**TECHNICKÁ UNIVERZITA V KOŠICIACH**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

# OPERAČNÉ SYSTÉMY

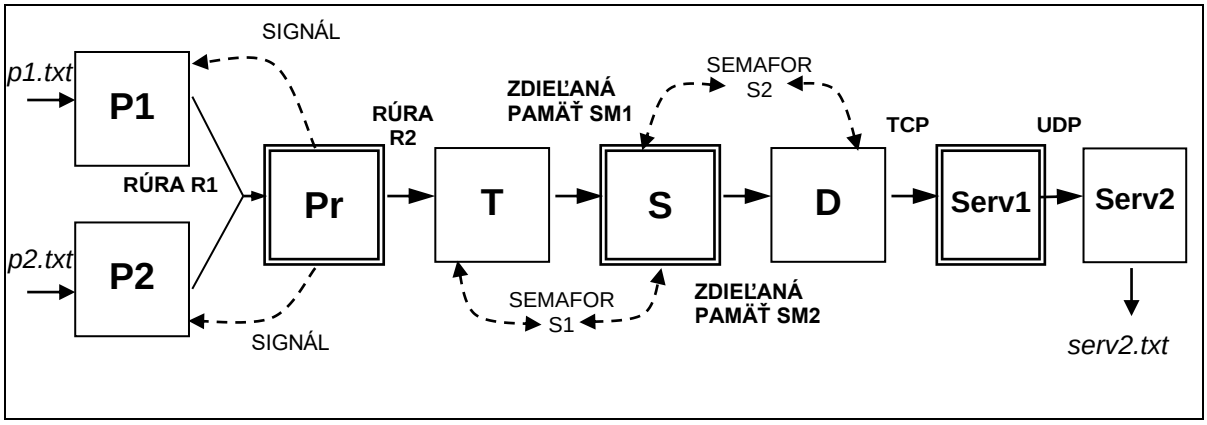
**(ZADANIE IPC)**

**Danyil Yedelkin**
**Yehor Oliinyk**
**Viyaleta Senkavets**

# Obsah

# 1. Zadanie Unix1

## 1.1 SCHÉMA




- procesy vyznačené zvýrazneným okrajom sú programy, ktoré budú pri kontrole zadania dodané. Teda treba vypracovať iba programy P1, P2, T, D a Serv2.

## 1.2 POPIS PROCESOV

**PROCES Zadanie (vypracovaný študentom, nie je zakreslený v diagrame)**

### Spúšťanie

*zadanie <číslo portu 1> <číslo portu 2>*

### Funkcia

Parametre hlavného programu sú čísla portov pre servery Serv1 (TCP) a Serv2 (UDP). Tieto čísla je potrebné týmto procesom odovzdať. Program **Zadanie** nech vyhradí všetky zdroje (pre medziprocesovú komunikáciu) a nech spustí všetky procesy. Všetky procesy nech sú realizované ako samostatné programy.

**PROCES Pr**

### Spúšťanie

*proc_pr <pid procesu P1> <pid procesu P2>*
*<id čítaciaho konca rúry R1> <id zapisovacieho konca rúry R2>*

### Funkcia

Proces **Pr** pošle signál SIGUSR1 svojmu hlavnému procesu (Zadanie) na signalizáciu, že je pripravený. Proces **Pr** si údaje bude žiadať:

- Ak pošle signál (SIGUSR1) procesu **P1,** nech tento proces (P1) zapíše slovo prečítané zo súboru *p1.txt*.
- To isté platí aj pre proces **P2** a *p2.txt*. Teda, ak proces **Pr** pošle signál (SIGUSR1) procesu **P2** nech do rúry R1 zapíše slovo proces **P2** zo súboru *p2.txt*.

Proces **Pr** k slovu prijatému z rúry R1 pridá svoju značku a zapíše nové slovo do rúry R2.

**PROCES S**

### Spúšťanie

*proc_s <id zdieľanej pamäte SM1> <id semaforu S1> <id zdieľanej pamäte SM2>*
*<id semaforu S2>*

**Funkcia**
Proces **S** pošle signál SIGUSR1 svojmu hlavnému procesu (Zadanie) na signalizáciu, že je (proces S) pripravený. Proces **S** príjme slovo zo zdieľanej pamäte SM1 so synchronizáciou semaforom S1 (pozri časť o semaforoch v časti „Popis komunikácie“), pripíše k nemu svoju značku a zapíše ho do zdieľanej pamäte SM2 so synchronizáciou semaforom S2.

## PROCES Serv1

**Spúšťanie**
*proc_serv1 <číslo portu 1> <číslo portu 2>*

**Funkcia**
Proces **Serv1** vytvorí TCP server (na porte <číslo portu 1>), ktorý bude prijímať TCP pakety. Server prijaté slová označí svojou značkou a pošle ich ďalej na UDP server (port <číslo portu 2>). Čísla portov 1 a 2 sú argumenty hlavného procesu (pozri kapitolu „PROCES Zadanie“). Proces **Serv1** pošle signál SIGUSR1 svojmu hlavnému procesu (Zadanie) na signalizáciu, že je (proces Serv1) pripravený. TCP aj UDP server nech vytvorený na lokálnom počítači, teda na počítači „127.0.0.1“ (pozor, nie „localhost“!).

### 1.3 POPIS KOMUNIKÁCIE

**Semafor S1**
Pre semafor S1 je potrebné vytvoriť dvojicu semaforov. Proces **T** nech sa riadi podľa semaforu S1[0] a proces **S** sa bude riadiť podľa semaforu S1[1], pričom nastavený semafor S1[0] (rozumej nastavený na hodnotu 1) nech znamená, že proces **T** môže zapisovať do zdieľanej pamäte (SM1). Nastavený semafor S1[1] nech znamená, že proces **S** môže zo zdieľanej pamäte (SM1) údaje čítať.

**Semafor S2**
Pre semafor S2 je potrebné vytvoriť dvojicu semaforov. Proces **S** nech sa riadi podľa semaforu S2[0] a proces **D** sa bude riadiť podľa semaforu S2[1]. Pričom nastavený semafor S2[0] (rozumej nastavený na hodnotu 1) nech znamená, že proces **S** môže zapisovať do zdieľanej pamäte (SM2). Nastavený semafor S2[1] nech znamená, že proces **D** môže zo zdieľanej pamäte (SM2) údaje čítať.

