Aplikačné vs. systémové programy

- aplikačné programy
  - riešia aplikačné úlohy
  - nezávislé od HW, detailov OS
- systémové programy
  - podporujú používanie počítača
  - závislé od detailov OS
  - môžu byť závislé od HW
  - napr. OS, kompilátor, assembler, linker, loader, debugger, konfiguračné programy, ...

Zjednodušená štruktúra počítača

- procesor
  - aritmeticko-logická jednotka
  - riadiaca jednotka
  - registre
- hlavná pamäť
  - medzi ňou a procesorom sa často nachádza cache pamäť (alebo hierarchia cache pamätí)
- vstupno-výstupné rozhranie
  - pripojenie periférnych zariadení

Reprezentácia údajov

- dátový typ
  - množina hodnôt
  - množina operácií
- abstraktný dátový typ
  - množina abstraktných entít a operácií bez vzťahu k počítaču
- virtuálny dátový typ – entity prog. jazyka
- fyzický dátový typ
  - fyzicky uložené entity v HW počítača

Reprezentácia údajov

- v HW (register, pamäť) uložené ako reťazec bitov
- kódovanie
  - relácia medzi prvkami dátového typu a jeho reprezentáciou v bitovom reťazci

Celé čísla

- bez znamienka
  - BCD (binary coded decimal)
    - každá číslica desiatkovej sústavy reprezentovaná 4 bitmi
    - príklad: 1000 0110 = 86
  - v dvojkovej sústave
    - príklad: 1000 0110 = 134 = 0x86

Celé čísla

- znamienko + abs. hodnota
  - príklad: 0100 0110 = 70
1100 0110 = -70
  - dve reprezentácie 0, nepoužiteľnosť sčítačky na odčítanie
- 1's complement
  - kladné čísla majú najvyšší bit 0, -x = NOT(x)
  - príklad: 0100 0110 = 70, 1011 1001 = -70
  - sčítanie sa doplní o pripočítanie prenosu
  - 1111 1111 = 0000 0000

Celé čísla

- 2's complement
  - kladné čísla majú najvyšší bit 0
  - -x = NOT(x) + 1
  - príklad: 0100 0110 = 70
1011 1010 = -70
  - sčítačka funguje aj so zápornými číslami (a teda aj na odčítanie)
  - najčastejšie používaný formát celých čísel

Čísla s pohyblivou rádovou čiarkou

- $(-1)^s * m * 2^{e-e_0}$
  - znamienko (s) – 1 bit (0=+, 1=-)
  - mantisa (m=1.xxx v dvojkovej sústave)
  - exponent (e, ukladá sa zväčšený o e_0)
  - IEEE single: m: 23 bitov, e: 8 bitov, e_0=127
    IEEE double: m: 52 bitov, e: 11 bitov, e_0=1023
  - príklad: 00111111100000000000000000000000 = 1.5

Jazyk assemblera

- príkazy
  - direktívy
    - riadiace príkazy pre assembler, napr. vyhradenie miesta, deklarácia typu symbolu, ...
  - strojové inštrukcie
    - zodpovedajú inštrukciám procesora
  - makropríkazy
    - používateľom definované makrá

Typy inštrukcií

- prenos dát (medzi reg. a pamäťou)
- aritmetické inštrukcie
  - sčítanie, odčítanie, násobenie, delenie, porovnanie, negácia, ...
- logické inštrukcie
  - AND, OR, XOR, NOT
- riadiace inštrukcie
- vstupno/výstupné inštrukcie
- iné

Registre i386

- 32 bitové eax, ebx, ecx, edx, esi, edi, ebp, esp (stack pointer), eip (instruction pointer)
- 16 bitové ax, bx, cx, dx, si, di, bp, sp
- 8 bitové ah, al, bh, bl, ch, cl, dh, dl
- segmentové cs, ds, es, ss, fs, gs
- príznaky (flags) (carry, overflow, sign, ...)
- riadiace, ladiace
- 80 bitové pre floating point (st0 – st7)

Adresné módy

- Registrový
  - mov %eax, %ebx mov ebx, eax
  - ebx := eax
- Nepriamy registrový
  - mov (%eax), %ebx mov ebx, dword ptr [eax]
  - ebx := MEM[eax]
- Nepriamy s doplnkom
  - mov 10(%eax), %ebx mov ebx, dword ptr [eax+10]
  - ebx := MEM[eax+10]

Adresné módy

- Nepriamy s doplnkom a indexom
  - mov 10(%eax, %esi, 4), %ecx
  - mov ecx, dword ptr [eax+esi*4+10]
  - ecx := MEM[eax+esi*4+10]
- Priama adresa
  - mov 100, %eax mov eax, dword ptr [100]
  - eax := MEM[100]
- Priama hodnota
  - mov $5, %eax mov eax, 5
  - eax := 5

Adresné módy

- Relatívna adresa
  - ukladá sa adresa cieľa relatívne k EIP
  - v i386 sa používa pri skokoch a volaniach (jmp, call)
- Autoinkrementačný/autodekrementačný
  - adresa v registri sa automaticky zväčší/zmenší po (pred) vykonaním inštrukcie
  - v i386 sa používa len pri „reťazcových“ inštrukciách (adresuje sa registrami esi a edi)

Štruktúra programu

- program sa bežne skladá z viacerých modulov
  - jednotlivé moduly môžu byť napísané v rôznych prog. jazykoch alebo v assembleri
- modul obsahuje *sekcie*
  - .text – kód
  - .data – inicializované premenné
  - .bss – neinicializované premenné
- sekcie spája *linker*

