

Sofia verzia 2016

Téma: ÚVOD – Man pages, chybový výstup

Kľúčové slová	UNIX/Linux manual, <názov služby> UNIX/Linux example, Linux documentation project		
Ciele	Zapamätať si:	účel a funkciu man pages (manuálových stránok) v UNIX/Linuxe	
	Porozumieť:	parametrom príkazu man, premennej errno, funkcií perror a strerror	
	Aplikovať :	služby na spracovanie chýb	<ul style="list-style-type: none"> • problémy spojené s analýzou chýb pri kompliacii • používanie služieb štandardného výstupu chybových správ v programoch • problémy týkajúce sa nájdenia informácií ohľadom jednotlivých služieb jadra
	Vyriešiť:		
Odhadovaný čas	35 min		
Scenár	Sofia sa prvý raz stretla s OS UNIX/Linux. Nemá žiadnu predstavu o tom, ako tento systém pracuje, kde má nájsť o ňom potrebné informácie a ako sa zorientovať v možnostiach a službách, ktoré jej ponúka. Zároveň by potrebovala získať skúsenosť v práci so štandardným výstupom chybovým chybových správ, pretože pomocou neho bude schopná nájsť odpovede na množstvo problémov, s ktorými sa stretne pri tvorbe svojich programov.		

POSTUP:

Internet

Prvým zdrojom informácií pre Sofiu o OS UNIX/Linux (v dnešnej dobe skoro pri všetkom) je internet. Otvorí si teda svoj oblúbený internetový prehliadač. Keďže potrebuje nejaké informácie o OS UNIX/Linux, do príslušnej kolónky prehliadača vpíše „unix manual“, alebo „Linux manual“. Z veľkého množstva výsledkov vyhľadávania si postupne vyberie tie, ktoré jej vyhovujú. Postupne, ako sa bude dozvedať o jednotlivých službách v OS UNIX/Linux, môže na internete vyhľadávať informácie týkajúce sa konkrétnej služby tak, že do vyhľadávača vpíše názov tejto služby spolu so slovom unix resp. Linux – napr.: „open() unix“, alebo „open() Linux“.

Man pages

Ďalším zdrojom informácií môžu byť pre Sofiu manuálové stránky (man pages), ktoré sú súčasťou každej distribúcie OS Linux/Unix. Sofia postupne zistí, že väčšina zdrojov na internete o nejakých službách OS UNIX/Linux je kópiou man pages.

Man pages sa rozdeľujú na niekoľko častí. V každej časti sú príkazy/služby, ktoré spolu logicky súvisia. Rozdelenie je nasledovné:

Sofia verzia 2016

Časť	Popis
1	spustiteľné programy a príkazy shellu
2	služby jadra operačného systému
3	služby knižníc operačného systému
4	špeciálne súbory (obyčajne v adresári /dev/)
5	formáty súborov, protokolov a štruktúry jazyka C
6	Hry
7	rôzne (dohody, protokoly, znakové normy, rozvrhnutie súborového systému,...)
8	administrácia systému
9	rutiny jadra operačného systému (nie je to štandardná časť man pages)

 Pre podrobnejšie informácie zadá príkaz `man 1 man`.

O tom, ktorá časť čo zahŕňa, sa Sofia môže dozvedieť z úvodu (intro) každej z nich.

 Prečítať `man 1 intro`, `man 2 intro`, `man 3 intro`, ...

Čo má však Sofia robiť v tom prípade, ak nevie, v ktorej časti man pages sa potrebná služba nachádza? V tom jej môže pomôcť príkaz:

`man -f <názov služby>`

Teda ak Sofia nevie, v ktorej časti man pages sa nachádzajú informácie o službe `open()`, zadá:

`man -f open`

Po zistení čísla časti (pri službe `open()` je to „2“) už len stačí, ak zadá:

`man 2 open - alebo len - man open`

Druhá možnosť je v prípade neznámych služieb trochu riskantná, pretože operačný systém môže poskytovať niekoľko manuálových stránok pre zadanú službu a teda `man <služba>` jej môže vrátiť zlý manuál.

V prípade, ak Sofia nevie, ktorú službu vlastne hľadá, resp. akú službu by mala použiť, môže využiť prepínač „-k“, ktorý vypíše služby operačného systému obsahujúce zvolené slovo aj s stručným popisom. Napríklad by Sofia chcela vedieť, ktoré služby sa týkajú práce s vlastníkom (súboru):

`man -k owner`

Ak si Sofia našla a prečítaла manuálovú stránku, tak potom potrebuje opustiť man pages. Dozvedela sa, že k tomu jej stačí iba stlačenie klávesy „q“.

Sofia verzia 2016

Linux - dokumentačný projekt, knihy a iné zdroje

Jedným z mnohých zdrojov informácií môže byť pre Sofiu aj „Linux - dokumentačný projekt“ (voľne dostupný na internete - <http://www.tldp.org/>). Obsahuje vcelku detailný popis činnosti OS Linux. Zahŕňa však skôr praktické použitie služieb jadra, než vysvetlenie ich syntaxe. O nej sa Sofia môže dozvedieť okrem z vyššie spomínaných zdrojov aj zo špecializovaných publikácií.

Hlavičkové súbory

Sofia môže získať vedomosti o potrebných službách aj z hlavičkových súborov, o ktorých sa dozvedela, že ich je potrebné pripojiť k programu pre správnu funkčnosť služby jadra. Hlavičkové súbory jazyka C sa nachádzajú v adresári Linuxu „/usr/include/“, prípadne „/usr/include/sys/“. Takže, ak sa Sofia bude chcieť niečo dozvedieť o službe `open()`, najprv si zistí (napr. pomocou manuálu), aké hlavičkové súbory potrebuje táto služba a potom si v adresári „/usr/include/“ otvorí potrebný hlavičkový súbor pomocou príkazu „`cat <meno_hlavickoveho_suboru>`“, ktorý zobrazí jeho obsah. Alebo, ešte lepšie, použije svoj oblúbený textový editor na otvorenie tohto hlavičkového súboru.

Zdrojové kódy

OS Linux je Open source, čo znamená, že zdrojové kódy jednotlivých systémových volaní sú voľne prístupné (prezerateľné). To vytvára ďalšiu možnosť pre Sofiu, ako sa oboznámiť s funkčnosťou jednotlivých služieb. Zdrojový kód jadra Linuxu je umiestnený v „/usr/src/linux/“, teda ak by Sofia cítila potrebu hlbšieho pochopenia činností pamäti, procesov alebo zariadení, zdrojové kódy sú jej plne k dispozícii.

Podtéma: Štandardný chybový výstup

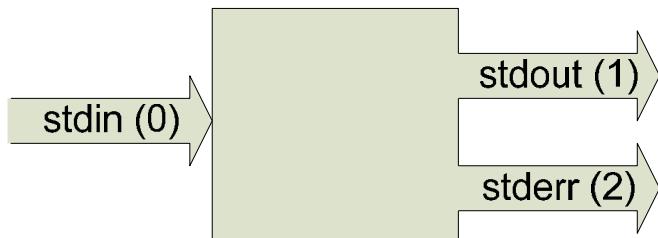
Kľúčové slová	errno unix, perror() unix		
Ciele	Zapamätať si:	význam a funkciu štandardného chybového výstupu	
	Porozumieť:	koncepciu chybového výstupu, syntaxi príkazov a služieb	
	Aplikovať:	príkazy a služby	
	Vyriešiť:	odchytenie a spracovanie chybových stavov v programoch	
Odhadovaný čas	15 min		
Scenár	Sofia už ovláda syntax a parametre jednotlivých služieb jadra a napísala si program. Po kompliacii a spustení jej program nefunguje tak, ako predpokladala. Sofia potrebuje zistiť, o akú chybu ide a ako ju má odchýtiť.		

KRÁTKY ÚVOD:

Štandardné prúdy

Sofia sa dozvedela, že Systém Linux/Unix obsahuje tzv. **štandardné prúdy**, ktoré predstavujú vstupné a výstupné kanály medzi počítačovým programom a jeho okolím. Dočítala sa, že existujú tri vstupno/výstupné kanály:

- štandardný vstup **stdin**
 - štandardný výstup **stdout**
 - štandardný chybový výstup **stderr**
- } **man stdin**



STDIN:

Predstavuje štandardný vstupný kanál, z ktorého programy zvyčajne čítajú dátu z klávesnice. Tento štandardný vstup, podobne ako aj výstup a chybový výstup, môžeme považovať za súbor. Všetkým súborom, s ktorými sa pracuje, jadro systému prideľuje špeciálne malé celé nezáporné číslo – **deskriptor**¹. Hodnota deskriptora pre štandardný vstup je ”0“.

STDOUT:

¹Deskriptor je odkaz na štruktúry (tabuľka deskriptorov) v jadre systému, pomocou ktorého sa bude k súboru pristupovať iba pomocou systémových volaní. Všetky deskriptory, ktoré proces (nás program) vlastní, buď zdedil od svojich rodičov alebo tieto deskriptory získal niektorým systémovým volaním.

Sofia verzia 2016

Do štandardného výstupného kanálu programy posielajú (zapisujú) potrebné dátu. Tie sa implicitne zobrazia na štandardnom výstupnom zariadení (monitor). Jeho deskriptor má hodnotu “1”.

STDERR:

Prostredníctvom štandardného chybového výstupu programy vypisujú na výstupné zariadenie chybové správy. Jeho deskriptor má hodnotu “2”.

Zlyhanie volania systémovej služby

Každá služba jadra vracia návratovú hodnotu, ktorá určuje, či služba bola vykonaná korektne, alebo pri spracovaní služby sa vyskytla chyba. Výskyt chyby sa signalizuje špeciálnou návratovou hodnotou služby (spravidla hodnota -1). Bližšiu špecifikáciu typu chyby môžeme nájsť v globálnej premennej `errno`.

Premenná `errno`:

- využíva hlavičkový súbor `errno.h`
- `errno` je typu int a je to globálna premenná nastavená na hodnotu 0
- ak volaná služba jadra sa vykoná korektne tak hodnota premennej `errno` je 0
- ak volaná služba jadra sa nevykoná korektne, tak nastaví hodnotu typu chyby; ak sa vyskytla chyba, nemôže mať hodnotu 0
- podrobný zoznam chýb je v **man 3 errno**

Chybové hodnoty sú celé čísla, ktoré sú definované v hlavičkovom súbore `errno.h`. Tieto hodnoty sú štandardne pomenované symbolickými konštantami tvorenými veľkými písmenami, vždy začínajúcimi písmenom „E“, napríklad EACCES, EINVAL.

POSTUP:

 Prečítať **man 3 errno**.

► **Príklad** využitia premennej `errno` (program musí obsahovať hlavičkový súbor `errno.h`):

```
if (systemcall() == -1) {  
    int errsv = errno;  
    printf("Chyba pri volani systemcall()\n");  
    if (errsv == ...) { ... ;}  
}
```

Doplňte (podľa **man 3 errno**):

Kód chyby	Význam
EACCES	Prístup zamietnutý
ENOENT	
ENOTDIR	
EINVAL	

Funkcia perror():

Syntax:

```
#include <stdio.h>
void perror(const char *s);
```

Funkcia perror() na základe hodnoty errno generuje opis chyby posledného systémového volania priamo na štandardný chybový výstup (stderr). Argumentom funkcie perror() je reťazec, ktorý sa zobrazí pred samotným opisom chyby. V praxi sa ako parameter používa názov systémovej služby doplnený o nejaký jednoznačný reťazec, aby mal programátor prehľad, kde chyba vznikla.

 Podrobnejšie v **man 3 perror**.

► **Príklad** využitia funkcie perror():

```
if (sluzba() == -1) {
    perror("sluzba()");
}
```

Funkcia strerror():

Syntax:

```
#include <string.h>
char *strerror(int errnum);
```

Funkcia strerror() vracia reťazec opisujúci číslo chyby, ktoré je jej argumentom. Hlavičkový súbor string.h je potrebný pri použití funkcie strerror().

► **Príklad** otvorenia súboru a vypísania chybového hlásenia v prípade vyskytnutia sa chyby. V príklade sme použili službu open(), s ktorou sa Sofia detailnejšie zoznámi na niektorom z ďalších cvičení. Pre tento príklad jej stačí vedieť, že touto službou otvára súbor.

Program 1

```
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

int main()
{
    int fd; //premenna pre ulozenie navratovej hodnoty
    fd = open("subor0", O_RDONLY); //otvorenie suboru
    if(fd == -1) //odchyt chyby pri neotvorení suboru
    {
        printf("Vypis premennej errno: %d\n", errno);
        printf("Vypis pomocou služby strerror:%s\n", strerror(errno));
    }
    perror("Vypis funkcie perror pre funkciu open");
    return 0;
}
```

 Podrobnejšie v `man 3 strerror`.

Doplňte:

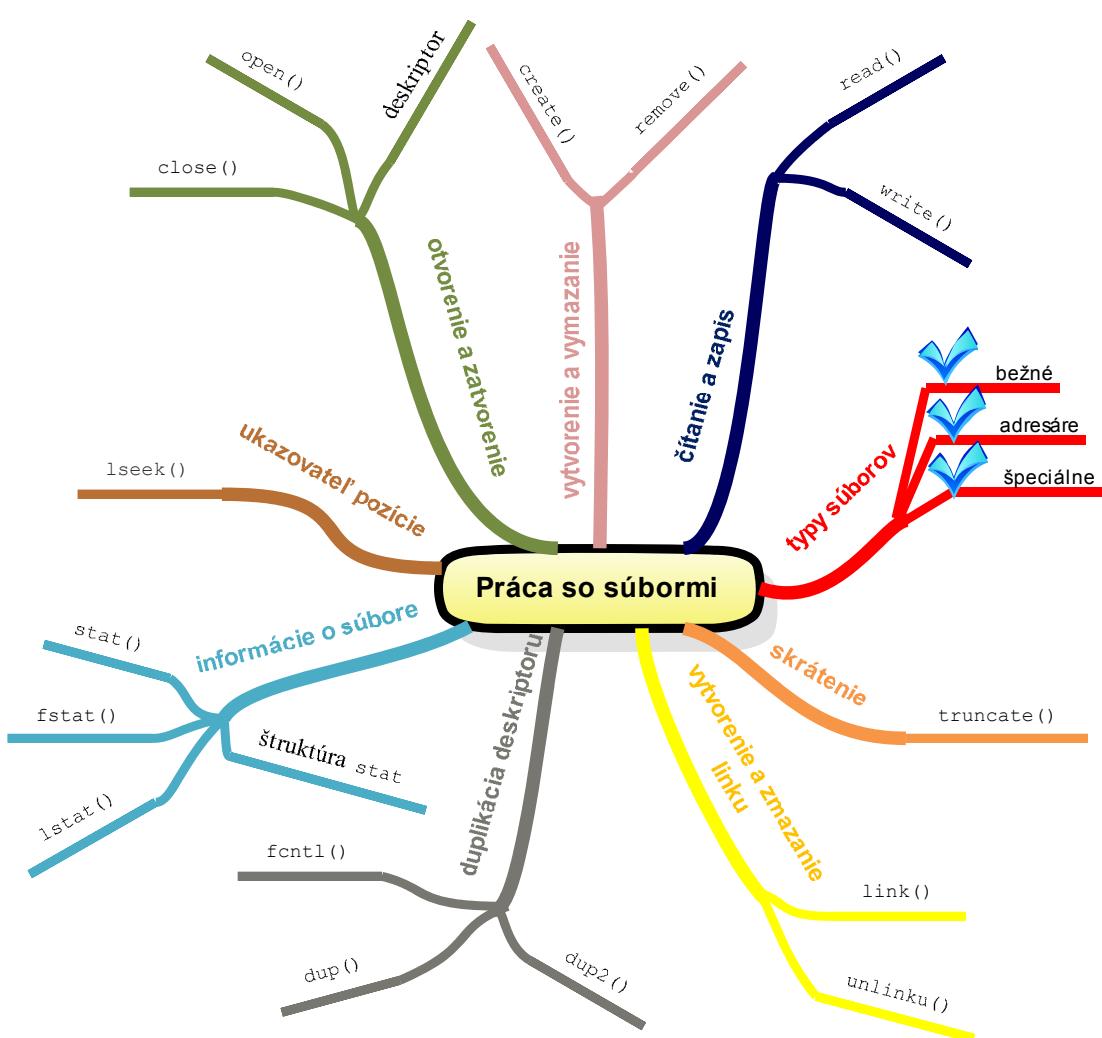
Pri používaní systémových služieb sa bežne môže stať, že služba neskončí správne a počas jej vykonávania nastane chyba. To indikuje návratová hodnota služby, ktorá je v takomto prípade väčšinou _____ (doplňte hodnotu z manuálových stránok).

Zistiť presne, o aký typ chyby ide, Sofia môže pomocou systémovej premennej _____.

Premenná `errno` je typu int a vyžaduje hlavičkový súbor _____.

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vytvorte súbor s názvom `subor0`, ktorý sa bude nachádzať v adresári, kde sa nachádza predchádzajúci Program 1 na otvorenie súboru. Aký bude výpis programu uvedeného v tejto podtémě ak súbor `subor0` bude existovať a aká bude hodnota premennej `errno`?



Téma: Práca so súbormi v OS UNIX/Linux

Kľúčové slová	Súborový systém OS UNIX/Linux, súbory, práca so súbormi, i-uzol				
Ciele	Zapamätať si:	základné služby jadra pre prácu so súbormi			
	Porozumieť:	parametrom služieb a súvislostiam medzi službami			
	Aplikovať:	služby jadra pre : <ul style="list-style-type: none">• otvorenie, zápis, čítanie zo súboru• získanie informácií o súbore• nastavenie prístupových práv• vymazanie súboru			
	Vediet:	využiť získané skúsenosti pri tvorbe programov			
Odhadovaný čas	105 minút				
Scenár	Sofia už vie pracovať s manuálovými stránkami a vie už odchytíť chybové návratové hodnoty služieb jadra svojho programu. Sofia potrebuje pre základnú prácu v OS UNIX/Linux a pre tvorbu programov základné poznatky o službách jadra pre prácu so súbormi.				

POSTUP:

Táto kapitola sa zameriava na:

- **Systémové volania:**
 - `read()`, `write()`
 - `open()`, `close()`
 - `lseek()`
 - `dup()`, `dup2()`
 - `stat()`, `fstat()`, `lstat()`
 - `link()`, `unlink()`, `remove()`
 - `truncate()`

Podtéma: Služby jadra – `read()` a `write()`

Kľúčové slova	<code>read()</code> , <code>write()</code> , deskriptor	
Ciele	Zapamätať si:	syntax služieb - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://unixhelp.ed.ac.uk/ http://linux.about.com/od/commands/l/blcmdl2_read.htm http://linux.about.com/library/cmd/blcmdl2_write.htm
	Porozumieť:	<ul style="list-style-type: none"> • argumentom služieb • návratovým hodnotám • pojmu kanál • významu súvisiacich služieb (<code>open()</code>, <code>create()</code>, <code>dup()</code>, <code>lseek()</code>) • chybovým hláseniam
	Aplikovať:	služby <code>read()</code> a <code>write()</code> pri práci so súbormi
	Viedet:	využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	15 minút	
Scenár	Sofia má za úlohu načítať a upraviť súbor v jej adresári. Zistila, že pre vyriešenie tejto úlohy jej pomôžu služby <code>read()</code> a <code>write()</code> . Aby však ich mohla využiť, potrebuje sa ich naučiť používať.	

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra pre vstup/výstup:

Všetky vstupy a výstupy sa realizujú prostredníctvom funkcií: `read()` a `write()`:

Syntax:

```
#include <unistd.h>
read(int fd, char *buf, size_t count);
write(int fd, const char *buf, size_t count);
```

Sémantika:

- `read()` načíta *count* bajtov z kanálu *fd* do vyrovnávacej pamäte *buf* a vráti počet načítaných bajtov; vráti 0, keď už predtým dosiahla koniec súboru alebo -1 pri chybe
- `write()` zapíše *count* bajtov do kanálu *fd* z vyrovnávacej pamäte *buf* a vráti počet zapísaných bajtov, alebo -1 pri chybe

KROK2 - pochopiť parametre služieb:

Pre obidve služby je prvým argumentom deskriptor súboru². Druhý argument je buffer³, do ktorého budú dátá zapisované, alebo budú z neho čítané. Tretí argument udáva počet slabík, ktoré budú prenesené.

KROK3 – aplikovanie služieb v programe:

1. program - Nasledujúci program otvorí súbor s názvom *subor1*, načíta z neho 20 znakov, vypíše ich na štandardný výstup a súbor zatvorí.

```
#include <fcntl.h>

int main(int argc, char **argv)
{
    int des;                                //deskriptor otvorenego suboru
    int i;                                    //buffer, do ktoreho nacitavame
    char buf;                                 //otvorime subor

    des=open("subor1",O_RDONLY);             //nacitame z neho 20 znakov
    for(i=0;i<20;i++)
    {
        read(des,&buf,1);                   //a vypiseme na standardny vystup
        write(1,&buf,1);
    }
    close(des);                            //subor znova zatvorime
    return 0;
}
```

Príklad skompilujeme `gcc sub1.c` a spustime `./a.out`.

Každé volanie služby `read()` vráti počet bytov, ktoré boli skutočne systémom prenesené. Ak je počet prenesených bajtov nižší, ako je zadaná (požadovaná) hodnota uvedená vo volaní služby, je to príznakom konca súboru. Pri zápisе službou `write()` je vrátená hodnota rovná počtu skutočne zapísaných bajtov. Ak je hodnota rôzna od zadanej hodnoty uvedenej vo volaní služby, je to znamenie chyby (zvyčajne presiahnutie určitých nastavených limitov).

2. program - Napíšeme jednoduchý program, ktorý kopíruje dátá zo štandardného vstupu na štandardný výstup. Program sa ukončí stlačením kombinácie kláves `Ctrl+C`.

```
main () /*kopíruje vstup na výstup*/
{
    char buf;
    int n;
    while ((n = read(0,&buf,1)) > 0) //citanie zo standardneho vstupu
        write(1,&buf,1);           //zapisanie na standardny vystup
    return 0;
}
```

Príklad kompliacie `gcc -o sub2 sub2.c` a spustenie programu `./sub2`

² Deskriptor je možné získať prostredníctvom služby `open()`.

³ Buffer = vyrovnávacia pamäť.

Sofia verzia 2016

 Podrobnejšie informácie o službách `write()` a `read()` si môžete pozrieť v `man 2 read` a `man 2 write`.

Sofia verzia 2016

Podtéma: Služby jadra – open() a close()

Kľúčové slova	open(), close(), flags, inode.h	
	<p>Zapamätať si:</p> <p>syntax služieb - prečítať si manuálové stránky v Unix/Linuixe, Linux dokumentačný projekt, zdroje na internete: http://unixhelp.ed.ac.uk/ http://www.ee.surrey.ac.uk/Teaching/Unix/ http://linux.about.com/od/commands/l/blcmdl2_open.htm http://linux.about.com/library/cmd/blcmdl2_close.htm</p>	
Ciele	<ul style="list-style-type: none"> • parametrom flags a mode • významu súvisiacich služieb (create(), lseek(), read(), umask()) • chybovým hláseniam <ul style="list-style-type: none"> • služby open() a close() pri práci so súbormi • flagy podľa aktuálnych potrieb 	
	Viedet:	využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	20 minút	
Scenár	<p>Sofia pri riešení svojej úlohy zistila, že pred prácou so súborom potrebuje daný súbor otvoriť. Použije na to službu open(), ale pre efektívnu prácu so súborom potrebuje sa naučiť pracovať s tzv. flagmi. Keď ukončí prácu so súborom, tak ho zatvorí pomocou služby close().</p>	

POSTUP:

KROK1- naučiť sa syntax a sémantiku služby jadra open():

Pomocou služba jadra open() získame deskriptor súboru pre čítanie alebo zápis, resp. môžeme vytvoriť nový súbor.

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(char *pathname, int flags, mode_t mode)
```

Sémantika:

- open() vracia - deskriptor súboru alebo -1, pri chybe

KROK2 - pochopíť parametre služby:

Služba open() otvorí súbor uvedený v parametri *pathname* pre čítanie alebo zápis, podľa toho, ako je to špecifikované argumentom *flags* a akú návratovú hodnotu vráti deskriptor pre otvorený súbor.

KROK3 - pochopiť účel parametrov *flags* a *mode*:

Parameter *flags* môže byť (okrem iného) špecifikovaný jednou alebo kombináciou viacerých z nasledujúcich položiek:

O_RDONLY	Otvoriť súbor len pre čítanie
O_WRONLY	Otvoriť súbor len pre zápis
O_RDWR	Otvoriť súbor pre zápis aj čítanie
O_APPEND	Doplnenie pre každý zápis
O_CREAT	Vytvoriť súbor ak neexistuje
O_TRUNC	Skrátiť veľkosť súboru na 0
O_EXCL	Chyba, ak súbor už existuje

Ak otváraný súbor ešte neexistuje, je možné jeho prístupové práva (tejto problematike sa budeme venovať neskôr) nastaviť parametrom *mode*. Parameter *mode* je tvorený jednou alebo kombináciou viacerých z nasledujúcich položiek, definovaných v *sys/inode.h*:

IREAD	00400	čítanie pre majiteľa
IWRITE	00200	zápis pre majiteľa
IEXEC	00100	vykonávanie pre majiteľa
	00070	čítanie, zápis a vykonávanie pre skupinu
	00007	čítanie, zápis a vykonávanie pre ostatných

KROK4 - naučiť sa syntax a sémantiku služby jadra *close()* :

Táto funkcia ukončí prácu s otvoreným súborom.

Syntax:

```
#include <unistd.h>
int close (int filedes);
```

Sémantika:

- *close()* vracia: 0 keď OK alebo -1, pri chybe

KROK5 – pochopiť parametre služby:

Argument *filedes* je deskriptor otvoreného súboru. Uzavretie súboru spôsobí vyprázdenie vyrovnávacích pamäti a taktiež odomknutie všetkých zámkov naložených na súbor.

 Podrobnejšie informácie o službách *open()* a *close()* si môžete pozrieť v **man 2 open** a **man 2 close**.

KROK6 – aplikovanie služieb v programe:

1. program -Vytvoríme program, ktorý otvorí súbor len na čítanie. Názov súboru je zadaný z príkazového riadku.

```
#include <fcntl.h>
#include <stdio.h>

int main (int argc, char* argv[])
{
    const char* const filename=argv[1];
    //meno suboru z prikazoveho riadku
    int fd = open (filename, O_RDONLY);           //otvorenie suboru
    printf("file descriptor %d meno suboru %s\n",fd ,filename);
    close(fd);                                     //zatvorenie suboru
    return 0;
}
```

Úloha – modifikácia programu

Vytvorte súbor s názvom *subor1* a zapíšte do neho ľubovoľný text. Ak súbor existuje, otvorte ho na zápis a zapisovaný text pridajte na koniec súboru. Vytvorený súbor bude mať povolený zápis a čítanie jeho majiteľom.

2. program – Riešenie zadanej úlohy.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int des;                                // premenna na ulozenie deskriptora
    des = open("subor1",O_CREAT|O_APPEND|O_WRONLY,S_IRUSR|S_IWUSR); // vytvorime / otvorime súbor
    printf("file descriptor %d suboru \n",des);
    write(des, "Dvadsatznakovy text\n",20); // Zapiseme do súboru
    close(des);                            // zatvorime súbor
    return 0;
}
```

A čo ak sa súbor nepodarí otvoriť? V takom prípade volanie `open()` vráti „-1“ a v globálnej premennej `errno` sa nastaví číslo chyby. Takže otvorenie súboru spolu s ošetrením chyby môže vyzerat takto:

```
if((des=open("subor1",O_RDONLY)) == -1){ //otvorime súbor
    perror("Otovrenie suboru SUBOR1");
}
```

Sofia verzia 2016

Podtéma: Služba jadra – lseek()

Kľúčové slova	lseek(), SEEK_SET, SEEK_END, SEEK_CUR		
Ciele	Zapamätať si:	syntax služby - prečítať si manuálové stránky v Unix/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://linux.about.com/library/cmd/blcmdl2_lseek.htm	
	Porozumieť:	<ul style="list-style-type: none">• pojmom:<ul style="list-style-type: none">- priamy prístup- ukazovateľ súboru• funkciám jednotlivých parametrov• chybovým hláseniam	
	Aplikovať:	službu lseek() pri práci so súbormi	
	Viedť:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	15 minút		
Scenár	Sofia napreduje v riešení svojej úlohy a chce sa naučiť efektívnejšie pracovať so súbormi v OS UNIX/LINUX. Zistila, že k tomu by jej mohla pomôcť služba jadra lseek(), ktorou môže prechádzať súbor a nastavovať pozíciu v súbore, preto sa ju chce naučiť používať.		

POSTUP:

Ukazovateľ aktuálnej pozície v súbore je miesto v súbore (konkrétny bajt), na ktorom sa bude vykonávať nasledujúca operácia `read()` alebo `write()`

KROK1 – naučiť sa syntax a sémantiku služby jadra lseek():

Bežný spôsob práce so súbormi je sekvenčný (ukazovateľ aktuálnej pozície v súbore sa priebežne zvyšuje). V prípade potreby je možné súbory čítať alebo do nich zapisovať na ľubovoľnej pozícii. Služba jadra lseek() umožňuje posunúť sa na ľubovoľné miesto v súbore bez toho, aby bolo nutné súbor čítať, alebo do neho zapisovať.

Syntax:

```
#include <sys/types.h>
#include <unistd.h>
long lseek (int fd, long offset, int origin);
```

Sémantika:

- lseek() vracia: nový offset keď je všetko OK alebo -1, pri chybe

KROK2 - pochopiť parametre služby:

Služba lseek() nastaví pozíciu v súbore (určeného deskriptorom súboru `fd`), na miesto určené posunutím `offset`, vzhľadom na pozíciu určenú argumentom `origin` môžeme špecifikovať nasledujúce pozície:

SEEK_SET	pozícia kurzoru s hodnotou začiatku súboru
SEEK_CUR	aktuálna pozícia kurzoru v súbore
SEEK_END	pozícia kurzoru v súbore s hodnotou konca súboru

Sofia verzia 2016

Nasledujúce čítanie alebo zápis do súboru sa uskutoční na tejto pozícii. Argument `offset` je typu long, argumenty `fd` a `origin` sú typu int. Argument `origin` môže mať hodnoty 0, 1 a 2, ktoré určujú, že posunutie je merané od počiatku, od práve aktuálnej pozície v súbore alebo od konca súboru.

Pozícia je typu long a preto je nevyhnutné v prípade uvedenia konštanty ju špecifikovať ako konštantu typu long (L za hodnotou konštanty) alebo pretypovať.

 Pre podrobnejšie informácie zadaj príkaz `man 2 lseek`.

Napr. nastavenie na koniec súboru (append) sa uskutoční nasledovne :

```
lseek(fd, 0L, SEEK_END);
```

Nastavenie na začiatok súboru (rewind):

```
lseek(fd, 0L, SEEK_SET);
```

Poznamenajme, že argument 0L je možné písť v tvare (long)0.

KROK3 – aplikovanie služby v programe:

Nasledujúce príklady ukazujú použitie služby jadra `lseek()`.

1. program – Zisti dĺžku súboru, ktorého meno je zadané z klávesnice a vypíše ju na štandardný výstup.

```
#include <stdio.h>
#include <fcntl.h>

int main(void)
{
    int handle;
    char meno[80];
    long l;

    printf("\nzadaj meno suboru :");
    scanf("%s", meno);           //nacitanie nazvu subora z klavesnice
                                //otvorenie suboru iba na citanie
    if ((handle = open(meno, O_RDONLY)) == -1){
        perror("open()");
        return(handle);
    }                           //nastavenie pozicie na koniec suboru
    if ((l = lseek(handle, 0L, SEEK_END)) == -1){
        perror("lseek()");
        close(handle);           //chyba pri nastavovani pozicie
                                //uzatvorenie suboru
    }
    else {
        printf("Subor <%s> je dlhy %ld bajtov.\n", meno, l);
        }                         //vypisanie dlzky suboru
    close(handle);             //uzatvorenie suboru
    return(0);
}
```

2. program - Nasledujúci príklad vypíše znak nachádzajúci sa na zadanej pozícii v súbore. Názov súboru a pozícia v súbore sú zadané z príkazového riadku.

```
#include <stdio.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
    int fd;
    off_t offset;
    char *name,
    buf[5];
    long poz;
    if (argc != 3){
        printf("Chybny pocet argumentov\n");
    }
    else{
        name = argv[1];
        offset = atoi(argv[2]);
        if ((fd=open(name, O_RDONLY)) == -1){
            perror("open()");
        }
        if((poz=lseek(fd, 0L, SEEK_END))<offset){
            printf("Subor neobsahuje tolko znakov\n");
        }
        if(lseek(fd, offset, SEEK_SET)==-1){
            perror("lseek()");
        }
        else {
            read(fd, buf, 1);
            printf("Vypis znaku zo suboru:%c\n", buf[0]);
        }
    }
    return (0);
}
```

Sofia verzia 2016

Podtéma: Služby jadra – dup(), dup2()

Kľúčové slova	dup(), dup2(), deskriptor		
Ciele	Zapamätať si:	syntax služieb - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://linux.about.com/library/cmd/blcmdl2_dup.htm	
	Porozumieť:	<ul style="list-style-type: none">účelu jednotlivých parametrovchybovým hláseniam	
	Aplikovať:	služby dup() a dup2() pri práci s deskriptormi	
	Viedet:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	10 minút		
Scenár	Sofia sa chce naučiť použiť obyčajný súbor ako štandardný vstup alebo výstup. Zistila, že pre riešenie tohto problému jej pomôžu služby jadra dup() a dup2(). Tieto služby slúžia na duplikáciu deskriptora otvoreného súboru.		

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služieb dup() a dup2():

Ako štandardný vstup alebo výstup môže slúžiť aj obyčajný súbor. Na tento účel môžeme využiť službu jadra dup(). Základný princíp činnosti je v tom, že dup() zduplicuje deskriptor, ktorý dostane ako argument a duplikát uloží na prvú voľnú pozíciu v tabuľke deskriptorov.

Syntax:

```
#include <unistd.h>
int dup (int oldfd);
int dup2 (int oldfd, int newfd);
```

Sémantika:

- Vracia: nový deskriptor alebo -1, pri chybe

KROK2 – pochopíť parametre služieb:

Argument *oldfd* je deskriptor otvoreného súboru pre službu dup() aj pre službu dup2(). Argumentom *newfd* služba dup2() specifikuje hodnotu nového deskriptora. Ak je *newfd* momentálne otvorený, je najprv zatvorený. Ak sa *oldfd* rovná *newfd*, potom dup2() vráti *newfd* bez jeho zatvorenia. Potom je nový deskriptor vrátený ako hodnota služieb zdieľajúcich rovnaké miesto v tabuľke súborov, ako argument *oldfd*.



Pre podrobnejšie informácie zadaj príkaz **man 2 dup** a **man 2 dup2**.

KROK 3 – aplikovanie služieb v programe:

Zduplikovanie deskriptoru neznamená, že sa znova otvorí ten istý súbor. Súbor ostane otvorený iba raz. Asi najdôležitejším dôsledkom je, že zostane iba jediný ukazovateľ na aktuálnu pozíciu v súbore. Nasledujúce dva príklady by to mohli trochu objasniť:

```
#include <fcntl.h>
#include <sys/stat.h>
int main(int argc, char **argv)
{
    int des1;
    int des2;
                                //vytvorime / otvorime subor
des1=open("subor1" , O_CREAT | O_WRONLY , S_IRUSR | S_IWUSR);

des2=open("subor1" , O_WRONLY);      //druhykrat ho netreba vytvorit
                                    //staci otvorit
write(des1,"Toto v subore nebude vobec\n",27); //zapiseme do neho
write(des2,"Toto bude v subore len raz\n",27); //a este raz
close(des1);                      //zatvorime subor
close(des2);                      //a znova zatvorime
return 0;
}
```

Výsledok bude, že v súbore bude zapísaný iba druhý text. Kedže pri každom z otvorených súboroch mame nezávislé ukazovatele na pozíciu v súbore a oba po otvorení súboru ukazovali na začiatok súboru, prepísal sa prvý text druhým. Kedže bol súbor dvakrát otvorený, treba ho aj dvakrát zatvoriť. V prípade zduplikovania deskriptora:

```
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    int des1;
    int des2;

    des1=open("subor1" , O_CREAT | O_WRONLY , S_IRUSR | S_IWUSR);
                                //vytvorime / otvorime subor
    des2=dup(des1);          //zduplicujeme deskriptor
                            //zatvorime prvý deskriptor
    write(des1,"Toto bude v subore\n",19); //zapiseme do neho
    write(des2,"Toto tam bude tiez\n",19); //a este raz
    close(des1);             //zatvorime prvý deskriptor
    close(des2);             //a aj druhý deskriptor
    return 0;
}
```

V súbore budú zapísané obidva texty, kedže ukazovateľ na aktuálnu pozíciu v súbore je len jeden a ten sa po prvom zápise posunie. Súbor sa zatvorí, ak sa zatvorí posledný deskriptor na neho.

Sofia verzia 2016

KROK4 – naučiť sa ďalší spôsob duplikácie:

Ďalším spôsobom duplikovania deskriptora súboru je služba `fcntl()`.
V skutočnosti :

```
dup(int filedes);
```

je ekvivalentný s

```
fcntl(int filedes, F_DUPFD, 0);
```

Podobne:

```
dup2(int filedes, int filedes2);
```

je ekvivalentný s

```
close(int filedes2);  
fcntl(int filedes, F_DUPFD, int filedes2);
```

Podtéma: Služby jadra – stat(), fstat(), lstat()

Kľúčové slova	stat(), fstat(), lstat(), i-uzol		
Ciele	Zapamätať si:	syntax služieb - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://linux.about.com/library/cmd/blcmdl2_stat.htm	
	Porozumieť:	<ul style="list-style-type: none"> • štruktúre i-uzla • funkciám jednotlivých parametrov • významu súvisiacich služieb (create(), dup(), open()) • chybovým hláseniam 	
	Aplikovať:	služby stat(), fstat(), lstat() pre získanie informácií o stave súboru	
	Vedieť:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	15 minút		
Scenár	Sofia potrebuje získať informácie o súbore, ktorý je uložený v jej adresári. Zistila, že na to jej poslúžia služby jadra stat(), lstat() alebo fstat(). Teraz Sofia potrebuje rozpoznať účel použitia týchto služieb.		

POSTUP:

KROK1 - naučiť sa syntax a sémantiku služieb jadra stat(), fstat(), lstat():
Tieto služby jadra využívame na získanie informácií o súbore a adresári.