**Príklad komunikácie medzi procesom T a procesom S – semafor S1**

Pozn.: Pre semafor S2 je to analogické.

| Hodnota semaforu | Význam |
|---|---|
| **0** | **červená** |
| **1** | **zelená** |

**Proces T sa riadi semaforom S1[0] a proces S sa riadi semaforom S1[1]. To znamená, že kým ma proces T na svojom semafore** (S1[0]) **hodnotu 0** (t.j. červená) **– tak stále čaká. To isté platí pre proces S. To znamená, že kým proces S má na svojom semafore** (S1[1]) **hodnotu 0** (červená) **– to znamená, že musí čakať.**

Semafor treba inicializovať do nasledovného stavu:

**S1[0] = 1** (zelená/môžeš) **S1[1] = 0** (červená/stoj)
- tento stav znamená, že proces T môže do zdieľanej pamäte zapisovať. Proces S čaká, má červenú, nemôže čítať.

**Proces T má zelenú** (teda môže zapisovať)**, lebo má svoj semafor** (S1[0]) **nastavený na hodnotu 1** (zelená)**. Teda do zdieľanej pamäte zapíše svoje údaje. Teraz je potrebné, aby proces T umožnil procesu S údaje čítať. Preto zmení semafory na nasledovný stav:**

**[**Najprv zmení svoj semafor (semafor S1[0]) z hodnoty **1** na hodnotu **0** (teda zo zelenej na červenú) a semafor procesu S (teda S1[1]) zmení z hodnoty **0** na hodnotu **1** (z červenej na zelenú, aby mu naznačil, že údaje sú zapísané a môže ich teda čítať.)**]**

**S1[0] = 0** (červená/stoj) **S1[1] = 1** (zelená/môžeš)
- tento stav znamená, že proces T nemôže do zdieľanej pamäte zapisovať, má čakať. Proces S má zelenú, má v zdieľanej pamäti pripravené údaje a môže údaje z pamäte prečítať.

**Teraz už môže semafor S zo zdieľanej pamäte údaje prečítať, lebo jeho semafor** (S1[1]**) sa zmenil z hodnoty 0** (červená) **na hodnotu 1** (zelená)**. Prečíta teda údaje zo zdieľanej pamäte a vymení farby semaforov** (t.j. zmení sebe zo zelenej na červenú – z **1** na **0** – a procesu T zmení z červenej na zelenú – z **0** na **1** – aby mohol do pamäte zapisovať)**.**

**Súbory p1.txt, p2.txt a serv2.txt**
Súbory p1.txt a p2.txt budú obsahovať slová (rozumej reťazce znakov každé v novom riadku), pričom veľkosť slova počas putovania medzi procesmi nepresiahne dĺžku 150 znakov. Súbor serv2.txt nech obsahuje výsledné slová zapísané každé v samostatnom riadku.

**Poznámka**: Pre účely vývoja programov a testovania je možné programy **Pr**, **S** a **Serv1** stiahnuť zo stránok systému na odovzdávanie zadaní.

**Upozornenie:**
**Pozor, nachádzate sa v paralelnom prostredí! To znamená, že poradie vykonávania procesov nemusí byť rovnaké na rôznych systémoch. Preto dbajte na synchronizáciu a vyhnete sa problémom „predbiehania“ procesov!**

### 1.4 „Čo všetko teda potrebujem spraviť??“

1. Potrebujem spraviť programy
   a. **P1** (zdrojový text: **proc_p1.cpp**, spustiteľný program: **proc_p1**),
   b. **P2** (zdrojový text: **proc_p2.cpp**, spustiteľný program: **proc_p2**),
   c. **T** (zdrojový text: **proc_t.cpp**, spustiteľný program: **proc_t**),
   d. **D** (zdrojový text: **proc_d.cpp**, spustiteľný program: **proc_d**),
   e. **Serv2** (zdrojový text: **proc_serv2.cpp**, spustiteľný program: **proc_serv2**) a
   f. **Zadanie** (zdrojový text: **zadanie.cpp**, spustiteľný program: **zadanie**).

   A takisto súbor **makefile** na skompilovanie všetkých zadaní.

2. Musím dodržať názvy programov a ich zdrojových textov uvedené v zátvorkách! Názvy súborov zdrojových textov nech sú dodržané tiež.

3. Musím dodržať stanovené názvy vstupných súborov (p1.txt, p2.txt) a výstupného súboru (serv2.txt). Dbať na veľké a malé písmená.

4. Vyhotovené **zdrojové texty** programov spolu so súborom **makefile** (programy Pr, S a Serv1 nie – tie budú pri kontrole dodané) treba zbaliť, najlepšie vo formáte *.tar.gz (napr. zadanie.tar.gz) a odoslať na server.

### 1.5 Príklad súboru makefile

```
all: zadanie proc_p1 proc_p2 proc_t proc_d proc_serv2

zadanie: zadanie.cpp
      g++ zadanie.cpp -o zadanie

proc_p1: ...A...
      ...B...
```

Vyššie je uvedený príklad, ako sa zo zdrojového súboru *zadanie.cpp* vyrobí (skompiluje) hlavný program *zadanie*. Ďalej uveďte, ako sa majú skompilovať ostatné programy. Namiesto časti *...A...* vypíšte ktoré súbory sú potrebné na skompilovanie a vytvorenie programu *proc_p1*. V časti *...B...* uveďte konkrétny kompilačný príkaz, ktorý vyvolá shell, aby vytvoril (skompiloval) program *proc_p1*.

### 1.6 Priebeh kontroly zadania

Pre názornosť a pre predstavu, ako sa vykonáva kontrola je tu uvedený stručný priebeh kontroly odovzdaného zadania:

1. Zavolá sa študentom vytvorený súbor *makefile*, pomocou ktorého sa vytvoria potrebné binárne súbory procesov.
2. Systém nájde a zabezpečí potrebné knižnice na spustenie programov zadania a kontroly.
3. Systém dodá programy *proc_pr*, *proc_s* a *proc_serv1*.
4. Pripravia sa vstupné súbory *p1.txt* a *p2.txt*.
5. Spustí sa hlavný študentom vytvorený program *zadanie*.
6. Po skončení behu celej sústavy procesov sa vyhodnotí súbor *serv2.txt*.