Štruktúra modulu v assembleri (GNU as)

```
.text
.global f
f: mov a, %eax
   mov %eax, b
   ret

.data
a: .long 10

.bss
.global b
b: .long 0
```

Niektoré direktívy

- .global
  - deklaruje symbol ako globálny, t.j. dostupný aj z iných modulov
- .text, .data, .bss
  - prepína sekcie
- .byte, .word, .long
  - vyhradzuje miesto príslušnej veľkosti (1, 2, 4B) a definuje hodnotu, ktorá sa doň uloží

Vybrané inštrukcie i386

- add a, r r:=r+a
- adc a, r r:=r+a+(hodnota bitu prenosu)
- sub a, r r:=r-a
- sbb a, r r:=r-a-(hodnota bitu prenosu)
- inc r r:=r+1 (nemení príznaky)
- dec r r:=r-1 (nemení príznaky)
- neg r r:=-r
- cmp a, b nastaví príznaky podľa b-a

Vybrané inštrukcie i386

- mul a — edx:eax := eax * a
- imul a — ako mul, ale znamienkové
- div a — eax := edx:eax / a, edx:=zvyšok
- idiv a — ako div, ale znamienkové
- and a, r — r:=r AND a
- or a, r — r:=r OR a
- xor a, r — r:=r XOR a
- not r — r:=NOT r

Vybrané inštrukcie i386

- mov s, d d:=s
- xchg a, b a:=:b (výmena obsahu)
- push a ESP-=4, MEM[ESP]:=a
- pop a a:=MEM[ESP], ESP+=4
- pushf push flags
- popf pop flags

Vybrané inštrukcie i386

- jmp addr skok na adresu
  - rel. adr. alebo adr. v pamäti/registri
- call addr volanie funkcie
- ret návrat z funkcie
- j*cond* rel8 podmienený skok, ak *cond*
  - z(e) =0, nz(ne) !=0
  - b(c) <0, ae(nc) >=0, a >0, be <=0 – neznam. por.
  - l <0, ge >=0, g >0, le <=0 – znam. porovnanie

Práca so zásobníkom

- Zásobník umožňuje ukladať položky dát spôsobom LIFO.
- Na i386 v 32-bit. móde je veľkosť základnej položky 4B.
- Zásobník rastie smerom „dolu“ (t.j. od vyšších adries k nižším).
- Na najnovšiu položku ukazuje register ESP.
- Na adresovanie sa používajú registre ESP a EBP v nepriemych módoch s doplnkom.

Práca so zásobníkom

- Zásobník (stack) sa používa napr. pre uloženie lokálnych premenných a argumentov funkcií.
  - Pred volaním funkcie sa uložia hodnoty argumentov do zásobníka.
  - Inštrukcia call uloží do zásobníka návratovú adr.
  - Vyhradí sa miesto pre lokálne premenné.
  - Pred návratom sa odstránia lok. premenné.
  - Inštrukcia ret vyberie návratovú adresu.
  - Odstránia sa argumenty.

Práca so zásobníkom (C volacia konvencia)

arg. 3	16(%ebp)
arg. 2	12(%ebp)
arg. 1	8(%ebp)
návrat. adr.	
odlož. EBP	(%ebp)
lok. prem. 1	-4(%ebp)
lok. prem. 2	-8(%ebp)

Volanie funkcie

```
push arg3
push arg2
push arg1
call f
add $12, %esp
```

Začiatok funkcie

```
push %ebp
mov %esp, %ebp
sub $8, %esp
```

Koniec funkcie

```
mov %ebp, %esp
pop %ebp
ret
```

Práca so zásobníkom (Pascal volacia konvencia)

- Argumenty sa ukladajú v poradí (pri C v opačnom poradí).
- Argumenty neodstraňuje volajúci, ale inštrukcia `ret n`, kde n je počet B, ktoré majú byť odstránené zo zásobníka.
- Neumožňuje funkcie s premenlivým počtom argumentov.

Vracanie hodnôt z funkcie

- Malé hodnoty sa zvyčajne vracajú v registri.
  - na i386 v registri EAX, prípadne dvojici EDX:EAX
- Veľké hodnoty (napr. štruktúra) sa vracajú tak, že volajúci poskytne (ako argument) adresu, kam sa má hodnota výsledku uložiť.
  - v gcc na i386 sa posiela ako prvý argument (posledný vložený) a odstraňuje sa použitím inštrukcie `ret $4`.

Assembler – prekladač

- prekladá program v jazyku assemblera do strojového kódu
- vytvára pomocné informácie o module potrebné pre linker a loader
- vstupný riadok má tvar
  `[návestie:] inštrukcia|direktíva operandy`
- návestie slúži na odkazovanie sa na adresu
- hodnotu návestiu priradí assembler

Assembler – prekladač

- lokálne návestia
  - môžu sa predefinovávať
  - majú tvar N:, kde N je kladné číslo
  - odkazuje sa na ne:
    - nasledujúci výskyt: Nf
    - predchádzajúci výskyt: Nb
- definícia konštánt
  - MENO = výraz
- výrazy – bežné aritm. operácie

Assembler – prekladač

- na začiatku nastaví hodnotu LC = 0
- keď potrebuje priradiť hodnotu návestiu, použije aktuálnu hodnotu LC
- LC zvyšuje vždy o dĺžku inštrukcie, resp. podľa pokynu direktívy (napr. pri definovaní dát podľa dĺžky dát)
- podľa počtu prechodov cez vstup delíme assemblery na jednoprechodové a dvojprechodové

