Operační systémy

Řízení výpočtu

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

Reprezentace hodnot

- záporná čísla jsou v doplňkovém kódu – dvojkový doplněk (zápornou hodnotu dostaneme tak, že provedeme inverzi bitů a přičteme 1) $\Longrightarrow$ snadná manipulace
- znaménková a beznaménkové typy (`unsigned int` vs. `int`)!!!
- pokud se hodnota nevejde do rozsahu typu $\Longrightarrow$ přetečení/podtečení

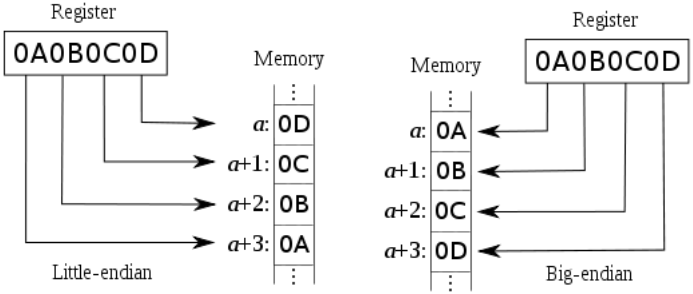
```
char a = 127 + 1;            // => -128
unsigned char c = 255 + 1;   // => 0
char b = -10 - 120;          // => 126
```

BCD (Binary Coded Decimal)

- čísla v desítkové soustavě 4b na cifru

Více-bytové hodnoty

- různé způsoby uložení více-bytových hodnot (endianita)
- liší se mezi procesory $\Longrightarrow$ potřeba brát v úvahu při návrhu datových formátů a protokolů
- little-endian: hodnoty jsou zapisovány od nejméně významného bytu
- big-endian: hodnoty jsou zapisovány od nejvýznamějšího bytu



- Little-endian: x86, Amd64, Alpha, ...
- Big-endian: SPARC, IBM POWER, Motorola 68000, ...
- Bi-endian: ARM, PowerPC, SparcV9, IA-64, ... (za určitých okolností lze přepínat)

Řetězce

- uložení řetězců záleží na zvoleném kódování (viz text Operační systémy)

Řetězce (1/2)

ASCII (American Standard Code for Information Interchange)

- způsob kódování znaků
- původně použité 7bitové hodnoty (později rozšířeny na 8 bitů)
- řídící znaky (CR, LF, BELL, TAB, backspace atd.)
- národní abecedy – horní polovina tabulky, kódování ISO-8859-X, Windows-125X, atd.

Unicode

- znaková sada (definuje vazbu číslo $\Leftrightarrow$ znak)
- několik tzv. *rovin* po 65535 znacích (v současnosti 110.000+ znaků)
- první rovina se nazývá základní (Basic Multilingual Plane, BMP) – znaky západních jazyků

UCS (Universal Character Set)

- způsob kódování znaků Unicode
- pevně daná velikost
- UCS-2 – 16 bitů na znak, odpovídá základní rovině UNICODE
- UCS-4 – 32 bitů na znak, všechny znaky UNICODE

Řetězce (2/2)

UTF-8 (Unicode Transformation Format)

- kódování znaků s proměnlivou délkou
- zpětně kompatibilní s ASCII

bity	rozsah UNICODE	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6
7	0000–007f	0xxxxxxx					
11	0080–07ff	110xxxxx	10xxxxxx				
16	0800–ffff	1110xxxx	10xxxxxx	10xxxxxx			
21	10000–1fffff	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	200000–3ffffff	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	4000000–7fffffff	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-16

- proměnlivá délka kódování znaku
- rozšiřuje UCS-2
- varianty UTF-16BE or UTF-16LE
- Byte Order Mark (BOM) – umožňuje určit typ kódování (0xffef nebo 0xfeff)