## 2. Dodefinovanie zadania

K zadaniu sú dodané procesy *proc_pr, proc_s, proc_serv1*, textové súbory *p1.txt a p2.txt.* Tieto súbory obsahujú reťazce znakov(slova oddelené znakom "nového riadku"). V zadaní treba dorobiť procesy P1, P2, T, D, Serv2.
Všetky súbory:

- Makefile
- p1.txt
- p2.txt
- proc_d.c
- proc_p1.c
- proc_p2.c
- proc_pr
- proc_s
- proc_serv1
- proc_serv2
- proc_serv2.c
- proc_t.c
- zadanie.c
- serv2.txt (Po skončení behu celej sústavy procesov sa vyhodnotí súbor serv2.txt)

Úloha bola dorobiť procesy: p1, p2, t, d, serv2 aj súbor zadanie.c .

## 3. (a) Popis relevantných štruktúr, algoritmov, dátových typov, konštánt

*struct sembuf* - štruktúra slúži na prácu so semaforom pri službe jadra semop().
*struct sockaddr_in* - štruktúra je potrená pre služby s prácou so servermi.

```
struct sockaddr_in {
    short            sin_family;       // 2 bytes e.g. AF_INET, AF_INET6
    unsigned short   sin_port;    // 2 bytes e.g. htons(3490)
    struct in_addr   sin_addr;     // 4 bytes see struct in_addr, below
    char             sin_zero[8];     // 8 bytes zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // 4 bytes load with inet_pton()
};
```

### Proces zadanie

**Spúšťanie procesu:** *zadanie <TCP port> <UDP port>.*
**Algoritmus:**

- Proces pripraví všetky potrebné prostriedky pre synchronizáciu v paralelnom prostredí a argumenty pre jednotlivé procesy a hneď ich aj spustí.

### Proces p1

**Spúšťanie procesu:** *proc_p1 <identifikačné číslo rúry 1 na zápis >*
**Algoritmus:**

- Proces otvorí súbor *p1.txt* a načíta slovo do rúry 1, ak mu *proces_pr* pošle signál.

### Proces p2

**Spúšťanie procesu:** *proc_p1 <identifikačné číslo rúry 1 na zápis > <>*

**Algoritmus:**

- Proces otvorí súbor *p2.txt* a načíta slovo do rúry 1, ak mu *proces_pr* pošle signál.

### Proces T

**Spúšťanie procesu:** *proc_t <semafor S1, identifikačné číslo zdieľanej pamäte 1, identifikačné číslo rúry 2 pre čítanie >*

**Algoritmus:**

- Proces pomocou semaforu zakáže čítanie zo zdieľanej pamäte 1 , načíta z rúry 2 slovo a zapíše do zdieľanej pamäte 1.

### Proces D

**Spúšťanie procesu**: *proc_d <identifikačné číslo zdieľanej pamäte 2, identifikačné číslo semaforu 2, číslo TCP portu>*

**Algoritmus :**

- Proces pomocou semaforu zakáže zápis do zdieľanej pamäte 2, načíta zo zdieľanej pamäte 2 slovo po znak nového riadka a pošle ho cez TCP port na server.

### Proces Serv2

**Spúšťanie procesu** *proc_serv2< číslo UDP portu>*

**Algoritmus :**

- Proces vytvorí UDP server, prijme slovo od TCP servera a zapíše ho do súboru *serv2.txt*

## 3. (b) Dátové typy

### Proces zadanie

pid procesov

```
int Server1, Server2, D, P1, P2, PR, T, S;
```

deklarácia pipov

```
int ppr1[2], ppr2[2];
```

pomocne premenne na uloženie id procesov p1, p2, id pipov a semaforov:

```
char pipe1_read_char[10];
char pipe2_read_char[10];
char pipe2_write_char[10];
char pipe1_write_char[10];

char semaphore1_char[10];
char semaphore2_char[10];

char sharedmem1_char[10];
char sharedmem2_char[10];

char P1_descriptor[10];
char P2_descriptor[10];

char server1_char[10];
char server2_char[10];
```

pomocne premenne pre ukončenie programu

```
int progress = 0, ended = 0;
```

shm a semafory

```
int serv1, serv2, shmem1, shmem2;
```

**Proces p1**

id rury1

```
int output;
```

buffer size

```
size_t len = 200;
```

ukazovateľ na file descriptor na súbor p1.txt

```
FILE *fd;
```

ukazovateľ na zdieľanú pamäť

```
char * buffer = NULL;
buffer = calloc(len, sizeof(char));
```

size line súboru fd

```
ssize_t readLine = getline(&buffer, &len, fd);
ssize_t writeLine = write(output, buffer, strlen(buffer));
```

**Proces p2**

id rury1

```
int output;
```

buffer size

```
size_t len = 200;
```

ukazovateľ na file descriptor na súbor p1.txt

```
FILE *fd;
```

ukazovateľ na zdieľanú pamäť

```
char * buffer = NULL;
buffer = calloc(len, sizeof(char));
```

size line súboru fd

```
ssize_t readLine = getline(&buffer, &len, fd);
ssize_t writeLine = write(output, buffer, strlen(buffer));
```

**Proces T**

id rúry 2 , semaforu 1 , zdieľanej pamäte

```
int pipe = atoi(argv[1]), sharedmem = atoi(argv[2]), semphr = atoi(argv[3]);
```

ukazovateľ na zdieľanú pamäť aj pomocná premenná pre data

```
char writeString[200];
char* smbuffer;
smbuffer = shmat(sharedmem, NULL, 0);
```

pomocná premenná pre cyklus

*char element;*
*for( ; element != '\n'; )* – cyklus, ktorý používajú pomocnú premennú element
*int i = 0;* - premenná, ktorá označuje umiestnenie kurzora v pamäte writeLine

**Proces D**

kód pre zdieľaná pamäť
```
int memory = atoi(argv[1]);
```

kód pre semafor
```
int semafor = atoi(argv[2]);
```

tcp port
```
int port = atoi(argv[3]);
```

Kód pre socket TCP
```
int socket = socket(AF_INET, SOCK_STREAM, 0);
```

kontrola pre kódu pre socket TCP
```
if(socked < 0)
{
      perror("Socket proc_d");
      exit(1);
}
```

kód pre ukazovateľ na zdieľanú pamäť
```
char* shared_memory;
shared_memory = shmat(memory, NULL, 0);
```

kontrola pre kódu pre ukazovateľ na zdieľanú pamäť
```
if(shared_memory == NULL)
{
      perror("Problem with shmat");
      exit(1);
}
```