Dvojprechodový assembler

- 1. prechod
  - číta vstupný text a priraďuje adresy každej inštrukcii alebo dátam
  - vytvára tabuľku symbolov obsahujúcu mená návestí a príslušnú hodnotu LC v čase definície návestia
  - môže doplniť vstupný text o ďalšie informácie
- 2. prechod
  - generuje strojový kód, využíva informácie z 1. p. (hlavne tabuľku symbolov)

Jednoprechodový assembler

- číta vstup len raz
- problém vzniká s návestiami, ktoré sú definované neskôr, ako sú použité
- musí vytvoriť zoznam nedefinovaných návestí s informáciou, kam treba doplniť ich hodnotu, a po skončení prechodu dopniť na príslušné miesta adresy návestí

Makrá

- makro – pomenovaná postupnosť inštrukcií
  - môže mať aj parametre
- pri použití sa nahradí svojím telom
- makro vs. funkcie(podprogramy)
  - volanie funkcie je opráciou procesora
  - „volanie“ makra procesor nevidí
  - makro je vhodné
    - pre krátke postupnosti
    - keď je dôležitá rýchlosť

Makrá

- definícia

```
.macro pridaj co, kam=%eax
   add \co, \kam
.endm
```

- použitie
  - `pridaj $6, %ebx`
  - `pridaj $6`
  - `pridaj kam=%ecx, co=$6`

Makroprocesor

- úloha
  - nájsť a uložiť definície makier
  - nájsť volania makier a nahradiť ich telom makra (so substitúciou parametrov)
- makroprocesor môže byť nezávislý od assemblera alebo môže byť integrovaný
- podľa počtu prechodov delíme makroprocesory na jednoprechodové a dvojprechodové

Dvojprechodový makroprocesor

- 1. prechod
  - hľadá definície makier a ukladá si ich do tabuľky makier spolu so zoznamom parametrov a ich default hodnotami
- 2. prechod
  - číta zdrojový text, ignoruje definície makier
  - ak nájde volanie makra, rozvinie ho podľa jeho definície v tabuľke
  - inak skopíruje riadok na výstup bez zmeny
- nemôže spracovať vnorené definície makier

Jednoprechodový makroprocesor

- v jednom prechode hľadá definície a zároveň robí rozvíjanie makier
- definícia makra musí predchádzať jeho použitiu
- môže byť zakomponovaný do prvého prechodu assemblera

Linker (spájač)

- programy pozostávajú z modulov
- každý modul je samostatne spracovaný kompilátorom alebo assemblerom
  - výsledkom je „object file“ – objektový súbor
- linker spája jednotlivé objektové súbory
  - spája jednotlivé sekcie rovnakého typu z rôznych modulov
  - relokuje adresy

Objektový súbor

- identifikácia
  - meno modulu, čas kompilácie, informácie pre linker
- tabuľka globálnych symbolov
  - mená, typy a relatívne adresy exportovaných symbolov
- tabuľka externých symbolov
  - mená použitých a nedefinovaných symbolov a relatívne adresy miest, kde treba doplniť adresu symbolu

Objektový súbor

- tabuľka relokácií
  - zoznam miest, ku ktorým je potrebné pripočítať počiatočnú adresu
- obsah inicializovaných sekcií
  - najmä sekcia .text (kód) a .data (inicializované dáta)
- adresa začiatku programu
- ladiace informácie

Knižnice

- mnohé funkcie sú často využívané mnohými programami
  - často sa implementujú v samostatných moduloch, ktoré sa potom pripájajú k programom
  - takéto moduly sa zvyčajne uložia do tzv. knižníc (archívov objektových súborov)
  - z knižnice linker potom vyberie potrebné moduly
  - príkladom je štandardná knižnica libc, ktorá obsahuje štandardné funkcie jazyka C vrátane rozhraní k systémovým volaniam

Práca linker-a

- načíta informácie o jednotlivých moduloch, buduje globálnu tabuľku symbolov
  - ak nájde nedefinované symboly, prehľadá určené knižnice, aby v nich našiel moduly, ktoré príslušné symboly definujú
  - ak nájde nedefinované symboly, vyhlási chybu
- pridelí miesto v pamäti jednotlivým sekciám zo všetkých modulov (sekcie rovnakého typu ukladá za seba)

Práca linker-a

- vykoná relokácie
  - na miesta určené údajmi z tabuľky relokácií a z tabuľky externých symbolov pripočíta príslušné hodnoty (adresa externého symbolu, začiatok sekcie)
- zapíše výsledok do súboru
  - absolute load module – obsahuje definitívny obraz programu, musí sa zaviesť na určenú adresu
  - relative load module – obsahuje relokačnú tabuľku, je potrebné robiť ďalšie relokácie

Loader (zavádzač)

- načíta súbor vytvorený linkerom
- umiestni ho do pamäte
  - v prípade relative load module vykoná relokácie podľa skutočnej adresy
- pridelí pamäť pre neinicializované dáta (sekcia .bss)
- odovzdá riadenie programu

Čas viazania (binding)

- kedy je ukončené mapovanie symbolických mien na fyzické adresy
  - počas písania programu
  - počas prekladu programu
  - počas linkovania (absolute load module)
  - počas loadovania
  - pri uložení bázovej adresy do registra
  - pri vykonávaní inštrukcie
- virtuálne vs. fyzické adresy

