnějších problémů
- Rozdělení programu na části, které jsou provedeny „atomicky“ (ACI)

```

void transfer (Account a1, Account a2, int amount){
    atomic {
        a1.balance += amount;
        a2.balance -= amount;
    }
}

```

- Změny se neprovádí přímo, ale ukládají se do logu, v případě „commitu“ se ověří, jestli došlo ke kolizi
- Zjednodušení vývoje – na úkor režie
- Omezení: transakci musí být možné provést vícekrát
- ```
void transfer (Account a1, Account a2, int amount){
    atomic {
        a1.balance += amount;
        a2.balance -= amount;
        lunchTheMissiles();
    }
}
```
- Možnost explicitně ovládat transakce retry, orElse
- Problém: jak se vypořádat s hodnotami mimo transakční paměť (např.: existující knihovny)
- Různé varianty a implementace
- Podpora HW pro transakční paměť -> omezená až žádná (Intel Haswell, IBM BlueGene/Q, ROCK // projekt zrušen po odkoupení Oraclem)
- Podpora na straně SW (Software Transactional Memory – STM)
- Podpora jako knihovny/rozšíření Javy/C#/C++
- Výzkum Haskel [Microsoft], Fortress/DSTM [SUN]
- Jazyky zaměřené na paralelní zpracování, např.: Clojure



# DALŠÍ PROSTŘEDKY MEZIPROCESNÍ KOMUNIKACE

## MEZIPROCESNÍ KOMUNIKACE

- IPC – Inter-process communication
- Procesy oddělené, potřeba kooperace
  - Sdílení informací
  - Zrychlení výpočtu (rozdělení úlohy na podúlohy)
  - Souběžná činnosti
  - Modularita
  - Oddělování privilegií // zvýšení bezpečnosti
- Základní kategorie
  - Sdílená paměť
  - Zasílání zpráv
  - Synchronizace
  - Vzdálené volání procedur

// nahrávání od toť

- Rozlišujeme různé charakteristiky
  - Zda komunikují dva příbuzné (mající společného rodiče) nebo zcela cizí procesy
  - Zda komunikující proces může jen číst či jen zapisovat data
  - Počet procesů zapojených do komunikace
  - Zda jsou komunikující procesy synchronizovány, např. čtecí proces čte, až je co číst
  - Zda jsou v rámci jednoho systému

## SDÍLENÁ PAMĚŤ

- Procesy sdílejí kousek paměti → nutná spolupráce se správou paměti
- Čtení i zápis, náhodný přístup
- Deklarace, že paměť je sdílená + namapování do adresního prostoru
- Velikost úseku paměti i adresa zaokrouhleny na násobky stránek paměť (typicky 4 kB)
- Paměť může být namapována na různé adresy

#111

// P1 a P2 má namapovanou stránku (logická paměť) na jeden rámec (fyzická paměť)

## WINDOWS

- Používá se mechanismus pro mapování souborů do paměti
- `CreateFileMapping`, `MapViewOfFile`
- Lze použít i stránkovací soubor

## UNIX

- `Shmget` – vytvoří/najde úsek sdílené paměti s daným klíčem (nastavení oprávnění)
- `Shmat` a `shmdt` – namapuje odmapuje sdílenou paměť s adresního prostoru

## SIGNÁLY

- Mechanismus podobný přerušení (asynchronní volání)
- Základní forma komunikace v unixech
- Proces může definovat vlastní handlersy těmto signálům
- Proces je možné zaslat jeden z celočíselných signálů
- Některé speciální určení (případně nasatavené na implicitní handlersy)
  - `SIGOUT` – ukončení procesu (`Ctrl + C`)
  - `SIGQUIT` – ukončení procesu
  - `SIGSTOP` – pozastavení procesu
  - `SIGCONT` – pokračování pozastaveného potomka
  - `SIGHLD` – zastavení signálu potomka // pokud si proces neuloží hodnotu vrácenou potomkem a zanikne -> vznikne zombie proces
- Race-conditions
- Nelze zasílat složitější zprávy