Zásobník

- procesor má vyčleněný úsek paměti pro zásobník (LIFO) $\Longrightarrow$ mezivýpočty, návratové adresy, lokální proměnné, ...
- vyšší prog. 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 vrcholu zásobníku)
- uložení/odebrání hodnot pomocí operací:

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

POP r/m             ;; mov op, [esp]
                    ;; add esp, 4
```

- registr ESP musí vždy obsahovat číslo, které je násobek čtyř

Volání podprogramů/funkcí

- k volání podprogramu se používá operace `CALL r/m/i` $\Longrightarrow$ uloží na zásobník hodnotu registru IP a provede skok

```
push eip            ;; tato operace neexistuje
jmp <addr>
```

- k návratu z funkce se používá operace `RET` $\Longrightarrow$ odebere hodnotu ze zásobníku a provede skok na adresu danou touto hodnotou
- použití zásobníku umožňuje rekurzi

Volání funkcí

- předání parametrů
- vytvoření lokálních proměnných
- provedení funkce
- odstranění informací ze zásobníku
- návrat z funkce, předání výsledku

Konvence volání funkcí (1/2)

- způsob, jakým jsou předávány argumenty funkcím, jsou jen konvence (specifické pro překladač, i když často součástí specifikace ABI OS)
- předávání pomocí registrů (dohodnou se urč. registry), příp. zbývající argumenty se uloží na zásobník
- předávání argumentů čistě přes zásobník
- kdo odstraní předané argumenty ze zásobníku? (volaná funkce nebo volající?)
- 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 volání funkcí (2/2)

- 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
- návratová hodnota se na i386 obvykle předává pomocí registru EAX, příp. EDX:EAX

Rámec funkce (stack frame)

- při volání funkcí se na zásobníku vytváří tzv. rámec (stack frame)
- obsahuje předané argumenty, adresu návratu, příp. lokální proměnné
- k přístupu k tomuto rámci se používá registr EBP

Volání funkce s konvecím cdecl (1/4)

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 obsah 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>`)

Návrat z funkce

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 z funkce `ret`
5. odstraníme argumenty ze zásobníku (lze použít přičtení k ESP)

Obsah zásobníku

```
                 ...
                 argument n
                 ...
EBP + 12   -->   argument 2
EBP + 8    -->   argument 1
                 návratová hodnota
                 původní EBP
EBP - 4    -->   první lokální proměnná
EBP - 8    -->   druhá lokální proměnná
                 ...
ESP        -->   poslední lokální proměnná
```

Volání funkce s konvecím cdecl (3/4)

Volání funkce

```
push arg2          ;; druhy argument
push arg1          ;; prvni argument
call func
add esp, 8         ;; odstrani oba argumenty ze zasobniku
```

Tělo funkce