Dynamické linkovanie

- moduly môžu byť prilinkované až pri spustení programu alebo až pred prvým použitím funkcie z modulu
- štandardné knižničné funkcie
  - šetrí diskový priestor, nevyžaduje nové linkovanie všetkých programov pri zmene knižničnej funkcie
- „rozšírenia“ programov
  - šetrí pamäť, umožňuje konfigurovať funkcionalitu bez potreby linkovania

Príklad

a.s

```
.global d1, b2, f1

.data
d1: .long 6
d2: .long 4

.bss
b1: .long 0
b2: .long 0

.text
f1: mov d1, %eax
mov %eax, b1
mov %eax, e1
mov d2, %eax
mov %eax, b2
mov %eax, e2
ret
```

b.s

```
.global e1, e2, _start

.data
e1: .long 8

.bss
e2: .long 0

.text
_start:
call f1
pushl $0
call exit
```

Príklad

```
$ as -o a.o a.s
$ objdump -d a.o

a.o:     file format elf32-i386

Disassembly of section .text:

00000000 <f1>:
   0:   a1 00 00 00 00          mov    0x0,%eax
   5:   a3 00 00 00 00          mov    %eax,0x0
   a:   a3 00 00 00 00          mov    %eax,0x0
   f:   a1 04 00 00 00          mov    0x4,%eax
  14:   a3 00 00 00 00          mov    %eax,0x0
  19:   a3 00 00 00 00          mov    %eax,0x0
  1e:   c3                      ret
```

```
$ objdump -rt a.o

a.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000004 l       .data  00000000 d2
00000000 l       .bss   00000000 b1
00000000 g       .data  00000000 d1
00000004 g       .bss   00000000 b2
00000000 g       .text  00000000 f1
00000000         *UND*  00000000 e1
00000000         *UND*  00000000 e2


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000001 R_386_32          d1
00000006 R_386_32          .bss
0000000b R_386_32          e1
00000010 R_386_32          .data
00000015 R_386_32          b2
0000001a R_386_32          e2
```

Príklad

```
$ as -o b.o b.s
$ objdump -d b.o

b.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   e8 fc ff ff ff          call   1 <_start+0x1>
   5:   6a 00                   push   $0x0
   7:   e8 fc ff ff ff          call   8 <_start+0x8>
```

```
$ objdump -rt b.o

b.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 g       .data  00000000 e1
00000000 g       .bss   00000000 e2
00000000 g       .text  00000000 _start
00000000         *UND*  00000000 f1
00000000         *UND*  00000000 exit


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000001 R_386_PC32        f1
00000008 R_386_PC32        exit
```

Príklad

```
$ ld -r -o rel.o a.o b.o
$ objdump -d rel.o

rel.o:     file format elf32-i386

Disassembly of section .text:

00000000 <f1>:
   0:   a1 00 00 00 00          mov    0x0,%eax
   5:   a3 00 00 00 00          mov    %eax,0x0
   a:   a3 00 00 00 00          mov    %eax,0x0
   f:   a1 04 00 00 00          mov    0x4,%eax
  14:   a3 00 00 00 00          mov    %eax,0x0
  19:   a3 00 00 00 00          mov    %eax,0x0
  1e:   c3                      ret
  1f:   90                      nop

00000020 <_start>:
  20:   e8 fc ff ff ff          call   21 <_start+0x1>
  25:   6a 00                   push   $0x0
  27:   e8 fc ff ff ff          call   28 <_start+0x8>
```

```
$ objdump -rt rel.o

rel.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000004 l       .data  00000000 d2
00000000 l       .bss   00000000 b1
00000004 g       .bss   00000000 b2
00000000 g       .data  00000000 d1
00000020 g       .text  00000000 _start
00000008 g       .bss   00000000 e2
00000000 g       .text  00000000 f1
00000000         *UND*  00000000 exit
00000008 g       .data  00000000 e1


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000001 R_386_32          d1
00000006 R_386_32          .bss
0000000b R_386_32          e1
00000010 R_386_32          .data
00000015 R_386_32          b2
0000001a R_386_32          e2
00000021 R_386_PC32        f1
00000028 R_386_PC32        exit
```

Príklad

```
$ ld -o dyn -dynamic-linker /lib/ld-linux.so.2 a.o b.o -lc
$ objdump -d dyn

dyn:     file format elf32-i386

Disassembly of section .plt:

08048164 <.plt>:
 8048164: ff 35 60 92 04 08       pushl  0x8049260
 804816a: ff 25 64 92 04 08       jmp    *0x8049264
 8048170: 00 00                   add    %al,(%eax)
 8048172: 00 00                   add    %al,(%eax)
 8048174: ff 25 68 92 04 08       jmp    *0x8049268
 804817a: 68 00 00 00 00          push   $0x0
 804817f: e9 e0 ff ff ff          jmp    8048164 <f1-0x20>
Disassembly of section .text:

08048184 <f1>:
 8048184: a1 b0 91 04 08          mov    0x80491b0,%eax
 8048189: a3 6c 92 04 08          mov    %eax,0x804926c
 804818e: a3 b8 91 04 08          mov    %eax,0x80491b8
 8048193: a1 b4 91 04 08          mov    0x80491b4,%eax
 8048198: a3 70 92 04 08          mov    %eax,0x8049270
 804819d: a3 74 92 04 08          mov    %eax,0x8049274
 80481a2: c3                      ret
 80481a3: 90                      nop

080481a4 <_start>:
 80481a4: e8 db ff ff ff          call   8048184 <f1>
 80481a9: 6a 00                   push   $0x0
 80481ab: e8 c4 ff ff ff          call   8048174 <f1-0x10>
```