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char * pathname, struct stat * buf);
int fstat (int filedes, struct stat * buf);
int lstat (const char * pathname, struct stat * buf);
```

Sémantika:

- Všetky tri služby vracajú: 0 keď OK alebo -1, pri chybe

KROK2 – pochopíť parametre služieb:

Prvý argument *pathname* alebo *filedes* špecifikuje súbor. Druhý argument je ukazovateľ na dátovú štruktúru, ktorú služba vyplní. Služba jadra stat() získava informácie o súbore podľa špecifikovaného mena súboru, fstat() získava informácie o už otvorenom súbore, lstat() je podobná stat(), ale keď ide o symbolický link⁴, získava informácie o tomto linku, a nie o súbore, na ktorý link ukazuje.

 Pre podrobnejšie informácie zadaj príkaz **man 2 stat**.

⁴ Symbolická linka v podstate predstavuje súbor, v ktorom je zapísané alternatívne meno súboru. Pri "normálnom" používaní sa symbolická linka tvári ako súbor, na ktorý odkazuje.

KROK3 – pochopíť štruktúru stat:

Všetky tieto služby jadra, pri ich úspešnom volaní, vyplňia štruktúru `stat`, ktorej obsah je nasledujúci:

```
struct stat{  
    mode_t  st_mode;      /* typ súboru & prístupové práva */  
    ino_t   st_ino;       /* číslo i-nodu */  
    dev_t   st_dev;       /* číslo zariadenia (file system) */  
    dev_t   st_rdev;      /* číslo zariadenia pre spec. súbory */  
    nlink_t st_nlink;     /* počet odkazov (linkù) */  
    uid_t   st_uid;       /* user ID */  
    gid_t   st_gid;       /* group ID */  
    off_t   st_size;      /* veľkosť v bajtoch */  
    time_t  st_atime;     /* čas posledného prístupu */  
    time_t  st_mtime;     /* čas poslednej modifikácie */  
    time_t  st_ctime;     /* čas poslednej zmeny súboru */  
    long    st_blksize;   /* najlepšia veľkosť I/O bloku */  
    long    st_blocks;    /* počet alokovaných 512B blokov */  
};
```

KROK4 – aplikovanie služieb v programe:

Uvedený príklad zobrazí informácie o type súborov zadaných z príkazového riadku.

```
#include <sys/types.h>  
#include <stdio.h>  
#include <sys/stat.h>  
  
int main(int argc, char *argv[]){  
    int             i;  
    struct stat     buf;  
    char            *ptr;  
  
    for (i = 1; i < argc; i++) {  
        printf("%s: ", argv[i]);                                //vypis  
        if (lstat(argv[i], &buf) < 0) {perror("lstat()"); continue;}  
                                         //určenie typu suboru  
        if      (S_ISREG(buf.st_mode))  ptr = "regular";  
        else if (S_ISDIR(buf.st_mode))  ptr = "directory";  
        else if (S_ISCHR(buf.st_mode))  ptr = "character special";  
        else if (S_ISBLK(buf.st_mode))  ptr = "block special";  
        else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";  
        else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";  
        else if (S_ISSOCK(buf.st_mode)) ptr = "socket";  
        else                            ptr = "*** unknown mode ***";  
        printf("%s\n", ptr);  
    }  
    return(0);  
}
```

Sofia verzia 2016

V položke `st_mode` sú uložené informácie o type súboru. Tieto informácie sú tu uložené ako bitový súčet (OR, čiže operátor `|`) rôznych príznakov.

Typ súboru	makro
regulárny súbor	<code>S_ISREG</code>
adresár	<code>S_ISDIR</code>
Obycajny subor	<code>S_IFREG</code>
znakový špeciálny súbor	<code>S_ISCHR</code>
blokový špeciálny súbor	<code>S_ISBLK</code>
FIFO	<code>S_ISFIFO</code>
symbolický link	<code>S_ISLNK</code>
soket	<code>S_ISSOCK</code>

Sofia verzia 2016

Podtéma: Služby jadra link(), unlink() a remove()

Kľúčové slova	link(), unlink(), remove()	
Ciele	Zapamätať si:	syntax služieb - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://linux.about.com/library/cmd/blcmdl2_link.htm http://linux.about.com/library/cmd/blcmdl2_unlink.htm http://linux.about.com/od/commands/l/blcmdl3_remove.htm
	Porozumieť:	<ul style="list-style-type: none">• pojmu link• funkciám jednotlivých parametrov• chybovým hláseniam
	Naučiť sa:	služby link(), unlink(), remove() pri práci so súbormi a adresármami
	Viedet:	využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	10 minút	
Scenár	Aby Sofia mohla vytvoriť odkaz na súbor a vymazať súbor. Musí porozumieť pojmu link a naučiť sa používať služby link(), unlink() a remove().	

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služieb jadra link() a unlink():

Na jeden fyzický súbor (t.j. na rovnaký i-node) môže odkazovať viac adresárových položiek. Tieto sa vytvoria pomocou tzv. pevného linku službou jadra link(). Pre zrušenie odkazov slúži služba jadra unlink().

Syntax:

```
#include <unistd.h>
int link (const char *existingpath, const char newpath);
int unlink (const char *pathname);
```

Sémantika:

- link() a unlink() vracia: 0 keď OK alebo -1, pri chybe

KROK2 – pochopit’ parametre služieb:

Služba jadra link() vytvorí novú položku adresára newpath, ktorá odkazuje na existujúcu položku existingpath. Iba superužívateľ môže vykonať link na adresár. Ak už newpath existuje, je vrátená chyba. Vytvorí sa len posledná časť newpath, zbytok cesty už musí existovať.

Služba jadra unlink() odstráni položku adresára a dekrementuje linku podľa pathname. Ak existujú na súbor aj iný link, dátu v súbore ostanú prístupné cez ostatné linky. Ak sa vyskytne pri volaní chyba, súbor sa nezmení. K odstráneniu súboru však musíme mať práva zápisu a vykonávania v adresári, kde sa daný súbor nachádza.

 Pre podrobnejšie informácie zadaj príkaz `man 2 link` a `man 2 unlink`.

KROK3 – aplikovanie služieb v programe:

1. program – Nasledujúci program využíva služby jadra `link()` a `unlink()` pre premenovanie súboru. Program najprv vytvorí synonymum medzi pôvodným a novým súborom a potom pôvodný súbor zruší prostredníctvom služby jadra `unlink()`.

```
#include <stdio.h>

main(int argc, char **argv)
{
    printf("staremeno:%s novemeno:%s\n", argv[1], argv[2]);
    if (argc > 3 || argc < 3){           //kontrola počtu argumentov
        printf("Chybny pocet argumentov!\n");
    }
    else if (link(argv[1], argv[2]) == -1){ //vytvorenie linku
        perror("link()");
    }
    else if (unlink(argv[1]) == -1){       //existujúci subor
        perror("unlink()");
    }
    printf("done\n");
    return(0);
}
```

2. program - Nasledujúci príklad otvorí súbor a potom ho „odpojí“. Program pred ukončením počká 15 sekúnd.

```
#include <stdio.h>
#include <fcntl.h>

int des;
int main(void)
{
    if((des=open("tempfile", O_RDWR | O_CREAT)) < 0){
        perror("open()");
    }
    if(unlink("tempfile") < 0){           //unlinkneme ho
        perror("unlink()");
    }
    printf("file unlinked\n");
    sleep(15);                           //pockame 15 sec
    close(des);
    printf("done\n");
    return(0);                           //koniec programu
}
```

KROK4 – využitie služby `unlink()`:

Služba `unlink()` je často využívaná programami na to, aby sa uistili, že dočasný súbor (temporary file) nebude ponechaný v pamäti po tom, ako program skončí. Program otvorí/vytvorí súbor volaniami `open()`/`create()` a hneď volá službu `unlink()`. Súbor nie je vymazaný pretože je otvorený. Až po tom, ako proces zatvorí súbor, je súbor vymazaný.

Sofia verzia 2016

Ak je *pathname* symbolický link, `unlink()` odstráni symbolický link, nie súbor, na ktorý link odkazuje. Neexistuje služba na odstránenie súboru odkazovaného symbolickým linkom odovzdaním mena linku.

KROK5 – naučiť sa syntax a sémantiku služby `remove()`:

Odstrániť súbor alebo adresár tiež môžeme funkciou `remove(3)`. Pre súbory je volanie `remove()` identické `unlink()`, pre adresáre je `remove()` identické `rmdir()`.

Syntax:

```
#include <stdio.h>
int remove(const char *pathname);
```

Sémantika:

- Návratové hodnoty: 0 ak OK alebo -1, ak nastala chyba.

Služba jadra `remove()` odstráni súbor, ktorý je špecifikovaný parametrom *pathname*.



Pre podrobnejšie informácie zadaj príkaz `man 3 remove`.

Sofia verzia 2016

Podtéma: Služba jadra – truncate()

Kľúčové slova	turncate()
Ciele	Zapamätať si: syntax služby - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://www.scit.wlv.ac.uk/cgi-bin/mansec?3C+truncate http://unixhelp.ed.ac.uk/CGI/man-cgi?truncate+2 Porozumieť: <ul style="list-style-type: none">• parametrom služby truncate()• chybovým hláseniam Aplikovať: službu truncate() pri práci so súbormi Vedieť: využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	5 minút
Scenár	Sofia má súbor, ktorému potrebuje zmeniť veľkosť. Potrebuje zistiť, akými spôsobmi by to mohla urobiť.

POSTUP:

Niekedy sa môže vyskytnúť situácia, keď potrebujeme skrátiť súbor „odrezaním“ dát z konca súboru, alebo naopak, súbor predĺžiť. Skrátenie obsahu súboru na nulu môžeme vykonať aj flagom `O_TRUNC` služby jadra `open()`, nielen službou `truncate()`. Naopak, služba `truncate()` je ďaleko flexibilnejšia, keďže umožňuje presne definovať novú veľkosť súboru.

KROK1 – naučiť sa syntax a sémantiku služby jadra `truncate()`:

Syntax:

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
```

Sémantika:

- `truncate()` vracia: 0 ak sa proces uskutočnil bez chýb alebo -1, ak nastala chyba.

KROK2 – pochopíť parametre služby:

const char *pathname	- názov súboru, ktorému chcem zmeniť veľkosť
int off_t lenght	- nová dĺžka v bytoch

Ak predchádzajúca veľkosť súboru bola väčšia než `length`, dáta za `length` nebudú prístupné. Ak volanie „predlží“ súbor, dáta medzi starým a novým koncom súboru budú načítané ako 0.

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Pomocou služby `write()` uložte obsah premennej (napr. zložitejšia štruktúrovaná premenná) resp. premenných do súboru a následne ich v ďalšom programe načítajte pomocou služby `read()`. Overte uloženie prvkov štruktúry v súbore.
- Nakopírujte existujúci súbor, ktorého meno je uvedené ako prvý argument príkazového riadku do súboru, ktorého meno je odovzdané programu ako druhý argument.
- Overte činnosť služby `open()` s nasledujúcimi príznakmi (flagmi), resp. ich kombináciami: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_CREAT`, `O_EXCL`, `O_TRUNC`.
 - Otvorte súbor pre zápis a čítanie (ked' neexistuje, nech je vytvorený).
 - Otvorte súbor pre zápis na koniec súboru.
 - Otvorte súbor s prepísaním obsahu.
 - Otvorte súbor bez prepísania obsahu.
- Použite službu `lseek()` na nastavenie novej pozície v súbore a zistenie aktuálnej pozície v súbore.
- Použite službu jadra `dup()` a `dup2()` v spojení so službami `open()`, `create()`, `write()`, `close()` na zapisovanie do súborov a na štandardný vstup/výstup.
- Na štandardný výstup vypíšte informácie o súbore – dĺžku, dátum vzniku/prístupu/..., UID, ... a zapíšte tieto informácie do novovytvoreného súboru.
- Vyskúšajte si službu jadra `link()` na súbor a vypíšte informácie o súbore, ktorý vznikol službou `link()`.
- Vyskúšajte si zmeniť veľkosť súboru pomocou služieb `truncate()` a `open()`.



Sofia verzia 2016

Téma: Prístupové práva

Kľúčové slova	maska prístupových práv, read, write, execute	
Ciele	Zapamätať si:	koncepciu prístupových práv a čo je maska prístupových práv
	Porozumieť:	<ul style="list-style-type: none">• pojmu maska• pojmom user, group, other• rozdiely medzi súbormi a adresármi
	Aplikovať:	služby jadra spojené s riadením prístupových práv
	Viedieť:	využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	30 minút	
Scenár	Sofia sa po tom, ako sa naučila pracovať so súbormi a adresármi, zameria teraz na prístupové práva. Potrebuje sa s nimi naučiť efektívne pracovať. Predstavme si situáciu, keď Sofia vytvorí súbor a nechce, aby si hočikto iný, okrem nej, mohol súbor zmeniť či dokonca zmazat. Práve pre takéto situácie je dobré ovládať prístupové práva.	

POSTUP:

Táto kapitol sa zameriava na:

- **Systémové volania:**
 - umask()
 - chmod()
 - chown()

KRÁTKY ÚVOD

KROK 1 - úvod do prístupových práv:

Prístupové práva sú rozdelené na čítanie (read), zápis (write) a vykonávanie (execute). Pri adresároch právo čítania znamená možnosť výpisu adresára, právo zápisu umožňuje vytvárať súbory v adresári a právo vykonávania povoluje zmenu aktuálneho adresára na iný adresár a jeho podadresáre. Prístupové práva k súboru sú zaznamenané v i-uzle v bitovom tvare, kde „1“ indikuje pridelenie práva použitia.

KROK 2 - výpis prístupových práv:

Výpis prístupových práv a ďalších údajov uskutočňujeme príkazom `ls` s voľbou `-l`. V prvom stĺpci je zobrazený typ súboru a za ním prístupové práva v symbolickom tvare. Typ (prvý znak prvého stĺpca) je kódovaný takto:

- - znamená obyčajný súbor
- **d** indikuje adresár
- **c** a **b** sú príkazy znakového a blokového špeciálneho zariadenia

V druhom stĺpci je počet odkazov na súbor. V treťom a štvrtom je vlastník a skupina súborov (napríklad súborov patriacich projektu `project`). V ďalších stĺpcach je veľkosť, dátum vytvorenia a meno súboru.

► Príklad výpisu príkazu `ls -l`:

```
$ls -l
total 72
-rw-r---- 1 martin project      58 May   12 13:02 File1
-rw-r---- 1 martin project      58 May   12 13:03 File2
-rwxr-x-x 1 martin project    0469 May   12 13:03 program
-rw-r---- 2 martin project      21 May   12 13:02 Text
-rw-r---- 2 martin project      21 May   12 13:02 Text12
drw-r---- 2 martin project     128 May   13 10:05 Reserve
$
```

Pomocou príkazu `whoami` sa zobrazí vaše práve používané užívateľské meno, pod ktorým vás systém pozná. Je to veľmi užitočný príkaz ak máte viaceré kont v systéme a prepíname sa medzi nimi. Každý používateľ je členom jednej alebo viacerých skupín. Na zistenie ku ktorej skupine patríte, použite príkaz: `groups`.

KROK 3 - kategórie používateľov

Užívatelia pracujúci pod OS UNIX/Linux sú delení do troch kategórií:

- **u (user)** vlastník súboru
- **g (group)** pracovná skupina, do ktorej vlastník súboru patrí (pracovná skupina je každému používateľovi pridelená správcom systému)
- **o (others)** ostatní (t.j. členovia ostatných pracovných skupín)

► Príklad

Prístupové práva súboru								
vlastník			skupina			ostatní		
read	write	execute	read	write	execute	read	write	execute
1	1	0	1	0	0	0	0	0

Uvedený súbor môže vlastník čítať a môže doňho zapisovať, užívatelia rovnakej skupiny môžu súbor čítať a ostatní nemôžu nič. Vyššie uvedené práva sa zobrazujú v symbolickom tvare

\$ rw-r----

Podtéma: Služba jadra – umask()

Kľúčové slova	umask(), prístupové práva, príkaz umask
Ciele	Zapamätať si: syntax služby - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://www.scit.wlv.ac.uk/cgi-bin/mansec?2+umask
	Porozumieť: • prístupovým právam súborov • maske práv • službe umask()
	Aplikovať: službu umask() na nastavenie masky práv a s ňou súvisiacou službou create() pre vytvorenie súboru
	Viedieť: využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	10 minút
Scenár	Sofia, ako študentka, má svoje prístupové práva. Chce však nastaviť masku práv (prístupových práv) pre vytváranie jej súborov a tým si zaručiť, že so súborom sa bude pracovať len tak, ako si to ona nastaví, použije službu umask().

POSTUP:

Pri vytváraní súborov službami `open()` alebo `creat()` špecifikujeme prístupové práva novovytvoreného súboru v zdrojovom kóde programu pomocou parametra `mode`. Taktôto je program skompilovaný a už nám neumožňuje tieto práva meniť. Avšak počas behu programu môžeme ešte ovplyvniť práva vytváraného súboru pomocou tzv. masky práv. Maska špecifikuje, ktoré práva budú z hodnoty parametra `mode` odobrané. Maska práv je väčšinou nastavovaná raz – pri prihlásení – konfiguračným súborom shell-u a zvyčajne už nie je menená. Spravidla je nastavená na hodnotu 022 (odobratie práva zápisu do súboru pre skupinu a ostatných). Ak chceme nastaviť špecifickú masku práv, môžeme tak urobiť pred spustením programu príkazom `umask` alebo v programe službou `umask()`.

KROK 1 - naučiť sa syntax a sémantiku služby jadra umask():

Syntax:

```
#include <sys/stat.h>
#include <sys/types.h>
mode_t umask(mode_t cmask);
```

Sémantika:

- Návratová hodnota: predošlá maska

KROK2 – pochopíť parameter služby:

Maska práv je používaná vždy pri vytváraní každého nového súboru alebo adresára. Argument `cmask` môže byť niektorý z prvkov uvedených v nasledujúcej tabuľke:

st mode	význam
S_IRUSR	Čítanie – používateľ
S_IWUSR	Zápis – používateľ
S_IXUSR	Vykonávanie – používateľ
S_IRGRP	Čítanie – skupina
S_IWGRP	Zápis – skupina
S_IXGRP	Vykonávanie – skupina
S_IROTH	Čítanie – ostatní
S_IWOTH	Zápis – ostatní
S_IXOTH	Vykonávanie – ostatní

 Pre podrobnejšie informácie zadaj príkaz **man 2 umask**.

KROK3 – aplikovanie služby v programe:

Nasledujúci program vytvorí dva súbory, jeden s použitím masky 0 (žiadne práva nebudú odobraté) a druhý s maskou, ktorá odoberá práva čítania a zápisu pre skupinu a ostatných.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int main(void)
{
    umask(0);                                //nastavenie masky 0
    if (creat("foo", RWRWRW) < 0)
        perror("creat ()");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH); //obmedzenia masky
    if (creat("bar", RWRWRW) < 0)              //citanie a zapis
        perror("creat ()");                   //pre skupinu a ostatnych
    exit(0);
}
```

Po spustení programu môžeme pozorovať, ako bola nastavená maska práv:

```
$ umask          zistenie aktuálnej hodnoty masky práv
022
$ ./a.out
$ ls -l foo bar
-rw----- 1 sar          0 Sep 20 21:20 bar
-rw-rw-rw- 1 sar          0 Sep 20 21:20 foo
$ umask          kontrola, či sa hodnota masky práv zmenila
022
```

Sofia verzia 2016

Podtéma: Služba jadra – chmod()

Kľúčové slova	chmod(), fchmod(), príkaz chmod		
	Zapamätať si:	syntax služieb chmod() a fchmod() http://www.cl.cam.ac.uk/cgi-bin/manpage?2+chmod	
Ciele	Porozumiet:	<ul style="list-style-type: none">pojmu maskapojmom user, group, other	
	Aplikovať:	služby jadra spojené s riadením prístupových práv	
	Vedieť:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	10 minút		
Scenár	Sofia si vytvorila súbor, ktorý môže prezerať a upravovať iba ona. Potrebuje ho sprístupniť svojim spolužiakom na prezeranie. Pre zmenu práv súborov Sofia chcela použiť príkaz chmod, ale jej OS UNIX/Linux tento príkaz nepodporuje (možno ho niekto zmazal) a tak sa musí pokúsi použiť službu jadra chmod().		

POSTUP:

KROK 1 – naučiť sa syntax a sémantiku služby jadra chmod():

Pomocou tejto služby môžeme meniť prístupové práva k súboru. Z bezpečnostných dôvodov len užívateľ root alebo vlastník súborov môže meniť práva súboru.

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (const char *pathname, mode_t mode);
```

Sémantika:

- Návratová hodnota: 0 ak OK alebo -1, ak sa vyskytla chyba

KROK2 – pochopíť parametre služby:

Prvý argument *pathname* obsahuje názov súboru, ktorému chceme zmeniť prístupové práva argumentom *mode*. Môžeme použiť číselné, alebo symbolické hodnoty uvedené v nasledujúcej tabuľke:

mode	význam
S_ISUID	Vykonávanie pre používateľa (vlastníka)
S_ISGID	Vykonávanie pre skupinu
S_ISVTX	saved-text (sticky bit) POZOR! Pri nastavovaní S_ISVTX (sticky bitu) musíte mať privilégia super-používateľa
S_IRWXU S_IRUSR S_IWUSR	Čítanie, zápis a vykonávanie pre používateľa (vlastníka) Čítanie pre používateľa (vlastníka) Zápis pre používateľa (vlastníka)

S_IXUSR	Vykonávanie pre používateľa (vlastníka)
S_IRWXG	Čítanie, zápis a vykonávanie pre skupinu
S_IROTH	Čítanie pre ostatných
S_IWGRP	Zápis pre skupinu
S_IXGRP	Vykonávanie pre skupinu
S_IRWXO	Čítanie, zápis a vykonávanie pre ostatných
S_IWOTH	Zápis pre ostatných
S_IXOTH	Vykonávanie pre ostatných

KROK 3 – aplikovanie služby v programe:

Nasledujúci program nastaví práva súboru *bar* na uvedenú hodnotu bez ohľadu na aktuálnu hodnotu tzv. prístupových bitov. Pre súbor *foo* sme nastavili práva na základe jeho aktuálnych práv a to tak, že sme zavolali službu *stat()* na získanie aktuálnych práv a potom sme ich upravili. Explicitne sme zaplnili set-group-ID bit a práva na vykonávanie pre skupinu. Použili sme súbory z predchádzajúceho programu, ktorý ich vytvoril.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main(void)
{
    struct stat statbuf;
    /* zapneme prístupový bit skupiny a vypneme jej práva pre
     * vykonávanie */
    if (stat("foo", &statbuf) < 0)
        perror("stat()");
    if (chmod("foo", (statbuf.st_mode | S_IXGRP) | S_ISGID) < 0)
        perror("chmod()");
    /* nastavíme práva na "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        perror("chmod()");
    exit(0);
}
```

Po spustení programu môžeme pozorovať zmenu prístupových práv súborov:

```
$ ls -l foo bar
-rw-r--r-- 1 sar          0 Dec 7 21:20 bar
-rw-rwSr-- 1 sar          0 Dec 7 21:20 foo
```

 Pre podrobnejšie informácie zadaj príkaz **man 2 chmod**.

Sofia verzia 2016

Podtéma: Služba jadra – chown()

Kľúčové slova	chown() , fchown() , lchown() , príkaz chown		
Ciele	Zapamätať si:	syntax služieb - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete: http://www.cl.cam.ac.uk/cgi-bin/manpage?2+chown	
	Porozumieť:	• príkazu chown • službám chown() , fchown() , lchown()	
	Aplikovať:	služby jadra spojené s riadením prístupových práv	
	Viedet:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	10 minút		
Scenár	Sofia má kamaráta, ktorý potrebuje zmeniť vlastníka súboru. Sofia mu poradila, aby použil príkaz chown alebo službu jadra chown().		

POSTUP:

Pokial' vlastníte nejaký súbor, môžete zmeniť jeho vlastníka alebo skupinu (len takú skupinu, ktorej sme členmi.). Akonáhle niekomu pridelíte vlastnícke práva, stratíte s tým spojené privilégiá ako schopnosť zmeniť oprávnenie prístupu a zmeniť vlastníka alebo skupinu. Administrátor smie zmeniť vlastníctvo aj skupinu ktoréhokoľvek súboru.

KROK 1 – naučiť sa syntax a sémantiku služby jadra chown():

Syntax:

```
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

Sémantika:

- Návratová hodnota: 0 ak OK alebo -1, ak sa vyskytla chyba

KROK 2 – pochopiť parametre služby:

ID vlastníka a ID skupiny súboru, pomenovaného parametrom *path*, sa mení špecifikáciou argumentov *owner* a *group*. Vlastník súboru môže zmeniť skupinu na skupinu, v ktorej je jej členom, avšak táto možnosť povolená len pre superpoužívateľa.

Služba jadra chown() vymaže set-user-id a set-group-id bity súboru ako prevenciu pred neúmyselným alebo zlým vytvorením programov nastavujúcich tieto bity, ak nie sú vykonávané s právami superpoužívateľa.



Pre podrobnejšie informácie zadaj príkaz **man 2 chown**.

KROK3 – aplikovanie služby v programoch

Vytvoríme program, ktorý zmení skupinu súboru na základe id alebo názvu existujúcej skupiny. ID alebo názov skupiny a meno súboru sú programu odovzdané ako argumenty.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <grp.h>
#include <stdlib.h>

struct group *gr,*getgrnam();
struct stat stbuf;
int gid;
int status;

main(int argc,char *argv[])
{
    register c;
    if(argc < 3)                                //kontrola poctu argumentov
        printf("pouzitie: chgrp gid subor ... \n");
        exit(4);
    }
    if(isnumber(argv[1])){                      //kontrola ci id skupiny je cislo
        gid = atoi(argv[1]);                     //zmena retazca na cislo
        }
    else {                                       //kontrola skupiny ci sme jej clenmi
        if((gr=getgrnam(argv[1])) == NULL) {
            printf("neznama skupina: %s\n",argv[1]);
            exit(4);
            }
        gid = gr->gr_gid;
        }
    for(c=2; c<argc; c++) {
        stat(argv[c], &stbuf);      //zistenie vlastnika pre zadane subory
        if(chown(argv[c], stbuf.st_uid, gid) < 0) {
            perror(argv[c]);          //zmena skupiny pre zadane subory
            status = 1;
            }
        }
    exit(status);
}

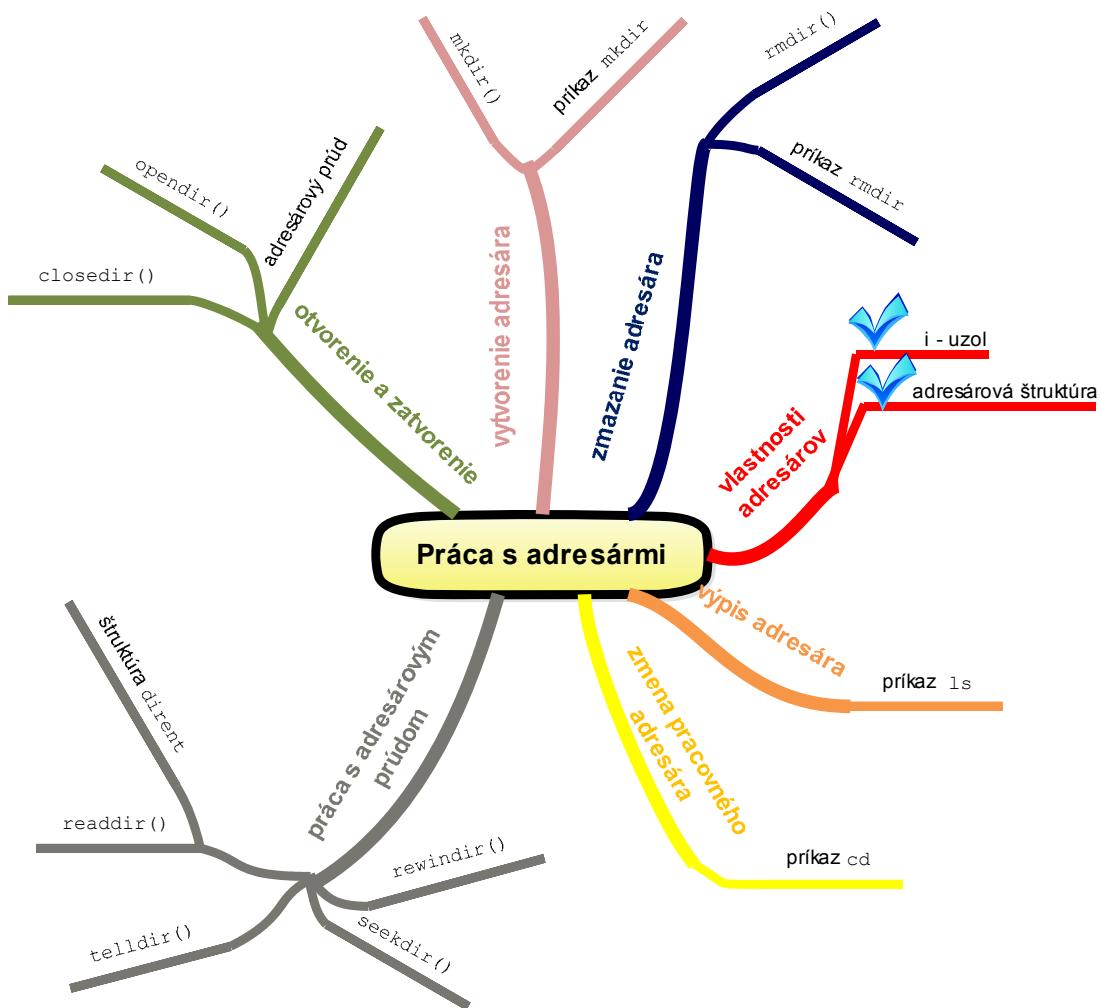
isnumber(char *s) //pomocna funkcia na kontrolu ci argument je cislo
{
    register c;
    while(c = *s++) {
        if(!isdigit(c))return(0);
        }
    return(1);
}
```

Po spustení programu môžeme pozorovať zmenu skupiny súboru:

```
$ls -l file
-rw-r----- 1 root root 58 May 12 13:02 file
$./pristp3 users file
$ls -l file
-rw-r----- 1 root marko 58 May 12 13:03 file
```

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vyskúšajte si zistenie aktuálnej hodnoty masky práv. Nastavenie masky práv tak, aby každý mal prístup k vytvorenému súboru.
- Vyskúšajte si zmeniť práva súboru vo vašom adresári pomocou príkazu `chmod` aj službou jadra `chmod()`. Zistite rozdiel medzi použitím služby `chmod()` a `fchmod()`.
- Vyskúšajte si zmeniť vlastníka súboru vo vašom adresári pomocou príkazu `chown` aj službou `chown()`. Zistite rozdiel medzi použitím služieb `chown()`, `fchown()`, `lchown()`.



Téma: Práca s adresármí v OS UNIX/Linux

Kľúčové slová	adresáre, directories, i-uzol, dirent, ls, cd, mkdir, rmdir		
Ciele	Zapamätať si:		<ul style="list-style-type: none"> • typy adresárov • služby jadra pre prácu s adresármí
	Porozumieť:		<ul style="list-style-type: none"> • parametre služieb • štruktúre i-uzlov
	Aplikovať:		<p>štruktúry a služby na:</p> <ul style="list-style-type: none"> • otvorenie, zápis, čítanie z adresára • získanie informácií o adresári • nastavenie prístupových práv • vymazanie adresára
	Viedieť:		využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	20 min		
Scenár	Sofia už vie o systéme súborov ⁵ OS UNIX/Linux, ktorý má tvar stromu. Je preto vhodná doba, aby si vytvorila vlastný adresár. V tejto kapitole sa Sofia naučí všetky príkazy, ktoré potrebuje na vytvorenie, premenovanie, odstránenie, presun a kopírovanie vlastných adresárov a súborov.		

POSTUP:

Táto kapitola sa zameriava na:

- **Príkazy:**
 - ls
 - cd
 - mkdir
 - rmdir
- **Systémové volania:**
 - mkdir()
 - rmdir()
 - opendir()
 - closedir()
 - readdir()
 - telldir()
 - rewdir()
- **Štruktúra:**
 - dirent

⁵ Systém súborov (ang. file system) v UNIXe je uložený na pevných diskoch, skladá sa z niekoľkých stromových štruktúr, tzv. zväzkov. Podrobnejšie informácie Linux – dokumentačný projekt.

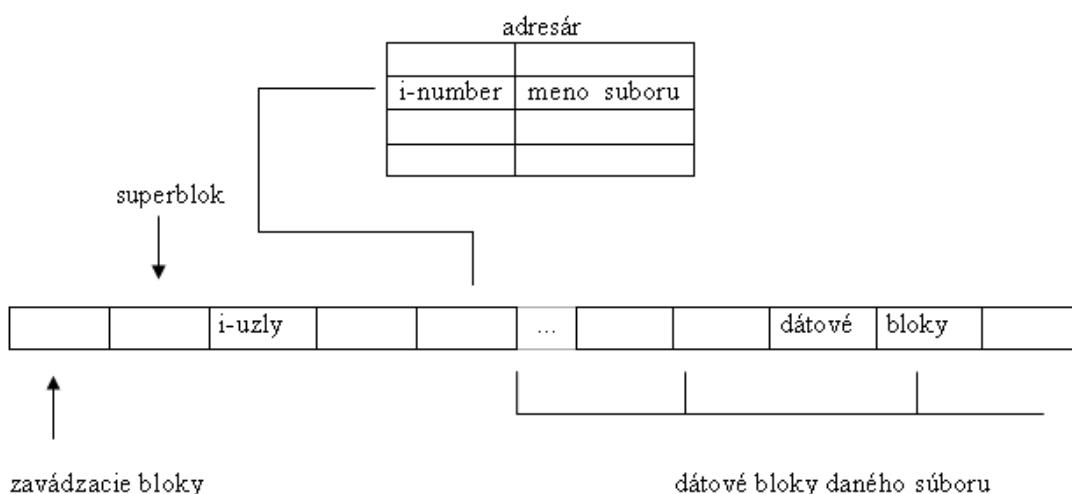
KRÁTKY ÚVOD

KROK1 – pochopit' pojmom adresár a i-uzol:

Adresár je zvláštny súbor, ktorý obsahuje zoznam s menami súborov a čísel ich uzlov. Jednoznačne tak každému súboru priraďuje *i-uzol*⁶ (cez i-number).

Jednotlivé i-uzly sú v systéme rozlíšené číslom. Číslo i-uzlu je jednoznačné v rámci jedného zväzku, preto nestačí k jednoznačnej identifikácii súboru. K nej je treba okrem čísla i-uzlu aj zväzok, na ktorom súbor leží. Pri vytváraní súboru určí OS doposiaľ voľný i-uzol, ktorý bude daný súbor reprezentovať. Jeho veľkosť je 64 bytov.

Súbor je určený z hľadiska jadra UNIXu číslom i-uzlu a z hľadiska používateľa cestou od koreňového adresára k súboru a menom súboru.



Obr.1

KROK2 – pochopit' štruktúru super_blok:

Štruktúra `super_blok` obsahuje informácie o súborovom systéme uloženom na médiu. Jeho formát môžeme opísť nasledujúcou štruktúrou:

```
struct super_block {
    inode_nr s_ninodes;           /* počet i-node */
    zone_nr s_nzones;             /* počet zón na zväzku */
    unshort s_imap_blocks;        /* počet blokov bit mapy i-nodes */
    unshort s_zmap_blocks;        /* počet blokov bit mapy zón */
    zone_nr firstdatazone;        /* číslo prvej dátovej zóny */
    short int s_log_zone_size;    /* počet blokov v zóne (log2 pomeru
                                    blok/zonu, */
                                    /* => ľahký prepočet bitovým posuvom) */
    file_pos s_max_size;          /* maximálna dĺžka súboru */
    int s_magic;                  /* číslo identifikujúce platný super blok*/
};
```

⁶ Identifikačný uzol. Skratka pochádza z anglického slova index node, inode.

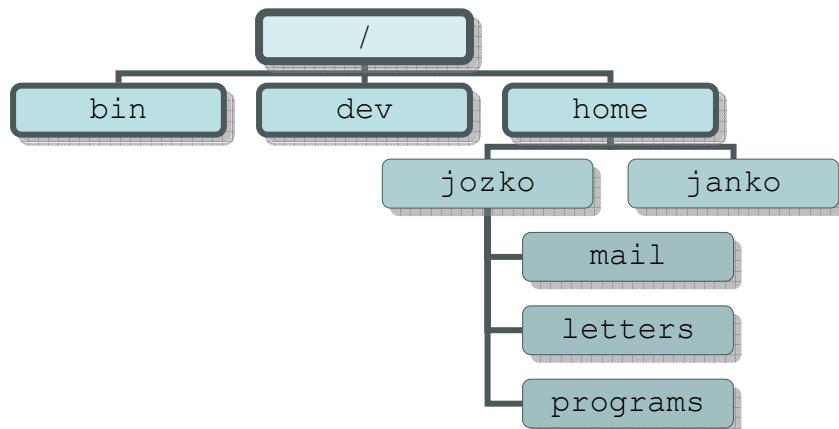
KROK3 – oboznámiť sa s adresárovou štruktúrou v OS UNIX/Linux:

Úplné meno súboru je zoznam všetkých adresárov, ktorými vede cesta od koreňového adresára k súboru s pripojeným menom súboru. Adresáre sa oddeľujú (okrem konečného) znakom „/“.

Domovský adresár (zvyčajne /home/meno_uzivatela) - každý užívateľ ma od administrátora pridelený adresár, do ktorého vstupuje po prihlásení do systému.

Adresár /home je sám podadresárom koreňového adresára (/), ktorý je vrcholom celej hierarchie a v jeho podadresároch sú uložené všetky systémové súbory. Koreňový adresár obsahuje aj adresár /bin, kam sa ukladajú systémové programy (binárne súbory), adresár /etc je určený pre ukladanie systémových konfiguračných súborov a v adresári /lib sú uložené systémové knižnice.

Súbory, ktoré reprezentujú fyzické zariadenia a ktoré poskytujú rozhranie pre tieto zariadenia sa zväčša nachádzajú v adresári /dev.⁷



Obr.2

Operačný systém UNIX/LINUX pracuje so všetkými vstupnými zariadeniami ako so súbormi (pozri obr.2). Z uvedeného vyplýva, že prvým krokom je otvorenie súboru. Pod týmto rozumieme špecifikáciu *mena_suboru* a vytvorenie *kanálu*, cez ktorý budeme k súboru pristupovať. Ďalším krokom je samotná práca so súborom, môže zahŕňať ľubovoľnú kombináciu činností nad súborom. Posledným krokom pri práci so súborom je jeho uzavretie.

⁷ Viac informácií o rozdelení súborového systému v os UNIX vid' man pages, príkaz man hier

Podtéma: Príkaz – ls (list)

Kľúčové slova	ls (list), man ls, unix		
Ciele	Zapamätať si:	príkaz ls:	<ul style="list-style-type: none"> • prečítať si manuálové stránky v Unixe /Linuxe, Linux dokumentačný projekt • zdroje na internete: http://unixhelp.ed.ac.uk/CGI/man-cgi?ls http://www.hmug.org/man/1/ls.php
	Porozumieť:	štruktúre i-uzlov	príkaz ls výpis obsahu adresára:
	Aplikovať:		<ul style="list-style-type: none"> • z výpisu určiť či sa jedná o súbor alebo adresár • rozpoznať koreňový adresár • strom adresárov
Odhadovaný čas	20 min		
Scenár	Sofia sa prihlásila do systému. Nevie v ktorom adresári sa práve nachádza resp. potrebuje zistieť čo v danom adresári má uložené. Preto musí prehľadať jednotlivé adresáre. Zistila, že na zorientovanie sa v adresárovej štruktúre a k vypísaniu obsahu adresára jej poslúži príkaz ls, ktorý jej vypíše obsah aktuálneho adresára.		

POSTUP:

KROK1- naučiť sa používať príkaz ls:

Na zorientovanie sa v adresárovej hierarchii a k vypísaniu obsahu adresára jej poslúži príkaz ls (list) (v skutočnosti sa jedná o rozpis obsahu i-uzlov). Príkaz ls zobrazí na štandardnom výstupe jednostípcový výpis pracovného adresára.

Syntax:

ls [-volba..] [meno_suboru...]

 Pre podrobnejšie informácie zadaj príkaz **man ls**.

Pre príkaz ls existuje viacero možností dodefinovanie:

Volba za ls	Popis príkazu
-a	vypíše všetky súbory aj tie čo začínajú znakom bodka
-B	nevypisujú sa súbory ktoré končia znakom '~' (takto sa označujú záložné súbory)
-I vzorka	nevypisujú sa súbory ktoré vyhovujú zadanej vzorke. Vzorku zadávame pomocou znakov '*' '?' napr. *.tar spôsobí že sa nebudú vypisovať súbory s koncovkou tar

Sofia verzia 2016

-l (long)	<p>Vypíšu sa rozsiahlejšie informácie o súbore</p> <ul style="list-style-type: none"> • typ súboru • prístupové práva • počet hard links, toto je vlastne počet pevných odkazov na súbor • meno vlastníka • meno skupiny • veľkosť v bajtoch • dátum a čas poslednej zmeny • meno adresára alebo súboru
-R (rekurzivne)	Rekurzívne vypisuje obsah adresárov (vypisuje aj obsahy podadresárov)
-C (column)	Výpis po stĺpcach
-x (across)	Výpis zotriedený vodorovne
-F (function)	Oznámi, ktoré z vypísaných súborov sú adresáre a ktoré sú spustiteľné súbory
-t (time)	Výpis v poradí podľa doby zmeny
-d (directory)	Vypisuje informácie o adresári
-r (reverse)	Výpis v opačnom poradí

KROK2 – zjednodušiť si prácu v OS UNIX/Linux:

Niekteré znaky využívané v UNIX/LINUXe:

Znak	Meno	Funkcia
~	Tilda	Skratka do domovského adresára
*	Hviezdička “ * ”	Náhradný znak
?	Otáznik	Náhradný/Pomocný znak
[]	Hranaté zátvorky	Hranice rozsahu príkazu
;	Bodkočiarka	Oddel'ovanie príkazov

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Zadaj príkaz pre zmenu svojho domovského adresára. Aký príkaz je potrebné použiť?

- Zadaj príkaz, ktorým zistíš v akom adresári sa práve nachádzaš? Aký príkaz je potrebné použiť?

- Ak zadáš nasledujúci príkaz `$ls` čo sa vypíše na obrazovku?

- Použi znak pre oddelenie príkazov a v jednom kroku zmeň svoj aktuálny adresár a vypíš obsah zvoleného.

Sofia verzia 2016

Podtéma: Príkaz – cd (change directory)

Kľúčové slova	cd (change directory), man cd, unix	
Ciele	Zapamätať si:	príkaz cd: <ul style="list-style-type: none">• prečítať si manuálové stránky v Unixe /Linuxe, Linux Dokumentačný projekt• zdroje na internete: http://bama.ua.edu/cgi-bin/man-cgi?cd http://www.scism.sbu.ac.uk/law/UnixStuff/cd.html
	Porozumieť:	stromovej adresárovej štruktúre
	Aplikovať:	príkaz cd na zmenu pracovného adresára
	Viedet':	zmeniť svoj pracovný adresár
Odhadovaný čas	5 min	
Scenár	Sofia nenašla hľadaný súbor vo svojom adresári. Potrebuje sa dostať o úroveň vyššie alebo nižšie. Pre zmenu pracovného adresára jej poslúži príkaz cd. O úspešnom prevedení príkazu sa dá presvedčiť príkazom pwd (print working directory)	

POSTUP:

KROK1 – naučiť sa používať príkaz cd:

Syntax:

 \$ cd [meno_adresara]

Ak sa nezadá žiadne meno adresára, nastaví sa ako pracovný adresár domovský adresár užívateľa. To isté sa udeje, ak sa zadá ako meno adresára znak '~'.

V prípade, že sa ako meno adresára zadá znak '-' tak sa nastaví predchádzajúci pracovný adresár. Ako meno adresára je možné zadať i dve bodky (..), tieto označujú návrat v stromovej štruktúre o jednu úroveň nahor. Cestu môžeme zadať absolútne od začiatku stromu (od koreňa) vtedy začína znakom / (lomka) (pozor zmena oproti MS-DOS kde sa používa znak \ (spätná lomka)). Cestu môžeme zadať aj relatívne voči aktuálnemu adresáru, vtedy začneme písť rovno bez lomítka.

Pri zmene pracovného adresára využívame príkaz cd (change directory):

```
$ cd /usr  
$ pwd  
/usr  
$
```

Ak Sofia použije príkaz cd bez argumentu, nastavuje si tak domovský adresár.