```
push ebp
mov ebp, esp
sub esp, n         ;; vytvori misto pro lokalni promenne
push ...           ;; ulozi obsah pouzivanych registru
...                ;; telo funkce
pop ...            ;; vrati hodnoty registru do puvodniho stavu
mov esp, ebp       ;; odstrani lokalni promenne
pop ebp
ret
```

Volání funkce s konvecím cdecl (4/4)

- první argument leží na adrese [ebp + 8], druhý na [ebp + 12], atd.
- první lokální proměnná na [ebp - 4], druhá na [ebp - 8], atd.

Uchovávání registrů

- uchovávání všech použitých registrů na začátku každé funkce nemusí být efektivní
- 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

Další vlastnosti ISA procesorů x86

- FPU (floating-point unit): výpočty s čísly s řádovou čárkou (80bitová čísla; zásobníkový procesor; registry ST0–ST7)
- MMX: multimediální instrukce, SIMD, práce s celými čísly (64bitové registry MM0–MM7 sdílené s FPU)
- SSE, SSE2: instrukce pro práci s čísly s plovoucí řádovou čárkou (SIMD; jednoduchá i dvojitá přesnost; 128bitové registry XMM0–XMM7)

AMD64 (1/2)

- 64bitové 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 rXw (např. r8w)
  - spodních 8 bitů jako registry rXb (např. r8b)
- nové 128bitové registry xmm8-xmm15
- nejnovější procesory s AVX (Sandy Bridge, Bulldozer) rozšiřují xmm0-xmm15 na 256bitů (registry ymm0-ymm15)
- adekvátní rozšíření operací (prefix REX); omezení délky instrukce na 15 B
- v operacích je možné používat jako konstanty maximálně 32bitové hodnoty $\Longrightarrow$ výjimkou je operace (`movabs r, i`)

AMD64 (2/2)

- rozšíření adresního prostoru
- fyzicky adresovatelných typicky 2^{36} až 2^{46} B paměti (virtuální paměť 2^{48} B)

Režimy práce

- 64bitová ISA je velice podobná 32bitové $\Longrightarrow$ minimální režie
- **Long mode**: dva submody (ve kterých jsou k dispozici 64bitové rozšíření)
  - 64-bit mode: OS i aplikace v 64bitovém režimu
  - compatibility mode: umožňuje spouštět 32bitové aplikace v 64bitovém OS
- **Legacy mode**: režimy pro zajistění zpětné kompatibility (protected mode, real mode)
- pro výpočty s čísly s plovoucí řádovou čárkou se používají operace SSE, SSE2

Volací konvence

- větší množství registrů umožňuje efektivnější volání funkcí (podobné fastcall)
- možnost zakódovat strukturovanou hodnotu do registru
- zarovnání zásobníku na 16 B
- sjednocení volácích konvencí (v rámci platformy)

AMD64: Volací konvence – Windows

- první 4 argumenty: rcx, rdx, r8, r9
- čísla s plovoucí řádovou čárkou přes: xmm0-xmm3
- na zásobníku se vytváří stínové místo pro uložení argumentů
- zbytek přes zásobník
- návratové hodnoty přes rax nebo xmm0

```
// a -> rcx, b -> xmm1, c -> r8, d -> xmm3
void foo(int a, double b, int c, float d);

sub rsp, 0x28              ; (0x20 + 0x08 -- kvuli zarovnani po call)
movabs rcx, <addr: msg>
call printf
add rsp, 0x28
```

- caller-saved: rax, rcx, rdx, r8, r9, r10, r11
- callee-saved: rbx, rbp, rdi, rsi, rsp, r12, r13, r14, r15

AMD64: Volací konvence – Windows

Pozice	Obsah	Rámec
...	...	předchozí
`rbp + 64`	6. argument	
`rbp + 56`	5. argument	
...	...	
`rbp + 16`	prostor pro uložení registrů	
`rbp + 8`	návratová adresa	aktuální
`rbp`	původní `rbp`	
`rbp - 8`	lok. proměnné a nespecifikovaná data	
...	...	
`rsp`	...	

AMD64: Volací konvence – Unix (Linux)

- prvních 6 argumentů: rdi, rsi, rdx, rcx, r8, r9
- čísla s plovoucí řádovou čárkou přes: xmm0-xmm7 (počet použitých XMM registrů musí být v registrů AL)
- zbytek přes zásobník (zprava doleva)
- návratové hodnoty přes rax nebo xmm0
- pod vrcholem zásobníku oblast 128 B (červená zóna) pro libovolné použití