Príklad

```
$ objdump -rtRT dyn

dyn:     file format elf32-i386

SYMBOL TABLE:
080480d4 l    d  .interp    00000000
080480e8 l    d  .hash  00000000
080480fc l    d  .dynsym    00000000
0804811c l    d  .dynstr    00000000
08048136 l    d  .gnu.version    00000000
0804813c l    d  .gnu.version_r  00000000
0804815c l    d  .rel.plt   00000000
08048164 l    d  .plt   00000000
08048184 l    d  .text  00000000
080491b0 l    d  .data  00000000
080491bc l    d  .dynamic   00000000
0804925c l    d  .got   00000000
0804926c l    d  .bss   00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
080491b4 l       .data  00000000              d2
0804926c l       .bss   00000000              b1
08049270 g       .bss   00000000              b2
080491bc g     O .dynamic   00000000                    _DYNAMIC
080491b0 g       .data  00000000              d1
080481a4 g       .text  00000000              _start
08049274 g       .bss   00000000              e2
0804926c g       *ABS*  00000000              __bss_start
08048174       F *UND*  000000e2              exit@@GLIBC_2.0
08048184 g       .text  00000000              f1
0804926c g       *ABS*  00000000              _edata
0804925c g     O .got   00000000              _GLOBAL_OFFSET_TABLE_
08049278 g       *ABS*  00000000              _end
080491b8 g       .data  00000000              e1
```

```
DYNAMIC SYMBOL TABLE:
08048174      DF *UND*  000000e2  GLIBC_2.0   exit


DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049268 R_386_JUMP_SLOT   exit


$ ldd dyn
        libc.so.6 => /lib/libc.so.6 (0x40024000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2
(0x40000000)
```

Systémové volania

- umožňujú programom využívať služby OS
- na spodnej úrovni sú realizované využitím špeciálnych inštrukcií, ktoré zabezpečia predanie riadenie do jadra OS
  - softvérové prerušenie (Linux na i386: int $0x80)
  - volacie brány (call gates)
  - špeciálne inštrukcie
- programy využívajú štand. knižnice, ktoré zakrývajú detaily volania

Vytváranie nového procesu

- `pid_d fork(void)`
  - sys/types.h, unistd.h
  - vytvorí kópiu procesu
  - rodičovi vráti PID nového procesu
  - dieťaťu vráti 0
  - pri chybe vráti -1
  - oba procesy sú inak identické a môžu pokračovať v činnosti nezávisle na sebe

Ukončenie procesu

- `void exit(int status)`
  - stdlib.h
  - ukončí proces s návratovou hodnotou status
- `pid_t wait(int *status)`
  - sys/types.h, sys/wait.h
  - počká na ukončenie dieťaťa, vráti jeho PID
  - ak status != NULL, uloží informácie o návratovej hodnote (`WEXITSTATUS(hodnota)`)

„Démonizácia“ procesu

- `int daemon(int nochdir, int noclose)`
  - nochdir – prikazuje nezmeniť aktuálny adresár na /
  - noclose – prikazuje nepresmerovať štand. vstup a výstup na /dev/null
  - funkcia zabezpečí odpojenie procesu od riadiaceho terminálu a jeho beh na pozadí
  - vráti 0 (ok) alebo -1 (chyba)

Vstup/výstup

- File Descriptor – číslo identifikujúce otvorený súbor/zariadenie/socket/...
  - 0 – štandardný vstup
  - 1 – štandardný výstup
  - 2 – štandardný chybový výstup
- Operácie
  - open, close – otvorenie, zatvorenie
  - read, write, lseek – čítanie, zápis, posun pozície

open

- `int open(const char *pathname, int flags[, mode_t mode])`
  - `sys/types.h, sys/stat.h, fcntl.h`
  - otvorí súbor pathname spôsobom určeným flags, pri vytvorení súboru požaduje práva mode
  - vráti -1 pri chybe, inak deskriptor
  - flags
    - O_RDONLY – otvorí len na čítanie
    - O_WRONLY – otvorí len na zápis
    - O_RDWR – otvorí aj na čítanie aj na zápis

open, close

- flags môžu byť doplnené (|) o:
  - O_CREAT (ak neexistuje, vytvorí)
  - O_EXCL (chyba, ak existuje)
  - O_TRUNC (skráti na 0)
  - O_APPEND (pred každým zápisom sa posunie na koniec)
  - O_SYNC (synchrónne zápisy)
- `int close(int fd)`
  - unistd.h
  - zatvorí deskriptor fd, vrári 0 alebo -1

read, write

- `ssize_t read(int fd, void *buf, size_t count)`
  - unistd.h
  - prečíta z fd do buf najviac count bytov
  - vráti počet prečítaných alebo -1
- `ssize_t write(int fd, const void *buf, size_t count)`
  - unistd.h
  - zapíše do fd z buf count bytov
  - vráti počet zapísaných alebo -1

lseek, fsync

- `off_t lseek(int fildes, off_t offset, int whence)`
  - sys/types.h, unistd.h
  - nastaví aktuálnu pozíciu súboru filedes na offset bytov od miesta určeného whence:
    - SEEK_SET – od začiatku
    - SEEK_CUR – od aktuálnej pozície
    - SEEK_END – od konca
  - vráti novú aktuálnu pozíciu alebo -1
- `int fsync(int fd)`
  - vyprázdni (zapíše) cache