```
$ cd  
$ pwd  
/usr/peter  
$
```

Sofia verzia 2016

Zmena pracovného adresára podlieha samozrejme kontrole oprávnení vstupu do adresára podľa prístupových práv je dovolený príznakom "x" vo výpise atribútov adresára.

cd /var/log	nastaví pracovný adresár /var/log
cd ../run	nastaví pracovný adresár /var/run
cd -	naspäť sa nastaví /var/log ako pracovný adresár
cd	a teraz sa nastaví domovský adresár užívateľa
cd ~	toto urobí to isté
cd ..	nastaví pracovný adresár o jednu úroveň vyššie

 Pre podrobnejšie informácie zadaj príkaz **man cd**.

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Nastav svojho domovský adresár. Aký príkaz je nutné použiť?
- Chod o úroveň vyššie. Vypíš obsah ľubovoľne zvoleného adresára.
- Vráť sa do domovského adresára.
- Použi príkaz **pwd** na overenie aktuálneho adresára. Aká bude odpoved?

Sofia verzia 2016

Podtéma: Služby jadra – mkdir() a rmdir()

Kľúčové slova	mkdir(), rmdir(), mkdir, rmdir, unix	
Ciele	Zapamätať si:	<p>príkazy mkdir, rmdir a služby mkdir(), rmdir():</p> <ul style="list-style-type: none">• prečítať si manuálové stránky v Unix /Linux, Linux dokumentačný projekt• Zdroje na internete:<ul style="list-style-type: none">◦ príkaz mkdir: http://unixhelp.ed.ac.uk/CGI/man-cgi?mkdir◦ príkaz rmdir: http://bama.ua.edu/cgi-bin/man-cgi?rmdir+2
	Porozumieť:	parametrom služieb
	Aplikovať:	<ul style="list-style-type: none">• službu na vytvorenie nového adresára• službu na zmazanie adresára
	Vedieť:	využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	7 min	
Scenár	Ak Sofia potrebuje vo svojom domovskom adresári vytvoriť nový adresár, použije na to príkaz mkdir. Ak by ho potrebovala vymazať, urobí to príkazom rmdir. Pri použití príkazu nastane chyba. Sofia zistila, že je aj iná možnosť ako vytvoriť adresár a to pomocou služieb jadra mkdir() a rmdir().	

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služieb jadra mkdir() a rmdir():
Funkcia mkdir() vytvorí prázdný adresár a funkcia rmdir() zruší prázdný adresár.

Syntax mkdir():

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *pathname, mode_t mode);
```

Syntax rmdir():

```
#include <unistd.h>
int rmdir (const char *pathname);
```

Sémantika:

- mkdir() a rmdir() vracia: 0 keď OK alebo -1, pri chybe.

 Pre podrobnejšie informácie zadaj príkaz **man 2 mkdir**, **man 2 rmdir**.

KROK2 – pochopíť parametre služby:

Systémové volanie mkdir() slúži na vytváranie adresárov a je ekvivalentné príkazu mkdir. Vytvorí nový adresár a pomenuje ho podľa parametru pathname. Prístupové

Sofia verzia 2016

práva k adresáru sú špecifikované v parametri `mode` a rovnako, ako u vol'by `O_CREAT` systémového volania `open()`, sú podmienené nastavením premennej `umask`.
Systémové volanie `rmdir()` odstraňuje adresáre, ale iba v prípade, že sú prázdne. Príkaz `rmdir` využíva práve túto službu.

KROK3 – naučiť sa používať príkaz `mkdir`:

Sofia použije príkaz `mkdir` (make directory) na vytvorenie adresára.

```
$ mkdir adr1          // vytvorenie adresára "adr1"
$ ls -l              // výpis obsahu adresára
total 2

drwxr-xr-x 2 sofia group 32 May 13 11:27 adr1

$ cd adr1           //nastavenie na adresár "adr1"
$ mkdir adr2         //vytvorenie podadresára "adr2"
$ cd                 //nastavenie na domovský adresár
```

KROK4 – naučiť sa používať príkaz `rmdir`:

Pre jeho dodatočné zrušenie (vymazanie) jej poslúži príkaz `rmdir` (remove directory):

```
$ rmadir adr1
rmadir: testy: Directory not empty
```

Sofia si však nevšimla, že jej adresár nie je prázdny. Adresár totiž môžeme zrušiť, len ak je prázdny, takže píšeme:

```
$ rmadir adr1/adr2
$ rmadir adr1
$
```

Pre odstránenie adresára príkazom `rmdir` musia byť splnené nasledujúce podmienky:

1. Adresár musí byť prázdny
2. Používateľská identifikácia musí mať oprávnenie pre zápis a pre spustenie v rodičovskom adresári
3. Adresár nesmie byť súčasne pracovným adresárom používateľa

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vytvorte v domovskom adresári adresár *skúška*.
- Nastavte adresár *skúška* ako pracovný adresár.
- Vytvorte súbor *prvy.txt* v ktorom budú nasledovne riadky: toto je prvý riadok
toto je druhý riadok
- Vytvorte adresár *pomocný* (v adresári *skúška*).
- Skopírujte súbor *prvy.txt* do adresára *pomocný* pod menom *druhy.txt*.
- Premenujte súbor *treti.txt* na *stvrty.txt*.
- Vymažte adresár *skúška* so všetkým, čo obsahuje.

Sofia verzia 2016

Podtéma: Funkcie pre prácu s adresármí

Kľúčové slova	opendir(), closedir(), readdir(), telldir(), seekdir(), rewinddir(), dirent, unix
Ciele	<p>Zapamätať si:</p> <p>funkcie pre prácu s adresármí:</p> <ul style="list-style-type: none">• prečítať si manuálové stránky v Unix /Linux, Linux dokumentačný projekt• zdroje na internete:<ul style="list-style-type: none">○ dirent: http://www.opengroup.org/onlinepubs/007908799/xsh/dirent.h.html○ opendir(): http://www.opengroup.org/onlinepubs/007908799/xsh/opendir.html○ closedir(): http://www.hmug.org/man/3/closedir.php○ readdir(): http://www.opengroup.org/onlinepubs/007908799/xsh/readdir.html○ telldir(): http://ccrma.stanford.edu/planetccrma/man/man3/telldir.3.html○ seekdir(): http://bama.ua.edu/cgi-bin/man-cgi?seekdir+3C
	Porozumieť:
	Aplikovať:
	Viedet:
Odhadovaný čas	20 min
Scenár	Sofia potrebuje vypísať obsah svojho adresára, preto použije príkaz <code>ls</code> . Pri použití príkazu nastane chyba. Sofia vie, že obsah adresára sa dá zistiť aj bez použitia príkazu <code>ls</code> a to použitím funkcií pre čítanie a prácu s adresármí.

POSTUP:

Pre získanie základných informácií o súboroch musíme vedieť, aké súbory sa v adresároch nachádzajú. Na to nám slúžia nižšie uvedené funkcie, ktoré sú deklarované v hlavičkovom súbore `dirent.h`. Poznamenajme, že v tomto prípade nejde priamo o služby jadra, ale o nadstavbové funkcie, ktoré služby jadra využívajú vo svojom tele.

KROK1 – oboznámiť sa s adresárovými štruktúrami:

Adresárové funkcie sú deklarované v hlavičkovom súbore `dirent.h`. Ako základ pre manipuláciu s adresárom využívajú štruktúru `DIR`. Ukazovateľ na túto štruktúru sa nazýva adresárový prúd, funguje podobným spôsobom ako súborový prúd (`FILE *`)

v prípade manipulácie s bežnými súbormi. Vlastné adresárové záznamy sú vrátené v štruktúre `dirent`, ktoré sú taktiež deklarované v súbore `dirent.h`.

Štruktúra `dirent`, špecifikujúca adresárové záznamy, obsahuje nasledujúce položky:

```
struct dirent{  
    int_t d_ino;           /* číslo i-uzla */  
    off_t d_off;          /* offset na nasledujúci dirent */  
    unsigned short d_reclen; /* veľkosť súboru */  
    unsigned char d_type; /* typ súboru */  
    char d_name[];        /* názov súboru */  
};
```

KOK2 – naučiť sa syntax a sémantiku funkcií pre prácu s adresárm:

Syntax:

```
#include <sys/types.h>  
#include <dirent.h>  
  
DIR *opendir (const char *pathname);
```

- Vráti: ukazovateľ na adresárový prúd `DIR` ked' OK alebo inak `NULL` pri chybe

```
struct dirent *readdir (DIR *dp);
```
- Vráti: ukazovateľ na štruktúru `dirent` ked' OK, inak `NULL` pri chybe alebo pri konci súboru

```
void rewinddir (DIR *dp);
```
- Nevracia žiadnu hodnotu

```
int closedir (DIR *dp);
```
- Vráti: 0 ked' OK alebo -1 pri chybe

KOK3 – pochopíť parametre funkcií

Funkcia `opendir()` nám otvorí adresár uvedený v parametri `pathname`. Pomocou funkcie `readdir()` prečítame obsah adresára, ktorý je prístupný cez adresárový prúd `DIR *dp` (`dp` – deskriptor adresára), pričom nám funkcia `readdir()` vracia ukazovateľ na štruktúru `dirent`. Pri prehliadaní adresára funkciou `readdir()` nie je zaručené, že budú vypísané všetky súbory (a podadresáre) v danom adresári, pokial' súčasne v rovnakom adresári iné procesy vytvárajú alebo mažú súbory.

Funkcia `rewinddir()` nám resetne pozíciu v adresárovom prúde `DIR *dp` na začiatok a `closedir()` zavrie adresárový prúd a uvoľní s ním združené zdroje.

KOK4 – oboznámiť sa s ďalšími funkciami `telldir()` a `seekdir()`:

Syntax:

```
#include <sys/types.h>  
#include <dirent.h>  
long int telldir(DIR *dirp);
```

Sémantika:

Funkcia `telldir()` vracia hodnotu, ktorá udáva aktuálnu pozíciu v adresárovom prúde. Môže ju potom využiť na nastavenie prehľadávania adresára od aktuálnej pozície.

Syntax:

```
#include <sys/types.h>
#include <dirent.h>
void seekdir (DIR *dirp, long int loc);
```

Sémantika:

Táto funkcia nastavuje smerník na adresárovú položku adresárového prúdu *dirp*. Hodnota *loc*, ktorá definuje príslušnú pozíciu, by mohla byť získaná z volania funkcie *telldir()*. Nemá žiadnu návratovú hodnotu.

KROK5 – aplikovanie služieb v programe:

1. program - Tento program nám vypíše obsah aktuálneho adresára.

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char **argv)
{
    DIR *adresar;
    struct dirent *subor;
    adresar=opendir(".");
    while ((subor=readdir(adresar))!=NULL) {
        printf("%s\n",subor->d_name);
    }
    closedir(adresar);
}
```

Funkcia *opendir()* otvorí adresár, ktorého názov je zadaný v programe (pripadne aj s cestou). V tomto prípade je to aktuálny adresár. Funkcia vráti ukazovateľ na adresárový prúd *DIR *adresar*, ktorý obsahuje informácie o adresári a pomocou ktorého sa k adresáru bude ďalej pristupovať.

Položky zapísané v adresári prečítame pomocou funkcie *readdir()*. Táto funkcia postupne číta položky adresára, pri každom volaní vráti nasledujúcu položku. Ak v adresári žiadna ďalšia položka nie je, vráti NULL. Položka adresára je vrátená ako ukazovateľ na štruktúru *dirent*. Štruktúra *dirent* obsahuje informácie o súbore, ktoré sú uložené v adresári. Pre nás bude zaujímavá iba položka *d_name*, čo je reťazec obsahujúci názov súboru, ku ktorému položka patri.

2. program - Tento program prehľadá adresár zadaný z príkazového riadku a pomocou funkcií *seekdir()* a *telldir()* sa následne vrátíme na položku adresára zadanú ako argument programu.

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    off_t offset;
    DIR *pDir;
    struct dirent *pDirent;
    int of=0,i=1;
```

```
if (argc == 3) {                                //kontrola argumentov
    of=atoi(argv[2]);
    printf ("Otvarany Adresar: %s\n", argv[1]);
    if((pDir = opendir(argv[1])) == NULL){        //otvorenie adresara
        perror("opendir()");
        exit(0);
    }
    while((pDrent = readdir(pDir)) != NULL){      //citanie poloziek adresara
        if(strcmp(".",pDrent->d_name)==0||strcmp("../",pDrent->d_name)==0)
            continue;                //.. a .. (aktualny a domovský adresar) ignoruje
        printf("%d.polozka: %s\n", i, pDrent->d_name);
        i++;
        if(i==of){
            offset=telldir(pDir);    //nastavenie offsetu na polozku
            printf("Offset(telldir) pre %d-tu polozku je %d,\n",i,offset);
        }
        printf("Pouzitim seekdir sa vrati na %d.polozku\n",of);
        seekdir(pDir, offset);           //vratenie na polozku offsetom
        pDrent = readdir(pDir);         //nacitanie polozky
        printf("%d.polozka je: %s\n",of, pDrent->d_name);
    }
    else printf("Chyba argument programu!\n");
    return 0;
}
```

Príklad spustenia predchádzajúceho programu:

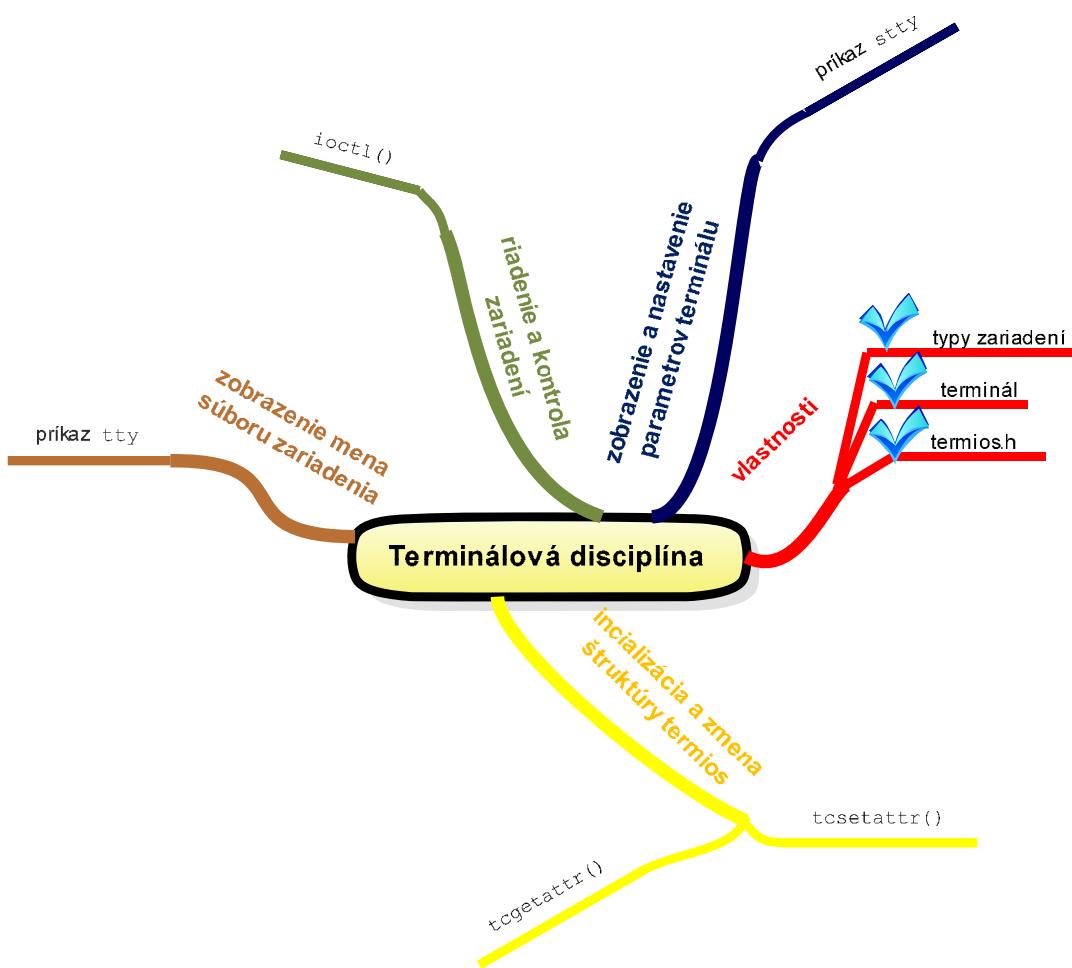
```
$ ./adres2 . 3
$
```

Takže zhrnutie postupu ako prečítať všetky položky adresára:

1. Otvoriť adresár pomocou `opendir()`.
2. Čítať položky adresára pomocou `readdir()` až kým nevráti `NULL`.
3. Zatvoriť adresár pomocou `closedir()`.

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vyskúšajte si prácu s adresárom pomocou služieb jadra `stat()`, `opendir()`, `readdir()`, `closedir()` a pod.
- Vytvorte program, ktorý výpiše celú adresárovú štruktúru aktuálne nastaveného adresára.



Sofia verzia 2016

Téma: Ovládanie zariadení, terminálová disciplína

Kľúčové slová	terminálová disciplína, terminál, termios, /dev/tty	
Ciele	Zapamätať si:	služby jadra a príkazy na ovládania terminálov a iných zariadení
	Porozumieť:	mechanizmu ovládania zariadení
	Aplikovať:	služby jadra a príkazy na zmenu nastavenia terminálov a zariadení
	Viedieť:	<ul style="list-style-type: none">• zmeniť základné nastavenie terminálu, napr. na vypnutie echo• využiť zmenu nastavenia terminálu v zadaniach a ďalších programoch
Odhadovaný čas	60 min	
Scenár	Sofia chce chrániť svoju aplikáciu heslom, aby k nej nemal prístup administrátor. Potrebuje vypnúť echo. Pritom narazila na tému Ovládanie zariadení, terminálová disciplína. V tejto téme sa naučí, akým spôsobom UNIX/Linux komunikuje so zariadeniami a ako je možné túto komunikáciu používať pri tvorbe svojich programov. Bude schopná nastavovať parametre terminálu počas behu programu aj mimo neho.	

POSTUP:

Táto kapitola sa zameriava na:

- **Príkazy:**
 - tty
 - stty
- **Služby:**
 - ioctl()
 - tcgetattr()
 - tcsetattr()
- **Štruktúra:**
 - termios

KRÁTKY ÚVOD

KROK1 – pochopíť ovládanie zariadení v OS UNIX/Linux:

Operačný systém UNIX/Linux, podobne ako väčšina operačných systémov, komunikuje s technickým vybavením počítača pomocou programových komponentov nazývaných ovládače zariadení (device drivers). Ovládače zariadení obsahujú detaľy komunikačného protokolu medzi jadrom OS a technickým vybavením počítača, ktoré umožňujú systému komunikovať so zariadením prostredníctvom štandardného rozhrania.

V OS UNIX/Linux sú ovládače zariadení súčasťou jadra alebo sa do jadra zavadzajú ako špeciálne moduly. OS UNIX/Linux však poskytuje mechanizmus, pomocou ktorého môžu procesy komunikovať s ovládačmi zariadení, teda aj s technickými zaradeniami počítača, prostredníctvom objektov podobných súborom. Tieto objekty sa nachádzajú v súborovom systéme a programy ich môžu otvárať, čítať z nich a zapisovať do nich, ako by to boli obyčajné súbory.

KROK2 - typy zariadení v OS UNIX/LINUX:

Existujú dva základné typy ovládačov zariadení:

- Znakové zariadenie reprezentujú technické zariadenia, ktoré zapisuje dátu ako postupnosť bajtov. Patria sem porty, terminály alebo zvukové karty.
- Blokové zariadenia reprezentujú zariadenia, ktoré čítajú alebo zapisujú dátu v blokoch stanovej veľkosti. Umožňujú priamy prístup k dátam uloženým na zariadení. Patria sem napr. diskové zariadenia.

Výpis niektorých blokových zariadení:

Zariadenie	Meno	Hlavné číslo	Vedľajšie číslo
Prvá disková jednotka	/dev/fd0	2	0
Druhá disková jednotka	/dev/fd1	2	2
Prvý SCSI CD-ROM disk	/dev/scd0	11	0
Druhý SCSI CD-ROM	/dev/csd1	11	1

Výpis niektorých znakových zariadení:

Zariadenie	Meno	Hlavné číslo	Vedľajšie číslo
Prvý sériový port	/dev/ttys0	4	64
Druhý sériový port	/dev/ttys1	4	65
Prvý virtuálny terminál	/dev/tty1	4	1
Druhý virtuálny terminál	/dev/tty2	4	0
Aktuálne zariadenie terminálu	/dev/tty	5	0

KROK3 – terminál:

OS UNIX/Linux používa hostiteľský počítač a k nemu je pripojený Terminal (niekedy virtuálny- napr. cez program putty).

Terminal je zariadenie, ktoré sprostredkováva vstupy a výstupy hostiteľského počítača. V reči OS UNIX/Linux sa terminálu zvyčajne hovorí TTY. Meno „tty“ je odvodene od slova „teletype“. Každé zariadenie je v UNIX/Linux-e reprezentované špeciálnym súborom. Tieto súbory sa spravidla nachádzajú v adresári /dev, napr. súbor /dev/tty reprezentuje aktuálne používaný terminál.

Sofia verzia 2016

Špeciálny súbor `/dev/tty` je zástupcom (logickým zariadením) pre riadiaci terminál (klávesnicu a obrazovku alebo okno) procesu, pokiaľ ho má.

Ked' potrebujeme presmerovať časti programu, ktoré komunikujú s používateľom, ale pritom zachovať ostatný vstup a výstup, musíme túto interakciu viest' mimo štandardného výstupu a štandardného chybového výstupu⁸. Dá sa to dosiahnuť zapisovaním priamo na terminál. UNIX/Linux nám to zjednodušuje a poskytuje nám špeciálne zariadenie s názvom `/dev/tty/`, ktorým je vždy aktuálny terminál alebo relácia (angl. session). Pretože UNIX/Linux so všetkým zaobchádza ako so štandardným súborom, môžeme pri čítaní a zápisе na zariadenie `/dev/tty` používať štandardné operácie so súbormi.

⁸ Štandardného výstupu, štandardného chybového výstupu - pozri tému Úvod resp. LDP

Podtéma: Reprezentácia zariadení

Kľúčové slová	tty, stty, echo	
Ciele	Zapamätať si:	reprezentáciu zariadení v UNIX/LINUXe: <ul style="list-style-type: none"> • prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, • zdroje na internete: http://unixhelp.ed.ac.uk/CGI/man-cgi?tty http://linux.about.com/library/cmd/blcmdl1_tty.htm http://www.scit.wlv.ac.uk/cgi-bin/mansec?7D+tty
	Porozumieť:	mechanizmu prístupu k zariadeniam
	Naučiť sa:	príkazy na manipuláciu so zariadeniami
	Viedieť:	<ul style="list-style-type: none"> • zistiť vlastnosti zariadenia • zmeniť nastavenia zariadenia
Odhadovaný čas	20 min	
Scenár	Sofia potrebuje pochopíť princíp ovládania zariadení, zisťovať a nastavovať ich rôzne parametre (napr. rýchlosť komunikácie cez sieťový port)	

POSTUP:

KROK1 – naučiť sa používať príkaz **tty:**

Syntax:

\$ tty [volba...]

Príkaz **tty** zobrazí úplné meno súboru, ktorý reprezentuje aktuálne zariadenie štandardného vstupu a výstupu, ktorým je obyčajne terminál. Ak použijeme služby `read()`, `write()` na tento súbor (po jeho otvorení službou `open()`), môžeme zapisovať na terminál, alebo čítať znaky z terminálu.

Zadajte príkaz **tty** a doplňte odozvu na tento príkaz: _____ .
Výstup bude vyzeráť napríklad takto:

/dev/tty01

V tomto prípade je meno terminálu **tty01**. Príkaz **tty** v skutočnosti zobrazí meno súboru `/dev/tty01`, ktoré obsahuje systémové rozhranie terminálu. Nazýva sa špeciálny súbor.

► **Príklad**

Nasledujúce riadky programu otvoria súbor s aktuálnym zariadením štandardného výstupu a zapíšu doňho reťazec vo funkcií `fprintf()`. Súbor sa nakoniec zatvorí.

POZOR: Vo funkcií `fopen()` treba doplniť názov a cestu k zariadeniu, ktoré sme zistili príkazom **tty**.

```
FILE *out = fopen("_____", "w");
fprintf(out, "Toto je vypis cez subor");
fclose(out);
```

 Detailnejší manuál k príkazu tty: **man 1 tty**

KROK2 – naučiť sa používať príkaz stty (set TTY, nastav TTY):

Príkaz stty zobrazuje a nastavuje parametre terminálu. Umožňuje riadiť širokú škálu nastavenia terminálu. Týchto nastavení je niekoľko desiatok. Väčšinou budeme príkaz stty používať na kontrolu.

Syntax:

```
$ stty [-F zariadenie] [--file=zariadenie] [nastavenie..]
```

Spustením príkazu sa zobrazia základné nastavenia terminálu, ako je rýchlosť, vypisovanie echo, a pod. Príkazom stty -a zobrazíme všetky nastavenia terminálu.

 Pre podrobnejšie informácie pozrieť **man 1 stty**.

► **Príklad** na vypnutie echo (vypisovania znakov) pomocou príkazu stty z príkazového riadka (pozor, neplatí pre niektoré typy shellov):

1. zadajte príkaz stty -echo
2. po tomto príkaze sa na obrazovku terminálu nevypisujú žiadne znaky pri stláčaní kláves
3. pre návrat do režimu vypisovania echo zadajte príkaz stty echo.

Nesprávne tvrdenie (o vypisovaní vstupu pri vypnutom echu) prečiarknite:

- a. UNIX/Linux registruje príkaz, aj keď sa nevypisuje na obrazovku. To znamená, že aj keď príkaz stty echo nevidíte na obrazovke, vykoná sa.
- b. UNIX/Linux neregistruje príkaz, keď sa nevypisuje na obrazovku. To znamená, že ak príkaz stty echo nevidíte na obrazovke, nevykoná sa.

4. po zadaní príkazu stty echo sa zapne vypisovanie echo.

V poslednom príklade je zrejmé, že ak zadáme parameter príkazu stty so znakom ‘-‘ (stty -echo), tak sa dané nastavenie vypne a ak zadáme parameter príkazu stty bez znaku ‘-‘ (stty echo), tak sa dané nastavenie zapne.

Podtéma: **Služba jadra – ioctl()**

Kľúčové slová	ioctl(), man ioctl()		
Ciele	Zapamätať si:	syntax služby ioctl(): <ul style="list-style-type: none"> • prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt • zdroje na internete: http://www.scit.wlv.ac.uk/cgi-bin/mansec?2+ioctl http://linux.about.com/library/cmd/blcmdl2_ioctl.htm 	
	Porozumieť:	parametrom služby	
	Aplikovať:	službu ioctl() pri nastavovaní zariadení	
	Vedieť:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	10 min		
Scenár	Systémové volanie ioctl() je viacúčelové rozhranie na riadenie technických zariadení. Sofia ho potrebuje poznať pre skvalitnenie svojej práce s terminálmi a zariadeniami.		

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra ioctl():

Služba jadra ioctl() poskytuje rozhranie pre riadenie technických zariadení (terminály, disky, pásky...). Je to volanie jadra, ktoré zabezpečuje kontrolu zariadení.

Syntax:

```
#include <unistd.h>
int ioctl(int fildes, int cmd, ...);
```

Sémantika:

- V prípade úspešného vykonania funkcia vracia hodnotu inú ako -1, ktorá závisí od kontrolnej funkcie zariadenia. Ak nastane chyba, návratová hodnota je -1 a ERRNO je nastavené na indikáciu chyby.

KROK2 - pochopiť parametre služby:

Volanie ioctl() vykonáva činnosť určenú parametrom *cmd* nad objektom, ktorý popisuje deskriptor *fildes*. V závislosti na funkciách podporovaných konkrétnym zariadením sa môže použiť tretí parameter.

Toto volanie jadra realizuje mnohé funkcie s terminálmi, zariadeniami, schránkami a prúdmi. Parametre *fildes* a *cmd* sú posielané prislúchajúcemu súboru ktorý je špecifikovaný deskriptorom a sú implementované ovládačom zariadenia. Táto kontrola je občas používaná na non-stream zariadeniach zo základnými vstupno-výstupnými službami vykonávanými systémovými volaniami *read()* a *write()*.



Pre podrobnejšie informácie pozrieť **man 2 ioctl** alebo kompletný zoznam ioctl() príkazov v **man 2 ioctl_list**.

KROK3 – aplikovanie služby v programe:

Vytvoríme program, ktorý si po spustení vyžiada od používateľa prihlasovacie meno a heslo. Pred zadaním hesla program potlačí vypisovanie znakov na terminál pomocou služby `ioctl()`. Echo je znova zapnuté po zadaní hesla a heslo sa späť vypíše na obrazovku.

```
#include <termio.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE 120

main()
{
    struct termio d_str,d_nov;
    char meno[SIZE];
    char heslo[SIZE];

    printf("\nZadaj svoj prihlasovacie meno:");
    scanf("%s",meno);                                //nacitanie z klavesnice
    ioctl(0,TCGETA,&d_str);                         //nacitanie struktury
    terminalu
    d_nov=d_str;                                    //zalohovanie nastavenia
    terminalu
    d_nov.c_lflag=~ECHO;                            //vypnutie echa
    ioctl(0,TCSETA,&d_nov);
    printf("Zadaj heslo:");
    scanf("%s",heslo);                              //nacitanie hesla z klavesnice
    ioctl(0,TCSETA,&d_str);                         //zapnutie echa
    printf("\nEcho bolo znova zapnute");
    printf("\nTvoje heslo je: %s\n", heslo);
    return 0;
}
```

Podtéma: Štruktúra termios, Funkcie – tcgetattr() a tcsetattr()

Kľúčové slová	termios, funkcie tcgetattr(), tcsetattr()				
Ciele	Zapamätať si:	<ul style="list-style-type: none"> • štruktúru termios • syntax funkcií tcgetattr() a tcsetattr() • prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt 			
	Porozumieť:	parametrom funkcií			
	Aplikovať:	tieto funkcie pri nastavovaní zariadení			
	Viedieť:	využiť získané skúsenosti pri tvorbe programov			
Odhadovaný čas	15 min				
Scenár	Sofia chce vytvoriť program, ktorý si na začiatku od používateľa vypýta zadanie hesla. Potrebuje zabrániť tomu, aby sa pri zadaní hesla vypisovali zadávané znaky na obrazovku. Potrebuje vypnúť echo pomocou štruktúry termios.				

POSTUP:

KROK1 – pochopit’ štruktúru termios:

Zmenu parametrov terminálu je možné dosiahnuť zmenou príznakov, načítaných do štruktúry termios . Jej štruktúra je definovaná v hlavičkovom súbore v termios.h alebo termbits.h (pre Linux).

Hodnoty, pomocou ktorých môžeme riadiť terminál, sú v štruktúre termios zoskupené do niekoľkých skupín:

- Vstup
- Výstup
- Riadiaci
- Lokálny
- Špeciálne riadiace znaky

Štruktúra termios je deklarovaná nasledovne:

```
#include <termios.h>

struct termios {
    tcflag_t c_iflag;           //vstupný režim
    tcflag_t c_oflag;           //výstupný režim
    tcflag_t c_cflag;           //riadiaci režim
    tcflag_t c_lflag;           //lokálny režim
    cc_t c_cc[NCCS];           //špeciálne riadiace znaky
}
```

Pre bežné použitie sú zaujímavé iba príznaky pre posledné dva režimy.

- `tcflag_t c_lflag` je často definovaný ako `unsigned int` alebo `unsigned long`
- `cc_t` typ je nastavený na `unsigned char`.

Okrem toho ešte existuje štruktúra `termio`, ktorá predchádzala `termios`. Je definovaná v `termios.h`. Jej obsah je takmer totožný s `termios`, ukladá však iba 8 špeciálnych znakov. V operačnom systéme zostala z dôvodu kompatibility. V systéme sú definované funkcie, ktoré priradia údaje z `termio` do `termios` a naopak. Pozri `termios.h`.

KROK2 – naučiť sa syntax a sémantiku funkcií na ovládanie terminálu:

Štruktúru `termios` môžeme pre terminál inicializovať prostredníctvom funkcie `tcgetattr()`. Parametre terminálu nastavíme prostredníctvom funkcie `tcsetattr()`. Môžeme otestovať a modifikovať rôzne flagy a špeciálne znaky pre zvolenú pracú terminálu.

Syntax:

```
#include <termios.h>
int tcgetattr(int fd, struct termios *termptr);
int tcsetattr(int fd, int act, const struct termios *termptr);
```

Sémantika:

- Obe vrátia: 0 keď OK alebo -1, pri chybe

KROK3 – pochopit' parametre funkcií:

Funkcia `tcgetattr()` zapíše aktuálne parametre terminálu do štruktúry, na ktorú odkazuje smerník `termptr`. Ak nastavíme terminál na iné hodnoty a chceme ich vrátiť späť do pôvodného stavu, stačí použiť znova `termptr`, do ktorého sme uložili pôvodné nastavenia

Pole `act`, ktoré využíva funkcie `tcsetattr()`, riadi spôsob aplikovania zmien. Môže mať nasledujúce tri hodnoty:

Parameter <code>act</code>	Význam
TCSANOW	Zmení hodnoty ihned.
TCSADRAIN	Zmení hodnoty po dokončení aktuálneho vstupu.
TCSAFLUSH	Zmení hodnoty po dokončení aktuálneho vstupu, ale zruší celý vstup, ktorý je aktuálne k dispozícii a nebol ešte vrátený funkciou <code>read</code> .

 Podrobnejšie v `man 3 termios`.