## ROURY

- Typická vlastnost unixových OS (ale podpora i ve Windows)
- Mechanismus umožňující jednosměrnou komunikace mezi procesy
- Komunikace dvou procesů (jeden zapisující konec, druhý čtecí konec)
- First-In-First-Out
- Umožňuje propojit vstupy a výstupy procesů →kompozice do větších celků
- V shellu: `cat foo.log | grep „11/11/2011“ | wc -l` // vrátí počet řádků kde je 11/11/2011
- Využití společně se standardním vstupem (`stdin`) a výstupem (`stdout`)

- Typické použití
  - Rodičovský proces vytvoří rouru voláním `pipe` (dva popisovače souborů – zápis a čtení)
  - Po zavolání `fork()` potom dědí oba tyto popisovače
  - Rodič i potomek zavírají nepotřebné popisovače
  - Je možné zapisovat /číst z/do jednotlivých popisovačů souborů
- u procesu lze přenastavit popisovače pro `stdin` a `stdout`, aby ukazovaly na konec roury
- rodič může propojit dva potomky (oba dědí popisovače)
- v Linuxu velikost bufferu 1 MB
- pokud je plný, zapisující proces je pozastaven, pokud je prázdný, čtecí proces je pozastaven
- více čtenářů/písařů → race-condition (operace nemusí být atomické)

### POJMENOVANÉ ROURY (FIFO)

- soubory, který se chová jako roura
- volání a program `mkfifo`
- umožňuje komunikaci nepříbuzných procesů

### PSEUDO-ROURY

- systém emulující (MS-DOS), ale vytváří mezi soubory
- `dir | sort | more`  
→ `dir > 1.tmp && sort <1.tmp > 2.tmp && more <2.tmp>`

## ZASÍLÁNÍ ZRPÁV

- message passing
- obecný mechanismus komunikace mezi procesy (→ různé varianty)
- vhodný pro počítače se sdílenou pamětí i pro distribuované systémy
- základní operace“
  - `send (dest, message)`
  - `receive (src, message)`
- `send` i `receive` → jako blokuující/neblokuující operace
  - `send` i `receive` blokuující – synchronizace
  - `send` neblokuující, `receive` blokuující – příjemce čeká na zprávu
  - `send` i `receive` neblokuující

## ADRESACE

- **přímá** – vhodné pro kooperující procesy
- **nepřímá**
  - zprávy jsou zasílány do fronty (mailbox) odkud jsou vyzvedávány příjemcem
  - různé varianty 1:1, 1:N, N:1, M:N
- **zprávy**: hlavička + tělo zprávy → (odlišování od volání)
- **tělo zprávy**: pevná vs. proměnlivá velikost
- **hlavička**: typ zprávy, zdroj, cíl, délka zprávy, (kontrolní informace, priorita)

## VZÁJEMNÉ VYLOUČENÍ

- zasíláním zpráv lze implementovat vzájemné vyloučení
- využívá se blokující `receive`
- společná schránka obsahuje žádnou nebo jednu zprávu → token udávající, že lze vstoupit do kritické sekce
  - pokud je ve schránce jedna zpráva

## ZASÍLÁNÍ ZPRÁV V OS

### POSIX MESSAGE QUEUE

- nepoužívá se často, není součástí std. knihovny (librt)
- velikost fronty a zpráv pevná (definovaná při otevření)
- `mq_open`, `mq_send`, `mq_receive`
- v současnosti se používá spíš D-BUS

### WINDOWS

- událostmi řízený systém
- zprávy zasílané jednotlivým oknům (všechno je okno)
- smyčka událostí součástí funkce `WinMain`

## DALŠÍ MECHANIZMY

- Remote Procedure Calls
  - klient volá zástupnou funkci (stub)
  - zástupná funkce provede převod parametrů a odešle zprávu
  - jádro předá zprávu cílovému počítači
  - server zpracuje příchozí zprávu (provede převod parametrů)

- provede se funkce
- odpověď je vrácena opačným způsobem
- #114
- různá implementace – CORBA, .NET Remoting, Java Remote Method Invocation, XML-RPC, SOAP
- Windows: DDE, COM, clipboard, Mailslots, pojmenované roury (i přes síť)
- (unixové) sockety – jako síťové rozhraní, ale lokální (→ rychlejší)

Zkouška: 5-6 otázek 20 minut, ústní zkouška

i otázky typu: jak se řeší skoky, jak se řeší bitové operace, proč něco tak je... atd