**Proces Serv2**

port UDP
```
int firstPort = atoi(argv[1]);
```

socket
```
int firstSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

súbor
```
int file = open("serv2.txt", O_CREAT | O_WRONLY | O_TRUNC, 0777);
```

štruktúra
```
struct sockaddr_in client; //the basic structure for all syscalls and
functions that deal with internet addresses
```

buffer
```
char buffer[200];
```

# 4. Analýza problematiky

Najprv spustime súbor Makefile, pomocou `make all`
Ďalej prejdeme do súboru zadanie.c.
Program *Zadanie* urobí *:*

- vyhradí všetky zdroje pre medzi všetkými procesormi komunikácii
  - rúry *R1* a *R2*
  - semafory *S1* a *S2*
  - zdieľané pamäte *SM1* a *SM2*
- spúšťa procesy: `proc_p1`, `proc_p2`, `proc_pr`, `proc_t`, `proc_s`, `proc_d`, `proc_serv1`, `proc_serv2`. Po ich spustení počká istý čas aby sa vykonali procesy korektne.

Po skončení celej sústavy procesov sa vyhodnotí súbor `serv2.txt` , ale ak sa program `proc_serv2.c` neukončí správne, teda sa procesy neukončia a ostanú visieť TCP a aj UDP spojenie a budeme mať súbor z error, napr. `proc_pr.err`.

# 5. Popis navrhovaného riešenia

## 5.1 Návrh riešenia

Najprv musíme mať v kóde každého procesu if(), ktoré kontrolujú pravopis argumentov. Naďalej, náš hlavný procesor `zadanie` musí pozrieť , či bol zadaný (zavolaný) predchádzajúci proces. Toto nastavuje poradie programu a správne volá každý procesy. A kedy všetko to bolo správne a ostatný proces dokončí svoju prácu, to v tomto prípade náš program sa ukončí z kódom 0, čo znamená, že všetko fungovalo to správne (dobre).

## 5.2 Dátové štruktúry

### 5.2.1 Rúry (Pipe)

*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 potomkov nadradeného procesu alebo o procesy 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.

*Rúra* je jednosmerný komunikačný prostriedok. Údaje zapisované na jednom konci rúry sú prečítané na druhom konci rúry. 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 potomkovho procesu.

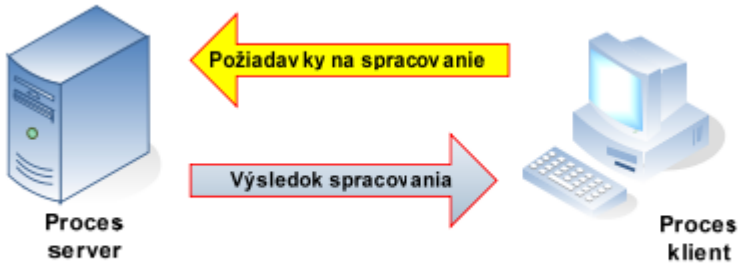
### 5.2.2 Semafory

***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 , ktorá obmedzuje prístupu k zdieľaným prostriedkom OS Unix. 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 operácii pochádzajú od zakladateľa semaforov, pána Edsger Wybe Dijkstra. V je skratka slova „verhoog“, čo znamená v holandčine zvýšiť. P je podľa pána Dijkstru skratka zo zloženého slova „prolaag“, čo znamená skús-a-zníž. Operácie sú definované nasledovne:

- ***Operácia P*** pokúšajúca sa odpočítať z hodnoty semaforu. Ak je hodnota na semafore väčšia ako 0 operácia sa vykoná. Ak je hodnota na semafore 0, operácia sa vykonať nedá a proces zostane zablokovaný, pokiaľ iný proces nezvýši hodnotu na semafore.
- ***Operácia V*** zvýši hodnotu na semafore a môže spôsobiť odblokovanie zablokovaného procesu.

### 5.2.3 Model Klient – Server

Jedným zo základných modelov pre komunikáciu medzi procesmi prostredníctvom socketov je ***model klient–server***. Tento model je založený na existencii procesov servera a klienta. Proces server vykonáva pasívnu úlohu na tom istom počítači alebo na inom počítači. Poskytuje určitú službu, prostriedky, výkon klientskym procesom a čaká na ich požiadavky. 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 líšiť používateľským prostredím.




Obr.1

Server sám inicializuje a následne svojou činnosťou pozastaví, pokiaľ 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 pochopiť, 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 sietí. Komunikačný systém pozostáva z týchto častí:

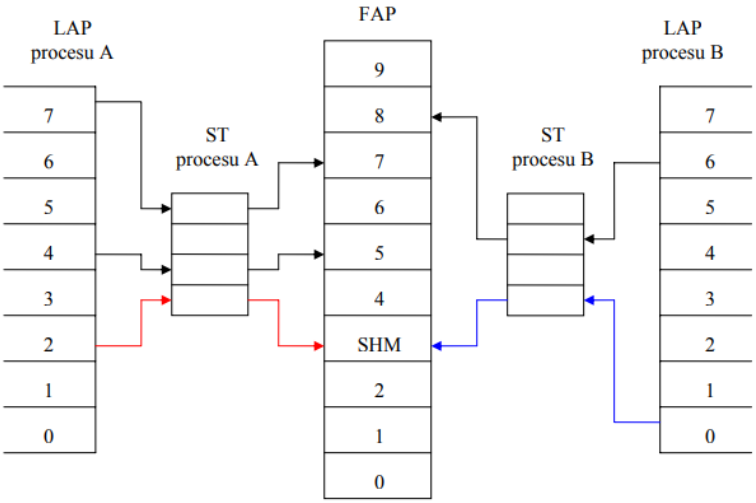
- IP adresa je adresa stroja (vzťahuje sa na jeho sieťové rozhranie), kde je komunikujúci proces vykonávaný. Pomocou nej dokážu s týmto procesom komunikovať iné procesy v rámci počítačovej siete. IP adresu tvoria štyri bajty (v súčasnosti sa rozširuje na 16 bajtov).
- Port je celočíselný identifikátor komunikujúceho procesu, na ktorom sú vybavované požiadavky procesov (Na strane servera sa používajú známe porty do 1024 a na strane klienta sa využívajú dynamicky prideľované 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 tretej vrstve TCP/IP modelu sa využívajú protokoly.

### 5.2.4 Zdieľaná pamäť

***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 a objaví sa v LAP tohto procesu. Iné procesy si potom môžu tento segment zdieľanej pamäti vo FAP „pripojiť“ ku svojmu vlastnému LAP. To znamená, že rovnaký segment operačnej pamäte počítača sa objaví v LAP niekoľkých procesov (pozri Obr. 2). V praxi je tento proces zložitejší, pretože dostupná pamäť v skutočnosti pozostáva z fyzickej pamäti a stránok pamäti (stránkovacia tabuľka), ktoré sú odložené na disku.

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 uskutočňovanie. Problémy spojené so synchronizáciou prístupu budú podrobnejšie vysvetlené v téme Synchronizácia procesov.




Obr. 2

## 5.3 Spôsob synchronizácie

O časovej závislosti hovoríme v prípadoch, keď výsledok priebehu dvoch alebo viacerých procesov závisí od relatívnej rýchlosti ich vykonávania. Pre riešenie časovej závislosti musí OS poskytnúť mechanizmy pre **synchronizáciu** vykonávania paralelných procesov.

**Semafor S1 a S2**

Pre semafor S1 (S2) sa vytvorí dvojica semaforov. Proces T sa riadi podľa semaforu S1[0] (S2[0]) a proces S (D) sa riadi podľa semaforu S1[1] (S2[1]), pričom nastavený semafor S1[0] (S2[0]) (rozumie sa nastavený na hodnotu 1) znamená, že proces T (S) môže zapisovať do zdieľanej pamäte SM1 (SM2). Nastavený semafor S1[1] (S2[1]) znamená, že proces S môže zo zdieľanej pamäte SM1 (SM2) údaje čítať.

## 5.4 Algoritmy

- **Proces P1**