Spracovanie chýb

- Keď funkcia vráti -1, nastaví globálnu premennú errno (errno.h).
- `char *strerror(int errnum)`
  - string.h
  - vráti reťazec popisujúci zadanú chybu
- `void perror(const char *s)`
  - stdio.h
  - vypíše chybu určenú errno na stderr

Blokovanie procesu

- Štandardné volania I/O alebo wait zostanú čakať, kým úspešne neskončia.
- Spôsobuje to problém, ak proces potrebuje reagovať na viac ako jednu udalosť, napr.:
  - čítať z viac ako 1 deskriptoru,
  - občas volať wait, no robiť aj niečo iné.

waitpid

- `pid_t waitpid(pid_t pid, int *status, int options)`
  - sys/types.h, sys/wait.h
  - pid == -1 – čaká na ľubovoľné dieťa (ako wait)
  - options == WNOHANG – ak žiadne dieťa neskočilo, vráti 0 namiesto čakania

select

- ```
  int  select(int n,
  fd_set *readfds,
  fd_set *writefds,
  fd_set *exceptfds,
  struct timeval *timeout)
  ```
- `FD_CLR(int fd, fd_set *set)`
- `FD_ISSET(int fd, fd_set *set)`
- `FD_SET(int fd, fd_set *set)`
- `FD_ZERO(fd_set *set)`
  ```

select

- sys/select.h
- monitoruje 3 množiny deskriptorov:
  - readfds – či je možné čítať
  - writefds – či je možné zapisovať
  - exceptfds – či došlo k výnimkám
- timeout – určuje, ako dlho má select čakať, ak je NULL, tak nekonečne
  - struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
    };
- n = max deskriptor + 1

select

- vráti počet deskriptorov, kde nastala očakávaná udalosť
  - 0 = došlo k timeoutu
  - -1 = došlo k chybe
- množiny obsahujú tie deskriptory, kde došlo k udalosti
- FD_CLR, FD_SET vymaže/pridá deskriptor do množiny
- FD_ZERO vyprázdni množinu
- FD_ISSET testuje prítomnosť v množine

Signály

- Signály sú udalosti (chyby, externé udalosti) na ktoré môže proces reagovať.
- `sighandler_t signal(int signum, sighandler_t handler)`
  - signal.h, typedef void (*sighandler_t)(int);
  - inštaluje handler pre signál signum
    - SIG_IGN – ignorovať
    - SIG_DFL – štandardné správanie
  - vráti predchádzajúci handler

Signály

- SIGHUP – zatvorenie riadiaceho terminálu, pri démonoch zvyčajne požiadavka na rekonfiguráciu
- SIGINT – Ctrl+C
- SIGPIPE – zápis do zatvorenej rúry/socketu
- SIGTERM – žiadosť o ukončenie procesu
- SIGQUIT – Ctrl+\ - žiadosť o okamžité skončenie
- SIGCHLD – ukončenie dieťaťa
- SIGKILL – násilné ukončenie procesu

Vyvolanie signálu

- `int raise(int sig)`
- `int kill(pid_t pid, int sig)`
  - signal.h, sys/types.h
  - raise vygeneruje určený signál
  - kill pošle určený signál určenému procesu
    - pid = -1 – každému procesu

„Automatický“ wait

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>

void childdied(int sig)
{
  while (waitpid(-1, NULL, WNOHANG) > 0);
  signal(SIGCHLD, childdied);
}

int main()
{
  signal(SIGCHLD, childdied);
  char buf[128];
  int pid;
  pid = fork();
  if (pid == -1) { perror("fork"); exit(1); }
  if (pid == 0)
  {
    ...
  }
  else
  {
    ...
  }
}
```

Sieťová komunikácia

- komunikuje sa pomocou socket-ov
  - socket predstavuje koncový bod komunikácie
  - so socket-om sa manipuluje prostredníctvom file descriptor-a, podobne ako so súborom
- typy socketov
  - SOCK_DGRAM
    - nespoľahlivá komunikácia bez spojenia, protokol UDP
  - SOCK_STREAM
    - spoľahlivá komunikácia so spojením (funguje ako rúra pre byty), protokol TCP

TCP

- server (čaká na pripojenie klienta)
  - vytvorí socket (socket)
  - nastaví adresu (bind)
  - začne počúvať (listen)
  - príjme spojenie (accept), dostane nový deskriptor
  - komunikuje na deskriptore prijatého spojenia (send, recv, write, read, select)
  - požiada o ukončenie spojenia (shutdown)
  - zatvorí (zruší) socket (close)

TCP

- klient (nadväzuje spojenie)
  - vytvorí socket (socket)
  - môže nastaviť adresu (bind)
  - požiada o spojenie (connect)
  - komunikuje na deskriptore (send, recv, write, read, select)
  - požiada o ukončenie spojenia (shutdown)
  - zatvorí (zruší) socket (close)

UDP

- server/klient
  - vytvorí socket (socket)
  - môže nastaviť lokálnu adresu (bind) a adresu partneta (connect)
  - komunikuje na deskriptore (sendto, recvfrom, send, recv, write, read, select)
    - send a write môže použiť, len ak použil connect
  - zatvorí (zruší) socket (close)
- dáta sa prenášajú po jednotlivých správach

Adresy

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    u_int16_t      sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;  /* internet address */
};