Lokálne režimy

Lokálne režimy sa nastavujú pomocou prepínačov ukladaných do pola `c_lflag` štruktúry `termios`. Najdôležitejšie sú:

- ECHO – povolí lokálne vypisovanie znakov
- ECHOE – po zachytení znaku Erase ho premení na sekvenciu Backspace – medzera – Backspace
- ICANON – povolí spracovanie kanonického vstupu

► **Priklad** na potlačenie (vypnutie) vypisovania echa:

```
...
struct termios nastavenia;
tcgetattr(fileno(stdin), &nastavenia);
nastavenia.c_lflag &= ~ECHO;
...
```

► **Priklad** povolenie (zapnutie) vypisovania echa (explicitne):

```
...
struct termios nastavenia;
tcgetattr(fileno(stdin), &nastavenia);
nastavenia.c_lflag |= ECHO;
...
```

Povolenie vypisovania echa môžeme dosiahnuť aj použitím implicitných hodnôt nastavenia terminálu tým, že nastavíme terminál na hodnoty, aké mal pred potlačením vypisovania echa. Je dôležité, aby program obnovil pôvodné nastavenie terminálu na hodnoty, ktoré boli nastavené pred jeho spustením. Povinnosťou programu je vždy najprv tieto hodnoty uložiť a po skončení ich zasa obnoviť!

KROK4 – oboznámiť sa s módmí terminálu:

Terminal I/O má dva módy:

1. Kanonický mód – v tomto móde terminálový vstup je spracovávaný ako text.
Terminal vracia najviac jeden riadok pre čítanie.
2. Nekanonický mód – vstupné znaky nie sú rozdelené do riadkov.

Kanonický mód je defaultne nastavený. Napr., ak shell presmeruje štandardný vstup na terminál a použijeme `read()` a `write()` na kopírovanie štandardného vstupu na štandardný výstup, terminál je v kanonickom móde, a každý `read()` vracia najviac jeden riadok.

Programy ako napr. editor `vi` používajú nekanonický mód. Príkazy môžu byť jednotlivé znaky a nie sú ukončené znakom nového riadku. Taktiež, tento editor nepotrebuje systémové spracovanie špeciálnych znakov, ktoré môžu presahovať príkazy editora.

Pole zo štruktúry `termios c_cc` slúži dvom odlišným účelom podľa toho, či je nastavený kanonický alebo nekanonický lokálny režim terminálu.

Pre kanonický režim sa nastavujú hodnoty poľa s indexmi ako VERASE, VKILL, VQUIT, atď.

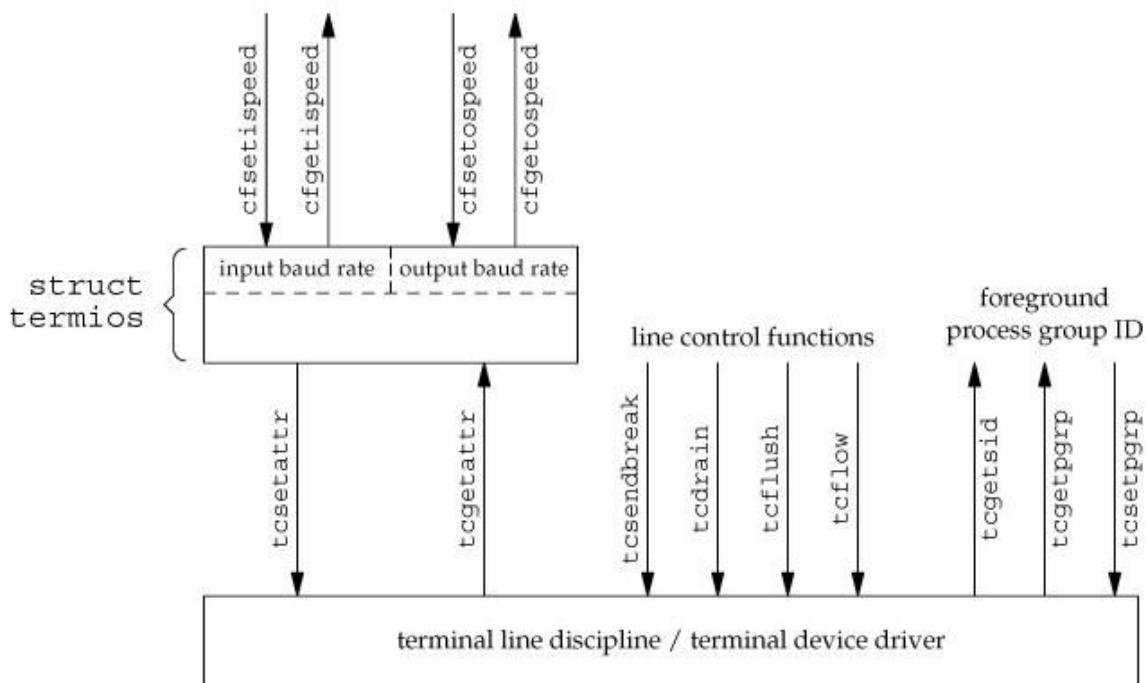
struktura.c_cc[VERASE] = ascii hodnota znaku, ktorý bude použitý pre operáciu erase
struktura.c_cc[VKILL] = ascii hodnota znaku pre signál kill

Pre nekanonický režim sú najzaujímavejšie indexy VMIN a VTIME, ktoré riadia čítanie vstupu.

struktura.c_cc[VMIN] = MIN
struktura.c_cc[VTIME] = TIME

Môžu nastať štyri prípady:

- MIN = 0, TIME = 0 - služba `read()` ihned končí. Ak sú dostupné nejaké znaky, budú vrátené
- MIN = 0, TIME > 0 - služba `read()` skončí až bude k dispozícii nejaký znak alebo uplynie TIME. Funkcia vráti počet načítaných znakov.
- MIN > 0, TIME = 0 - služba `read()` čaká, kým nenačíta aspoň MIN znakov a potom vráti počet načítaných znakov.
- MIN > 0, TIME > 0 - služba `read()` najprv čaká na prvý znak. Služba končí po načítaní aspoň MIN znakov alebo ak doba medzi načítaním dvoch znakov prekročí TIME.



Obr.1
Vzťahy medzi jednotlivými I/O funkciami

POZNAMKA: V UNIXe/Linuxe na čítanie znakov z klávesnice, na rozdiel od prostredia niektorých iných OS, sa používa iba služba `read()`. Preto ak chceme čítať znaky z klávesnice bez čakania (= testovať, či bol stlačený nejaký kláves, alebo nie) musíme zmeniť terminálovú (linkovú) disciplínu tak, ako je to uvedené vyššie.

KROK5 – aplikácia funkcií v programoch:

Program vypne echo a vyžiada zadanie hesla. Po zdaní hesla je echo znova zapnuté a zadávané znaky vypísané na obrazovku. Zmena parametrov terminálu je vykonaná pomocou funkcií `tcgetattr()` a `tcsetattr()`.

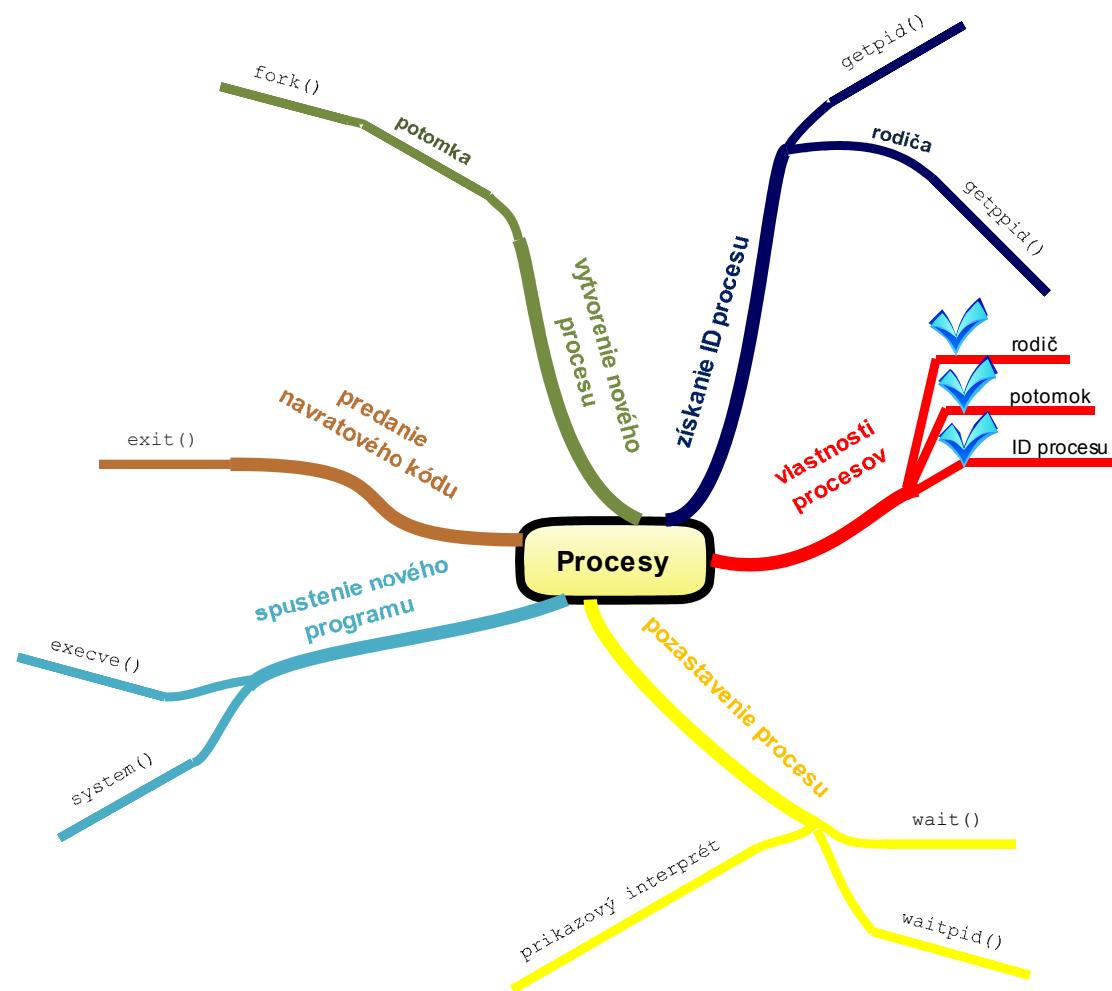
```
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#define PASSWORD_LEN 8

int main()
{
    struct termios initialsettings, newrsettings;
    char password[PASSWORD_LEN + 1];

    tcgetattr(fileno(stdin), &initialsettings);
                                //ziskame nastavenia standardneho vstupu
    newrsettings = initialsettings;          //kopie povodneho nastavenia
    newrsettings.c_lflag &= ~ECHO;           //vypneme priznak ECHO
    printf("Enter password: ");
                                //zadanie hesla
    if (tcsetattr(fileno(stdin), TCSAFLUSH, &newrsettings) != 0){
        //zabranime vypisovanie znakov na obrazovku
        fprintf(stderr, "Could not set attributes \n");
    }
    else {
        fgets(password, PASSWORD_LEN, stdin);      //nacitame heslo
        tcsetattr(fileno(stdin), TCSANOW, &initialsettings);
                                //nastavime povodne nastavenia terminalu
        fprintf(stdout, "\nYou entered %s \n", password);
                                //zobrazime heslo
    }
    exit(0);
}
```

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vyskúšajte si ďalšie funkcie, ktoré poskytujú príkazy `stty` a `tty`.
- Vyskúšajte si ďalšie funkcie, ktoré poskytuje služba jadra `ioctl()`.
- Do už existujúceho programu s určitou funkciou implementujte zadávanie hesla na začiatku behu programu. Rozhranie pre zadávanie hesla navrhnite tak, aby sa heslo nezobrazovalo a aby každé heslo malo konštantnú dĺžku 5 znakov. Pri zadaní piateho znaku hesla sa overovanie správnosti vykoná automaticky (t.j. bez použitia klávesu Enter).
- Vytvorte program, ktorý načíta vstup z klávesnice a vypíše ho na štandardný výstup. Po vypísaní vstupu opäť program načíta vstup z klávesnice, ale pred vstupom z klávesnice potlačí vypisovanie echo pomocou štruktúry `termios`. (Nastavenie terminálu na konci programu vráťte do pôvodného stavu!).
- Vytvorte program, ktorý pred vstupom z klávesnice potlačí vypisovanie echo pomocou štruktúry `termios` s využitím nekanonického režimu (Nastavenie terminálu vráťte do pôvodného stavu!). Nastavte nekanonický režim a hodnoty MIN a TIME:
 1. MIN = 2, TIME = 0
 2. MIN = 0, TIME = 5000
 3. MIN = 5, TIME = 2000



Téma: Procesy

Kľúčové slová	proces, rodič, potomok, program		
Ciele	Zapamätať si:		<ul style="list-style-type: none"> • syntax základných služieb jadra pre prácu s procesmi • hodnoty niektorých parametrov
	Porozumieť:		<ul style="list-style-type: none"> • princípu činnosti procesov • vytváraniu a pridelovaniu činností podriadeným potomkom
	Aplikovať:	Služby jadra pre:	<ul style="list-style-type: none"> • vytvorenie procesu • získanie ID procesu a jeho rodiča • priradenie činnosti procesu • pozastavenie vykonávania procesu
	Vyriešiť:		specifické problémy týkajúce sa práce s procesmi
Odhadovaný čas	50 min		
Scenár	Sofia doteraz pracovala s programami, ktoré vykonávajú jednu cieľenu úlohu od začiatku až do konca. A preto ju trápi otázka – ako prinútiť program, aby vykonával viaceru úlohu, ktoré by poprípade mohli byť na sebe závislé.		

POSTUP:

Táto kapitola sa zameriava na:

- **Systémové volania:**
 - getpid()
 - fork(), getppid()
 - execve()
 - wait(), wiatpid()
 - exit()

KRÁTKY ÚVOD

Čo je to proces? – je to prostredie, v ktorom sa realizujú programy (ako vesmír, v ktorom sa nachádza planéta, čiže náš „program“). Toto prostredie má napr. svoj adresný priestor, sú mu pridelené systémové zdroje. Proces ma pri svojom vzniku pridelené PID, ktoré je v systéme unikátné. Pre získanie PID aktívnych procesov slúži príkaz ps (jeho parametre si Sofia samostatne naštuduje). Každý proces môže vytvoriť ďalší proces. Medzi procesmi takto vzniká vzťah „rodič – potomok“. Bližšie informácie o procesoch – študijná literatúra (napr. skriptá).

Podtéma: Služby jadra – getpid()

Kľúčové slová	getpid(), pid process	
Ciele	Zapamätať si:	návratové hodnoty služby
	Porozumieť:	jej použitiu pri získavaní identifikačných čísel procesov
	Aplikovať:	<ul style="list-style-type: none"> • túto službu pre získanie identifikačných čísel procesov • jej návratové hodnoty
	Vyriešiť:	problémy pri orientovaní sa v hierarchii procesov
Odhadovaný čas	5 min	
Scenár	Aby Sofia pri práci s väčším počtom procesov nestratila orientáciu bude využívať ID procesu pre jeho identifikáciu. Kvôli tomu sa najprv oboznámi so službou jadra getpid().	

POSTUP:

Na programové zistenie ID procesu Sofia použije službu getpid().

KROK1 – naučiť sa syntax a sémantiku služby jadra getpid():

 Pre podrobnejšie informácie zadaj príkaz **man 2 getpid**.

KROK2 – aplikovanie služby v programe:

Použitie tejto služby je veľmi jednoduché, ale jej samostatné použitie v programe nemá veľký význam. Iná situácia nastáva, keď aktuálny proces vytvorí svojho potomka a je potrebné sa medzi nimi zorientovať. Napriek tomu si aj tak ukážeme typické použitie spomínanej služby.

Sofia zostaví jednoduchý program, prostredníctvom ktorého vypíše ID procesu. Ako v každom programe, Sofia najprv pripojí potrebné hlavičkové súbory a následne zrealizuje výpis čísla procesu.

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

int main()
{
    printf("ID procesu je %d\n", getpid());
    return 0;
}
```

KROK3:

Aké bolo číslo procesu, v ktorom bol spustený Váš program?:

Výstup z programu:

```
$
ID procesu je .....
$
```

Podtéma: Služba jadra – fork(), getppid()

Kľúčové slová	<code>fork()</code> , <code>man fork()</code> , <code>getppid()</code> , <code>return value</code> ,	
Ciele	Zapamätať si:	rozdiel medzi návratovými hodnotami služby <code>fork()</code>
	Porozumieť:	<ul style="list-style-type: none"> • hlavne návratovým kódom • tvorbe podriadeného procesu (zdedenie vlastností rodiča) • vykonávaniu programu po vytvorení nového procesu
	Aplikovať:	<ul style="list-style-type: none"> • tieto služby pri tvorbe nových procesov • návratové hodnoty služieb
	Vyriešiť:	tvorbu dvoch a viacerých procesov
Odhadovaný čas	15 min	
Scenár	Aby mohla Sofia naplno využiť možnosti, ktoré jej ponúka práca s procesmi, musí sa naučiť vytvárať nové procesy. V tejto kapitole sa naučí používať aj službu na získanie ID procesu rodiča, ktorý potomka vytvoril. K čomu jej to bude? Hlavne jej to pomôže zorientovať sa vo vzťahu rodič - potomok.	

POSTUP:

KROK1 - naučiť sa syntax a sémantiku služby jadra `fork()`:

Pri vytváraní procesov služba `fork()` vytvorí (takmer) identický proces – klon (to, ktoré vlastnosti zdedí potomok od rodiča, si Sofia pozrie v študijnej literatúre.). Keďže v oboch procesoch spracovanie pokračuje za volaním `fork()`, je veľmi dôležité rozumieť návratovým hodnotám služby v jednotlivých procesoch:

- **Návratovou hodnotou služby `fork()` v rodičovskom procese je ID jeho potomka a v potomkovi je návratovou hodnotou „0“.**

 Pre podrobnejšie informácie zadaj príkaz `man 2 fork`.

KROK2 – aplikovanie služby v programe:

1. program - Sofia si pripraví program pomocou, ktorého si môže vytvoriť nový proces. Pripojí potrebné hlavičkové súbory.

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
```

Sofia definuje premennú pre uloženie návratovej hodnoty funkcie `fork()` :

```
    int pid;
```

Proces - rodič vypíše štandardnú vetu „Hello world“ a vytvorí nový proces pomocou

```
fork():

    printf("Hello World!\n");
    pid = fork();
```

Na základe otestovania návratovej hodnoty služby `fork()` program vypíše hlásenia, buď „Ja som syn“ alebo „Ja som rodič“.

```
if (pid == 0) printf(" Ja som syn.\n");
else printf(" Ja som rodič.\n");
}
```

Kedže po vykonaní služby `fork()` v systéme sa vykonávajú dva identické procesy, v kóde je využitá návratová hodnota služby `fork()`, aby aj rodič a aj potomok vedeli sami seba identifikovať. Po výpise potomok ukončí svoju činnosť. Je potrebné si uvedomiť, že rodič **nečaká** na ukončenie potomka. Vykonávanie rodiča prebieha ďalej, paralelne s vykonávaním potomka (fungujú ako dva nezávislé procesy). Samozrejme, existujú prostriedky, ako zabezpečiť, aby rodič počkal, kým sa jeho potomok neukončí, ale o tom si povieme neskôr.

2. program - Sofia chce vytvoriť program, ktorý bude testovať na základe návratovej hodnoty, či vytvorenie procesu potomok prebehlo v poriadku.

```
#include <stdio.h>
int main(void)
{
    printf("Ja som rodič, kto je viac?\n");
    switch(fork())
```

Na základe návratovej hodnoty služby `fork()` pridelí hlásenie o tom, čo je vykonávané potomkom, rodičom alebo hlásenie o prípadnej chybe vytvorenia potomka.

```
{
case 0: //toto vykonava potomok
    printf("Potomok sa hlasí\n");
    break;
case -1: //toto vykonáva rodič, ak sa nepodarilo vytvorit potomka
    printf("Nastala chyba\n");
    break;
default: //navratova hodnota je PID potomka
    //toto vykonáva rodič ak je všetko v poriadku
    printf("Rodič sa hlasí\n");
}
```

ÚLOHA – modifikácia programu

Pre názornosť využitia naučenej služby `getpid()` a lepšie pochopenie služby `fork()` rozšírite výpisy o identifikáciu ID procesu. Všimnite si výsledok po skompilovaní a spustení programu ☺.

KROK3:

Aké boli čísla procesov rodič a potomok , v ktorom bol spustený Váš program?:

Výstup z programu:

```
$  
Ja som rodič, kto je viac? ID procesu je .....  
Potomok sa hlasí ID procesu je .....  
Rodič sa hlasí ID procesu je .....  
$
```

KROK4 - využitie služby `getppid()`:

Služba jadra `getppid()` ma rovnakú syntax, ako služba `getpid()`, len s tým rozdielom, že jej návratová hodnota je ID rodiča volajúceho procesu!

 Pre podrobnejšie informácie zadaj príkaz `man 2 getppid`.

Sofia pripojí potrebné hlavičkové súbory a vypíše na štandardný výstup ID procesov pomocou služieb `getppid()` a `getpid()`.

```
#include <stdio.h>  
#include <sys/types.h>  
#include <unistd.h>  
  
int main()  
{  
    //vypíše ID procesov  
    printf("ID procesu je %d\n", getpid());  
    printf("ID jeho roduca je %d\n", getppid());  
}
```

ÚLOHA – modifikácia programu

Službu `getppid()` skúste použiť v prípade, keď si sami vytvoríte potomka (jedného, dvoch) nejakého procesu. Kombinujte ju so službou `getpid()` a sledujte návratové hodnoty.

Podtéma: Služba jadra – execve()

Kľúčové slová	execve()		
Ciele	Zapamätať si:	<ul style="list-style-type: none"> • syntax služby • syntax štruktúry predávanej ako argument tejto službe 	
	Porozumieť:	<ul style="list-style-type: none"> • princípu a použitiu tejto služby • jej parametrom, ktoré sa predávajú spustenému programu • návratovým hodnotám 	
	Aplikovať:	<ul style="list-style-type: none"> • službu na nahradenie zdedenej činnosti v podriadenom procese inou činnosťou • službu na spustenie programu s príslušnými parametrami 	
	Vyriešiť:	proces rozhodovania sa, ktorú službu skupiny exec použiť	
Odhadovaný čas	10 min		
Scenár	Teraz sa Sofia potrebuje naučiť, ako v procese spustiť iný program, než ten, ktorý proces - potomok zdedil pri vytvorení od svojho rodiča. Procesu - potomkovi je potrebné „povedať“, aby vykonával nejaký iný program. A práve na to môže Sofia využiť služba jadra execve().		

POSTUP:

Vytváranie nových procesov nemá pre Sofiu veľký význam, pokiaľ nové procesy vykonávajú ten istý program, ktorý zdedili od svojho rodiča. Novovskytvorenému procesu môže priradiť nový program použitím služby jadra execve() :

KROK1 – naučiť sa syntax a sémantiku služby jadra execve() :

Syntax:

```
#include <unistd.h>
int execve(const char *path, char *const argv[], char *const envp[]);
```

Sémantika:

- služba execve() pri úspešnom vykonaní nevracia návratovú hodnotu, -1 pri chybe

KROK2 – pochopíť parametre služby:

Teraz si bližšie vysvetlíme parametre služby execve() v našom prípade – jej prvým parametrom je názov spustiteľného programu, ktorý má vykonávať proces. Ďalším parametrom má byť pole argumentov, ktoré chceme danému spúšťanému programu predáť. Posledný parameter určuje vlastnosti prostredia (environment) spúšťaného programu.

 Podrobnejšie informácie o službe execve() si môžete pozrieť v **man 2 execve**.

KROK3 - aplikovanie služby v programe:

Sofia si vytvorí nový proces, pričom mu hned' priradí vykonávanie programu *child*. *Child* je jednoduchý program, ktorý vypíše ID svojho procesu. Následne pozastaví svoju činnosť na jednu sekundu a vypíše opäť svoje ID.

```
//Program child
#include <stdio.h>
int main (void)
{
    printf("Process[%d]: potomok v case vykonavania ...\\n", getpid());
    sleep(1);
    printf("Process[%d]: potomok pri ukonceni ...\\n", getpid());
}
```

V tomto príklade bol použitá služba `sleep()`, ktorá pozastaví vykonávanie na určitý počet sekúnd, definovaný parametrom tejto služby. Bližšie informácie – `man sleep`.

```
//Program pre vytvorenie noveho procesu
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    if (fork() == 0) {           //toto je potomok
        execve("child", NULL, NULL);
        exit(0);                //tu sa nikdy nedostane
    }
}
```

Proces (potomok) po zavolení služby `execve()` zostáva ten istý. Zmení sa iba kód, ktorý proces vykonáva. Pôvodný kód procesu (teda náš program) sa nahradí kódom programu *child*. Takisto sa nahradia aj údaje pôvodného programu (premenne, konštanty a alokovaná pamäť).

Teraz bude ďalej pokračovať nadriadený proces. Vypíše ID procesu, pozastaví svoju činnosť na dva sekundy a následne vypíše svoje ID pri zaznamenaní toho, že proces sa ukončuje:

```
//toto je uz rodic
printf("Process[%d]: Rodic v case vykonavania ...\\n", getpid());
sleep(2);
if(wait(NULL) > 0)           // potomok konci
    printf("Process[%d]: rodic zaznamenal ukoncenie potomka \\n",
           getpid());
printf("Process[%d]: Rodic konci ...\\n", getpid());
}
```

Stručné vysvetlenie „`if(wait(NULL) > 0)`“ : služba `wait()` pozastaví vykonávanie volajúceho procesu po dobu, kym sa neukončí jeho proces - potomok. Jej parametrom je **smerník** na stavový buffer (celočíselná hodnota) alebo NULL. Ak sa použije ako parameter celočíselná hodnota, služba uloží stavovú informáciu do stavového buffra, na ktorý ukazuje táto hodnota (je to smerník). V našom prípade sme použili ako parameter NULL, pretože nepotrebujeme uložiť do buffra žiadnu stavovú informáciu pre neskôršie

použitie. Služba `wait()` v prípade úspešného volania vráti ID ukončeného procesu, preto sme použili „`wait(NULL) > 0`“.

KROK4 - oboznámiť sa s príbuznými službami k službe jadra `execve()`:

Vyššie bola spomenutá služba jadra `execve()`. Okrem nej existujú aj služby knižník `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`. Všetky tieto varianty sa líšia typom a počtom parametrov, preto v rámci bližšieho si osvojenia predávania programov potomkom si prečítajte manuálové stránky aj k týmto službám.

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg0, ..., (char *)0);
int execlp(const char *file, const char *arg0, ..., (char *)0);
int execle(const char *path, const char *arg0, ..., (char *)0, char
*const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Služby s menom, v ktorom sa nachádza parameter `path`, využívajú premennú prostredia PATH na vyhľadanie spustiteľného programu. Ak sa spustiteľný program nenachádza v PATH, je nutné použiť ako argument meno programu s absolútou, resp. relatívnou cestou k danému programu.

KROK5 - aplikovanie služieb v programe:

V tomto programe Sofia spustí nový program v hlavnom procese pomocou služby `execlp()` bez toho, aby tento nový program spúšťala v novovytvorenom procese.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Spustenie ps pomocou execlp\n");
    execlp("ps", "ps", "-ax", (char *)0);
    printf("Koniec.\n");
    exit(0);
}
```

Výstup z programu:

```
$ ./pexec
Spustenie ps pomocou execlp
PID TTY STAT TIME COMMAND
...
1262 pts/1 S 0:00 /bin/bash
1465 pts/1 S 0:01 emacs Makefile
1514 pts/1 R 0:00 ps -ax
$
```

Po spustení programu `pexec` si Sofia všimla jednu podstatnú vec – vo výpise chýba správa „Koniec“. Čo sa stalo? Program vypísal prvú správu a potom zavolal `execlp()`. Táto služba spustila vykonávanie nového kódu z nového vykonateľného súboru špecifikovaného vo volaní `execlp()` služby (čiže program `ps`). Po skončení programu `ps` sa ukázal nový shell prompt. Nevykonal sa návrat do programu `pexec`, takže sa nevypísala posledná správa „Koniec“.

KROK6 - oboznámiť sa s ďalšou možnosťou priradenia programu:

Sofia sa dozvedela, že novovytvorenému procesu môže priradiť nový program nielen pomocou služby jadra `execve()` a služieb skupiny `exec`, ale tiež pomocou služby `system()`. Pričom služba `system()` využíva pri svojej činnosti služby `fork()`, `execve()` a `waitpid()`.

Syntax:

```
#include <stdlib.h>
int system (const char *string)
```

Táto služba vykoná program, ktorý je jej odovzdaný ako parameter vo forme retázca a čaká na jeho ukončenie. Pre názornosť malý príklad:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Spustenie ps s parametrami cez sluzbu system\n");
    system("ps -ax");
    printf("Done\n");
    exit(0);
}
```



Pre podrobnejšie informácie zadaj príkaz `man system`.

Program zavolá službu `system()` s parametrom `ps -ax`. Služba `system()` spúšťa program v shelli. Tento shell sa spúšťa ešte predtým, než samotný program použitím služby `fork()`. Pričom tu vzniká závislosť na používanom shelli a prostredí (environment). Spustenie programu `ps` sa uskutoční pomocou služby `execve()`, pričom nás program (hlavný proces) nemôže vykonávať žiadnu ďalšiu úlohu pokial' neskončí program `ps` svoju činnosť. Tu sa využíva služba `waitpid()`, ktorá čaká na ukončenie spusteného programu.

Podtéma: Služba jadra – `wait()`, `waitpid()`

Kľúčové slová	<code>wait()</code> , <code>waitpid()</code> , return value		
Ciele	Zapamätať si:		<ul style="list-style-type: none"> • syntax oboch služieb • typ argumentu slúžiaceho na uloženie návratového kódu podriadeného procesu
	Porozumiet:		<ul style="list-style-type: none"> • synchronizácií činnosti medzi procesmi • parametrom služby • návratovým kódom
	Aplikovať:		<ul style="list-style-type: none"> • služby na pozastavenie vykonávanie procesu na dobu, kym sa neukončí potomok procesu • návratové hodnoty služieb pre ďalšie potreby
	Vyriešiť:		problémy jednoduchej synchronizácie medzi procesmi
Odhadovaný čas	10 min		
Scenár	Pri riešení zadanej úlohy Sofia narazila na problém – nevedela, ako prinútiť rodiča, aby počkal na ukončenie činnosti vytvoreného potomka. Riešením tohto problému je použitie služieb jadra <code>wait()</code> a <code>waitpid()</code> .		

POSTUP:

Niekedy je užitočné zistiť, čí potomok procesu už skončil a s akým návratovým kódom. Na toto slúži služba jadra `wait()`.

KROK1 – naučiť sa syntax a sémantiku služieb jadra `wait()` a `waitpid()`:

✍ Pre podrobnejšie informácie zadaj príkaz `man 2 wait`.

Argument `pid` služby `waitpid()` môže nadobúdať tieto hodnoty:

- „-1“ čaká na ukončenie ľubovoľného potomka. V tomto stavu je ekvivalentné k službe `wait()`
- „>0“ čaká na ukončenie potomka presne daným PID
- „== 0“ čaká na ukončenie ľubovoľného potomka, ktorého skupinové ID je rovnaké s ID skupiny volajúceho procesu.
- „<-1“ čaká na ukončenie potomka, ktorého skupinové ID je rovnaké s absolútou hodnotou `pid`.

KROK2 – aplikovanie služieb v programe:

1. **program** - Po pripojení potrebných hlavičkových súborov Sofia deklaruje celočíselnú premennú `status`, ktorá poslúži na uloženie stavovej informácie potomka pri ukončení jeho činnosti a ďalšiu celočíselnú premennú pre uloženie návratovej hodnoty funkcie `fork()`:

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

int main() {
    int pid;
    int status;
    //Vytvorí potomka
    pid = fork();
    //Otestuje, ci vytvorenie potomka prebehlo bez problémov
    if (pid == -1) {
        perror("zly fork");
        exit(1);
    }
    else
        if(pid == 0) {
            printf("riadenie ma na starosti potomok\n");
            printf("jeho pid je: %d\n", getpid());
            exit(51); /*nahodne zvoleny parameter */
        }
        else {
            printf("pid ziskane cez wait je: %d\n", wait(&status));
            printf("status = %d\n", status);
            printf("Rodic pokračuje vo vykonávaní programu.\n");
        }
    return 0;
}
```

Služba `waitpid()` vykonáva podobnú funkciu ako služba `wait()`, avšak s tým rozdielom, že čaká na ukončenie konkrétneho procesu identifikovaného pomocou jeho ID, ktoré sa predá ako argument tejto službe.

KROK3 – oboznámiť sa s pojmom zombie procesy:

V systéme ostane proces (potomok), ktorý už dávno skončil a jeho jediným účelom je odovzdať návratovú hodnotu. Pretrváva dovtedy, pokiaľ sa rodič tiež neukončí, resp. pokiaľ nepreberie návratovú hodnotu, t.j. nezavolá službu jadra `wait()`. To je dôvod, prečo záznam v systémovej tabuľke procesov o potomkovi nie je okamžite odstránený po jeho ukončení. Hoci samotný proces nie je aktívny, nadálej zostáva v systéme ako tzv. proces - mátoha („zombie“ proces).

2. program - Ako názorný príklad „zombie“ procesu si Sofia vytvorila program, v ktorom rodič aj potomok vypíšu určitý počet správ – konkrétnie potomok 2 správy a rodič 10 správ. Keďže potomok vypíše menej správ než rodič, ukončí sa skôr ako rodič a ostane v systéme ako zombie proces. Pokiaľ sa o jeho odstránenie nepostará proces `init`.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    pid_t pid;
    char *message;
    int n;
    printf("spustenie fork programu\n");
    pid = fork();
    switch(pid)
    {
        case -1:
            perror("nepodarilo sa vytvorit potomka");
            exit(1);
        case 0:
            message = "Toto je potomok";
            n = 2;
            break;
        default:
            message = "Toto je rodič";
            n = 10;
            break;
    }
    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```

Ked' Sofia po skompilovaní spustí tento program príkazom `./forkz`. Po skončení potomka a pred ukončením rodiča zadá príkaz `ps -al` do novo otvoreného okna, uvidí podobný výpis:

```
$ ps -al
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
004 S 0 1273 1259 0 75 0 - 589 wait4 pts/2 00:00:00 su
000 S 0 1274 1273 0 75 0 - 731 schedu pts/2 00:00:00 bash
000 S 500 1463 1262 0 75 0 - 788 schedu pts/1 00:00:00 oclock
000 S 500 1465 1262 0 75 0 - 2569 schedu pts/1 00:00:01 emacs
000 S 500 1603 1262 0 75 0 - 313 schedu pts/1 00:00:00 fork
003 Z 500 1604 1603 0 75 0 - 0 do_exi pts/1 00:00:00 fork <defunct>
000 R 500 1605 1262 0 81 0 - 781 - pts/1 00:00:00 ps
```

Ked' potomok ukončí svoju činnosť, musí odovzdať svoju návratovú hodnotu svojmu rodičovi. Ak však rodič tuto hodnotu neprevzme pomocou služby jadra `wait()`, potomok nemôže ukončiť svoju činnosť. Je potrebné si uvedomiť, že rodič nečaká na ukončenie potomka. Vykonávanie rodiča prebieha paralelne s vykonávaním potomka (fungujú ako dva nezávislé procesy). Ak proces rodič ukončí svoju činnosť skôr ako proces potomok. Potom proces potomok sa stáva zombie procesom, ktorý zostane v tabuľke procesov dovtedy, pokým sa o jeho odstránenie nepostará proces `init` s PID 1. Čím väčšia je tabuľka procesov, tým je systém pomalší, preto by sa používatelia mali vyvarovať zombie procesov, nakoľko vyčerpávajú systémové zdroje.

Podtéma: Služba – exit()

Kľúčové slová	exit()		
Ciele	Zapamätať si:	účel tejto služby	
	Porozumieť:	<ul style="list-style-type: none"> • súvislosti tejto služby so službou <code>wait()</code> • návratovým kódom • stavovým makrám 	
	Aplikovať:	službu na ukončenie a odovzdanie návratového kódu (potomka) svojmu rodičovskému procesu	
	Vyriešiť:	tvorbu efektívnych programov	
Odhadovaný čas	10 min		
Scenár	Sofia potrebuje ukončiť proces – potomok, aby proces – rodič mohol pokračovať. Riešením tohto problému je použitie služby <code>exit()</code> .		

POSTUP:

Na okamžite ukončenie vykonávaného procesu a predanie statusu rodičovskému procesu nám poslúži služba `exit()`.

KROK1 – naučiť sa syntax a sémantiku služby jadra `exit()`:

☞ Pre podrobnejšie informácie použite príkaz - `man exit`

KROK2 – aplikovanie služby v programe:

Sofia pripojí potrebné hlavičkové súbory. V hlavnej funkcií deklaruje premennú na uloženie stavovej hodnoty a vytvorí nový proces, pričom na základe návratovej hodnoty služby `fork()` definuje činnosť potomka alebo rodiča.

```
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
int main()
{
    int status;
    if (fork() == 0) {
        /* toto vykona potomok, pozastavi svoje vykonavanie na 6 s */
        printf("pid potomka je: %d\n", getpid());
        printf("pid roduca je: %d\n", getppid());
        sleep(6);
        exit(51);
        /* hodnotu pre argument sluzby exit() sme zvolili nahodne */
    }
}
```

Ináč program vykoná inštrukcie rodiča. Ten počká na ukončenie potomka, pričom ak bol potomok ukončený normálne (kontrola použitím makra `WIFEXITED` na stavovú hodnotu získanú službou `exit()`), vypíše túto stavovú hodnotu pomocou makra `WEXITSTATUS`.