```
Begin main (int argc, char *argv[])
{
      if(argc  < 3){
                  print error about argc;
                  exit with error 1;
       }
      kill (getpid() , SIGUSR1);
      pipe = the first argument from line in int form (atoi(argv[1]));
      if (opening the first file (p1.txt) has a mistake) {
            print error for opening the file;
            exit with error 1;
      }

      signal(SIGUSR1, writePipe);
      signal(SIGUSR2, killPipe);

      if ( the opened file isn't NULL){
            close the file;
      }

      exit with successful (return 0);
} end main function

void killPipe()
{
      close the opened file;
      print about finished P1;
            exit with successful (return 0);
}

void writePipe()
{
      char* buffer = NULL;
      buffer = calloc(size_t size, sizeof(char));     //sizeof(char) – size
of char
      ssize_t readLine = getline(buufer, size_t size, the opened file);
      ssize_t writeLine = write(pipe, buffer, size of buffer);
```

```
      if(readLine is -1 or writeLine is -1)     //if it has mistake (error in
initialization)
      {
            print error message about problem;
            exit with error (return 1);
      }

      print about successful P1;
      free(buffer);
      print the new line ("\n");
}
```

- **Proces P2**

```
Begin main (int argc, char *argv[])
{
      if(argc  < 3){
                  print error about argc;
                  exit with error 1;
       }
      kill (getpid() , SIGUSR1);
      pipe = the first argument from line in int form (atoi(argv[1]));
      if (opening the second file (p2.txt) has a mistake) {
            print error for opening the file;
            exit with error 1;
      }

      signal(SIGUSR1, writePipe);
      signal(SIGUSR2, killPipe);

      if ( the opened file isn't NULL){
            close the file;
      }

      exit with successful (return 0);
} end main function

void killPipe()
{
      close the opened file;
      print about finished P2;
            exit with successful (return 0);
}

void writePipe()
{
      char* buffer = NULL;
      buffer = calloc(size_t size, sizeof(char));     //sizeof(char) – size
of char
      ssize_t readLine = getline(buufer, size_t size, the opened file);
      ssize_t writeLine = write(pipe, buffer, size of buffer);

      if(readLine is -1 or writeLine is -1)     //if it has mistake (error in
initialization)
```

```
      {
            print error message about problem;
            exit with error (return 1);
      }

      print about successful P2;
      free(buffer);
      print the new line ("\n");
}
```

- **Proces T**

```
the main function begin (int argc, char** argv)
{
      If(argc < 4){
            error about wrong num of parameters;
            exit Fail (return 1);
      }
      int pipe = first parameter;
      int sharedmem = second parameter;
      int semphr = third parameter;
      //all are in int form

      //add additional variables…
      //and checking their parameters

      while(1)
      {
            //change semafor to 0 (write the code)
            semop(semphr, structSmbuff, 1);
            //additional variables for while() and for()
            bzero(string to write, sizeof(char) * size);

            for(; additional variable != '\n'; )
            {
                  If(read(pipe, string to write[side], 1) < 0){
                        break; //end of for()
                  } else {
                        Additional variable (element) = string to write[side]
                        side++;
                  }
            }
            String to write[side - 1] = 0;
            //write copy the string pointer line of write to new smbuffer
            strcpy(smbuffer, writeString);
             writing(to file, smbuffer, size of smbuffer));

            //change semafor to 1(write the code)
            semop(semphr, structSmbuff, 1);
      }
} end of the main function
```

```
void writing(int file, char* smbuffer, size_t size)
{
      write(file, smbuffer, size);
      write(file, "\n", 1);
}
```

- **Proces D**

```
Begin of the main function (int argc, char** argv)
{
      Creating socket;
      Creating connecting;

      While(1)
      {
            Changing semafor
            Check semop(semafor, sem_b, 1) < 0
            If it's < 0, then we will have an error message and exit with
number 1

            If(write(socked, memory, size of memory + 1) != size of memory +
1){
                  Print problem writing into socket;
                  exit with error (return 1);
            } else {
                  sleep(1);
                  write(into file, memory, size of memory);
                  write(into file, "\n", 1);    //new line
            }
            Changing semafor
            Checking semafor
      }
} end of the main function
```

- **Proces Serv2**

```
begin with main function (int argc, char* argv[])
{
      Checking number of parameters, if != 2, than return error message and
exit with error(return 1)
      Create socket;

      Connecting the socket to the server;
      Creating the file serv2.txt
```

```
      Checking his opening, if it has an error, then print error message and
exit with code of error

      for(is active, when we write 10 words)
      {
            recv(socket, buffer, 200, 0);
            write(into file, buffer, size of buffer);
            write(into file, "\n", 1);    //new line
      }