/* Internet address. */
struct in_addr {
    u_int32_t      s_addr;     /* address in network byte order */
};
```

čísla portov sa udávajú v „sieťovom“ formáte

Adresy

- `uint16_t htons(uint16_t hostshort)`
- `uint16_t ntohs(uint16_t netshort)`
  - netinet/in.h
  - htons konvertuje short do sieťového formátu
  - ntohs konvertuje short zo sieťového formátu
- `char *inet_ntoa(struct in_addr in)`
  - netinet/in.h, arpa/inet.h
  - vráti textovú reprezentáciu adresy

Adresy

- `int inet_aton(const char *cp, struct in_addr *inp)`
  - konvertuje textovú reprezentáciu IP adresy na struct in_addr a vráti nenulu, ak je adresa ok, 0 ak nie je
- `in_addr_t inet_addr(const char *cp)`
  - podobne ako inet_aton, ale vracia adresu, INADDR_NONE pri chybe (zodpovedá 255.255.255.255 !!!)
- INADDR_ANY – znamená ľubovoľnú adresu

socket

- int socket(int domain, int type, int protocol)
  - sys/types.h, sys/socket.h
  - vytvorí socket a vrári deskriptor
  - domain == PF_INET (pre IPv4)
  - type
    - SOCK_STREAM pre TCP
    - SOCK_DGRAM pre UDP
  - protocol == 0
  - vytvorí socket a vráti deskriptor

bind

- `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`
  - sys/types.h, sys/socket.h
  - nastaví lokálnu adresu (IP adresa + port) socketu
  - vráti 0 (ok) alebo -1 (chyba)

```
int fd;
struct sockaddr_in sa;

sa.sin_family = AF_INET;
sa.sin_addr.s_addr = INADDR_ANY;
sa.sin_port = htons(cislo_portu);

fd = socket(PF_INET, SOCK_STREAM, 0);
bind(fd, &sa, sizeof(sa));
```

listen

- `int listen(int s, int backlog)`
  - sys/socket.h
  - nastaví socket do počúvacieho módu
  - backlog = max. počet spojení čakajúcich na prijatie
  - vráti 0 (ok) alebo -1 (chyba)

accept

- `int accept(int s, struct sockaddr *addr, socklen_t *addrlen)`
  - sys/types.h, sys/socket.h
  - príjme spojenie na sockete v počúvacom stave
  - v addr vráti adresu druhej strany
  - addrlen ukazuje na premennú obsahujúcu dĺžku adresy (po návrate udáva skutočnú dĺžku)
  - vráti nový deskriptor alebo -1 (chyba)
  - pôvodný deskriptor zostáva ďalej počúvať

accept - príklad

```
int fd, newfd;
struct sockaddr_in me, peer;
socklen_t peerlen;

fd = socket(PF_INET, SOCK_STREAM, 0);

me.sin_family = AF_INET; me.sin_addr.s_addr = INADDR_ANY;
me.sin_port = htons(cislo_portu);
bind(fd, &me, sizeof(me));
listen(fd, 5);

while (mam_bezat)
{
  peerlen = sizeof(peer);
  newfd = accept(fd, &peer, &peerlen);
  komunikuj(newfd, &peer);
}
close(fd);
```

connect

- `int  connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)`
  - sys/types.h, sys/socket.h
  - pri SOCK_STREAM vytvorí spojenie na zadanú adresu
  - pri SOCK_DGRAM nastaví adresu druhej strany
  - vráti 0 (ok) alebo -1 (chyba)

```
struct sockaddr_in dst; int fd;
dst.sin_family = AF_INET;
dst.sin_port = htons(cislo_portu);
dst.sin_addr.s_addr = inet_addr("127.0.0.1");