```

else {
    if(pid == -1){
        perror("nepodarilo sa vytvorit potomka");
        exit(1);
    }
    else {
        printf("pid = %d\n", wait(&status));
        printf("status = %x\n", status);
        if (WIFEXITED(status))
            printf("Status (cez makro):%d\n", WEXITSTATUS(status));
        exit(0);
    }
}
}

```

 Význam použitých makier je vysvetlený v manuálových stránkach k službe `wait()`.

KROK3:

Aký bol výpis programu uvedeného v KROKU2?:

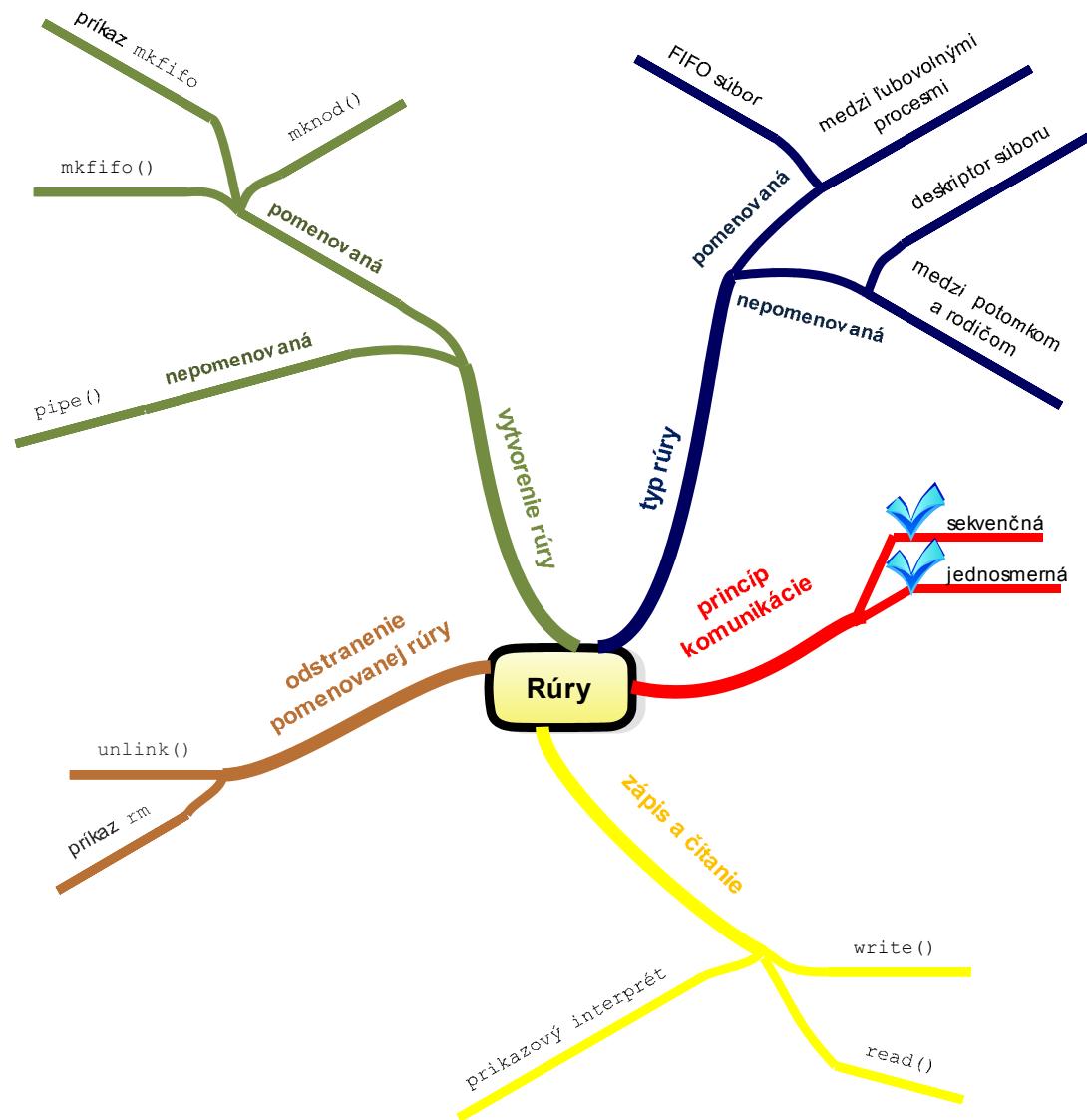
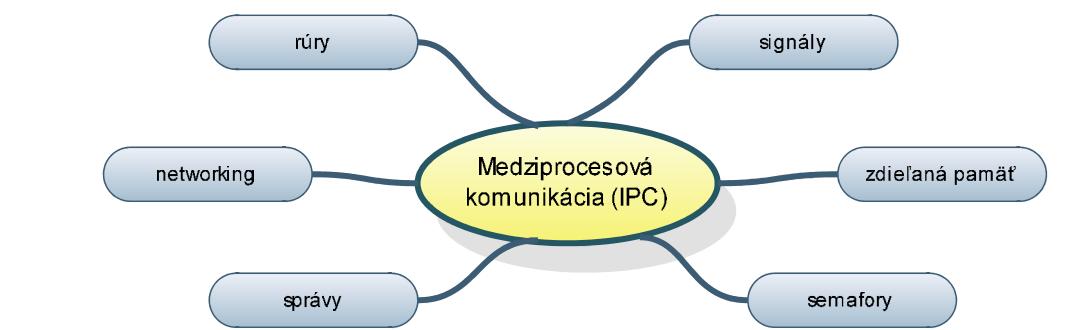
Výstup z programu:

```

$ pid potomka je: .....
pid rodica je: .....
pid = .....
status = .....
Status (cez makro): .....
$
```

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vyskúšajte si ďalšie služby skupiny `exec` a zistite rozdiel použitia medzi jednotlivými službami `exec`.
- Vytvorte program, v ktorom hlavný proces vytvorí jedného potomka. Proces potomok spustí vykonávanie programu "ls -al". Nepoužívajte službu `system()`. Proces rodič počká na ukončenie svojho potomka a vypíše jeho návratovú hodnotu a pid obidvoch procesoch.
- Vytvorte program, ktorý bude prijímať dve číslice ako argumenty. Nech hlavný proces vytvorí proces potomok, ktorý spočíta tieto číslice. Súčet týchto číslíc potomok bude vraciať ako svoj návratový kód. Hlavný proces tento výsledok nakoniec vypíše.



Téma: Komunikácia medzi procesmi – rúry - nepomenované, pomenované

Kľúčové slová	rúra, pomenovaná rúra, pipe, fifo		
Ciele	Zapamätať si:		<ul style="list-style-type: none"> • pojem medziprocesová komunikácia • nástroje na komunikáciu medzi procesmi • syntax služieb jadra a funkcií
	Porozumieť:		<ul style="list-style-type: none"> • pojmu rúra • rozdielu medzi pomenovanou a nepomenovanou rúrou (PIPE a FIFO)
	Aplikovať:		služby jadra určené na vytvorenie a komunikáciu prostredníctvom rúry
	Viedieť:		<ul style="list-style-type: none"> • využiť skúsenosti pri tvorbe programov • vytvoriť takýto prostriedok komunikácie medzi procesmi
Odhadovaný čas	60 min		
Scenár	Sofia už vie vytvárať procesy, avšak tento mechanizmus nedáva možnosť rodičovskému procesu komunikovať s procesom potomkom inak, než prostredníctvom argumentov, premenných prostredia a návratového kódu. V tejto kapitole sa Sofia naučí používať ďalší nástroj pre komunikáciu medzi procesmi, ktorý prekonáva tieto nedostatky - komunikácia pomocou rúr.		

POSTUP:

Táto kapitola sa zameriava na:

- **Systémové volania:**
 - pipe()
 - mkfifo()
 - mknod()

KRÁTKY ÚVOD

KROK 1 - úvod do medziprocesovej komunikácie:

Procesy v operačnom systéme UNIX sú chápane ako nezávislé entity, ktoré nemajú možnosť priamo komunikovať s ďalšími procesmi súbežne v systéme spracovávanými (proces nemôže pracovať s pamäťou iného procesu). Jadro operačného systému poskytuje viacero spôsobov komunikácie medzi nimi. Mechanizmus medziprocesovej komunikácie (InterProcess Communication, ďalej len IPC) umožňuje ľubovoľným procesom výmenu dát a synchronizáciu ich spracovania. V tradičných UNIX-ových systémoch sa stretávame s nasledovnými možnosťami IPC:

- **signály** - jedná sa o spôsob upozornenia procesu o výskytu asynchronnej udalosti. Nie sú posielané žiadne dáta. Procesy si môžu zasielať signály alebo samotné jadro môže vyvolať signál interne pri výskytu asynchronnej udalosti.
- **rúry** - sú najstarším komunikačným mechanizmom vo všetkých typoch UNIX-ov a majú obmedzujúce vlastnosti.
- **pomenované rúry** - dovoľujú komunikovať aj procesom, ktoré nie sú príbuzné.
- **System V IPC** - obsahuje mechanizmy komunikácie známe z kommerčných implementácií UNIX-u, ako sú mechanizmy *front správ* umožňujúcich procesom zasielať formátované postupnosti dát, *zdieľanej pamäte*, ked' procesy môžu zdieľať časti ich virtuálneho adresného priestoru a *semaforov*, pomocou ktorých procesy synchronizujú svoju činnosť
- **sockety** - umožňujú medziprocesovú komunikáciu nielen na lokálnej úrovni v rámci toho istého počítača, ale pomocou komplikovanejších sieťových protokolov umožňujúcich komunikovať procesom medzi počítačmi v rámci počítačovej siete spôsobom klient - server.

KROK2 - oboznámiť sa s pojmom rúra:

Nepomenovaná rúra (pipe) je komunikačný prostriedok, ktorý umožňuje dátovú komunikáciu medzi dvoma procesmi. Je dôležité poznamenať, že tieto procesy musia byť vzájomne príbuzné, teda musí sa jednať o vzťah typu rodič – potomok.

Princíp komunikácie pomocou rúr spočíva v tom, že údaje zapisované na jednom (zapisovacom) konci rúry sú prečítané na druhom (čítacom) konci rúry. Rúra je sériovou komunikáciou, čo znamená, že údaje zapísané na jednom konci rúry sú na druhom konci rúry prečítané v rovnakom poradí.

Nevýhody nepomenovanej rúry sú:

- Rúra je jednosmerný komunikačný prostriedok. Údaje zapisované na jednom konci rúry sú prečítané na druhom konci rúry (jednosmerný tok dát).
- Rúra je vytvorená volaním jadra `pipe()`. Pri vytvorení rúry systém obsadí 2 pozície tabuľky otvorených súborov procesu. Takto vzniknutú rúru dedí každý potomok. Ten sa môže na rúru pripojiť pre čítanie i zápis, rovnako ako sa na ňu môže pripojiť rodič, ale nikto iný, pretože rúra je súčasťou dedičstva, ktoré nemožno exportovať do prostredia iného než procesu potomok. Tento mechanizmus neumožňuje spojenie dvoch ľubovoľných procesov. Snaha odstrániť túto nevýhodu viedla k vytvoreniu tzv. pomenovaných rúr (FIFO).
- Rúry predstavujú samosynchronizujúci prostriedok, ale len na úrovni jedného volania jadra pre čítanie. Pri rozdelení čítania na viac volaní môže dochádzať ku konfliktom medzi procesmi, ktoré môžu s rúry čítať údaje súčasne.

Poznámka:

Pravdepodobne ste už s rúrami pracovali v príkazovom priadku. V príkazovom priadku sa rúra vytvorí pomocou znaku „|“. Napríklad nasledujúci príkaz vytvorí 2 procesy, jeden pre `ls` a jeden pre `less`.

```
% ls | less
```

Príkazový procesor medzi týmito procesmi vytvorí rúru spojujúcu štandardný výstup procesu `ls` so štandardným vstupom procesu `less`. Mená súborov na výstupe z programu `ls` sa odosielajú programu `less` v rovnakom poradí, v akom by sa vypisovali na terminál.

Podtéma: Služba jadra – pipe()

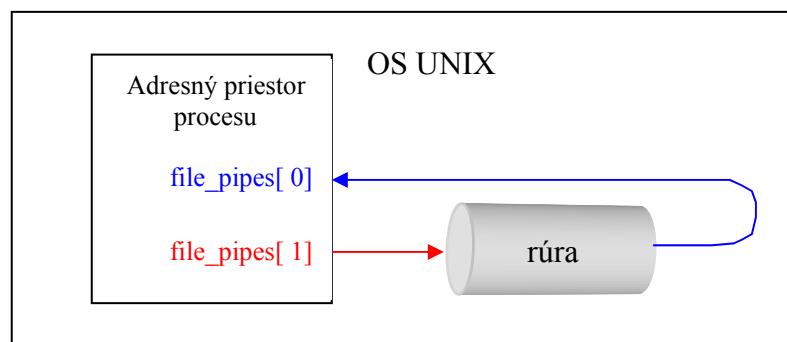
Kľúčové slova	who, pipe()		
Ciele	Zapamätať si:	<ul style="list-style-type: none"> • syntax služby pre vytvorenie nepomenovanej rúry • príkaz pre zistenie prihlásených používateľov • zdroje na internete: http://www.hmug.org/man/2/pipe.php http://www.die.net/doc/linux/man/man2/pipe.2.html 	
	Porozumiet:	<ul style="list-style-type: none"> • medziprocesovej komunikácií založenej na princípe rúr (pipe-ov) • argumentom služby pipe() 	
	Aplikovať:	službu pipe() na vytvorenie nepomenovanej rúry	
	Viedet:	<ul style="list-style-type: none"> • rozdiel medzi pomenovanou a nepomenovanou rúrou • využiť získané skúsenosti pri tvorbe programov 	
Odhadovaný čas	10 min		
Scenár	Sofia dostala za úlohu preniesť dát prostredníctvom rúry. Aby mohla túto úlohu vyriešiť, tak sa potrebuje naučiť vytvoriť rúru pomocou služby pipe() a realizovať prenos dát cez rúru.		

POSTUP:

UNIX umožňuje vytvoriť v jadre vyrovnavaciu pamäť, cez ktorú si môžu procesy vymieňať údaje. Dáta, ktoré jeden proces do rúry zapíše, druhý môže prečítať, ale aj v rámci jedného procesu. Táto vyrovnavacia pamäť sa nazýva rúra,(angl. pipe).

Jednoducho povedané, služba jadra pipe() vytvorí rúru a sprístupní ju prostredníctvom dvoch deskriptorov súborov. Jeden slúži na zápis dát, druhý na ich čítanie. Zápis a čítanie prebieha podobne ako pri súboroch (volania `read()` a `write()`).

Schematický je princíp činnosti rúr zobrazený vzhľadom na jeden proces na Obr.1:



Obr.1

KROK 1 - naučiť sa syntax a sémantiku služby jadra `pipe()`:

Aby Sofia mohla pracovať s rúrou potrebuje sa naučiť službu jadra `pipe()`. Táto služba umožňuje vytvoriť nepomenovanú rúru a sprístupniť ju na odovzdávanie dát medzi procesmi, ale aj v rámci jedného procesu.

Syntax:

```
#include <unistd.h>
int pipe(int file[2]);
```

Sémantika:

Ako argument sa službe `pipe()` odovzdáva dvojprvkové celočíselné pole. Toto pole služba `pipe()` vyplní dvoma novými deskriptormi a vráti návratovú hodnotu - nulu. Pri zlyhaní vráti hodnotu -1. Po volaní služby `pipe()` sa v prvom prvku poľa nachádza deskriptor pre čítanie z rúry (`file[0]`) a v druhom prvku deskriptor pre zápis do rúry (`file[1]`).

Manuálové stránky systému Unix definujú nasledujúce chyby:

Chyba	Popis
EMFILE	Proces používa príliš veľa deskriptorov súborov
ENFILE	Systémová tabuľka súborov je plná
EFAULT	Deskriptor súboru je neplatný



Podrobnejšie informácie o službe `pipe()` si môžete pozrieť v `man 2 pipe`.

KROK2 – aplikovanie služby v programe:

1. program - Program vytvorí rúru prostredníctvom volania `pipe()` a sprístupní ju pomocou dvoch deskriptorov v poli `file_pipes`. Do rúry sa zapíšu dátá pomocou deskriptora `file_pipes[1]`, prostredníctvom deskriptora `file_pipes[0]` ich môže proces prečítať. Rúra používa vyrovnávaciu pamäť obmedzenej veľkosti (cca 4KB), ktorá slúži na uloženie dát medzi volaniami `write()` a `read()`. Spravidla sa volania `read()` a `write()` vyskytujú v rôznych procesoch.

Program 1

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE    5

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data,
            strlen(some_data));
```

```
    printf("Zapis %d bytov\n", data_processed);
    data_processed = read(file_pipes[0], buffer, BUFSIZ);
    printf("Citanie %d bytov: %s\n", data_processed, buffer);
    exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

Výstup z programu:

```
$
Zapis 3 bytov
Citanie 3 bytov: 123
$
```

ÚLOHA – modifikácia programu

Pre názornosť využitia naučenej služby `pipe()` a lepšie pochopenie prenosu dát cez rúru v rámci jedného procesu rozšírite Program 1 o službu `execve()`. Služba `execve()` nám spustí vykonávanie programu, ktorý prečíta dáta, ktoré boli zapísané pred volaním služby `execve()`.

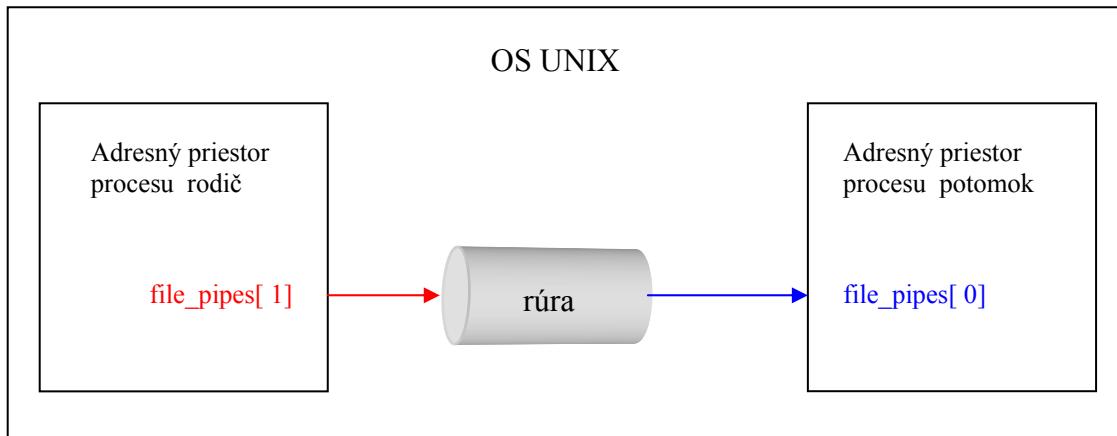
Podtéma: Rodičovské a dcérske procesy

Kľúčové slova	medziprocesová komunikácia		
Ciele	Zapamätať si:	<ul style="list-style-type: none"> • služby pre komunikáciu medzi procesmi prostredníctvom rúr • zdroje na internete: http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html 	
	Porozumieť:	komunikáciu medzi rodičovským procesom a potomkom pomocou rúry	
	Aplikovať:	služby na komunikáciu medzi procesmi prostredníctvom rúr	
	Viedieť:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	10 min		
Scenár	V predchádzajúcom príklade sa Sofia naučila vytvoriť rúru. Rúra je v prvom rade prostriedkom pre medziprocesovú komunikáciu pre pochopenie tohto problému jej pomôže nasledujúca podkapitola.		

POSTUP:

KROK1 – pochopíť komunikáciu medzi procesmi pomocou rúry:

Sofii vysvetlíme komunikáciu medzi rodičom a potomkom pomocou rúry na Obr.2.



Obr.2

Na začiatku rodičovský proces vytvorí dva deskriptory súborov ako pole deskriptorov o dvoch prvkoch, napr. `int file_pipes[2]`. Deskriptor `file_pipes[0]` sa zvyčajne používa na čítanie z rúry a `file_pipes[1]` na zápis do rúry. Volaním služby `pipe()` sa otvoria deskriptory rúry a sú prístupné na čítanie a zápis. Jadro nám teda poskytuje pamäť, pomocou ktorej môžu tieto procesy komunikovať.

KROK2 – aplikovanie služby v programe:

2. program – Aby sme zrealizovali architektúru zobrazenú na obr. 2, proces rodič najprv vytvorí rúru pomocou služby `pipe()`, potom pomocou služby jadra `fork()` sa vytvorí nový proces. Pokiaľ je volanie `fork()` úspešné, rodičovský proces môže do rúry zapísat dátu, ktoré potom prečíta proces potomok. Obidva procesy potom ukončia svoju činnosť po volaní služieb `write()` alebo `read()`.

Program 2

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;
    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) { // vetva potomok
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        }
        else { // vetva rodič
            data_processed = write(file_pipes[1], some_data,
                strlen(some_data));
            printf("Write %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Výstup z programu:

```
$ 
Read 3 bytes
Write 3 bytes: 123
$
```

Deskriptory súboru získané procesom ktorý rúru vytvára sú prístupné iba procesom - potomkom. Pri volaní služby `fork()`, deskriptory súboru sa dedia procesom potomkom. Odtiaľ vyplýva, že rúry môžu spojovať iba príbuzné procesy. V Program 2 `fork()` vytvorí potomka. Ten zdedí deskriptor súboru pre rúru. Rodičovský proces zapíše do rúry dátu, ktoré potom proces – potomok prečíta.

Podtéma: Rúra FIFO

Kľúčové slova	fifo , man fifo		
Ciele	Zapamätať si:		syntax služby <code>mkfifo()</code> , <code>mknod()</code> : <ul style="list-style-type: none"> • prečítať si manuálové stránky v Unixe/Linuxe • zdroje na internete: http://www.hmug.org/man/n/fifo.php http://www.die.net/doc/linux/man/man4/fifo.4.html
	Porozumieť:		<ul style="list-style-type: none"> • pojmu pomenovaná rúra • rozdielu medzi pomenovanou a nepomenovanou rúrou
	Aplikovať:		služby <code>mkfifo()</code> resp. <code>mknod()</code> na vytvorenie pomenovanej rúry
	Viedet:		využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	20 min		
Scenár	<p>Doposiaľ sa Sofia naučili ako môže prenášať údaje medzi dvoma procesmi, ktoré boli príbuzné. To však pre využívanie rúr nie je príliš praktické, najmä ak Sofia potrebuje prenášať jednotlivé údaje medzi nezávislými procesmi. To dosiahne pomocou pomenovaných rúr, ktorým sa hovorí aj rúry FIFO.</p>		

POSTUP:

KROK1 – pochopiť vlastnosti pomenovanej rúry:

Ako bolo spomenuté, nepomenované rúry neumožňujú komunikovať s procesom, ktorý nie je potomkom procesu. Dôvodom je to, že deskriptory je možné odovzdať iba potomkom. Tento nedostatok odstraňujú pomenované rúry (FIFO)

Pomenovaná rúra (FIFO) je špeciálny typ súboru, ktorý existuje ako záznam v súborovom systéme, ale chová sa ako nepomenovaná rúra. Tento „súbor“ slúži len ako položka – referenčný bod, ktorým procesy môžu pristupovať k rúre na základe mena tohto súboru.

Rozdiel medzi pomenovanou a nepomenovanou rúrou je v tom, že pomenovaná rúra je súčasťou súborového systému. Môže byť otvorená viacerými procesmi ako na čítanie tak aj na zápis. Tieto procesy môžu byť navzájom nezávislé.

Jadro podporuje jeden objekt pre každý FIFO súbor, ktorý je otvorený aspoň jedným procesom. FIFO musí byť otvorené na oboch koncoch (zápis aj čítanie), predtým než môže predávať dátu. Štandardne je otváranie FIFO blokované pokial nie je otvorený aj druhý koniec. (ak nie je otvorené na čítanie, nedá sa zapisovať a opačne).

KROK2 – naučiť sa syntax a sémantiku funkcie `mkfifo()`:

Ak chce Sofia vytvoriť FIFO súbor použije službu `mkfifo()` alebo príkaz `mkfifo`¹.

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Sémantika:

- `mkfifo()` vracia 0 ak sa proces uskutočnil bez chýb alebo -1 ak nastala chyba.

Jednotlivé parametre sú:

- `pathname` - cesta k danému súboru
- `mode` - prístupové práva

 Pre podrobnejšie informácie zadaj príkaz `man 3 mkfifo` a `man 1 mkfifo`.

KROK3 – aplikovanie služby v programe:

Sofia potrebuje vytvoriť pomenovanú rúru. Správu o jej úspešnom vytvorení rúry chce vypísať na štandardný výstup a následne program ukončiť.

1. Najprv do programu pridá potrebne hlavičkové súbory. Na vytvorenie pomenovanej rúry využije spomínanú funkciu `mkfifo()`.

```
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    int des_fifo;
    des_fifo=mkfifo("/tmp/my_fifo",0777);
    printf("des_fifo je %d",des_fifo);
    if(des_fifo==0) printf("uspesne sa podarilo vytvorit FIFO");
    else perror("chyba:");
    return 0;
}
```

2. Záznam o rúre si Sofia skontroluje pomocou príkazu:

```
ls -lF /tmp/my_fifo
```

3. Aký bol Vás výpis použitím predchádzajúceho príkazu:

Prvý znak výpisu je p, čo znamená, že sa jedná o rúru. Symbol | pridala na koniec riadku voľba - F príkazu ls a tiež označuje rúru.

¹ Ďalší príkaz ktorý môžeme použiť je `mknod`. Ten však nemusí byť dostupný vo všetkých unixovských systémoch, preto by sme mali dať prednosť príkazu `mkfifo`. Príkaz `mknod` viď. man pages.

Odstránenie pomenovanej rúry:

Rúru FIFO môžeme odstrániť pomocou príkazu `rm` alebo pomocou služby jadra `unlink()`.

Prístup k rúre FIFO:

Pretože pomenované rúry sú súčasťou súborového systému, majú jednu veľmi užitočnú vlastnosť - môžeme ich používať v príkazoch na mieste, kde môžeme normálne použiť názov súboru. Teraz si opíšeme správanie pomenovanej rúry v spojitosti s normálnymi príkazmi pre prácu so súbormi.

Do príkazového riadku skúste zadať nasledujúce príkazy: (obmena príkladov, čo sme si ukázali na začiatku)

1. Najprv sa pokúsime čítať z práznej rúry FIFO:

```
$ cat < /tmp/my_fifo
```

2. Teraz skúsime do rúry zapísat:

```
$ echo "sdsdasdf" > /tmp/my_fifo
```

3. Ak spustíme obidva príkazy súčasne, môžeme dátá odovzdať prostredníctvom rúry:

```
$ cat < /tmp/my_fifo & echo "Ja som rura" > /tmp/my_fifo
[1] 1316
Ja som rura

[1]+ Done                  cat < /tmp/my_fifo
$
```

Ako to funguje:**Prípad 1-2:**

V rúre neboli žiadne dátá preto príkazy `cat` a `echo` čakali kým do rúry nejaké dátá dorazia, resp. až ich začne nejaký iný proces čítať.

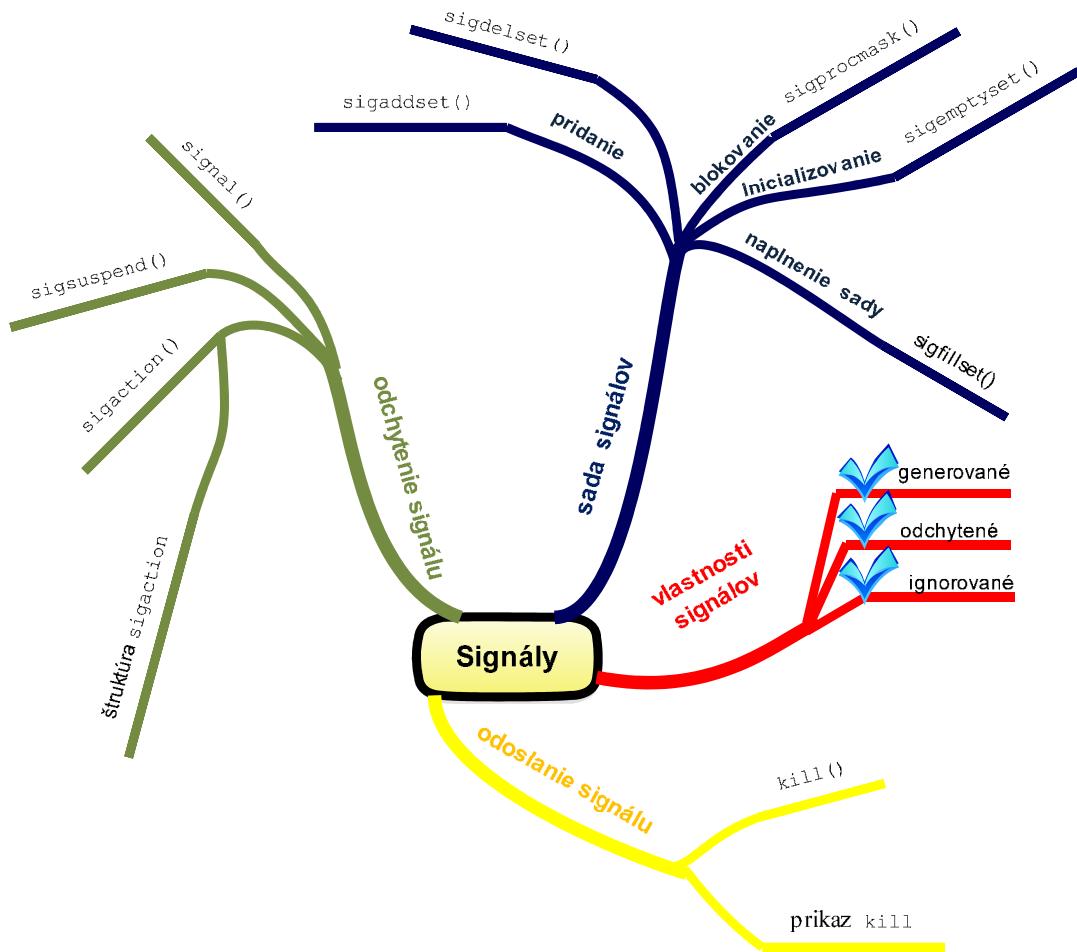
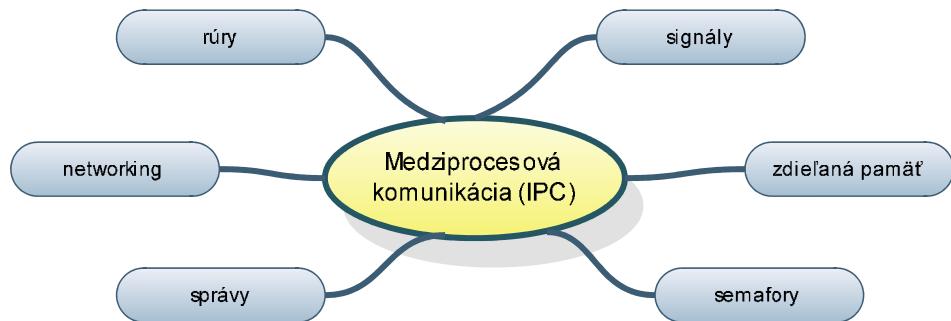
Prípad 3:

Proces `cat` je najprv zablokovaný na pozadí. Ako náhle dá príkaz `echo` k dispozícii nejaké dátá, príkaz `cat` ich prečíta a vytlačí na štandardný výstup.

Na rozdiel od rúry vytvorennej volaním `pipe()` existuje rúra FIFO v podobe pomenovaného súboru, nie ako otvorený deskriptor súboru, a pred tým ako sa do/z nej môžu zapisovať/čítať dátá musíme ju otvoriť. Rúra FIFO sa otvára/zatvára rovnakými funkciemi `open()`/`close()`, ktoré sme používali pre otváranie/zatváranie súborov.

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vytvorte program, ktorý pozostáva z 2 procesov a komunikácia medzi týmito procesmi prebieha pomocou rúr. (rodičovský proces teda načítava čísla z terminálu a proces - potomok ich vypisuje).
- Vytvorte rodiča aj potomka. Nech medzi sebou komunikujú pomocou dvoch rúr. Do jednej zapisuje rodič a do druhej syn. Nech obidvaja vypisujú to čo prečítali z rúry.
- Zamyslite sa, ako by ste realizovali komunikáciu medzi dvoma procesmi v oboch smeroch pomocou jedinej rúry.
- Vytvorte dva nezávislé programy, ktoré medzi sebou komunikujú pomocou pomenovanej rúry. Prvý program vytvorí pomenovanú rúru a zapíše do nej reťazec. Druhý program prečíta reťazec. Nech obidva programy vypíšu reťazec, ktorý bol zapísaný do rúry a prečítaný z rúry.



Téma: Signály

Kľúčové slova	medziprocesová komunikácia, procesy, signály		
Ciele	Zapamätať si:		<ul style="list-style-type: none"> základné princípy komunikácie medzi procesmi prostredníctvom signálov fakty o odchytávaní a posielaní signálov
	Porozumieť:		komunikácii procesov pomocou signálov
	Aplikovať:		<ul style="list-style-type: none"> služby posielajúce signály metódy odchytávania signálov metódy ignorovania signálov
	Vyriešiť:		zložitejšie príklady medziprocesovej komunikácie
Odhadovaný čas	36 min		
Scenár	Procesy sami o sebe dokážu vykonávať dôležité činnosti, ich sila sa prejavuje až v spojení s inými procesmi, teda ich komunikáciou. Jedným zo spôsobov komunikácie medzi procesmi sa Sofia dozvie v tejto téme.		

POSTUP:

Táto kapitola sa zameriava na:

- **Príkaz:**
 - o kill
- **Systémové volania:**
 - o signal()
 - o kill()
 - o alarm()
 - o sigsuspend()

Práca navyše:

- **Systémové volania:**
 - o sigaction()
 - o sigaddset()
 - o sigemptyset()
 - o sigfillset()
 - o sigdelset()
 - o sigprocset()
- **Štruktúra:**
 - o sigaction

Podtéma: Príkaz - kill

Kľúčové slová	kill, send signal				
Ciele	Zapamätať si:	<ul style="list-style-type: none"> • najdôležitejšie parametre tohto príkazu • jeho syntax 			
	Porozumieť:	funkciu a využitiu príkazu			
	Aplikovať:	pre potrebu ukončenia alebo signalizácie procesu			
	Vyriešiť:	špecifické situácie súvisiace s uviaznutím procesu			
Odhadovaný čas	3 min				
Scenár	Pokial' chce Sofia poslať signál inému procesu prostredníctvom príkazového interprétra ako úlohe na pozadí, použije práve príkaz <code>kill</code> . Aby ho mohla využívať, potrebuje sa ho naučiť používať.				

KRÁTKY ÚVOD

Signál je komunikačný prostriedok, ktorý generuje systém UNIX ako odpoveď na výskyt konkrétnej udalosti. Po jeho prijatí môže proces na tento signál reagovať. Sú buď generované operačným systémom alebo používateľským procesom. Signály môžu byť generované a odchytiavané alebo ignorované. Názvy signálov sú definované v hlavičkovom súbore `signal.h`.

KROK1 – oboznámiť sa so signálmi v Unix:

 Pre podrobnejšie informácie zadá Sofia príkaz `man 7 signal`.

Doplňte (podľa `man 7 signal`):

Názov	č.	Akcia	Popis
SIGINT	2	Term	Prerušenie terminálu (ekvivalentné s Ctrl+C z klávesnice)
SIGQUIT	3	Core	
SIGKILL	9	Term	Ukončenie procesu (tento signál nie je možné odchytiať ani ignorovať)
SIGTERM	15	Term	
SIGCHLD	17	Ign	Dcérsky proces bol ukončený alebo prerušený
SIGCONT	18		
SIGSTOP	19	Stop	
SIGWINCH	28	Ign	Signál zmeny veľkosti okna

POSTUP:**KROK2 – naučiť sa používať príkaz kill:**

Príkaz shellu `kill` slúži na okamžité zaslanie signálu jednému procesu alebo skupine procesov. Tento príkaz prijíma voliteľné číslo signálu (príkaz `kill` má prednastavenú hodnotu signálu SIGTERM) a PID procesu (ktoré obvykle zistíme príkazom `ps`), ktorému sa má daný signál poslat'.

 Pre podrobnejšie informácie zadá Sofia príkaz `man kill`.

Príklady:

kill 59	odošle signál SIGTERM procesu s PID = 59
---------	--

Ak je číslom procesu záporné číslo, signál pošle danej skupine procesov (PGID)

kill -59	odošle signál SIGTERM procesom skupiny PGID = 59
----------	--

Ak zadá ako PID číslo -1, signál pošle všetkým procesom okrem PID = 1
(`init` proces – proces, ktorý riadi spúšťanie iných procesov)

kill -1	odošle signál SIGTERM všetkým procesom okrem PID = 1
---------	--

Ak chce poslať signál trom individuálnym procesom oddelí ich medzerou.

kill 23 24 38	odošle signál SIGTERM procesom s PID = 23, 34 a 38
---------------	--

Najčastejšie však Sofia pravdepodobne použije poslanie signálu SIGKILL parametrom „-9“.

kill -9 2888	odošle signál SIGKILL procesu s PID = 2888
--------------	--

Podtéma: Služba jadra - signal()

Kľúčové slová	signal(), man signal				
Ciele	Zapamätať si:	<ul style="list-style-type: none"> jej syntax a syntax obslužnej funkcie argumenty tejto služby 			
	Porozumieť:	princípu čakania a odchytávania signálu			
	Aplikovať:	službu pri odchytávaní, resp. ignorovaní nejakého signálu			
	Vyriešiť:	efektívnu komunikáciu medzi dvoma alebo viacerými signálmi			
Odhadovaný čas	10 min				
Scenár	Sofia chce vytvoriť program, ktorý odchytí príchod špecifického signálu prostredníctvom príkazového interprétra. Zistila, že pre vyriešenie tejto úlohy jej pomôže služba <code>signal()</code> .				

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra `signal()`:

Hlavnou úlohou tejto služby je čakanie na špecifický signál, po prijatí ktorého sa vykoná nejaká obslužná funkcia (*signal handler*). Prostredníctvom nej môže proces vykonať špecifickú činnosť, ignorovať prijatý signál (SIG_IGN) alebo po jej vykonaní nastaviť implicitnú reakciu na daný signál (SIG_DFL).

Syntax:

```
#include <signal.h>
typedef void(*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

 Pre podrobnejšie informácie zadaj príkaz `man signal`.

Doplňte návratové hodnoty služby `signal()`:

- služba `signal()` vracia _____, pri chybe _____

KROK2 – aplikovanie služby v programe:

Sofia si vytvorí program, ktorý vypisuje reťazec „Hello world“ v sekundových intervaloch, pokiaľ nedostane signál SIGINT, ktorý pri normálnom nastavení terminálu reprezentuje kombinácia kláves Ctrl+C. Pri prvom stlačení Ctrl+C program vypíše, že signál bol odchytiený a zároveň nastaví pôvodné správanie na signál SIGINT. Po ďalšom stlačení kombinácie kláves už program končí.

Najprv Sofia pripojí potrebné hlavičkové súbory:

```
#include <signal.h>
#include <stdio.h>
```

Následne definuje obslužnú funkciu signálu, teda reakciu, čo sa stane, keď procesu (programu) bude poslaný signál:

```
void odchytenie(int sig){
    printf("Odchytenie signalu %d\n", sig);
    signal(SIGINT, SIG_DFL);
}
```

V hlavnej funkcií použije službu jadra `signal()` a v nekonečnom cykle bude po uplynutí jednej sekundy vypisovať reťazec „Hello world“ :

```
int main(){
    (void) signal(SIGINT, odchytenie);
    while(1){
        printf("Hello world\n");
        sleep(1);
    }
    return(0);
}
```

Tento program bude čakať na signál SIGINT (poslaný Sofiou pri stlačení CTRL+C). Počas doby čakania bude vypisovať „Hello world“. Po prijatí signálu sa aktivuje obslužná funkcia `odchytenie()`, v ktorej sa vypíše číslo prijatého signálu a pomocou argumentu SIG_DFL sa nastaví pôvodná reakcia na signál SIGINT, teda ukončenie programu. Ak by v obslužnej funkcií nebolo nastavenie pôvodnej reakcie, program by po každom stlačení CTRL+C zavolal obslužnú funkciu a neukončil by sa. Sofia by musela použiť príkaz `kill -9` s číslom tohto procesu, aby sa tento proces ukončil.