      Killing getppid();
      kill(getppid(), SIGUSR2);

      exit with success (return 0);
}
```

## 6. Záver a zhodnotenie

Hlavným účelom tohto programu je získať poznatky z oblasti medzi procesorovej komunikácie a synchronizácie medzi nimi. Pri plnení tejto úlohy sa viac ponoríte do problematiky procesov a signálov a preskúmate rôzne internetové zdroje, aby ste túto úlohu vyriešili.

## 7. Použitá literatúra a informačné zdroje


- https://hornad.fei.tuke.sk/~genci/Vyucba/OperacneSystemy/SKR2010-2011/08-1-semafory_090128-Popadic.pdf
- https://hornad.fei.tuke.sk/~genci/Vyucba/OperacneSystemy/SKR2010-2011/09-1-networking_090128-Popadic.pdf
- https://hornad.fei.tuke.sk/~genci/Vyucba/OperacneSystemy/SKR2010-2011/07-1-shm_090128-popadic.pdf
- https://sk.wikipedia.org/wiki/Semafor_(programovanie)
- https://www.opennet.ru/docs/RUS/linux_parallel/node7.html
- https://www.youtube.com/watch?v=kP02t8NOqJQ&list=PLdI4_a4AZXjHAHG4qqhIovDQnirO7rWAL&ab_channel=NickChi
- https://man7.org/linux/man-pages/man3/atoi.3.html
- http://linux.die.net/
- http://man.he.net/man2/
- http://sk.wikipedia.org/
- www.stackowerflow.com
- https://man7.org/linux/man-pages/
- https://www.ibm.com/docs/en/search/
- Sofia - (neautorizovaný materiál)


PRÍLOHY

## 8. Systémová príručka

### 8.1 Popis funkcií a štruktúr

- `shmat(int shmid, const void *shmaddr, int shmflg)` - function attaches the shared memory segment associated with the shared memory identifier, *shmid*, to the address space of the calling process.
- `semop(int semid, struct sembuf *sops, size_t nsops)` - The semop() function performs semaphore operations atomically on a set of semaphores associated with argument *semid*. The argument *sops* is a pointer to an array of sembuf data structures. The argument *nsops* is the number of sembuf structures in the array.
- `bzero(void *s, size_t n)` – function places *n* zero-valued bytes in the area pointed to by *s*.
- `socket(int domain, int type, int protocol)` – creates an endpoint for communication and returns a file descriptor that refers to that endpoint
- `bind(int socket, struct sockaddr *address, socklen_t address_len)` - assigns the address specified by &addres to the socket referred to by the file descriptor socket.
- `recv(int socket, void* buffer, size_t length, int flags)` – function shall receive a message from a connection-mode or connectionless-mode socket.
- `read(int fd, void* buf, size_t count)` – attempts to read up to count bytes from file descriptor fd into the budder starting at buf.
- `write()` – writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.

- *open(const char* pathname, int flags, mode_t mode)* – system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open().
- *fopen(const char* restrict pathname, const char* restrict mode)* – function opens the file whose name is the string pointed to by pathname and associates a stream with it.
- *close(int fd)* – closes a file descriptor, so that it no longer refers to any file and may be reused.
- *fclose(FILE* stream)* – function flushes the stream pointed to by stream and closes the underlying file descriptor.
- *getppid(void)* – function shall return the parent process ID of the calling process
- *fork(void)* – creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process .
- *sleep(int number)* – pause for number seconds.
- *execve(const char* pathname, char* const argv[], char *const envp[])* – executes the program referred to by pathname.
- *pipe(int pipefd[2])* – creates a pipe, a unidirectional data channel that can be used for interprocess communication.
- *signal(int signum, sighandler_t handler)* – sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a “signal handler”).
- *kill(pid_t pid, int sig)* – system call can be used to send any signal to any process group or process.
- *shmget(key_t key, size_t size, int shmflg)* – function shall return the shared memory identifier associated with key.
- *atoi(const char* nptr)* – function converts the initial portion of the string pointed to by nptr to int.
- *getline(char **restrict lineptr, size_t *restrict n, FILE* restrict stream)* – reads an entire line from stream, storing the address of the buffer containing the text into *lineptr.
- *exit(int status)* – function causes normal process termination and the least significant byte of status is returned to the parent.
- *shmdt(const void* shmaddr)* – function detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process.
- *shmctl(int shmid, int cmd, struct shmid_ds* buf)* – performs the control operation specified by cmd on the System V shared segment whose identifier is given in shmid.
- *semget(key_t key, int nsems, int semflg)* – system call returns the System V semaphore set identifier associated with the argument key.
- *semctl(int semid, int semnum, int cmd, ...)* – performs the control operation specified by cmd on the System V semaphore set identified by semid, or on the semnum-th semaphore of that set. (The semaphores in a set are numbered starting at 0).

## 9. Príručka používateľa

- **Účel programu**

Hlavným účelom tohto programu je získať poznatky z oblasti medzi procesorovej komunikácie a synchronizácie medzi nimi.

- **Popis spustenia**

Program sa spúšťa z terminálu príkazom *./zadanie <číslo TCP portu> <číslo UDP portu>*. Tento program je spustiteľný na každom počítači s operačným systémom Unix/Linux, pretože využíva služby jadra tohto systému. Pred spustením je nutná kompilácia programu príkazom *make all*, ktorý zabezpečí kompiláciu všetkých procesov vrátane hlavného. Spustenie programu vyžaduje vstupné programy p1.txt a p2.txt. Slova načítané zo súborov putujú medzi procesmi, ktoré postupne k nim pridávajú svoje značky a nakoniec sú zapísané do súboru *serv2.txt*.

## 10. Zdrojový text programu v jazyku C

### 10.1 Proces P1 (súbor proc_p1.c)

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

FILE *fd;
size_t len = 200;
int output;

void writePipe()
{
        char *buffer = NULL;
        buffer = calloc(len, sizeof(char));
        ssize_t readLine = getline(&buffer, &len, fd);        //read a
delimited record from fd (FILE*)
        ssize_t writeLine = write(output, buffer, strlen(buffer));
        if (readLine == -1 || writeLine == -1)
        {
                printf("With read || write pipe proc_p1\n");  //error message
                exit(EXIT_FAILURE);    //exit with failure
        }
        printf("P1: SUCCESSFUL \n");
        free(buffer);
        printf("\n");
}
void killPipe()
{
        fclose(fd);
        printf("P1: finished!\n");
```

```
        exit(EXIT_SUCCESS);
}
int main(int argc, char *argv[])
{
        if (argc < 3)  //check number of parameters
        {
                printf("Usage %s <ppid> <pipe[0]>\n", argv[0]);       //error
message
                exit(EXIT_FAILURE);    //failure exit
        }
        //The kill() function sends a signal to a process or process group
specified by pid (getppid()).
    //The kill() function is successful if the process has permission to send
the signal sig(SIGUSR1) to any of the processes specified by pid (getppid()).
    //If kill() is not successful, no signal is sent.
        kill(getppid(), SIGUSR1);
        output = atoi(argv[1]);               //write the pipe, atoi converts
to int
        if ((fd = fopen("p1.txt", "r")) == NULL)      //open the file and
checks it
        {
                perror("p1.txt wasn't opened!\n");    //error message
                exit(EXIT_FAILURE);    //failure exit
        }
        printf("Signal Send to write \n");    //write current situation
        signal(SIGUSR1, writePipe); //sets the disposition of the signal
SIGUSR1 to writePipe
        signal(SIGUSR2, killPipe); //sets the disposition of the signal
SIGUSR2 to killPipe
        printf("END proc_p1\n");       //write current situation
        while (1)
        {
        pause();
        }
        if (fd != NULL)
                fclose(fd);
        exit(0);
}
```