fd=socket(PF_INET, SOCK_STREAM, 0);
connect(fd, &dst, sizeof(dst));
```

shutdown

- `int shutdown(int s, int how)`
  - sys/socket.h
  - požiada o ukončenie spojenia (pre SOCK_STREAM)
  - how
    - SHUT_RD – už nebudeme čítať
    - SHUT_WR – už nebudeme písať
    - SHUT_RDWR – oboje
  - zvyčajne sa použije SHUT_WR, a čaká sa na ukončenie z druhej strany (prečítanie 0 B) a následne sa zavolá close.

recv, recvfrom, read

- `ssize_t recv(int s, void *buf, size_t len, int flags)`
- `ssize_t  recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)`
  - sys/types.h, sys/socket.h
  - čítajú z s max. len B do buf, vrátia počet prečítaných alebo -1 (0 znamená koniec spojenia)
  - recvfrom vracia aj informácie o adrese druhej strany (viď accept)

recv, recvfrom, read

- flags
  - MSG_PEEK
    - prečíta, ale aj ponechá na vstupe
  - MSG_DONTWAIT
    - nečaká na dáta
    - ak by inak čakal, vráti -1 a nastaví errno na EAGAIN
- read (viď vstup zo súborov)
  - dá sa použiť aj na socket, funguje analogicky ako recv s flags == 0

send, sendto, write

- `ssize_t send(int s, const void *msg, size_t len, int flags)`
- `ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)`
  - zapíše max. len B z msg do s
  - sendto pre SOCK_DGRAM obsahuje aj adresu druhej strany, send použije adresu nastavenú pomoco connect
  - vrátia počet zapísaných B alebo -1 (chyba)

send, sendto, write

- flags
  - MSG_DONTWAIT
    - nečaká na možnosť odoslať dáta
    - ak by musel čakať, vráti -1 a errno nastaví na EAGAIN
  - MSG_NOSIGNAL
    - zablokuje vznik signálu SIGPIPE
- write (viď výstup do súborov)
  - môže sa použiť aj na socket, funguje analogicky ako send s flags == 0
- Pri zápise môže vzniknúť signál SIGPIPE

Vstup z terminálu

- štandardné čítanie z terminálu
  - bufferované po riadkoch
  - interpretujú sa niektoré špeciálne znaky
  - automaticky sa robí „echo“
- niekedy to nemusí byť vhodné
  - potrebujeme reagovať na jednotlivé znaky
  - potrebujeme detekovať začiatok písania (select)
  - nechceme zobrazovať vstup (heslá)

Nastavenie terminálu

- `int tcgetattr(int fd, struct termios *termios_p)`
- `int tcsetattr(int fd, int optional_actions, struct termios *termios_p)`
  - termios.h, unistd.h
  - man termios
  - tcgetattr získa, tcsetattr nastaví atribúty terminálu
  - optional_actions
    - TCSANOW – nastaví hneď
    - TCSADRAIN – nastaví po odoslaní výstupu
    - TCSAFLUSH – ako TCSADRAIN a navyše ignoruje prijatý a neprečítaný vstup

Nastavenie terminálu

- struct termios obsahuje
  - tcflag_t c_iflag; vstupné atribúty
  - tcflag_t c_oflag; výstupné atribúty
  - tcflag_t c_cflag; riadiace atribúty
  - tcflag_t c_lflag; lokálne atribúty
  - cc_t c_cc[NCCS]; riadiace znaky
    - VINTR Ctrl+C výskyt vyvolá signál SIGINT
    - VQUIT Ctrl+\ výskyt vyvolá signál SIGQUIT
    - VERASE DEL,BS výskyt vymaže predch. znak
    - VKILL Ctrl+U vymaže celý riadok
    - VEOF Ctr+D signalizuje koniec vstupu
    - VSUSP Ctrl+Z výskyt vyvolá signál SIGSTP

Nastavenie terminálu

- vybrané lokálne atribúty
  - ISIG
    - keď sa vo vstupe vyskytne znak definovaný pre generovanie signálu, znak sa odstráni zo vstupu a pošle sa príslušný signál
  - ICANON
    - povoľuje kanonický mód (štandardne povolený), v ktorom sa vstup bufferuje po riadkoch a interpretujú sa znaky na mazanie znaku/riadku a znak pre koniec vstupu (a niekoľko ďalších)
  - ECHO
    - zapína „automatické echo“, t.j. prijatý znak sa automaticky posiela na výstup

Nastavenie terminálu

- príklad – vypnutie kanonického módu

```
struct termios termattr;

if (tcgetattr(0, &termattr) < 0)
{
  perror("tcgetattr");
  exit(1);
}

termattr.c_lflag &= ~ICANON;

if (tcsetattr(0, TCSANOW, &termattr) < 0)
{
  perror("tcsetattr");
  exit(1);
}
```

Nastavenie terminálu

- príklad – zapnutie kanonického módu

```
struct termios termattr;

if (tcgetattr(0, &termattr) < 0)
{
  perror("tcgetattr");
  exit(1);
}

termattr.c_lflag |= ICANON;

if (tcsetattr(0, TCSANOW, &termattr) < 0)
{
  perror("tcsetattr");
  exit(1);
}
```

Preklad z mien na IP adresy

- `struct hostent *gethostbyname(const char *name)`
  - netdb.h
  - vyhľadá zadané meno v určených databázach a vráti pointer na štruktúru hostent
  - pri chybe vráti NULL a nastaví h_errno:
    - HOST_NOT_FOUND
    - NO_ADDRESS, NO_DATA
    - NO_RECOVERY
    - TRY_AGAIN

Preklad z mien na IP adresy

```
struct hostent {
  char    *h_name;         /* official name of host */
  char    **h_aliases;     /* alias list */
  int     h_addrtype;      /* host address type */
  int     h_length;        /* length of address */
  char    **h_addr_list;   /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

- h_name meno
- h_aliases pole alternatívnych mien, končí NULL
- h_addrtype == AF_INET
- h_length dĺžka adresy
- h_addr_list pole adries, končí NULL

Práca s časom

- `time_t time(time_t *t)`
  - time.h
  - vráti aktuálny čas ako počet sekúnd od 0:00:00 UTC 1.1.1970
- `struct tm *gmtime(const time_t *timep)`
- `struct tm *localtime(const time_t *timep)`
  - vrátia smerník na struct tm, kde je rozpísaný zadaný čas ako UTC, resp. ako lokálny čas

Práca s časom

- `char *asctime(const struct tm *tm)`
- `char *ctime(const time_t *timep)`
  - vrátia textový reťazec popisujúci zadaný čas (v prípade ctime chápaný ako lokálny čas)
- `time_t mktime(struct tm *tm)`
  - vráti zadaný rozpísaný čas (chápaný ako lokálny čas) v tvare „počet sekúnd od 1.1.1970“

Práca s časom

```
struct tm {
    int     tm_sec;         /* seconds */
    int     tm_min;         /* minutes */
    int     tm_hour;        /* hours */
    int     tm_mday;        /* day of the month */
    int     tm_mon;         /* month */
    int     tm_year;        /* year */
    int     tm_wday;        /* day of the week */
    int     tm_yday;        /* day in the year */
    int     tm_isdst;       /* daylight saving time */
};
```

Spustenie programu

- `int execl(const char *path, const char *arg, ...)`
- `int execlp(const char *file, const char *arg, ...)`
  - unistd.h
  - nahradí proces novým procesom zo súboru path resp. file a odovzdá mu argumenty
  - prvý (povinný) argument je „meno programu“
  - posledný argument je NULL