KROK3:

Aké bolo číslo odchyteneho signálu vo výpise Vášho programu?:

Výstup z programu:

```
$
Hello World
Hello World
Hello World
Hello World
^C
Odchytenie signalu _____
Hello World
Hello World
Hello World
Hello World
$
```

Podtéma: Služba jadra - kill()

Kľúčové slová	kill()	
Ciele	Zapamätať si:	syntax tejto služby
	Porozumieť:	parametrom služby a jej spojeniu so službou signal()
	Aplikovať:	získané vedomosti pri posielaní signálov iným procesom v rámci programu aj mimo neho
	Vyriešiť:	zložitejšie príklady komunikácie medzi procesmi
Odhadovaný čas	5 min	
Scenár	<p>Služba signal() umožnila Sofii napísť program, ktorý čaká na príchod nejakého signálu. Opäť však Sofia musela poslať tento signál ručne. Aby sa zbavila tohto zásahu do činnosti programu, môže vytvoriť iný proces, ktorý bude za ňu posielat signál prostredníctvom služby kill().</p>	

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra kill() :

Ak sme schopní signály spracovať, musíme ich vedieť i zasielať. To nám umožňuje služba jadra kill(). Služba kill() umožňuje zaslať signál procesu alebo skupine procesov.

Syntax:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

☞ Pre podrobnejšie informácie zadaj príkaz **man 2 kill**.

Doplňte návratové hodnoty služby kill() :

- služba kill() vracia _____, pri chybe _____

KROK2 – aplikovanie služby v programe:

V nasledujúcim príklade po 5 sekundách pošle proces - potomok signál SIGALRM procesu – rodič a proces - potomok sa ukončí. Po prijatí signálu sa proces – rodič ukončí.

Sofia pripojí potrebné hlavičkové súbory a definuje globálnu premennú na príznak odchytenia signálu a obslužnú funkciu:

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

void ding(int sig){
    printf("Odchytanie signálu %d\n", sig);
}
```

V hlavnej časti programu vytvorí jedného potomka procesu. Tento potomok po uplynutí 5 sekúnd pošle svojmu procesu (rodičovi) signál SIGALRM:

```
int main()
{
    pid_t pid;
    printf("aplikacia bezi..\n");
    pid = fork();
    switch(pid) {
        case -1: perror("fork zlyhal");
                    exit(1);
        case 0:   sleep(5);
                    kill(getppid(), SIGALRM);
                    exit(0);
    }
}
```

Rodič bude čakať na daný signál, pričom využije službu `pause()`. Táto služba pozastaví vykonávanie programu do príchodu nejakého signálu:

```
default:  printf("cakanie na signal..\n");
           signal(SIGALRM, ding);
           pause();
           printf("koniec\n");
           exit(0);
    }
```

KROK3:

Aké bolo číslo odchyteneho signálu vo výpise Vášho programu?:

Výstup z programu:

```
$
aplikacia bezi..
cakanie na signal..
Odchytenie signalu _____
koniec
$
```

Podtéma: Služba jadra - alarm(), sigsuspend()

Kľúčové slová	raise signal, alarm() unix
Ciele	Zapamätať si: základnú syntax
	Porozumieť: využitiu tejto služby, jej rozdielu od služieb pozastavujúcich činnosť programu
	Aplikovať: službu pri vykonaní nejakej činnosti programu po vykonaní istej doby
	Vyriešiť: situácie spojené s plynutím času v programe
Odhadovaný čas	8 min
Scenár	Tieto služby nemajú také uplatnenie, ako predošlé služby, avšak Sofia sa môže stretnúť so situáciami, v ktorých sa jej táto služba hodí. V príklade tejto podtémy sa dozvie o takejto situácii.

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služieb:

Služba `sigsuspend()` umožní Sofii pozastaviť činnosť programu dovtedy, pokiaľ nie je programu poslaný nejaký signál. Aby si precvičila túto službu, môže s ňou experimentovať v spojení so službou `alarm()`. Služba `alarm()` umožňuje vytváranie časovačov a plánovačov. Služba zaistí, že volajúcemu procesu bude po `sec` sekundách zaslaný signál `SIGALRM`.

 Pre podrobnejšie informácie - `man sigsuspend` a `man alarm`.

KROK2 – aplikovanie služieb v programe:

Nasledujúci príklad pozastaví vykonávanie programu do príchodu signálu `SIGALRM`, ktorý sám sebe po uplynutí 5 sekúnd službou `alarm()`.

```
#include <signal.h>
#include <stdio.h>

void ding(int sig){
    printf("Odchytenie signalu %d\n", sig);
}
```

V hlavnej časti programu sa zavolá služba `alarm()`, ktorá by mala po 5 sekundách vyvoláť signál `SIGALRM`:

```
int main()
{
    sigset(SIGALRM, ding);
    printf("aplikacia bezi...\n");
    alarm(5);
```

Proces bude čakať na signál, pričom využije službu `sigsuspend()`. Táto služba pozastaví vykonávanie programu do príchodu nejakého signálu:

```
printf("čakanie na signal..\n");
signal(SIGALRM, ding);
sigsuspend(SIGALRM);
printf("koniec\n");
return(0);
```

{}

KROK3:

Aké bolo číslo odchyteneho signálu vo výpise Vášho programu?:

Výstup z programu:

```
$  
aplikacia bezi..  
cakanie na signal..  
Odchytenie signalu _____  
koniec  
$
```

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vytvorte program, ktorý bude generovať a zachytávať dva signály SIGUSR1 a SIGUSR2. Program bude posielat' tieto signály podľa hodnoty počítadla nastavenej od 0 do 25. Program pošle signál SIGUSR1, ak bude hodnota počítadla deliteľná číslom 5. Program pošle signál SIGUSR2, ak bude hodnota počítadla deliteľná číslom 3. Výskyt týchto signálov sa má zachytiť a vypísať na štandardný výstup programu.
- Vytvorte program, ktorý vytvorí nový proces potomok (simulujúci budík). Nech rodičovský proces žiada potomka (budík), aby ho ráno zobudil. Pošle procesu potomok signál. Potomok potvrdí, že ho zobudí. Pomocou funkcie `sleep()` sa odsimuluje noc. Potom nech potomok pošle rodičovskému procesu signál, že je čas vstávať. Rodičovský proces odpovie, že vstáva.
- Vytvorte program, ktorý vytvorí dva nové procesy, ktoré budú komunikovať cez nepomenovanú rúru. Jeden potomok zapíše do rúry dátu zadané z klávesnice a druhý túto správu prečíta z rúry. Tento cyklus sa opakuje päť krát. Na synchronizáciu zápisu do rúry a čítania z rúry použite signály. Proces rodič počká na ukončenie svojich potomkov.

**Podtéma: Služba jadra - `sigaction()` (`sigemptyset()`, `sigfillset()`,
`sigaddset()`, `sigdelset()`, `sigprocmask()`)**

		<code>sigaction()</code> , signal set, reliable signals
Ciele	Zapamätať si:	<ul style="list-style-type: none"> základné fakty o využití týchto služieb ich základnú syntax
	Porozumieť:	<ul style="list-style-type: none"> argumentu služby <code>sigaction()</code>, ktorý nastavuje príznak správania sa služby sadám signálov princípu pridávania signálov do sady
	Aplikovať:	<ul style="list-style-type: none"> všetky členy štruktúry <code>sigaction</code> pri nastavovaní správania sa procesu po prijatí signálu služby pracujúce so sadami signálov
	Vyriešiť:	problémy súvisiace s prijatím signálu v okamihu, keď proces ešte nedokončil činnosť spojenú s obsluhou predošlého signálu
Odhadovaný čas		10 min
Scenár		Sofia sa dočítala, že pre lepšiu spoľahlivosť a hlavne bezpečnosť programov sa majú používať definované sady signálov. Prostredníctvom týchto sád môže napríklad definovať, vzhľadom na ktoré signály má byť program imúnny, čo sa má stať po prijatí nejakého signálu. Využiť tieto sady môže práve prostredníctvom služieb, s ktorými sa naučí pracovať práve v tejto podtéme.

POSTUP 1:

KROK1 – naučiť sa syntax a sémantiku služby `sigaction()`:

Služba `sigaction()` poskytuje možnosť presnejšie špecifikovať chovanie obslužných rutín signálov.

Syntax:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

Služba jadra `sigaction()` využíva štruktúru `sigaction`, ktorej obsah je nasledovný:

```
struct sigaction{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *,void*);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

- Prvou položkou zoznamu štruktúry je obslužna funkcia `sa_handler`, s rovnakým významom ako pri službe `signal()`. Sofia namiesto druhej položky zoznamu štruktúry môže použiť príznaky `SIG_IGN` a `SIG_DFL`.

- Tretou položkou `sa_mask` zoznamu štruktúry definuje sadu signálov ešte predtým, než sa zavolá obslúžna funkcia. Patria sem signály, ktoré majú byť ignorované pri vykonávaní obslúžnej funkcie. Tým sa zabráni výskyt problému, ktorý často nastáva vtedy, keď je procesu poslaný nejaký signal v momente vykonávania obslúžnej funkcie.
- Posledná položka `sa_flags` modifikuje správanie sa obslúžnej funkcie signálu. Jeho hodnotami môžu byť:

Názov	Popis
<code>SA_NOCLDSTOP</code>	Negeneruje sa signál SIGCHILD, keď sa potomkovia končia
<code>SA_RESETHAND</code>	Nastavuje pôvodnú akciu na daný signál (význam ako <code>SIG_DFL</code>)
<code>SA_RESTART</code>	Reštartuje služby, ktoré môžu byť prerušené príchodom nejakého signálu
<code>SA_NODEFER</code>	Keď beží obslužná funkcia a proces zachytí nejaký signál, tento signál sa nepridá do sady signálov

KROK2 – oboznámiť sa so službami knižníc `sigemptyset()`, `sigaddset()`, `sigdelset()`, `sigprocmask()`:

Všetky tieto služby prijímajú sadu signálov ako svoj argument. Táto sada je typu `sigset_t`. Najdôležitejšími sú služby `sigemptyset()` a `sigfillset()`, ktoré by mala Sofia volať vždy pred ostatnými službami.

- služba `sigemptyset()` inicializuje sadu signálov
- služba `sigfillset()` naplní sadu signálov všetkými možnými signálmi
- služba `sigaddset()` pridá do sady nejaký signál
- služba `sigdelset()` odstráni jeden alebo niekoľko signálov zo sady. Táto služba je obvykle volaná po službe `sigfillset()`
- služba `sigprocmask()` blokuje prijatie niektorého signálu počas doby vykonávania obslúžnej funkcie

Najzaujímavejšou z týchto služieb je služba `sigprocmask()` slúžiaca na chránenie kritických sekcií programu, teda sekcií, ktorých vykonanie nesmie byť prerušené (napríklad príchodom nejakého signálu). Táto služba modifikuje aktuálnu signálovú masku. Prvý argument tejto služby určuje, ako má byť modifikovaná aktuálna signálová maska. Argument môže nadobúdať tieto hodnoty:

Názov	Popis
<code>SIG_BLOCK</code>	Signály v dodanej sade sú pridané k blokovaným signálom aktuálnej sady
<code>SIG_UNBLOCK</code>	Signály v dodanej sade sú odstránené zo sady blokovaných signálov
<code>SIG_SETMASK</code>	Kompletne nahradza signálovú masku. Špecifikovaná sada nahradí aktuálnu sadu, v ktorej sa nachádzajú blokované signály

 Pre podrobnejšie informácie si Sofia prečíta manuál k týmto službám!

KROK3 – aplikovanie služieb v programe:

1.program - Nasledujúci príklad je veľmi jednoduchý, aplikuje len služby `sigaction()` a `sigemptyset()`.

Sofia po pripojení potrebných hlavičkových súborov definuje obslužnú funkciu, ktoré bude len vypisovať jeden reťazec:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig){
    printf("Dostal som signal %d\n", sig);
}
```

V hlavnej funkcií Sofia najprv deklaruje štruktúru `sigaction`, nastaví obslužnú funkciu tejto štruktúry na funkciu `ouch()`, vyprázdní sadu signálov a príznak štruktúry inicializuje hodnotou „0“, pretože nepotrebuje nastaviť špeciálne správanie sa obslužnej funkcie. Následne Sofia zavolá službu `sigaction()`, ktorej na mieste prvého argumentu predá SIGINT, teda program bude odchytávať signál prerušenia. Nakoniec bude v slučke vypisovať reťazec „Hello World“ v sekundových intervaloch.

```
int main()
{
    struct sigaction act;
    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, 0);
    while(1){
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Výstup z programu:

```
$
Hello World!
Hello World!
Hello World!
^C
Dostal som signal 2
Hello World!
Hello World!
^C
Dostal som signal 2
Hello World!
Hello World!
^\
Quit
```

Program bude po každom stlačení klávesovej skratky CTRL+C vypisovať reťazec „Hello World“, pretože štruktúra `sigaction` je nastavená na opakované odchytenie tohto signálu SIGINT. Pre ukončenie programu môže Sofia stlačiť klávesovú skratku CTRL+\, prípadne procesu tohto programu pošle signál kill -9. Sofia môže skúsiť priradiť členovi `sa_flags` štruktúry `act` hodnotu SA_RESETHAND.

POSTUP 2:

V tejto časti sa Sofia naučí používať aj ďalšie služby pracujúce so sadou signálov.

Krátke príklady:

1. Blokovanie všetkých signálov okrem SIGUSR1 a SIGUSR2:

```
sigset_t sig_set; // deklarácia sady signálov
sigfillset(&sig_set); // naplnenie sady všetkými signálmi
sigdelset(&sig_set, SIGUSR1); // odstránenie signálu SIGUSR1 zo sady
sigdelset(&sig_set, SIGUSR2); // odstránenie signálu SIGUSR2 zo sady
sigprocmask(SIG_SETMASK, &sig_set, NULL); //zaevidovanie upravenej masky
```

2. Odblokovanie/odstránenie SIGINT, SIGQUIT:

```
sigset_t onesig, old_mask;
sigemptyset(&onesig); // inicializácia sady (jej vyčistenie)
sigaddset(&onesig, SIGINT);
sigaddset(&onesig, SIGQUIT); /* pridanie signálov do sady*/
sigprocmask(SIG_UNBLOCK, &onesig, &old_mask); /* ich odblokovanie,
uchovaniestarej sady v old_mask */
```

2. program - Teraz Sofia vytvorí komplexnejší program, v ktorom najprv povolí a potom zablokuje prijatie signálu SIGUSR1.

Najprv pripojí potrebné hlavičkové súbory a definuje obslužnú funkciu:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void catcher( int sig ){
    printf("som v obsluznej funkcii\n");
}
```

Deklaruje štruktúru `sigaction`, sadu signálov a nastaví štruktúru `sigaction` na obsluhu signálu SIGUSR1. Následne program sám sebe pošle tento signál:

```
int main()
{
    struct sigaction sigact;
    sigset_t sigset;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR1, &sigact, NULL );
    printf("pred prvym poslanim signalu\n");
    kill( getpid(), SIGUSR1 );
```

Teraz Sofia vyčistí štruktúru `sigaction` a pridá do nej len signál SIGUSR1. Nastaví masku signálu na danú sadu. Následne najprv oznamí a potom vykoná opäťovné poslanie signálu:

```
sigemptyset( &sigset );
sigaddset( &sigset, SIGUSR1 );
sigprocmask( SIG_SETMASK, &sigset, NULL );
printf("pred druhym poslanim signalu\n");
```

```

kill( getpid(), SIGUSR1 );
printf("po druhom poslani signalu\n");
return (0);
}

```

Výstup z programu:

```

$ pred prvym poslanim signalu
som v obsluznej funkcii
pred druhym poslanim signalu
po druhom poslani signalu
$
```

3. program - Tento program je modifikáciou predošlého príkladu. Sofia v ňom použije služby `sigfillset()` a `sigdelset()`.

Úvod príkladu je totožný:

```

#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void catcher( int sig ){
    printf("som v obsluznej funkcii\n");
}

int main()
{
    struct sigaction sigact;
    sigset(SIGSETSIGSET, &sigact);
    sigact.sa_mask = 0;
    sigact.sa_handler = catcher;
    sigaction(SIGUSR1, &sigact, NULL );
}

```

Sofia inicializuje sadu `sigset`, do ktorej pridá aj signál `SIGUSR1`, teda aj tento signál bude blokovaný. Nastaví signálovú masku upravenou maskou:

```

sigfillset( &sigset );
sigaddset( &sigset, SIGUSR1 );
sigprocmask( SIG_SETMASK, &sigset, NULL );

```

Následne pošle signál `SIGUSR1`:

```

printf("pred poslanim signalu SIGUSR1\n");
kill( getpid(), SIGUSR1 );

```

Teraz odstráni signál `SIGUSR1` zo sady, takže program už bude zachytávať tento signál:

```

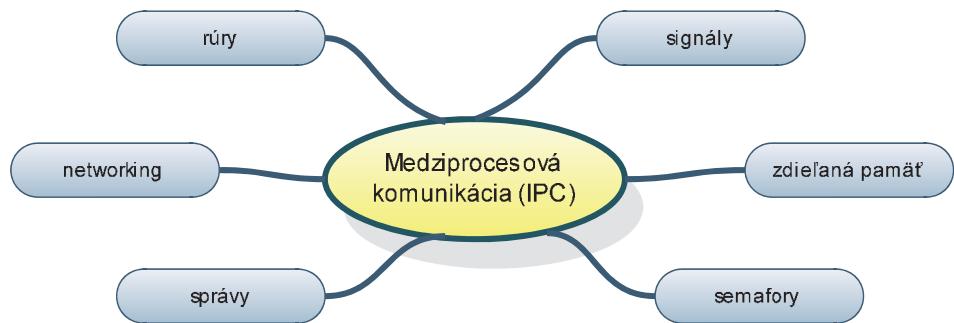
printf("pred odblokovanim signalu SIGUSR1\n");
sigdelset( &sigset, SIGUSR1 );
sigprocmask( SIG_SETMASK, &sigset, NULL );
printf("po odblokovani signalu SIGUSR1\n");
return (0);
}

```

V tomto príklade, ako aj v predošlom, si Sofia musí všimnúť jednu dôležitú skutočnosť. Ked' sa programu pošle signál, ktorého odchytenie je v programe blokované, tento signál sa nestratí, ale bude čakať, pokiaľ sa mu nepovolí odchytenie. Konkrétnie v tomto príklade po odblokovaní signálu SIGUSR1 bol tento signál automaticky programom odchytaný, takže sa hned' zavolala obslužná funkcia.

Výstup z programu:

```
$  
pred poslanim signalu SIGUSR1  
pred odblokovanim signalu SIGUSR1  
som v obsluznej funkcii  
po odblokovani signalu SIGUSR1  
$
```



Téma: Medziprocesová komunikácia – zdieľaná pamäť

	Kľúčové slova	medziprocesová komunikácia, procesy, zdieľaná pamäť
Ciele	Zapamätať si:	základné princípy komunikácie medzi procesmi prostredníctvom zdieľanej pamäte
	Porozumieť:	<ul style="list-style-type: none"> • pojmom stránka pamäte, adresný priestor • pojmom vyhradenie (alokácia), pripojenie, odpojenie, zrušenie (dealkácia) a riadenie • syntax a význam parametrov jednotlivých služieb
	Aplikovať:	služby jadra spojené s komunikáciou cez zdieľanú pamäť
	Viedieť:	<ul style="list-style-type: none"> • uvedomiť si výhody a nevýhody použitia zdieľanej pamäte (napr. oproti iným prostriedkom IPC) • využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	60 minút	
Scenár	Sofia sa v nasledujúcom cvičení zameria na medziprocesovú komunikáciu prostredníctvom zdieľanej pamäte. Sofia bude vytvárať programy, v ktorých bude potrebné zabezpečiť, aby do pamäte pristupovalo viacero procesov, ktoré začnú využívať zdieľanú pamäť na prenos dát medzi sebou.	

POSTUP:

Táto kapitola sa zameriava na:

- **Systémové volania:**
 - `shmat()`
 - `shmget()`
 - `shmdt()`
 - `shmctl()`

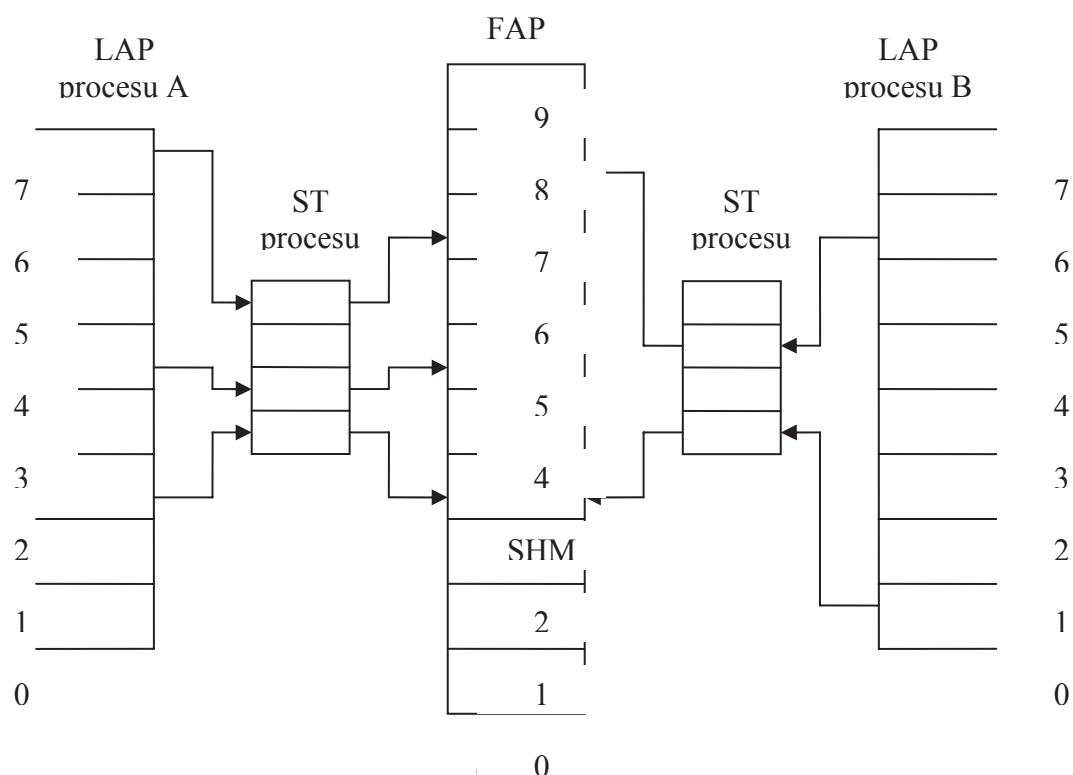
KRÁTKY ÚVOD

KROK 1 – oboznámiť sa s pojmom zdieľaná pamäť:

Zdieľaná pamäť je jedným z nástrojov IPC (Inter Process Communication). Je to veľmi efektívny spôsob odovzdávania dát medzi dvoma nezávislými procesmi. Budeme rozlišovať operačnú pamäť fyzicky prítomnú v počítači - **fyzický adresný priestor** (FAP) a pamäť ako abstrakciu, na ktorú sa jednotlivé programy (a tým aj procesy) odvolávajú pri adresácii - **logický adresný priestor** procesu (LAP).

Zdieľaná pamäť je pamäťový segment (špeciálna skupina rámcov vo FAP alebo odswapovaná na disku). Tento segment je mapovaný do adresných priestorov dvoch alebo viacerých procesov. Proces mapovania pamäťového segmentu do adresného priestoru procesu je nasledovný. Jeden proces vytvorí segment zdieľanej pamäte vo FAP. Procesy si potom môžu tento segment zdieľanej pamäti vo FAP „pripojiť“ ku svojmu vlastnému LAP. To znamená, že rovnaký segment operačnej pamäte vo počítača sa objaví v LAP niekoľkých procesov (pozri Obr. 1).

Ak do zdieľanej pamäte zapíše jeden proces, tieto zmeny budú ihneď viditeľné všetkým ostatným procesom, ktoré pristupujú k rovnakej zdieľanej pamäti. Pri súbežnom prístupe k zdieľaným dátam je potrebné zaistiť synchronizáciu prístupu, pretože zdieľaná pamäť neposkytuje žiadny spôsob synchronizácie. V tomto prípade zodpovednosť za komunikáciu padá na programátora, operačný systém poskytuje len prostriedky pre jej uskutočnenie. Problémy spojené so synchronizáciou prístupu budú podrobnejšie vysvetlené v téme Synchronizácia procesov.



Podtéma: Služby jadra - `shmat()`,`shmget()`,`shmdt()`,`shmctl()`

Kľúčové slova	<code>shmat()</code> , <code>shmget()</code> , <code>shmdt()</code> , <code>shmctl()</code>				
Ciele	Zapamätať si:	syntax služieb - prečítať si manuálové stránky v Unix/Linux, Linux dokumentačný projekt, zdroje na internete			
	Porozumiet:	<ul style="list-style-type: none"> • argumentom služieb • návratovým kódom • chybovým hláseniam 			
	Aplikovať:	služby <code>shmat()</code> , <code>shmget()</code> , <code>shmdt()</code> , <code>shmctl()</code> pre prácu so zdieľanou pamäťou			
	Vedieť:	využiť získané skúsenosti pri tvorbe programov			
Odhadovaný čas	60 minút				
Scenár	Sofia pracuje na zadaní, kde potrebuje vytvoriť segment zdieľanej pamäti na komunikáciu medzi procesmi. Aby mohla využívať zdieľanú pamäť ako medziprocesovú komunikáciu, musí sa oboznámiť so základnými službami, ktoré jej umožnia pracovať so zdieľanou pamäťou.				

POSTUP:

KROK1 - naučiť sa syntax a sémantiku služby jadra `shmget()`:

Sofia začne so službou jadra `shmget()`, ktorá jej pomôže získať identifikátor a vytvoriť zdieľanú pamäť.

Syntax:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Sémantika:

- služba `shmget()` vracia nezápornú celočíselnú hodnotu, čo je identifikátor zdieľanej pamäte, alebo pri chybe služba vráti hodnotu -1.

KROK2 - pochopiť parametre služby:

Parameter `key` predstavuje kľúč (číslo) pomocou ktorého chceme globálne identifikovať blok zdieľanej pamäte v operačnom systéme. V prípade, že takto identifikovaný blok už existuje, a použité príznaky (flagy) a prístupové práva to dovoľujú, je vrátený identifikátor na tento blok zdieľanej pamäte. V prípade, že žiadna zdieľaná pamäť s týmto identifikátorom neexistuje, je vytvorený nový segment zdieľanej pamäte.

Ak chceme zabezpečiť, aby nevznikali konflikty, resp. dať súboru programov, ktoré predpokladáme inštalovať na viacero systémov, prostriedok pre jednoznačné priradenie kľúča pre segment zdieľanej pamäte, je možné použiť službu `ftok()`. Táto služba vráti jednoznačný identifikátor segmentu zdieľanej pamäte na základe špecifikovaného súboru.

Existuje špeciálna hodnota kľúča menom `IPC_PRIVATE`, ktorá vytvorí zdieľanú pamäť prístupnú len danému procesu (nezohľadní sa kľúč). Druhý parameter `size` špecifikuje

počet bajtov požadovanej pamäte. Tretí parameter *shmflg* je tvorený príznakmi. Rozoberieme si dva základne príznaky:

IPC_CREAT - tento príznak znamená, že chceme vytvoriť zdieľanú pamäť. Ak je **IPC_CREAT** použité samostatne, *shmget()* vracia buď identifikátor pre novovytvorený segment, alebo identifikátor segmentu, ktorý už existuje a ma rovnakú hodnotu kľúča (key).

IPC_EXCL - je príznak, ktorý zabezpečí, že pokial' je na daný kľúč už zaregistrovaná zdieľaná pamäť, volanie zlyhá. Používa sa spoločne s príznakom **IPC_CREAT**.

Parameter *shmflg* obsahuje okrem príznakov aj prístupové práva., ktoré sú používané podobným spôsobom, ako pri vytváraní súborov. Pravdaže, tie sa aplikujú len pri vytváraní, ak k nemu dôjde.

 Pre podrobnejšie informácie zadaj príkaz **man 2 shmget**.

KROK3 - naučiť sa syntax a sémantiku služby jadra **shmat():**

Ked' operačný systém pamäť vytvorí, nebude prístupná žiadnemu procesu. Sofia ju sprístupní tak, že ju pripojí do adresného priestoru relevantného procesu. Toto dosiahne pomocou služby *shmat()*.

Syntax:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

Sémantika:

- Ak je volanie služby *shmat()* úspešné, vráti ukazovateľ (smerník) na prvý bajt zdieľanej pamäte, alebo pri chybe vráti hodnotu -1.

KROK4 - pochopiť parametre služby:

Prvý parameter *shm_id* je identifikátor zdieľanej pamäte, ktorý vrátila služba *shmget()*. Druhý parameter *shm_addr* je adresa, na ktorú by sme chceli aby bola zdieľaná pamäť pripojená v LAP. Ak je táto hodnota NULL, operačný systém sám rozhodne, kam zdieľanú pamäť pripoji. Parameter *shmflg* určuje, ako chceme zdieľanú pamäť pripojiť. Možnosti, ktoré sú pre nás zaujímavé sú **SHM_RDONLY** a **SHM_RDN**. V prípade **SHM_RDONLY** je zdieľaná pamäť sprístupnená len pre čítanie.

 Pre podrobnejšie informácie zadaj príkaz **man 2 shmat**.

KROK5 - naučiť sa syntax a sémantiku služby jadra **shmdt():**

Služba *shmdt()* odpojí zdieľanú pamäť od LAP aktuálneho procesu. Ako parameter použije ukazovateľ (smerník) na adresu, ktorý vrátila služba *shmat()*. Odpojením zdieľanej pamäť nie je systémom uvoľnená, ale iba znepristupnená aktuálnemu procesu.

Syntax:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Sémantika:

- služba `shmdt()` po úspešnom vykonaní vráti nulovú hodnotu, alebo pri chybe hodnotu -1.

 Pre podrobnejšie informácie zadaj príkaz `man 2 shmdt`.

KROK6 - naučiť sa syntax a sémantiku služby jadra `shmctl()`:

Poslednou dôležitou službou, ktorú Sofia potrebuje pri tvorbe programov, je služba `shmctl()`. Táto služba slúži na ovládanie zaregistrovaných zdieľaných pamäťí.

Syntax:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

Sémantika:

- služba `shmctl()` po úspešnom vykonaní vráti hodnotu 0 alebo -1, ak nastala chyba.

KROK7 - pochopiť parametre služby:

Prvý parameter služby `shmctl()` *shm_id* je lokálny identifikátor zdieľanej pamäte vrátený službou `shmget()`. Druhý parameter *cmd* je akcia, ktorá sa má vykonať s danou zdieľanou pamäťou. Tento parameter môže nadobúdať jednu z troch nasledujúcich hodnôt:

Hodnota	Popis
<code>IPC_STAT</code>	Skopíruje dátá z jadra OS pre segment zdieľanej pamäte <i>shm_id</i> do štruktúry <code>shmid_ds</code>
<code>IPC_SET</code>	Nastaví hodnoty asociavané so zdieľanou pamäťou podľa údajov obsiahnutých v štruktúre <code>shmid_ds</code> , pokial' na to má daný proces oprávnenie.
<code>IPC_RMID</code>	Zmaže segment zdieľanej pamäte.

Tretí parameter *buf* je smerník na štruktúru, ktorá obsahuje režimy prístupu (`struct shmid_ds`) a prístupové práva (`struct ipc_perm`) k zdieľanej pamäti.

Štruktúra `shmid_ds` obsahuje tieto prvky:

```
struct shmid_ds {
    struct ipc_perm shm_perm;      /* Ownership and permissions */
    size_t          shm_segsz;     /* Size of segment (bytes) */
    time_t          shm_atime;     /* Last attach time */
    time_t          shm_dtime;     /* Last detach time */
    time_t          shm_ctime;     /* Last change time */
    pid_t           shm_cpid;      /* PID of creator */
    pid_t           shm_lpid;      /* PID of last shmat()/shmdt() */
    shmat_t         shm_nattch;   /* No. of current attaches */
    ...
};
```

 Pre podrobnejšie informácie zadaj príkaz `man 2 shmctl`.

KROK8 – aplikovanie služieb v programe:

Ked' sú služby pre prácu so zdieľanou pamäťou Sofii už známe, môže ich využiť pri tvorbe programov. Sofia vytvorí dvojicu programov *shm1.c* a *shm2.c*. Program 2 (konzument), vytvorí segment zdieľanej pamäte a potom zobrazí dátu, ktoré do nej zapíše Program 3 (producent) tento segment pripojí a umožní doňho zapisovať dátu.

Najprv si vytvoríme hlavičkový súbor s názvom *shm_com.h*.

```
#define TEXT_SZ 2048

struct shared_use_st {
    int written_by_you;
    char some_text[TEXT_SZ];
};
```

Tento súbor definuje štruktúru, ktorú budeme využívať v obidvoch programoch. To, že boli do štruktúry zapísané dátá, povieme konzumentovi pomocou celočíselného príznaku *written_by_you* nastaveného na hodnotu 1 producentom. Ak sú dátá prečítané konzumentom, tak sa príznak *written_by_you* nastaví na hodnotu 0.

Program 2 (konzument) vytvorí segment zdieľanej pamäte a potom ho pripojí k svojmu adresnému priestoru. Akonáhle sú k dispozícii nejaké dátá, príznak *written_by_you* je nastavený na hodnotu 1 producentom. Program 2 prečíta ľubovoľný text uložený v premennej *shared_stuff* a zobrazí ho. Nakoniec konzument nastaví príznak *written_by_you* na hodnotu 0, aby naznačil, že dátu prečíta a čaká na ďalšie dátu. Na ukončenie cyklu konzumenta sa použije reťazec „end“. Program 2 potom zdieľanú pamäť odpojí a odstráni ju zo systému.

Po pridaní potrebných hlavičkových súborov pre prácu so segmentom zdieľanej pamäte si zavoláme službu *shmget()*, v ktorej špecifikujeme príznak *IPC_CREAT*

Program 2

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    int shmid;

    srand((unsigned int) getpid());

    shmid = shmget(1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
    //ziskanie identifikator shm
```

```

if (shmid == -1) {
    fprintf(stderr, "shmget failed\n");
    exit(EXIT_FAILURE);
}

```

Teraz sprístupníme zdieľanú pamäť programu:

```

shared_memory = shmat(shmid, (void *)0, 0);      //pripojenie shm
if (shared_memory == (void *)-1) {
    fprintf(stderr, "shmat failed\n");
    exit(EXIT_FAILURE);
}

printf("Memory attached at %X\n", (int)shared_memory);

```

V ďalšej časti programu priradíme segment `shared_memory` do premennej `shared_stuff` a zobrazíme ľubovoľný text uložený v premennej `shared_stuff`. Cyklus bude pokračovať, kým nebude premenná `shared_stuff` obsahovať reťazec `end`. Volanie služby `sleep()` zdrží konzumenta v kritickej sekcií, čo prinúti producenta počkať.

```

shared_stuff = (struct shared_use_st *)shared_memory;
shared_stuff->written_by_you = 0;          //povolime zapis producentovi
while(running) {
    if (shared_stuff->written_by_you) {           //ak mame co citat
        printf("You wrote: %s", shared_stuff->some_text);
        sleep( rand() % 4 );                      //nechame producenta cakat
        shared_stuff->written_by_you = 0;           //povolime zapis producentovi
        if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
            running = 0;             //ukoncenie cyklu konzumenta retazcom end
        }
    }
}

```

Nakoniec zdieľanú pamäť odpojíme a zrušíme:

```

if (shmdt(shared_memory) == -1) {                  //odpojenie shm
    fprintf(stderr, "shmdt failed\n");
    exit(EXIT_FAILURE);
}

if (shmctl(shmid, IPC_RMID, 0) == -1) {           //zrusenie shm
    fprintf(stderr, "shmctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}

```

Program 3 (producent) získa a pripojí rovnaký segment zdieľanej pamäte, pretože použije rovnaký kľúč. Potom požiada používateľa, aby zadal nejaký text. Ak príznak `written_by_you` je nastavený na hodnotu 1, producent vie, že konzument ešte dáta neprečítal a čaká. Až potom, keď konzument príznak `written_by_you` nastaví na hodnotu 0, môže Program 3 zapísať do zdieľanej pamäte nové dátu a príznak

written_by_you znova nastaví na hodnotu 1. Po vložení reťazca „end“ producent skončí a odpojí segment zdieľanej pamäte.