### 10.2 Proces P2 (súbor proc_p2.c)

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

FILE *fd;
size_t len = 200;
int output;

void writePipe()
{
        char *buffer = NULL;
        buffer = calloc(len, sizeof(char));
        ssize_t readLine = getline(&buffer, &len, fd);        //read a
delimited record from fd (FILE*)
        ssize_t writeLine = write(output, buffer, strlen(buffer));
        if (readLine == -1 || writeLine == -1)
        {
                printf("With read || write pipe proc_p2\n");  //error message
                exit(EXIT_FAILURE);    //exit with failure
        }
        printf("P2: SUCCESSFUL \n");
        free(buffer);
        printf("\n");
}
void killPipe()
{
        fclose(fd);
        printf("P2: finished!\n");
        exit(EXIT_SUCCESS);
}
int main(int argc, char *argv[])
{
        if (argc < 3)  //check number of parameters
        {
                printf("Usage %s <ppid> <pipe[0]>\n", argv[0]);       //error
message
                exit(EXIT_FAILURE);    //failure exit
        }
        //The kill() function sends a signal to a process or process group
specified by pid (getppid()).
    //The kill() function is successful if the process has permission to send
the signal sig(SIGUSR1) to any of the processes specified by pid (getppid()).
    //If kill() is not successful, no signal is sent.
        kill(getppid(), SIGUSR1);
        output = atoi(argv[1]);               //write the pipe, atoi converts
to int
        if ((fd = fopen("p2.txt", "r")) == NULL)      //open the file and
checks it
        {
                perror("p2.txt wasn't opened!\n");    //error message
                exit(EXIT_FAILURE);    //failure exit
        }
        printf("Signal Send to write \n");    //write current situation
```

```
        signal(SIGUSR1, writePipe); //sets the disposition of the signal
SIGUSR1 to writePipe
        signal(SIGUSR2, killPipe); //sets the disposition of the signal
SIGUSR2 to killPipe
        printf("END proc_p2\n");       //write current situation
        while (1)
        {
        pause();
        }
        if (fd != NULL)
                fclose(fd);
        exit(0);
}
```

### 10.3 Proces T (súbor proc_t.c)

```
#include <stdio.h>
#include <stdlib.h>     //atoi
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/shm.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <signal.h>
#include <termios.h>
#include <sys/sem.h>    // struct sembuf, semop

//write define SEM_UNDO..., since I have a problem with decleration in VS
Code
#define SEM_UNDO 0x1000
//struct for work with semop() and etc.
struct sembuf structSmbuff[1];

// создание дополнительных методов
// create methods
void error(char *error);    //for print and return error message
__sighandler_t exitSuccess();   //for successful exit
void change0();
void change1();
void writing(int file, char* smbuffer, size_t size);


int main (int argc, char** argv){

        if (argc < 4){
                error("Wrong num of parameters for proc_t");
        }
    // возможная замена atoi на strol, нужно ещё разобраться, почему
    //converts the string argument argv[1] to an integer (pipe),
    //converts the string argument argv[2] to an integer (sharedmem),
    //converts the string argument argv[3] to an integer (semphr);
    int pipe = atoi(argv[1]), sharedmem = atoi(argv[2]), semphr =
atoi(argv[3]); //советую поменять на strtol
https://stackoverflow.com/questions/17710018/why-shouldnt-i-use-atoi

    //struct sembuf structSmbuff[1] = NULL; //перенёс в глобальную
переменную, для того, чтобы легче было считывать показания в других методах
    char writeString[200]; // массив (хранилище) для хранение слов (размер,
которого составляет в 200 единиц символов)
```

```
    int side = 0;   // для обозначения местоположение курсора в массиве
writeString
    char* smbuffer;
        char element;

    //The shmat() function attaches the shared memory segment associated with
the shared memory identifier, sharedmem, to the address space of the calling
process.
    smbuffer = shmat(sharedmem, NULL, 0);

    //if smbuffer is NULL, than return the error message
    if (smbuffer == NULL) {
        error ("couldn'd attach shared memory");    //prints the error
message and returns EXIT_FAILURE
        exit(1);
        }

    //The kill() function sends a signal to a process or process group
specified by pid (getppid()).
    //The kill() function is successful if the process has permission to send
the signal sig(SIGUSR1) to any of the processes specified by pid (getppid()).
    //If kill() is not successful, no signal is sent.
    kill(getppid(), SIGUSR1);

    // продолжение кода Егора (попытка 1)

    // создание сигнала SIGUSR2 с последующим удачным выходом
    // signal() sets the disposition of the signal "SIGUSR2" to
"exitSuccess()"
    // SIGUSR2 1       Intended for use by user applications
    // If unsuccessful, signal() returns a value of SIG_ERR and a positive
value in errno
    signal(SIGUSR2, exitSuccess);