```
// a -> rdi, b -> xmm0, c -> rsi, d -> xmm1; 2 -> al
void foo(int a, double b, int c, float d);
```

- caller-saved: rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11
- callee-saved: rbx, rsp, rbp, r12, r13, r14, r15

AMD64: Volací konvence – Unix (Linux)

Pozice	Obsah	Rámec
`rbp + 16 + 8 *` n	n-tý argument na zásobníku	předchozí
...	...	
`rbp + 16`	0-tý argument na zásobníku	
`rbp + 8`	návratová adresa	aktuální
`rbp`	původní `rbp`	
`rbp - 8`	lok. proměnné a nespecifikovaná data	
...	...	
`rsp`	...	
`rsp - 128`	červená zóna	

Přerušení

- mechanismus umožnující reagovat na asynchronní události
- nejčastěji vyvolané vnějším zařízením (např. stisk klávesnice), které chce komunikovat s CPU
- pokud vznikne přerušení, činnost procesoru je zastavena a je vyvolána *obsluha přerušení*
- po skončení obsluhy přerušení program pokračuje tam, kde byl přerušen
- obslužné rutiny – velice podobné běžným funkcím
- procesor ví, kde jsou uloženy obslužné rutiny přerušení
- na x86 IDT (Interrupt Descriptor Table) – vektor 256 adres (256 přerušení – prvních 32 HW přerušení, zbylé pro SW přerušení); adresa IDT uložena v registru IDTR
- systém priorit (přerušení s nižší prioritou nemůže, přerušit pokud již běží přerušení s vyšší a musí počkat)

I/O zařízení (1/2)

Aktivní čekání

- procesor pracuje se zařízením přímo (instrukce `in`, `out` – zápis/čtení hodnoty z portu)
- výpočetně náročné (obzvlasť přenosy velkých dat); omezené na speciální operace (jen zápis/čtení)

DMA

- řadič DMA dostane požadavek: čtení/zápis + adresu v paměti
- předá požadavek řadiči zařízení (např. disku)
- zapisuje/čte data z/do paměti
- dokončení je oznámeno řadiči DMA
- DMAC vyvolá přerušení
- př. Tan p.277

Sdílení paměťového prostoru

- zařízení mají přímý přístup k operační paměti

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
- $\Longrightarrow$ jádro operačního systému
- CPU různé režimy práce:
  - privilegovaný – běží v něm jádro OS (umožňuje vše)
  - neprivilegovaný – běží v něm aplikace (některé funkce jsou omezeny)
- přechod mezi režimy pomocí *systémových volání* (SW přerušení, speciální instrukce, speciální volání)

Architektura jádra (1/2)

Monolitické jádro

- vrstvená architektura; moduly
- všechny služby pohromadě $\Longrightarrow$ lepší výkon
- problém s chybnými ovladači
- Linux, *BSD

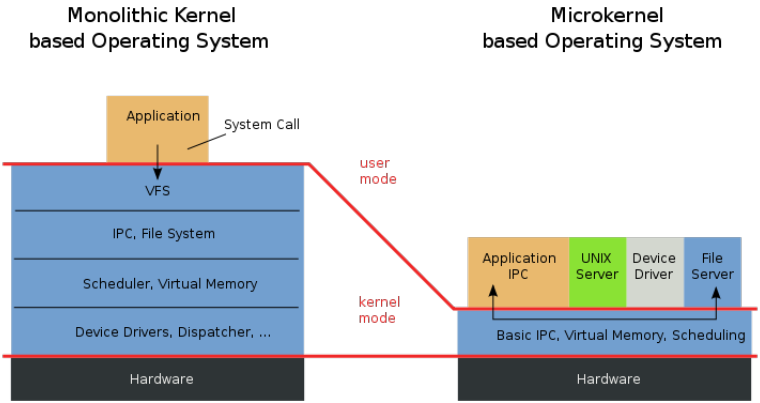
Mikrojádro

- poskytuje správu adresního prostoru, procesů, IPC
- oddělení serverů (služeb systému); běžné procesy se speciálními právy $\Longrightarrow$ bezpečnost
- možnost restartu servrů, pomalé IPC (přepínaní kontextu)
- MINIX, QNX, Mach

Hybridní jádro

- kombinuje prvky obou přístupů
- Windows NT, MacOS X

Architektura jádra (2/2)



Samostudium

- Keprt A. Operační systémy.
- Kapitoly 5–8, tj. strany 41–83.