Program 3

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    char buffer[BUFSIZ];
    int shmid;

    shmid = shmget(1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
                           //ziskanie identifikatora shm

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }

    shared_memory = shmat(shmid, (void *)0, 0);           //pripojenie shm
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Memory attached at %X\n", (int)shared_memory);
    shared_stuff = (struct shared_use_st *)shared_memory;
    while(running) {
        while(shared_stuff->written_by_you == 1) {
            sleep(1);           //cakanie na precitanie dat konzumentom
            printf("waiting for client...\n");
        }
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);           //zadanie retazca

        strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
        shared_stuff->written_by_you = 1; //povolime citanie konzumentovi

        if (strcmp(buffer, "end", 3) == 0) {
            running = 0;           //ukoncenie cyklu producenta retazcom end
        }
    }
    if (shmdt(shared_memory) == -1) {                   //odpojenie shm
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

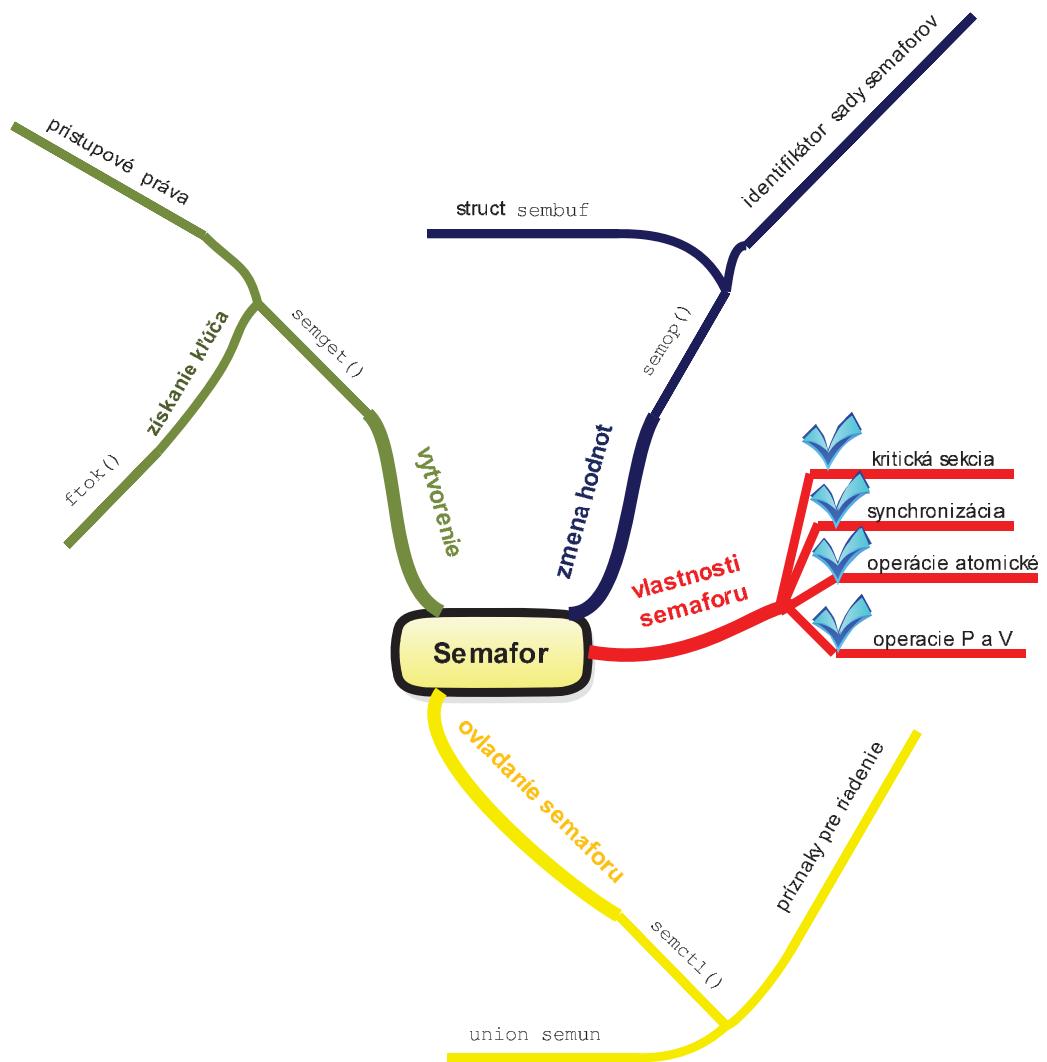
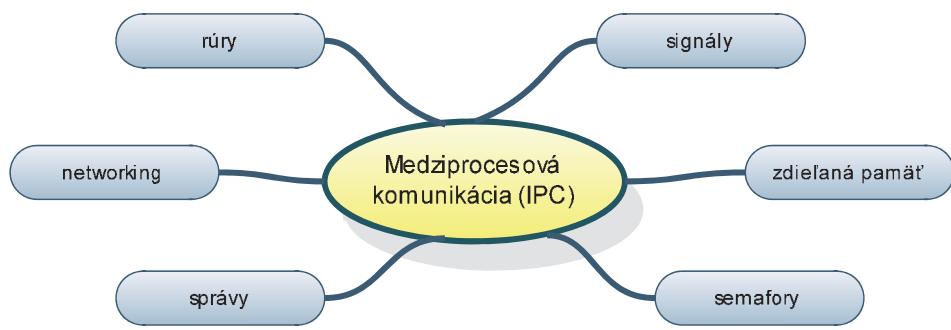
Ked' tieto dva programy spustíme, mali by sme dostať podobný výstup, ktorý sa bude lišiť hodnotou „memory attached“ a použitím nami zadávaných reťazcov slov:

```
$ ./shm1 & ./shm2
Memory attached at 50007000
Memory attached at 50007000
Enter some text: hello
You wrote: hello
waiting for client...
waiting for client...
Enter some text: Linux!
You wrote: Linux!
waiting for client...
waiting for client...
waiting for client...
Enter some text: end
You wrote: end
$
```

Nevýhodou navrhnutého riešenia je, že na synchronizáciu používame vlastný príznak *written_by_you*, ktorý vyžaduje nákladné čakanie (testovanie hodnoty príznaku v nepretržitom cykle a tým aj zbytočné zaťažovanie procesora). V skutočných programoch by sme na synchronizáciu čítania a zapisovania použili mechanizmus posielania správ (bud' prostredníctvom rúry alebo IPC správu), signálov alebo semaforov.

ÚLOHY NA SAMOSTANÚ PRÁCU:

- Vytvorte program, ktorý ilustruje použitie zdieľanej pamäte:
 1. Alokujte segment zdieľanej pamäte, pripojte sa k segmentu a zapíšte doň reťazec znakov.
 2. Odpojte segment zdieľanej pamäte.
 3. Následne znova pripojte segment zdieľanej pamäte, tentokrát na inej adrese, vypíšte reťazec zo zdieľanej pamäte, odpojte segment a dealokujte ho.
- Vytvorte program, ktorý vytvorí zdieľanú pamäť a nový proces-potomok, ktorý zapíše dátu do zdieľanej pamäte zadané z príkazového riadku. Proces rodič prečíta tieto dátu a vypíše na štandardný výstup. Synchronizácia medzi procesmi rodič a potomok sa realizuje pomocou signálov.



Téma: Medziprocesová komunikácia – synchronizácia procesov

Kľúčové slová	Medziprocesová komunikácia, synchronizácia, pasívne čakanie, semafory		
Ciele	Zapamätať si:	<ul style="list-style-type: none"> • techniky synchronizácie procesov • pojem kritická sekcia • synchronizácia procesov vo vzťahu producent – konzument • syntax jednotlivých služieb pre synchronizačný nástroj semafor 	
	Porozumieť:	<ul style="list-style-type: none"> • synchronizácii aktívnym a pasívnym čakaním. • použitiu semaforov v rámci synchronizácie procesov v IPC • argumentom jednotlivých služieb 	
	Aplikovať:	služby jadra týkajúce sa problematiky synchronizácie procesov prostredníctvom semaforov	
	Viedieť:	<ul style="list-style-type: none"> • uvedomiť si výhody použitia semaforov, ale aj ich náročnosť pri implementácii • využiť získané skúsenosti pri tvorbe programov 	
Odhadovaný čas	60 minút		
Scenár	Sofia ma za úlohu vytvoriť procesy, ktoré medzi sebou zdieľajú systémové prostriedky (napr. zdieľanú pamäť) a synchronizovať ich vykonávanie. Je to však rozsiahla problematika a preto sa bude zameriavať na synchronizáciu procesov pomocou semaforov. Ale najprv sa musí oboznámiť s pojмami ako napr. synchronizácia procesov alebo kritická sekcia. Po preštudovaní tejto kapitoly bude schopná vytvárať procesy využívajúce IPC.		

POSTUP:

Táto kapitola sa zameriava na:

- **Systémové volania:**
 - `semget()`
 - `semop()`
 - `semctl()`

KRÁTKY ÚVOD

KROK1 - úvod do synchronizácie procesov:

V OS UNIX/Linux môže súčasne bežať veľa procesov a môže existovať mnoho inštancií jedného programu (jeden program môže byť spustený niekoľkokrát súčasne). Tieto procesy môžu byť navzájom nezávislé alebo beh jedného procesu môže nejakým spôsobom závisieť od behu iného procesu. Môžu sa teda navzájom ovplyvňovať. Proces môže mať vo svojom kóde sekciu, nazvanú **kritická sekcia**, v ktorej môže používať zdieľané prostriedky systému alebo modifikovať zdieľané dátá (spoločné premenné, tabuľky, zdieľané súbory a iné). Tento prístup môže viesť k **nekonzistencii** dát vtedy, keď sa vo svojich kritických sekciách nachádzajú súčasne dva procesy, ktoré pracujú s tými istými spoločnými systémovými prostriedkami.

Príklad:

Rezervačný systém leteniek ma dve používateľské funkcie (z nášho pohľadu transakcie):

a) zrušenie N rezervácií zo dňa X a prevod na deň Y

b) pridanie M rezervácií.

a) `read_item(X)`

$X=X-N$

`write_item(X)`

`read_item(Y)`

$Y=Y+N$

`write_item(Y)`

b) `read_item(X)`

$X=X+M$

`write_item(X)`

V praxi sa vyžaduje vykonať tieto funkcie aspoň čiastočne paralelne. Uskutočníme ich ako transakcie T1 a T2. Samozrejme, že systém môže vykonávať naraz len jednu vnútrotransakčnú operáciu (read, write, výpočet a pod.), pričom je zrejmé, že postupnosť ich vykonávania môže byť rôzna - čo je problém (vznikajú konflikty prístupu k údajom), ktorý musí systém vyriešiť tak, aby stav dát oboch transakcií ostal konzistentný. Príklady možného paralelného vykonania transakcií T1 a T2
The lost update problem (stratená aktualizácia):

T1	T2
<code>read_item(X)</code> $X=X-N$	
	<code>read_item(X)</code> $X=X+M$
<code>write_item(X)</code> <code>read_item(Y)</code>	
	<code>write_item(X)</code>
$Y=Y+N$ <code>write_item(Y)</code>	

Potom je potrebné mať k dispozícii mechanizmy pre synchronizáciu dvoch alebo viacerých procesov navzájom a zabezpečiť odovzdávanie dát medzi procesmi.

Úlohou synchronizácie je zaistíť vzájomné vylúčenie paralelných procesov, ktoré využívajú zdieľané prostriedky. Prakticky to znamená, že sa vykonávanie procesov musí zosúladiť tak, aby sa vykonávanie ich kritických sekcií neprekryvalo v čase. Pri tom sa uplatňujú dva základné princípy:

1. **Synchronizácia aktívnym čakaním:**

- znamená, že sa odsun vstupu do kritickej sekcie uskutoční vložením pomocných (obyčajne prázdnych) inštrukcií do kódu procesu (dekkerov algoritmus, algoritmus pekára).

2. **Synchronizácia pasívnym čakaním:**

- znamená, že sa odsun vstupu do kritickej sekcie uskutoční dočasným pozastavením procesu, kym sa kritická sekcia neuvoľní (semafory, monitory).

KROK2 - oboznámiť sa s pojmom semafor:

Semafor je pasívny synchronizačný nástroj. Vo svojej najjednoduchšej podobe je semafor miesto v pamäti prístupné viacerým procesom. Semafor je celočíselná systémová „premenná“ nadobúdajúca povolené hodnoty $<0, \text{max} (\text{integer})>$, ktorá obmedzuje prístup k zdieľaným prostriedkom OS UNIX/Linux. Je to počítadlo, ktoré sa operáciami nad ním zvyšuje alebo znižuje, avšak nikdy neklesne pod nulu. Synchronizáciu zabezpečujú dve neprerušiteľné operácie P a V (buď sa vykoná celá naraz, alebo sa nevykoná vôbec). Názvy týchto operácií pochádzajú od tvorca semaforov, pána *Edsger Wybe Dijkstra*. V je skratka slova „*verhoog*“, čo znamená v holandčine *zvýšiť*. P je skratka zo zloženého slova „*prolaag*“, čo znamená *skús-a-zniž*. Operácie sú definované nasledovne:

- Operácia P sa pokúša odpočítať hodnotu jedna od hodnoty semaforu. Ak je hodnota semaforu väčšia ako 0, operácia sa vykoná. Ak je hodnota semaforu 0, operácia sa vykonat' nedá a proces zostane zablokovaný, pokial' iný proces nezvýši hodnotu semaforu.
- Operácia V zvýši hodnotu semaforu a môže spôsobiť odblokovanie zablokovaného procesu.

Podtéma: Služby jadra - semget()

Kľúčové slova:	semget()				
Ciele:	Zapamätať si:	syntax služby - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete			
	Porozumieť:	<ul style="list-style-type: none"> • argumentom služby • návratovým hodnotám • chybovým hláseniam 			
	Aplikovať:	službu <code>semget()</code> pri vytváraní synchronizačného nástroja semafor			
	Viedet:	využiť získané skúsenosti pri tvorbe programov			
Odhadovaný čas	10 minút				
Scenár	Aby Sofia mohla využívať semafor pri riešení svojej úlohy na synchronizáciu procesov v medziprocesovej komunikácii, musí si ho najprv vytvoriť. Z manuálu vyčítala, že na vytvorenie semaforu sa používa služba <code>semget()</code> a preto potrebuje sa ju naučiť používať.				

POSTUP:

KROK1 - naučiť sa syntax a sémantiku služby jadra `semget()`:

Služba `semget()` vytvorí novú sadu semaforov a vráti ich identifikátor (ten používajú ďalšie služby pre prácu so semaformi).

Najprv si objasníme ako sú semafory vnímané v OS UNIX/Linux. Semaforová sada (tzv. set alebo pool) je určitý počet semaforov, ktorý je identifikovaný unikátnym identifikátorom. V takejto sade môžeme mať viacero semaforov a identifikujete ich poradím (začínajúc od nuly). Operácie sa vykonávajú atomicky nad celou sadou (buď sa vykonajú všetky požadované, alebo sa nevykonajú vôbec).

Syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key, int num_sems, int sem_flags);
```

Sémantika:

- Služba `semget()` vracia identifikátor sady semaforov (nezáporné celé číslo), alebo `-1` pri chybe.

KROK2 - pochopiť parametre služby:

Prvý parameter `key` je celočíselná hodnota, ktorá umožňuje nezávislým procesom pristupovať k rovnakej sade semaforov. Podľa kľúča OS vytvorí sadu semaforov alebo použije existujúcu s rovnakým kľúčom. Existuje špeciálna hodnota kľúča semaforu `IPC_PRIVATE`, ktorá vytvorí sadu semaforov, ku ktorej môže pristupovať iba proces, ktorý ju vytvoril. Tento identifikátor musí tento proces doručiť priamo procesom ktoré ho potrebujú, väčšinou ide o ním vytvorené dcérské procesy (proces potomok).

Parameter `num_sems` reprezentuje počet semaforov v sade. Parameter `sem_flags` je množina príznakov, ktoré sú podobné príznakom služby `open()`. Špecifikuje prístupové práva k sade semaforov, ktoré fungujú ako prístupové práva k súboru. Navyše, môžu byť bitovo spočítané s hodnotou `IPC_CREAT`, ktorá zaistí vytvorenie novej sady semaforov. Nie je chybou nechať príznak `IPC_CREAT` nastavený a odovzdať službe kľúč existujúcej sady semaforov. Keď nie je potreba, ostáva príznak `IPC_CREAT` ignorovaný. Pomocou príznakov `IPC_CREAT` a `IPC_EXCL` je možné získať novú jedinečnú sadu semaforov. Ak už taká sada semaforov existuje, služba `semget()` vráti chybu.

 Pre podrobnejšie informácie zadaj príkaz `man 2 semget`.

KROK3 – aplikovanie služby v programe:

Sofia má za úlohu urobiť program, ktorý má vytvoriť tri sady semaforov s týmito nastavenými príznakmi `IPC_CREAT|IPC_EXCL`, `IPC_PRIVATE` a prístupovými pravami nastavenými na hodnotu `0666`. K získaniu kľúča pre službu `semget()` sa použije služba `ftok()` a príznak `IPC_CREAT`.

```
#include <stdio.h>
#include <sys/sem.h>

int main(void)
{
    int sem1, sem2, sem3;
    key_t ipc_key;

    ipc_key = ftok(".", 'S');      //ziskanie kluca pre sluzbu semget()
    if ((sem1 = semget(ipc_key, 3, IPC_CREAT | 0666)) == -1) {
        perror("semget: IPC_CREAT | 0666");
        }                                //vytvori sadu semaforov s pristupovymi pravami
    printf("sem1 identifikator: %d\n", sem1);

    if ((sem2 = semget(ipc_key, 3, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
        perror("semget: IPC_CREAT | IPC_EXCL | 0666");
        }
    //vytvori sadu semaforov ak uz exituje tak sluzba semget() vrati chybu
    printf("sem2 identifikator: %d\n", sem2);

    if ((sem3 = semget(IPC_PRIVATE, 3, 0666)) == -1) {
        perror("semget: IPC_PRIVATE");
        } //vytvori jedinecnu sadu semaforov pre proces ktory ju vytvoril
    printf("sem3 identifikator: %d\n", sem3);
    return 0;
}
```

Spustením predchádzajúceho programu Sofia získala dva identifikátory na sadu semaforov, ktoré sa vypíšu na štandardný výstup. Po ukončení programu vytvorené sady semaforov ostanú v systéme. Môže použiť príkaz `ipcs -s`, ten jej ukáže aktuálny stav vytvorených semaforov. Ak Sofia potrebuje odstrániť semafor zo systému, môže použiť príkaz `ipcrm -s sem_id`, kde `sem_id` je identifikátor sady semaforov.

Podtéma: Služby jadra - semctl()

Kľúčové slova:	semctl()				
Ciele:	Zapamätať si:	syntax služby - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete			
	Porozumieť:	<ul style="list-style-type: none"> • argumentom služby • návratovým hodnotám • chybovým hláseniam 			
	Aplikovať:	službu <code>semctl()</code> pri inicializácii a pri práci so semaforom			
	Viedet:	využiť získané skúsenosti pri tvorbe programov			
Odhadovaný čas	15 minút				
Scénár	Sofia už si vie vytvoriť sadu semaforov. Aby ju mohla využívať pre svoje procesy a pre riešenie zadanej úlohy, musí ju najprv inicializovať na hodnoty, ktoré potrebuje pre synchronizáciu procesov. Zistila, že na vyriešenie tohto problému sa používa služba <code>semctl()</code> .				

POSTUP:

KROK1 - naučiť sa syntax a sémantiku služby jadra `semctl()`:

Služba `semctl()` inicializuje (nastaví), alebo prečíta hodnoty semaforov zo sady semaforov alebo prípadne sadu semaforov odstráni zo systému.

Syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int sem_id, int sem_num, int command, ...);
```

Sémantika:

- Služba `semctl()` vracia rôzne hodnoty v závislosti na parametri `command`. V prípade hodnôt `SETVAL` a `IPC_RMID` vracia po úspešnom vykonaní 0, alebo pri chybe -1.

KROK2 - pochopiť parametre služby:

Prvý parameter `sem_id` je identifikátor sady semaforov, ktorý získame službou `semget()`. Parameter `sem_id` určuje, s ktorou sadowou semaforov sa má pracovať. S tým je spojený druhý parameter `sem_num`, ktorý naopak určuje, s ktorým semaforam z danej sady sa má pracovať (začínajúc od NULY). Parameter `command` špecifikuje akciu, ktorá sa má vykonať. Štvrtý parameter, ak je prítomný, je definovaný ako `union semun`, ktorý musí obsahovať minimálne nasledujúce prvky:

```
union semun {
    int val; /* hodnota SETVAL */
    struct semid_ds *buf; /* výrovnávacia pamäť pre IPC_STAT
                           a IPC_SET */
    unsigned short *array; /* pole pre GETALL, SETALL */}
```

Parameter *command* môže v službe `semctl()` nadobúdať rôzne hodnoty. My si uvedieme dve najpoužívanejšie z nich:

- `SETVAL` – slúži k inicializácii semafora určitou hodnotou. Požadovaná hodnota je odovzdaná ako prvok `val` štruktúry `union semun`. Semafor je potrebné nastaviť ešte pred prvým použitím.
- `GETVAL` – slúži na zistenie nastavenej hodnoty semaforu.
- `IPC_RMID` – slúži na zmazanie sady semaforov, keď už nie je potrebná.

 Pre podrobnejšie informácie zadaj príkaz `man 2 semctl`.

KROK3 – aplikovanie služby v programe:

Sofia dostala za úlohu vytvoriť sadu semaforov s troma semaformi a s prístupovými právami 0666. Nastaviť ich na tieto hodnoty 3,4,1 a vypísat čas a dátum vytvorenia sady semaforov pomocou služby `semctl()`.

```
#include <stdio.h>
#include <sys/sem.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    int sem_id, sem_value, i;
    key_t ipc_key;
    struct semid_ds sem_buf;
    static ushort sem_array[] = {3,1,4};
    ipc_key = ftok(".", 'S');
    if ((sem_id = semget(ipc_key, 3, IPC_CREAT | 0666)) == -1) {
        perror("semget: IPC_CREAT | 0666"); exit(1);
    }                                         //vytvorenie sady semaforov
    printf("Semafor ID: %d\n", sem_id);
    if (semctl(sem_id, 0, IPC_STAT, &sem_buf) == -1) {
        perror("semctl:IPC_STAT"); exit(2);
    }                                         //poskytne informacie o vytvorenjej sade semaforov
    printf("Vytvorený %s", ctime(&sem_buf.sem_ctime));
                                         //inicializacia sady semaforov
    if (semctl(sem_id, 0, SETALL, sem_array) == -1) {
        perror("semctl: SETALL"); exit(3);
    }
    for (i = 0; i < 3; ++i){                  //zobrazí hodnoty semaforov
        if ((sem_value = semctl(sem_id, i, GETVAL)) == -1) {
            perror("semctl: GETVAL"); exit(4);
        }
        printf("Semafor %d má hodnotu %d\n", i, sem_value);
    }
    if (semctl(sem_id, 0, IPC_RMID) == -1) {
        perror("semctl: IPC_RMID"); exit(5);
    }                                         //odstrani sadu semmaforov
    return 0;
}
```

Podtéma: Služby jadra - semop()

Kľúčové slova:	semop ()		
Ciele:	Zapamätať si:	syntax služby - prečítať si manuálové stránky v Unixe/Linuxe, Linux dokumentačný projekt, zdroje na internete	
	Porozumieť:	<ul style="list-style-type: none"> • argumentom služby • návratovým hodnotám • chybovým hláseniam 	
	Aplikovať:	službu <code>semop()</code> pri operáciach vykonávaných nad semaformami	
	Viedieť:	využiť získané skúsenosti pri tvorbe programov	
Odhadovaný čas	60 minút		
Scenár	Sofia pokračuje v riešení svojej úlohy. Potrebuje vykonať operáciu nad ňou vytvorenou a inicializovanou sadou semaforov. Aby zabezpečila synchronizovaný prístup do kritickej sekcie každého svojho procesu, musí sa naučiť efektívne používať službu jadra <code>semop()</code> .		

POSTUP:

KROK1 - naučiť sa syntax a sémantiku služby jadra `semop()`:

Služba `semop()` je používaná na vykonávanie operácií P a V nad sadou semaforov definovaných v parametri `sem_id`.

Prv si povieme ako `semop()` narába s množinou operácií. V prípade, že nie je príznakmi v operácii povedané inak, všetky operácie sa vykonajú atomicky a vykonajú sa len vtedy, ak je možné ich všetky vykonať. V prípade, že to možné nie je, `semop()` uspí volajúci proces.

Syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop (int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

Sémantika:

Doplňte návratové hodnoty služby `semop()`:

- služba `semop()` vracia _____, pri chybe _____

KROK2 - pochopiť parametre služby:

Prvý parameter `sem_id` je identifikátor sady semaforov, ktorý vráti služba `semget()`. Druhý parameter `sem_ops` je ukazovateľ na pole štruktúry, ktorá obsahuje aspoň tieto prvky:

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

Prvý člen záznamu (štruktúry) `sem_num` je číslo semaforu, nad ktorým sa má zo sady semaforov urobiť daná operácia. Druhý člen záznamu `sem_op` je hodnota alebo číslo (určujúce typ operácie), na ktorú má byť semafor zmenený. Hodnotu semaforu môže meniť o viac než 1. Často sa však používajú iba dve hodnoty:

- -1 zníženie hodnoty semaforu – P operácia
- +1 zvýšenie hodnoty semafory – V operácia

Operácie P a V vyjadrujeme pomocou číselnej hodnoty druhého člena záznamu `sem_op` štruktúry `sembuf`:

- Ak $sem_op > 0$ – jadro použije túto hodnotu na zvýšenie hodnoty semafora a odblokuje procesy, ktoré čakajú na zvýšenie hodnoty semafora (operácia V).
- Ak $sem_op = 0$ – jadro kontroluje hodnotu semafora, pokiaľ nie je nulová, zvýši počet procesov čakajúcich na nulovú hodnotu semafora a proces zablokuje.
- Ak $sem_op < 0$ – absolútна hodnota je rovná hodnote semafora alebo je menšia jadro pripočítá túto hodnotu (hodnota semafora je znížená – operácia P). Ak sa potom hodnota semafora rovná 0, jadro aktivuje všetky zablokované procesy, čakajúce na nulovú hodnotu semafora.
- Ak $sem_op < 0$ – absolútна hodnota je väčšia a alebo rovná hodnote semafora, jadro proces zablokuje.

Posledný člen záznamu štruktúry `sem_flg` obsahuje jeden z dvoch príznakov SEM_UNDO alebo IPC_NOWAIT.

- SEM_UNDO – operácie, ktoré sú vykonané s týmto príznakom sú navrátené po ukončení procesu. Táto operácia má zabezpečiť prípadné uchovanie konzistencie semaforu v prípade, že proces sa ukončí počas vykonávania kritickej sekcie; umožní operačnému systému tento semafor automaticky uvoľniť.
- IPC_NOWAIT – v prípade, že sa medzi operáciami narazí na operáciu, ktorá by vyžadovala uspatie procesu (nemôže byť totiž vykonaná) a zároveň má daná operácia príznak IPC_NOWAIT, tak funkcia neuspí volajúci proces, ale len vráti chybovú hodnotu -1.

Posledný parameter `num_sem_ops` je počet prvkov štruktúry `sembuf` (semaforov v poole), nad ktorými sa vykonávaju operácie P a V.

 Pre podrobnejšie informácie zadaj príkaz `man 2 semop`.

Podtéma: Príklad

KROK1 – aplikovanie služieb v programe:

Pre experimentovanie so semaformi použijeme nasledujúce programy *sem1.c* a *sem2.c*. Program *sem1.c* vytvorí, inicializuje a odstráni sadu semaforov. Pre riešenie nášho problému využijeme iba jednosemaforovú sadu (jednosemaforový pool). Program *sem2.c* využije sadu semaforov vytvorenú programom *sem1.c*. K indikácii vstupu a výstupu z kritickej sekcie programu použijeme dva rôzne znaky. Program *sem1.c* zobrazí pri vstupe i opustení kritickej sekcie programu znak X a program *sem2.c* zobrazí pri vstupe a opustení kritickej sekcie programu znak O. Pretože by mal do kritickej sekcie programu mať prístup vždy iba jeden proces, mali by sa všetky znaky X a O na výstupe objaviť v pároch. Pre striedanie prístupu do kritickej sekcie programu *sem1.c* a *sem2.c* využívajú službu *sleep()*, ktorá pozastaví vykonávanie programu.

Program *sem1.c*

```
#include <stdlib.h>                                // potrebne hlavičkové súbory
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun
{
    int val, arg1, arg2;      /* hodnota pre SETVAL */
    struct semid_ds *buf;    /* buffer pre IPC_STAT, IPC_SET */
    unsigned short int *array; /* pole pre GETALL, SETALL */
    struct seminfo *__buf;   /* buffer pre IPC_INFO */
};

static int set_semvalue(void);           //prototypy funkcií
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);
static int sem_id;                      // globálna premenná.

int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'X';
    srand((unsigned int) getpid()); //Nastaví generátor nahodných ciel
    arg1 = atoi(argv[1]);          //ziskame argumenty odovzdane programu
    arg2 = atoi(argv[2]);          //konvertujem string na integer

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
    if(sem_id== -1){
        perror("semget()");
        exit(EXIT_FAILURE);
    }
    printf("Program sem1.c ID semaforu: %d\n", sem_id);
    if (!set_semvalue()) {           //inicializacia semafora
        fprintf(stderr, "Failed to initialize semaphore\n");
        exit(EXIT_FAILURE);
    }
    sleep(2);
}
```

Cyklus 10 krát vkrocí do kritickej sekcie programu a náhodne počká. Funkcia *semaphore_p()* nastaví semafor na čakanie.

```

for(i = 0; i < 10; i++) {
    if (!semaphore_p()) exit(EXIT_FAILURE);
    printf("%c", op_char);fflush(stdout);
    pause_time = rand() % arg1;
    sleep(pause_time);
    printf("%c", op_char);fflush(stdout);
}

```

Po skončení kritickej sekcie voláme funkciu semaphore_v(), ktorá semafor nastaví na voľno, potom sa náhodne čaká a pokračuje ďalšou iteráciou cyklu. Nakoniec voláme del_semvalue na odstránenie ID semafora.

```

if (!semaphore_v()) exit(EXIT_FAILURE);
pause_time = rand() % arg2;
sleep(pause_time);
}
sleep(6);
printf("\n%d - finished\n", getpid());
del_semvalue();
exit(EXIT_SUCCESS);
}

```

Funkcia set_semvalue inicializuje semafor pomocou príkazu SETVAL, ktorý je odovzdaný ako parameter *command* služby semctl(). Je to nutné pred prvým použitím semaforu.

```

static int set_semvalue(void){
    union semun sem_union;
    sem_union.val = 1;
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
    return(1);
}

```

Funkcia del_semvalue má skoro rovnaký tvar, ale služba semctl() používa príkaz IPC_RMID, ktorý odstráni ID semaforu:

```

static void del_semvalue(void){
    if (semctl(sem_id, 0, IPC_RMID) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}

```

Funkcia semaphore_p zmení hodnotu semaforu na -1 (čakanie):

```

static int semaphore_p(void)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1;
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}

```

Funkcia `semaphore_v` nastavuje `sem_op` na 1, takže semafor sa stane opäťovne prístupným.

```
static int semaphore_v(void)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = 1;
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

Program `sem2.c`

```
#include <stdlib.h>                                // potrebne hlavickove subory
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>

static int semaphore_p(void);
static int semaphore_v(void);
static int sem_id;                                    // globalna premenna.

int main(int argc, char *argv[])
{
    int i, arg1, arg2;
    int pause_time;
    char op_char = 'O';
    srand((unsigned int) getpid()); //Nastaví generator náhodných čísel
    arg1 = atoi(argv[1]);          //ziskame argumenty odovzdane programu
    arg2 = atoi(argv[2]);          //konvertujem string na integer

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
    if(sem_id== -1){
        perror("semget()");
        exit(EXIT_FAILURE);
    }
    printf("Program sem2.c ID semaforu: %d\n", sem_id);
    sleep(1);

    for(i = 0; i < 10; i++) {
        if (!semaphore_p()) exit(EXIT_FAILURE);
        printf("%c", op_char);fflush(stdout);
        pause_time = rand() % arg1;
        sleep(pause_time);
        printf("%c", op_char);fflush(stdout);
        if (!semaphore_v()) exit(EXIT_FAILURE);
        pause_time = rand() % arg2;
        sleep(pause_time);
    }
    sleep(6);
    printf("\n%d - finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Funkcia `semaphore_p` zmení hodnotu semaforu na -1 (čakanie):

```
static int semaphore_p(void)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1;
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```

Funkcia `semaphore_v` nastavuje `sem_op` na 1, takže semafor sa stane opäťovne prístupným.

```
static int semaphore_v(void)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = 1;
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

Toto je príklad výstupu, ktorý získame, keď spustíme program `sem1.c` a `sem2.c` súčasne v jednom termináli s parametrami 3 a 2 pre službu `sleep()`:

```
$ ./sem1 3 2 & ./sem2 3 2

Program sem2.c ID semaforu: 229377
Program sem2.c ID semaforu: 229377

OOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXX

1083 - finished
1082 - finished
$
```

Je vidieť, že znaky X a O sú spárované a striedajú sa, čo znamená, že kritické sekcie programov boli spracované správne.¹⁰ Ak by sa sparované znaky X a O (dvojice) nestriedali pravidelne, môže to byť spôsobené vyťažením systému v danom momente spustenia programov v systéme. Striedanie spárovaných znakov X a O, čiže prístup do

¹⁰ Ak chcete uvedený program spustiť a nefunguje vám, skúste pred spustením programu zadať príkaz `stty -testop`, ktorý zabráni programu na pozadí generujúcemu výstup `tty`, aby generoval signál.

kritickej sekcií programov, môžeme ovplyvňovať pomocou parametrov odovzdaných programu *sem1.c* a *sem2.c*.

KROK2 - ako to funguje:

Program *sem1.c* a *sem2.c* najprv získajú identifikátor semaforu službou `semget()`. Klúč pre túto službu bol zvolený programátorom a príznak `IP_CREAT` zaistí vytvorenie semaforu, ak to bude nutné. Program *sem1.c* je zodpovedný za inicializáciu semafora, čo vykoná pomocou funkcie `set_semvalue()`, ktorá poskytuje zjednodušené rozhranie všeobecnejšej služby `semctl()`. Program *sem1.c* počká s odstránením semaforu, kým neskončí program *sem2.c*. Pokiaľ by semafor neboli zmazaný, existoval by v systéme, i keď by ho žiadne iné programy nepoužívali.

Program *sem1.c* a *sem2.c* potom vykoná desať iterácií cyklu, pričom v kritických a nekritických sekciách počká náhodne dlho. Kritická sekcia je strážená funkciami `semaphore_p()` a `semaphore_v()`, ktoré tvoria zjednodušenie všeobecnejšej služby `semop()`. Služba `sleep()` slúži tiež k tomu, aby bolo možné spustiť program *sem2.c* skôr, ako *sem1.c* vykoná väčšie množstvo cyklov. Pomocou funkcie `rand()` je v programe získavané pseudonáhodné číslo. Generátor je inicializovaný funkciou `srand()`.

ÚLOHY NA SAMOSTANÚ PRÁCU:

Príklad č.1:

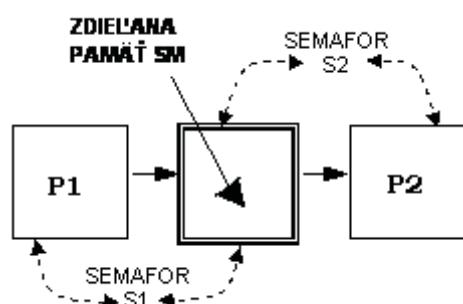
Majme dva programy, program A a program B, ich telá tvorí nasledujúci kód:

```
static int i; //zdielana premenna  
  
//proces A                                // proces B  
i=0;                                         i=0;  
while(i < 100){                            while(i > -100){  
    i++;                                     i++;  
}                                            }  
printf("vyhral to A");                      printf("vyhral to B");
```

Zodpovedzte nasledujúce otázky:

- Ktorý proces vyhrá?
- Skončí sa niekedy táto „súťaž“?
- Ak jeden skončí, skončí zároveň aj druhý?
- Pomôže, ak program A spustíme ako prvý?

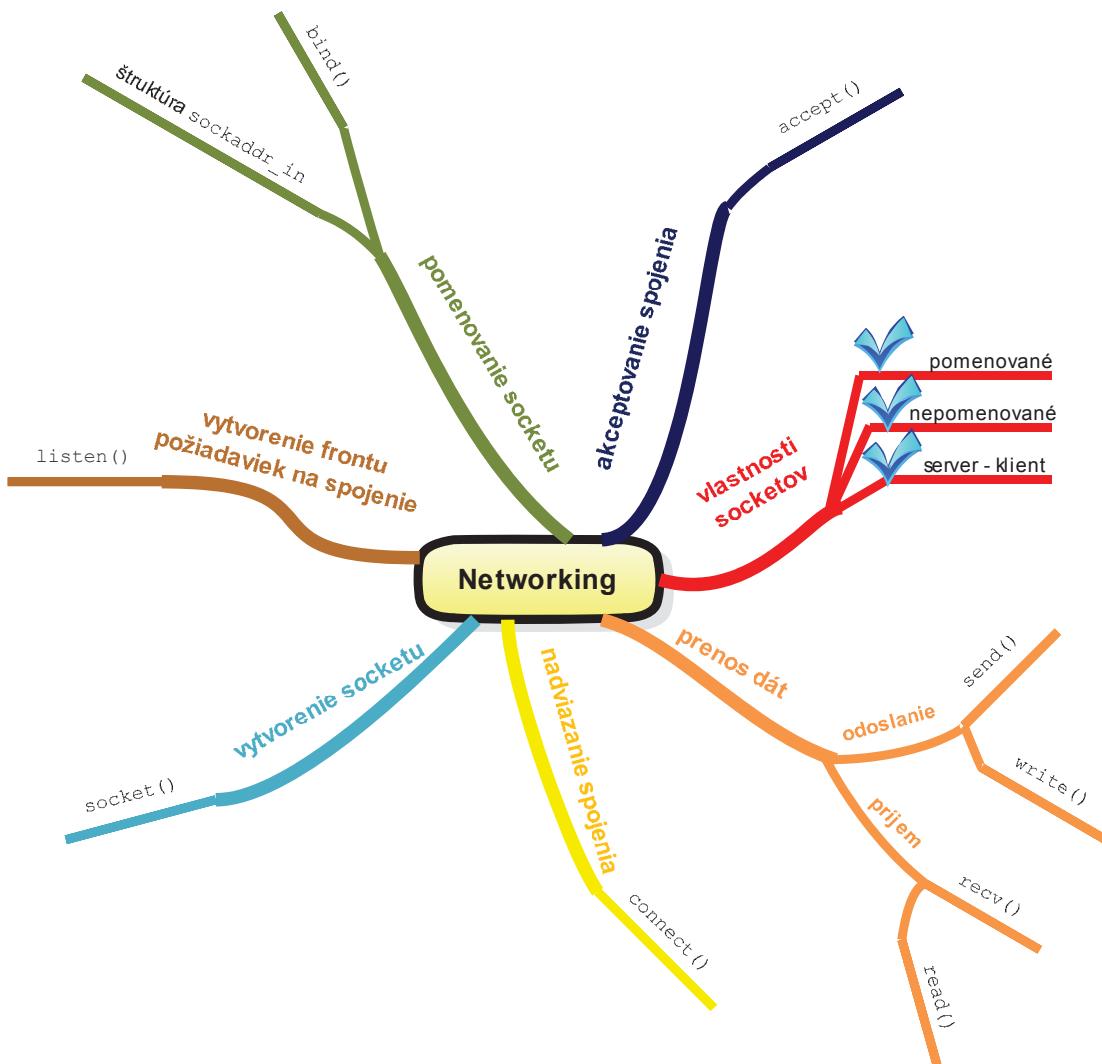
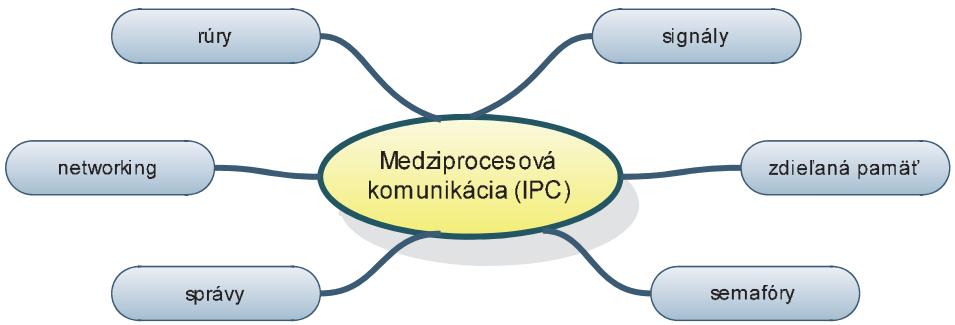
Príklad č.2:



Majme nasledujúcu schému. Vytvorte dva programy, ktoré budú čítať a zapisovať do zdieľanej pamäte. Nech program p1, zapisuje do zdieľanej pamäte slová zo vstupu a program p2 ich číta a vypisuje na výstup. Na vzájomnú synchronizáciu použite semafory.

Príklad č.3:

V tejto úlohe máme spolupracujúce procesy, ktoré komunikujú cez vyrovnanávajúcu pamäť obmedzenej veľkosti. Jedna skupina procesov produkuje informácie a vkladá ju do vyrovnanávacej pamäte, odkiaľ ju druhá skupina procesov vyberá. Aby mohli obidve skupiny procesov prebiehať paralelne, ich vykonávanie sa musí zosynchronizovať, t. j. producent musí mať vždy voľné miesto vo vyrovnanávacej pamäti pre uloženie dát a konzument musí mať vždy pripravené dátá na výber. Ak tomu tak nie je, proces, ktorý nemôže pokračovať v činnosti, musí počkať - producent na uvoľnenie miesta vo vyrovnanávacej pamäti, konzument na uloženie dát.



Téma: Sokety - siet'ová komunikácia

Kľúčové slová	klient, server, socket, medziprocesová komunikácia				
Ciele	Zapamätať si:		<ul style="list-style-type: none"> • základné princípy komunikácie medzi procesmi prostredníctvom socketov • model klient/server • syntax jednotlivých služieb 		
	Porozumieť:	priebehu komunikácie pomocou socketov			
	Aplikovať:	služby jadra spojené s komunikáciou cez sockety			
Vedieť:		<ul style="list-style-type: none"> • nadviazať spojenie pomocou socketov • využiť získané skúsenosti pri tvorbe programov 			
Odhadovaný čas	60 min				
Scenár	<p>Sofia dostala za úlohu vytvoriť procesy, ktoré by komunikovali v rámci počítačovej siete pomocou protokolov spojovanej služby. Až doteraz sa Sofia spoliehala na zdieľané zdroje systému jedného počítača. Jeden proces má plniť rolu servera a druhý rolu klienta. Pri analýze úlohy musí zistiť, aké ma použiť služby jadra na strane procesu-servera a procesu-klienta pri použití spojovanej komunikácie. Pri riešení tejto úlohy má použiť komunikačný nástroj <i>socket</i>, ktorý umožňuje procesom komunikovať prostredníctvom počítačovej siete.</p>				

POSTUP:

Táto kapitola sa zameriava na:

- **Systémové volania:**
 - `socket()`
 - `bind()`
 - `listen()`
 - `accept()`
 - `connect()`

KRÁTKY ÚVOD

Komunikačný mechanizmus *socket* je obojsmernou komunikačnou technológiou. Umožňuje komunikáciu medzi procesmi na tom istom počítači (tzv. sockety unixovej domény) alebo s procesmi vykonávanými na iných počítačoch prostredníctvom počítačovej siete (tzv. sockety internetovej domény). Možno s ním pracovať ako so súborom, má pridelený vlastný jedinečný deskriptor.

Model Klient – Server

Jedným zo základných modelov pre komunikáciu medzi procesmi prostredníctvom socketov je model klient–server. Tento model je založený na existencii dvoch typov procesov: procesu-servera a procesu-klienta. Proces-server vykonáva pasívnu úlohu - čaká na požiadavky od klientských procesov, ktorým poskytuje nejakú „službu“. Proces klient vykonáva aktívnu úlohu na tom istom počítači alebo na inom počítači. Je to proces odosielajúci požiadavky na spojenie a využívajúci služby procesu server. Klienti, ktorí spolupracujú s jedným typom servera, môžu byť rôzneho typu a môžu sa navzájom lísiť používateľským prostredím.



Obr.1

Server sám inicializuje a následne pozastaví svoju činnosť dovtedy, kým nepríde požiadavka zo strany klienta (pozri Obr. 1). Server a klient vzájomne kooperujú pri riešení jednotlivých úloh. Procesy typu klient väčšinou inicializuje používateľ. Je dôležité pochopíť, že nie počítač určuje, kto je klient a kto je server, ale proces, ktorý využíva sockety. Spolupráca klienta so serverom je zabezpečená prostredníctvom komunikačného systému a protokolov počítačových sieti. Komunikačný systém pozostáva z týchto časťí:

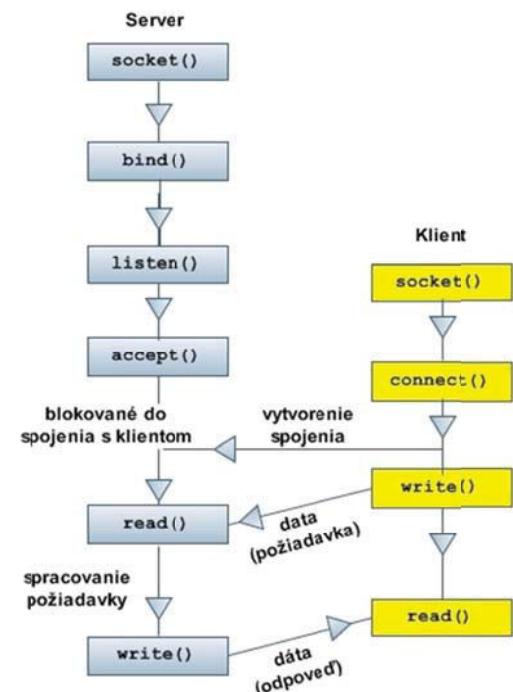
- *IP adresa* je adresa stroja (vzťahuje sa na jeho sietové rozhranie), kde je komunikujúci proces vykonávaný. Pomocou nej dokážu s týmto procesom komunikovať iné procesy v rámci počítačovej siete. IP adresu tvoria štyri bajty, interpretované ako 32-bitové celé číslo (IPv4; novší protokol IPv6 rozširuje IP adresu na 16 bajtov).
- *Port* je celočíselný identifikátor komunikujúceho procesu, na ktorom sú vybavované požiadavky procesov (na strane servera sa zvyčajne používajú tzv. známe porty (angl. well known) do 1024 a na strane klienta sa využívajú dynamicky pridelované od 1024).

Súbor protokolov **TCP/IP** (Transmission Control Protocol/Internet Protocol) je určený pre prepájanie heterogénnych sieti, t.j. sieti rôznych ako po stránke technickej, tak aj programovej. Protokol TCP/IP pozostáva zo skupiny protokolov, z ktorých pre prácu so

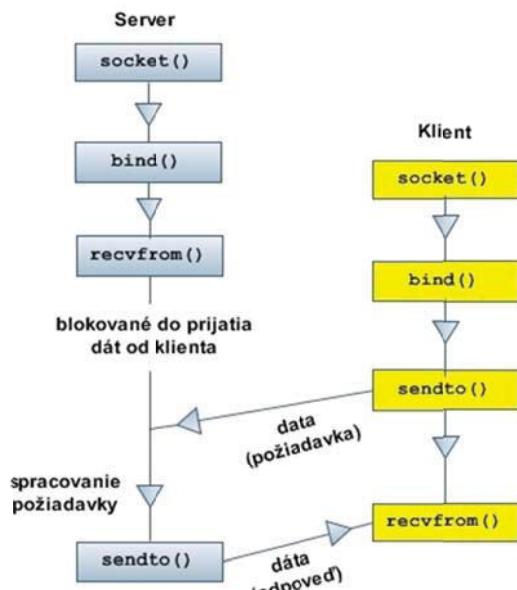
socketmi na štvrtej vrstve TCP/IP modelu sa využívajú protokoly uvedené v nasledujúcej tabuľke.

TCP/IP protokoly	
Spojované služby (protokoly TCP)	Nespojované služby (protokoly UDP)
 <ul style="list-style-type: none"> • Jeden partner zavolá • Druhý akceptuje volanie • Spojenie je vytvorené • Začína komunikácia • Jeden z partnerov ukončí komunikáciu uzavorením spojenia • Synchrónne • Okamžité potvrdenie • Rézia vytvorenia spojenia • Stavové 	 <ul style="list-style-type: none"> • Odosielateľ napíše list • Na obálku napiše adresu prijímateľa • Doručí list na poštu • Prijímateľ obdrží list (?) • Asynchronné • Zásadná neurčitosť • Rézia smerovania každej správy • Bezstavové

Soketové služby pre spojované protokoly (TCP)



Soketové služby pre nespojované protokoly (UDP)



Na strane procesu server (protokol TCP) musíme na rozdiel od procesu klient priradiť socketu adresu (službou `bind()`). Potom musíme vytvoriť front do ktorého sa budú ukladať požiadavky na spojenie (službou `listen()`). Požiadavky na spojenie musíme z frontu vyberať postupne (služba `accept()`). Ak vo fronte nie je žiadna požiadavka na

spojenie, proces server počká (bude uspatý), kým nejaká požiadavka nedôjde. Služba `accept()` nám vráti nový socket, pomocou ktorého budeme komunikovať s procesom-klientom, ktorý sa pripája na proces server systémovým volaním `connect()`.

Podtéma: Služba jadra - socket()

Kľúčové slová	socket(), man socket	
Ciele	Zapamätať si:	syntax služby socket(): <ul style="list-style-type: none"> zdroje na internete: http://www.rt.com/man/socket.2.html http://linux.die.net/man/2/socket
	Porozumieť:	<ul style="list-style-type: none"> využitiu služby socket() parametrom služby socket()
	Aplikovať:	službu socket() pri vytvorení socketu
	Viedieť:	<ul style="list-style-type: none"> využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	10 min	
Scenár	Pri riešenie svojej úlohy Sofia musí použiť komunikačný nástroj socket, ktorý slúži na komunikáciu medzi procesmi. Aby ho mohla využívať vo svojich procesoch, musí sa ho najprv naučiť vytvoriť a to pomocou služby jadra socket().	

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby socket():

Socket môžeme prirovnať k mobilnému telefónu, ktorý si Sofia kúpila bez SIM karty. Sofia má už telefón, ale nemôže ho využívať na telefonovanie. Podobne je to aj so socketom, ktorý nám vytvorí služba socket() ako koncový bod pre komunikáciu. Jeho návratová hodnota je deskriptor (niečo ako ID vytvoreného socketu – integer), ktorý môže byť použitý pre prístup k socketu v rámci procesu.

Syntax:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Sémantika:

- socket() vracia (socket) deskriptor pri úspešnom vykonaní alebo -1, ak nastane chyba



Pre podrobnejšie informácie - [man 2 socket](#).

KROK2 - pochopíť parametre služby:

Prvým parametrom je doména. Umožní Sofii určiť spôsob komunikácie, resp. určuje spôsob adresácie komunikačných uzlov a súbor dostupných protokolov (spôsobov komunikácie). Môžeme to prirovnať k výberu spôsobu komunikácie medzi ľuďmi – telefón, list, e-mail, skype a pod. Každý z týchto spôsobov komunikácie má svoj spôsob adresácie (tel. číslo, adresa bydliska, e-mailová adresa, skype adresa) a dostupné spôsoby komunikácie (napr. telefón – hovor alebo SMS). Ak chceme komunikovať prostredníctvom TCP/IPV4 protokolu, ako hodnotu parametra *domain* je potrebné

uviesť symbolickú konštantu `PF_INET`¹¹. Každá socketová doména používa vlastný adresový formát¹².

Typy domén:

DOMÉNY	POPIS
<code>PF_UNIX</code>	Lokálna komunikácia
<code>PF_INET</code>	IPv4 internetové protokoly
<code>PF_INET6</code>	IPv6 internetové protokoly
<code>PF_IPX</code>	IPX – Nowell protokoly
<code>PF_APPLETALK</code>	Appletalk DDP

Druhým parametrom je typ socketu, teda spôsob komunikácie – ešte raz - môžeme to prirovnáť k službám poskytovaným mobilným operátorom (SMS, hovory, internet, MMS, atď.), ktorého si Sofia vybraľa. Argument `type` Sofii určí typ socketu, ktorý určí charakteristiku komunikácie. Možné hodnoty sú:

- **`SOCK_STREAM`** – jedná sa o spojovanú transportnú službu, v doméne IPv4 ide o protokol TCP. Parameter `SOCK_STREAM` používame v prípade, že chceme vytvoriť najprv spojenie medzi socketmi. Odoslané dátá budú potvrdzované a budú určenému procesu doručené v poradí, v akom sme ich odoslali (ale môžu zostať aj nedoručené – v tom prípade bude detegovaná chyba).
- **`SOCK_DGRAM`** – je nespojovanou službou, v doméne IPv4 ide o protokol UDP. Tento socket môžeme používať pri posielaní správ s dopredu definovanou maximálnou veľkosťou, pričom nie je žiadna záruka, že správa bude doručená. Navyše, odosielateľ nemá možnosť sa dozvedieť, či správa doručená bola, alebo nie. Jedná sa o prenos bez vytvorenia spojenia medzi socketmi. Každá správa (datagram) musí obsahovať adresu cieľa.

Posledným parametrom je identifikátor protokolu. Pre naše potreby bude tento parameter nastavený na nulu. Hodnota nula znamená použitie defaultného protokolu¹³.

Ak chce Sofia používať protokol TCP/IP, zadáme ako posledný parameter hodnotu `IPPROTO_TCP`. Takže, ak bude chcieť vytvoriť socket pre spojovo orientovanú komunikáciu, použijeme službu `socket()` s týmito parametrami:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Typy parametrov `domain` a `type`, ktoré môžu byť použité spolu.

	<code>PF_UNIX</code>	<code>PF_INET</code>	<code>AF_NS</code>
<code>SOCK_STREAM</code>	Áno	TCP	SPP
<code>SOCK_DGRAM</code>	Áno	UDP	IDP
<code>SOCK_RAW</code>		IP	Áno
<code>SOCK_SEQPACKET</code>			SPP

¹¹ Minulosti sa používali symbolické konštandy začínajúce s `AF_` (address family) v službe `socket()`. Pozor si treba dať na služby, ktoré využívajú tieto symbolické konštandy v štruktúre `sockaddr` alebo od nej odvodenej štruktúry.

¹² Typy adresného formátu vidieť Linux dokumentačný projekt.

¹³ Kto ma záujem dozvedieť sa viac informácií nech použije manuálové stránky (man page) alebo Linux dokumentačný projekt.

KROK3 – aplikovanie služby v programe:

Sofia už pozná jednotlivé parametre služby `socket()`, preto môže vytvoriť jednoduchý program, v ktorom si vytvorí socket pre spojovanú komunikáciu s defaultne nastaveným protokol.

```
#include <sys/socket.h>
#include <stdio.h>

main() {
    int s;

    s = socket(PF_INET, SOCK_STREAM, 0); //vytvorenie socketu

    if(s == -1) perror("socket"); //kontrala sluzby socket()
    else
        printf("Socket vytvorený\n jeho deskriptor je %d \n",s);

    return 0;
}
```

KROK4:

Aké bolo číslo deskriptora socketu Vášho programu?:

Výstup z programu:

```
$
Socket vytvorený
jeho deskriptor je _____
$
```

Podtéma: Služby jadra - bind()

Kľúčové slová	bind(), man bind				
Ciele	Zapamätať si:	syntax služby bind(): <ul style="list-style-type: none"> zdroje na internete: http://www.hmug.org/man/2/bind.php http://ibm5.ma.utexas.edu/cgi-bin/man-cgi?bind+2 			
	Porozumieť:	<ul style="list-style-type: none"> dôvodu zviazania socketu s IP adresou a portom chybovým hláseniam parametrom služby bind() 			
	Aplikovať:	službu bind() pri práci so socketmi			
	Viedieť:	<ul style="list-style-type: none"> využiť získané skúsenosti pri tvorbe programov 			
Odhadovaný čas	20 min				
Scenár	Sofia si v prvej časti riešenie úlohy vytvorila socket pomocou služby jadra socket(). Aby mohol byť socket využívaný procesom v rámci skupiny počítačov, musí ho zviazať s IP adresou a portom. Nato sa využije služba jadra bind().				

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra bind():

Sofia navštívila pobočku vybraného mobilného operátora, aby si mohla kúpiť SIM kartu, ktorá jej priradí jedinečné telefóne číslo (IP adresa) a tiež jej poskytne služby v rámci jej telefónneho programu SMS, MMS, internet, hovory, atď. (port). Po vložení SIM karty do telefónu Sofia už môže využívať telefón na komunikáciu. Podobne ako SIM karta, tak aj služba bind() zviaže, resp. priradí socketu IP adresu a port. Po zviazaní službou bind() socket môžeme využívať na komunikáciu medzi procesmi v rámci počítačovej siete.

Syntax:

```
#include <sys/socket.h>
int bind (int socket, struct sockaddr *address, int address_len);
```

Sémantika:

- bind() vracia - 0 pri úspešnom vykonaní alebo -1, ak nastala chyba.

 Pre podrobnejšie informácie - [man 2 bind](#).

KROK2 - pochopiť parametre služby:

Prvým parametrom je `socket`. Parameter špecifikuje socket (prostredníctvom jeho deskriptora), ktorý má byť “zviazaný” s adresou. (deskriptor sme získali pomocou služby `socket()`.)

Druhým parametrom je `address`. Ukazuje na `sockaddr` štruktúru, formát ktorej je určený doménou alebo požadovaným správaním socketu. `Sockaddr` štruktúra zahrňuje štruktúry `sockaddr_in` a `sockaddr_un`, to závisí od toho, ktorá z podporovaných address family (rodiny adries) je práve využívaná.

Tretím parametrom je `address_len`. Určuje dĺžku štruktúry `sockaddr`, určenú parametrom `address`.

KROK3 - oboznámiť sa so štruktúrou `sockaddr` a jej položkami:

Proces potrebuje na nadviazanie spojenia socket, IP adresu stroja a portu ktorý je na nej určený pre pripájanie. Adresa je súčasťou štruktúry:

```
struct sockaddr {  
    unsigned short sa_family;    // typ komunikačného socketu AF nie PF  
    char     sa_data[14];        // 14 bytov protokolovej adresy  
};
```

Položka štruktúry `sa_data[14]` obsahuje cieľovú adresu a číslo portu pre daný socket. Pre IPv4 budeme používať odvodenú štruktúru `sockaddr_in`, ktorá je definovaná v hlavičkovom súbore `<netinet/in.h>` v tvare:

```
struct sockaddr_in {  
    short int      sin_family;    //typ komunikačného socketu AF nie PF  
    unsigned short int sin_port;   // 2 byty cislo portu  
    struct in_addr sin_addr;      // 4 byty actual IP address  
    unsigned char   sin_zero[8];    // zvyšnych 8 bytov nulujeme  
};
```

KROK4 – aplikovanie služby v programe:

Sofia si vytvorí program, v ktorom si vyskúša zviazať socket s IP adresou a portom pomocou služby `bind()`.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <netinet/in.h>  
#include <sys/socket.h>  
  
#define MYPORT 3490      //port, na ktorý sa budu užívateľia pripajat  
  
int main()  
{  
    int s;  
    struct sockaddr_in my_addr;      /* struktura obsahujuca informacie o  
                                     mojej adrese*/  
    int sin_size;  
  
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) == -1) {  
        perror("socket");           //vytvorenie socketu  
        exit(1);  
    }  
    my_addr.sin_family = AF_INET; //naplnenie struktury sockaddr_in  
    my_addr.sin_port = htons(MYPORT);  
    my_addr.sin_addr.s_addr = INADDR_ANY;  
    bzero(&(my_addr.sin_zero), 8);  
  
    //zviazanie socketu sluzbou bind()  
    if(bind(s,(struct sockaddr *)&my_addr, sizeof(struct sockaddr))== -1){  
        perror("bind");  
        exit(1);  
    }  
}
```

Opis predchádzajúceho programu:

Najprv si Sofia vytvorí socket pre spojovanú komunikáciu s defaultne nastaveným protokolom pomocou služby `socket()`. Aby mohla zviazať IP adresu a port so socketom službou `bind()`, musí najprv naplniť štruktúru `sockaddr_in`. Všetky príkazy začínajúce `my_addr` slúžia na napĺňanie štruktúry `sockaddr_in`. V závislosti od architektúry procesora, počítače uchovávajú čísla v pamäti rôznym spôsobom (endianita¹⁴). Funkcia `htons()` zabezpečí transformáciu endianity počítača na endianitu siete. Funkcia `bzero()` nám doplní reťazec `my_addr.sin_zero` o 8 núl.

Sofia sa musí rozhodnúť, akú IP adresu priradí socketu. Ak by za adresu počítača dosadila adresu 127.0.0.1 (*localhost*), v tom prípade by sa mohla na proces server pripojiť iba z lokálneho počítača. Ak chce očakávať spojenie z akéhokoľvek rozhrania, vloží do atribútu symbolickú konštantu `INADDR_ANY`.

Základnou podmienkou komunikácie je získanie adresy komunikujúcich strán. To môže byť vykonané v nasledujúcich krokoch:

Rozlíšenie adresy

V prípade, že máme meno uzla (host), ktorý sa má podieľať na komunikácii, jeho IP adresu zistíme pomocou funkcie `gethostbyname(char *hostname)`, ktorá vracia smerník na štruktúru `hostent`, ktorej definícia je:

```
struct hostent {  
    char* h_name;           //oficiálne meno hostu  
    char** h_aliases;       //smerník na zoznam aliasov (iných mien)  
    int h_addrtype;         //typ adresy  
    int h_length;           //dĺžka adresy  
    char** h_addr_list;     //smerník na zoznam adries ak ich ma viac  
};
```

Postupnosť bajtov

Na to, aby sme sformovali adresy do potrebného tvaru pre komunikáciu, použijeme sieťovú postupnosť bajtov. Našťastie, väčšina sieťových funkcií akceptuje adresy v hostovej postupnosti bajtov a vracajú výsledok v sieťovej postupnosti. Preto je zvyčajne potrebné zmeniť na sieťovú postupnosť bajtov jedine čísla portov, keďže pre zadávanie IP adresy vo zvyčajnom tvari sú systémom poskytované špeciálne funkcie (`inet_addr()` je funkcia, ktorá zmení reťazec znakov IP adresy do 4 bajtovnej sieťovej postupnosti). Ak chceme získať IP adresu v čitateľnom tvari, použijeme funkciu `inet_ntoa()`. Preklad čísel do sieťovej postupnosti vykonávajú nasledujúce funkcie:

- `htons()` – krátke celé čísla z hostovej postupnosti do sieťovej (pre porty).
- `ntohs()` – krátke celé čísla zo sieťovej do hostovej postupnosti (pre porty).
- `htonl()` – dlhé celé čísla z hostovej do sieťovej postupnosti (pre IP adresy).
- `ntohl()` – dlhé celé čísla zo sieťovej do hostovej postupnosti (pre IP adresy).

Sformovanie adresy

Formovanie adresy pre internetové protokoly sa uskutočňuje pomocou štruktúry `sockaddr_in`.

¹⁴ <https://sk.wikipedia.org/wiki/Endianita>

Podtéma: Služba jadra - listen()

Kľúčové slová	listen(), man listen()	
Ciele	Zapamätať si:	syntax služby listen(): <ul style="list-style-type: none"> zdroje na internete: http://www.rt.com/man/listen.2.html http://www.cl.cam.ac.uk/cgi-bin/manpage?2+listen
	Porozumieť:	<ul style="list-style-type: none"> parametrom služby listen() socketovému frontu
	Aplikovať:	službu listen() pri práci so socketmi
	Viedet:	<ul style="list-style-type: none"> využiť získané skúsenosti pri tvorbe programov
Odhadovaný čas	10 min	
Scenár:	Pri príjme požiadaviek na socket, musí Sofia pomocou serverového procesu vytvoriť front, kam sa ukladajú zatial' nevybavené simultánne požiadavky. Využije na to službu listen().	

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra listen():

Môžeme to prirovnáť k situácii (podržanie hovoru), keď Sofia pravé využíva telefón na hovor a na jej telefón príde požiadavka o ďalší hovor, ktorý je pre ňu dôležitý. Preto je nutné zaistiť, aby sa ďalšie prichádzajúce požiadavky od ďalších klientov nestratili zatial' čo prvá je obsluhovaná procesom serverom. Preto používame službu listen() na strane procesu server, ktorá vytvorí vyrovnávaciu pamäť pre uchovávanie požiadaviek na pripojenie. Ak je front plný a nejaký klient sa pokúsi pripojiť, bude spojenie odmietnuté.

Syntax:

```
#include <sys/socket.h>
int listen (int socket, int backlog);
```

Sémantika:

- listen() vracia - 0 pri úspešnom vykonaní alebo -1, ak nastane chyba

 Pre podrobnejšie informácie - [man 2 listen](#).

KROK2 - pochopíť parametre služby:

Prvým parametrom je *socket*. Argument určuje jedinečný identifikátor socketu vytvorený službou *socket()* s adresou priradenou službou *bind()*. Druhým parametrom je *backlog*. Argument *backlog* určuje maximálne množstvo simultánnych požiadaviek na spojenie. Horný limit je špecifikovaný symbolickou konštantou SOMAXCONN v hlavičkovom súbore *<sys/socket.h>*. Hodnota parametra *backlog* je nastavená štandardne na hodnotu 5.

Podtéma: Služba jadra - accept()

Kľúčové slová	accept(), man accept()		
Ciele	Zapamätať si:	syntax služby accept(): <ul style="list-style-type: none"> zdroje na internete: http://www.rt.com/man/accept.2.html http://www.manpagez.com/man/2/accept/ 	
	Porozumieť:	<ul style="list-style-type: none"> prijatiu – vyžiadaniu - ukončeniu spojenia chybovým hláseniam parametrom služby accept() 	
	Aplikovať:	službu accept() pri práci so socketmi	
	Viedet:	<ul style="list-style-type: none"> využiť získané vedomosti pri tvorbe programov 	
Odhadovaný čas	10 min		
Scenár:	Sofia pokračuje vo vytváraní procesu server, v ktorom už vytvorila socket zviazaný s IP adresou a portom. Už má front pre zapamävanie nevybavených požiadaviek. Teraz sa potrebuje naučiť, ako vybrať z frontu požiadavku na spojenie. Zistila, že k tomu jej poslúži služba accept().		

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby accept() :

Sofia prišla na jej telefón požiadavka o hovor. Ak chce vytvoriť telefonické spojenie a nadviazať komunikáciu, musí ju najprv potvrdiť. Podobne je to so službou accept(), ktorá sa využíva v procese server pre vyber požiadavky z frontu čakajúcich požiadaviek o spojenie a potvrdí ju. Pre každé prijaté spojenie sa vytvorí nový socket. Potom cez tento nový socket prebieha komunikácia. Tento sám o sebe nemôže pripať ďalšie spojenia, ale pôvodný (ktorý prijal požiadavku) socket je otvorený a ten môže pripať ďalšie spojenia. Ak je front prázdný, bez ďalších požiadaviek na spojenie, služba accept() blokuje proces server (uspí proces), až pokiaľ nie je prítomná požiadavka na spojenie.

Syntax:

```
#include <sys/socket.h>
int accept (int socket, struct sockaddr *restrict addr,
socklen_t *restrict len);
```

Sémantika:

- accept() vracia - nezáporný (socket) descriptor pri úspešnom vykonaní alebo -1, ak nastane chyba

 Pre podrobnejšie informácie - [man 2 accept](#).

KROK2 - pochopiť parametre služby:

Prvým parametrom je `socket`. Určuje socket, ktorý bol vytvorený službou `socket()`, bol zviazaný s adresou a má vytvorený backlog službou `listen()`. Druhým parametrom je `address`. Ukazuje na `sockaddr` štruktúru, ktorá bude obsahovať IP adresu a port klientskeho procesu (vzdialenosť stroja), ktorý sa pripojil k procesu

server. Jej formát je určený doménou alebo požadovaným správaním socketu. Môžeme nastaviť na NULL, čím určíme, že adresa nie je potrebná (v rámci jedného stroja). Tretím parametrom je `address_len`. Určuje dĺžku štruktúry `sockaddr`, určenú parametrom `address`. Ak je parameter `address` nastavený na NULL, potom je tento parameter ignorovaný.

Podtéma: Služba jadra - connect()

Kľúčové slová	connect(), man connect(), model klient/server		
Ciele	Zapamätať si:	syntax služby connect(): <ul style="list-style-type: none"> zdroje na internete: http://www.rt.com/man/connect.2.html http://www.manpagez.com/man/2/connect/ 	
	Porozumieť:	<ul style="list-style-type: none"> prijatie – vyžiadanie - ukončenie spojenia chybovým hláseniam parametrom služby connect() 	
	Aplikovať:	službu connect() pri práci so socketmi	
	Viedet:	<ul style="list-style-type: none"> využiť získané vedomosti pri tvorbe programov 	
Odhadovaný čas	20 min		
Scenár:	Sofia sa už naučila vytvárať servrovské procesy. Teraz je čas, aby vytvorila klientsky proces. Dozvedela sa, že tak ako pre serverovský proces, aj pre klientsky proces musí vytvoriť socket pomocou služby socket(). To sa jej podarilo. Teraz chce nadviazať spojenie so vzdialeným počítačom (serverom). Potrebuje svoj klientsky proces pripojiť na serverovský proces a vytvoriť spojenie medzi socketmi týchto procesov. K tomu jej pomôže služba connect().		

POSTUP:

KROK1 – naučiť sa syntax a sémantiku služby jadra connect():

Sofia potrebuje zavolať svojej kamarátke pomocou jej telefónu. Najprv musí zadat jej telefóne číslo (IP adresa), aby mohla uskutočniť telefonické spojenie. Podobne je to aj so službou connect(), ktorá vytvára spojenie medzi dvoma procesmi (využíva sa na strane klienta). Jadro nastaví komunikačné linky medzi socketmi, pričom oba sockety musia používať ten istý adresný formát a protokol. Táto služba vykonáva rôzne činnosti pre každý z nasledujúcich typov socketov:

- Ak socket je SOCK_DGRAM, služba connect() vytvorí peer adresu. Peer adresa identifikuje socket ktorému sú zaslané všetky dátá následnou službou send(). Taktiež identifikuje socket, z ktorého môžu byť dátá prijímané. Ale nie je žiadna záruka, že dátá budú doručené. Jedná sa o prenos bez vytvorenia spojenia medzi socketmi (nespojovaná služba).
- Ak socket je SOCK_STREAM, služba connect() sa pokúša nadviazať spojenie so socketom špecifikovaným parametrom serv_addr (spojovaná služba). Formát parametra serv_addr je určený doménou a požadovaným správaním socketu.

Syntax:

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);
```

Sémantika:

- connect() vracia - 0 pri úspešnom vykonaní alebo -1, ak nastane chyba.

 Pre podrobnejšie informácie - **man 2 connect**.

KROK2 - pochopiť parametre služby:

Prvým parametrom je `sockfd`, ktorý špecifikuje deskriptor socketu. Druhým parametrom je `address`. Ukazuje na `sockaddr` štruktúru, ktorá obsahuje IP adresu a port procesu (vzdialeného stroja - servera), na ktorý sa chceme pripojiť. Jej formát je určený doménou alebo požadovaným správaním socketu. Tretím parametrom je `address_len`. Určuje dĺžku štruktúry `sockaddr`, určenú parametrom `address`.

KROK3 – odosielanie a príjem dát¹⁵:

Odosielanie dát: Na odosielanie dát slúži služba `send()`

Syntax:

```
int send(int s, const void *msg, size_t len, int flags);
```

Príjem dát: Na príjem dát slúži služba `recv()`.

Syntax:

```
int recv(int s, void *buf, size_t len, int flags);
```

 Pre podrobnejšie informácie - **man 2 send** a **man 2 recv**.

Ukončenie spojenia: Socket uzavrieme rovnako ako súbor službou `close()`

Uzavretie socketu:

Socketové prepojenie môžeme ukončiť na serveri alebo u klienta pomocou služby `close()`. Socket sa musí vždy zatvárať na obidvoch stranách. Na serveri sa zatvára, keď `recv()` vráti nulu, ale `close()` sa môže zablokovať, pokial' ma socket ešte neodoslané dátá.

¹⁵ Na odosielanie a príjem dát cez socket môžeme využiť aj služby jadra `write()` a `read()`.

Podtéma: Príklad

KROK1 – aplikovanie služieb v programe:

Nasledujúce programy prezentujú jednoduchý serverovský a klientský proces. Používajú na komunikáciu sockety a spojovanú transportnú službu v Internetovej doméne IPv4. Predtým, než si opíšeme samotný kód, skompilujte obidva programy a spusťte ich, aby ste mali možnosť vidieť, čo robia. Klienta skopírujte do súboru *klient.c* a server do súboru *server.c*. Ideálne by bolo, aby ste server a klient spustili na dvoch rôznych počítačoch. Najprv spusťte server, potrebujete mu odovzdať číslo portu ako argument. Môžete si vybrať akokoľvek číslo medzi 1024 a 65535. Ak je ten náhodný port už používaný, server vám to oznámi a program skončí. Potom si zvoľte iné číslo a skúste znova.

Spustiť server môžete napríklad takto: **./server 51717**

Na to, aby ste spustili klienta, potrebujete mu odovzdať dva argumenty:

- 1 . adresu počítača, na ktorom beží proces server
- 2 . číslo portu, na ktorom server čaká na pripojenie.

Pripojenie k serveru môžeme uskutočniť takto: **./client alfa.intrak.sk 51717**

Klient vás požiada, aby ste zadali správu. Ak všetko ide tak ako má, server zobrazí Vašu správu na štandardnom výstupe, pošle potvrdenie správy klientovi a skončí. Klient zobrazí potvrdzujúcu správu od servera a skončí.

Ak spúšťate tieto dva programy na jednom PC, otvorte si jedno okno na server a jedno na klienta. Potom ako prvý argument pre klienta uvediete **localhost**.

Výstup z programu server:

```
$  
./server 51717  
Here is the message: posielam pozdrav serveru  
$
```

Výstup z programu klient:

```
$  
./klient localhost 51717  
Please enter the message: posielam pozdrav serveru  
I got your message  
$
```

Server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    //tato funkcia sa vyuziva ked systemove volanie zlyha
    perror(msg); //vypise spravu o chybe a ukonci program server
    exit(1);
}

int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clilen, n; //pomocne premenne
    char buffer[256]; //buffer pre ulozenie znakov zo socketu
    struct sockaddr_in serv_addr; //obsahuje adresu servera
    struct sockaddr_in cli_addr; //obsahuje adresu klienta

    if (argc < 2) { //kontrola poctu argumentov
        fprintf(stderr,"ERROR, no port provided\n");
        exit(1);
    }
    //vytvorenie socketu
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) error("ERROR opening socket");
    //naplnenie struktury sockaddr_in
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    //zviazanie socketu sluzbou bind()
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
        error("ERROR on binding");

    listen(sockfd,5); //vytvorime si front poziadaviek
    clilen = sizeof(cli_addr); //velkost struktury adresy klienta
    //akceptovanie spojenia
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,&clilen);
    if (newsockfd < 0) error("ERROR on accept");

    bzero(buffer,256); //spracovanie dat od klientov
    n = read(newsockfd,buffer,255);
    if (n < 0) error("ERROR reading from socket");

    printf("Here is the message: %s\n",buffer);

    n = write(newsockfd,"I got your message",18);
    if (n < 0) error("ERROR writing to socket");

    return 0;
}
```

Klient

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg)
{
    //tato funkcia sa vyuziva ked systemove volanie zlyha
    perror(msg); //vypise spravu o chybe a ukonci program server
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;           //pomocne premenne
    struct sockaddr_in serv_addr;   //obsahuje adresu servera
    struct hostent *server;         //informacie o vzdialenom pocitaci
    char buffer[256];              //buffer pre ulozenie znakov zo socketu

    if (argc < 3) {                //kontrola poctu argumentov
        fprintf(stderr,"usage %s hostname port\n", argv[0]);
        exit(0);
    }
    portno = atoi(argv[2]);          //cislo portu servera
                                    //vytvorenie socketu
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) error("ERROR opening socket");
                                    //hostname pc server
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);}
                                    //naplnenie struktury sockaddr_in
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr,
          server->h_length);
    serv_addr.sin_port = htons(portno);
                                    //vytvorenie spojenia
    if (connect(sockfd, (struct sockaddr *) &serv_addr,
                sizeof(serv_addr)) < 0)
        error("ERROR connecting");
                                    //spracovanie dat
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);       //zadanie znakov z klavesnice

    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
        error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
        error("ERROR reading from socket");
    printf("%s\n",buffer);
    return 0;
}
```

Vysvetlenie programu server¹⁶:

Pre program *server.c* vytvárame socket, ktorý využíva spojovanú transportnú službu v Internetovej doméne IPv4. Aby sme mohli komunikovať v rámci počítačovej siete, postačí k zviazaniu socketu službou `bind()` na strane servera iba port (na ktorom server akceptuje spojenia), ktorý získame ako parameter odovzdaný programu server. Potom musíme vytvoriť front, do ktorého sa budú ukladať požiadavky na spojenie (službou `listen()`). Požiadavky na spojenie musíme z frontu vyberať postupne (služba `accept()`). Ak vo fronte nie je žiadna požiadavka na spojenie, proces server počká, kým nejaká požiadavka nedôjde. Služba `accept()` nám vráti nový socket, pomocou ktorého budeme komunikovať s procesom-klientom. Na príjem a odosielanie dát využívame služby `read()` a `write()`, ktoré obsahujú počet znakov prečítaných alebo zapísaných. Server zobrazí správu od klienta na štandardnom výstupe, pošle potvrdenie správy klientovi a skončí.

Vysvetlenie programu klient:

Pre program *klient.c* vytvárame socket, ktorý využíva spojovanú transportnú službu v Internetovej doméne IPv4 (klientské sockety neviažeme službou `bind()`). Aby sme mohli komunikovať v rámci počítačovej siete so serverom, potrebujeme jeho IP adresu a port, ktoré získame ako parametre odovzdané programu klient. IP adresu počítača môžeme zadať ako *hostname* (názov počítača). K identifikácii počítača potrebujeme IP adresu, ktorú získame funkciu `gethostbyname()` (vyplní štruktúru `hostent *server` – informácie o vzdialenom počítači). Ked' už sme získali IP adresu a port, program vyplní štruktúru `sockaddr_in serv_addr`. Proces-klient sa pripája na proces server systémovým volaním `connect()`, ktoré využíva štruktúru `sockaddr_in serv_addr`. Na príjem a odosielanie dát využívame služby `read()` a `write()`, ktoré obsahujú počet znakov prečítaných alebo zapísaných. Klient vás požiada, aby ste zadali správu a odošle ju. Zobrazí potvrdzujúcu správu od servera a skončí.

ÚLOHY NA SAMOSTATNÚ PRÁCU:

- Vytvorte socket, ktorý bude využívať protokol UDP. Overte si činnosť služby `socket()` aj pre ostatné protokoly.
- Aký je rozdiel medzi socketmi na strane servera a na strane klienta?
- Modifikujte predchádzajúci program *server.c* tak, aby na ošetrenie prichádzajúceho spojenia využíval nový proces (`fork()`). Proces rodič programu server prijíma požiadavky na spojenie a jeho proces potomok vykonáva komunikáciu s klientom (prijíma dátu). Proces rodič pokračuje v akceptovaní ďalších požiadaviek o spojenie.
- Vytvorte dva nezávislé programy, ktoré používajú na komunikáciu sockety nespojovanej služby (protokol UDP) v Internetovej doméne.

¹⁶ Jednotlivé hlavičkové súbory – samostudium .