        //файл, с помощью которого можно узнать, какое слово получил сервак и
отослал
    //open the file
        int file = open("1T.txt", O_CREAT | O_WRONLY | O_TRUNC, 0777);
    // проверка, если с какой-то причины - файл не открылся
    //if opening the file has a problem, than returns the error message
    if(file == -1){
        error("problem with opening the file"); //returns the error message
    }

    // цикл (cycle)
        while(1){
                // вписывание семафору 0
                change0();
        //semop() performs operations on selected semaphores in the set
        //indicated by semphr.  Each of the 1 (nsops) elements in the array
        //pointed to by structSmbuff is a structure that specifies an
operation to
        //be performed on a single semaphore.  The elements of this
        //structure are of type struct sembuf, containing the following
        //members:
            //unsigned short sem_num;  /* semaphore number */
            //short          sem_op;   /* semaphore operation */
            //short          sem_flg;  /* operation flags */
        //Flags recognized in sem_flg are IPC_NOWAIT and SEM_UNDO.  If an
        //operation specifies SEM_UNDO, it will be automatically undone
        //when the process terminates.
                semop(semphr, structSmbuff, 1);
                side = 0;
```

```
                element = 0;
                bzero(&writeString, sizeof(char)*200);

                // чтение с pipeR2
        //read form the pipeR2
        for( ; element != '\n'; )
        {
            //read from pipe to writeString by 1 element
            if (read(pipe, &writeString[side], 1) < 0){
                                break;
                        } else {
                                element = writeString[side];
                                side++;
                        }
            //element = one of the element from writeString[]
        }

                writeString[side-1] = 0; //maybe should be side-1 but i'm not
sure




                // запись с переменной для помощи в память
        //copy information
        //The strcpy() function copies the string pointed by
writeString(source) (including the null character) to the smbuffer
(destination).
                strcpy(smbuffer, writeString);  // копирование строки
        //write into the file all smbuffer content
                writing(file, smbuffer, strlen(smbuffer));  //запись контента
в файл с массива, с последующим окончание заполнения, через "\n"
                side = 0;

                // вписывание семафору 1
                change1();
                semop(semphr, structSmbuff, 1);
        }

}

// метод для вывода ошибки
void error(char *error) {
    fprintf(stderr,"[./proc_t] %s", error);
    exit(EXIT_FAILURE);
}

//  ПРОДОЛЖЕНИЕ КОДА ЕГОРА
// метод для удачного выхода (причина создания - невозможно поменять void
exit() на __sighandler_t exit())
__sighandler_t exitSuccess(){
    exit(EXIT_SUCCESS);
}
// смена семафоры на 0 (write the theory before, look for the lines 75-86)
void change0(){
    structSmbuff[0].sem_num = 0;
        structSmbuff[0].sem_op = -1;
        structSmbuff[0].sem_flg = SEM_UNDO;
}
// смена семафоры на 1 (write the theory before, look for the lines 116-127)
void change1(){
    structSmbuff[0].sem_num = 1;
        structSmbuff[0].sem_op = 1;
```

```
        structSmbuff[0].sem_flg = SEM_UNDO;
}
// запись в файл
//write into the file smbuffer content with "\n" at the end
void writing(int file, char* smbuffer, size_t size){
    write(file, smbuffer, size);
    write(file, "\n", 1);
}
```

### 10.4 Proces D (súbor proc_d.c)

```
#include <stdio.h>
#include <errno.h>   // for errno
#include <limits.h>  // for INT_MAX, INT_MIN
#include <stdlib.h>  // for strtol
#include<stdbool.h>  //bool function
#include <signal.h>  //use signal
#include <netinet/in.h> //AF_INET.
#include <string.h>
#include <sys/sem.h> //sembuf
#include <unistd.h>  //I/O primitives fork,pipe
#include <sys/types.h> //pid_t
#include <sys/ipc.h> // All use a common structure type, ipc_perm to pass
information used in determining
#include <sys/shm.h>  //schmat
#include <arpa/inet.h> //inet_addr
#include <fcntl.h>


void killSignal()
{
        printf("Turn OFF\n");
        exit(EXIT_SUCCESS);
}

int main(int argc, char ** argv)
{

    int memory = atoi(argv[1]);
    int semafor = atoi(argv[2]);
    int port = atoi(argv[3]);

    char* shared_memory;
    shared_memory = shmat(memory, NULL, 0);
    if(shared_memory == NULL)
    {
        perror("Problem with shmat");
        exit(1);
    }
    else
    {
        printf("Memory work proc_d\n");
    }

    kill(getppid(), SIGUSR1);
    signal(SIGUSR2, killSignal);

    int socked = socket(AF_INET, SOCK_STREAM, 0);
    if(socked < 0)
    {
        perror("Socket proc_d");
        exit(1);
```

```
    }
    else
    {
        printf("Socket work proc_d\n");
    }

    //char buffer[1024];
        struct sembuf sem_b[1];

        struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
        serv_addr.sin_port = htons(port);
        bzero(&serv_addr.sin_zero, 8);

        if(connect(socked,(struct sockaddr*)&serv_addr,sizeof(serv_addr))<0)
    {
                perror("Connect proc_d");
                exit(1);
        }
    int fd=open("1D.txt",O_CREAT|O_TRUNC|O_WRONLY,0777);
    while(1)
    {
        sem_b[0].sem_num = 1;
        sem_b[0].sem_op = -1;
        sem_b[0].sem_flg = SEM_UNDO;
        if(semop(semafor, sem_b, 1) < 0)
        {
            perror("Semop proc_d");
            exit(1);
        }
        if (write(socked, shared_memory, strlen(shared_memory)+1) !=
(int)strlen(shared_memory)+1)
        {
                        printf("Problem write to socket ");
                        exit(EXIT_FAILURE);
                }
        else
        {
                        sleep(1);
                        write(fd, shared_memory,strlen(shared_memory));
                        write(fd, "\n",1);
                }

        //bzero(shared_memory, 1024*sizeof(char));
        sem_b[0].sem_num = 0;
        sem_b[0].sem_op = 1;
        sem_b[0].sem_flg = SEM_UNDO;
        if(semop(semafor,sem_b,1) < 0)
        {
                        perror("Semop2 proc_d");
                        exit(1);
                }

    }
}
```

### 10.5 Proces Serv2 (súbor proc_serv2.c)

```
#include <stdio.h>  //for perror()
#include <stdlib.h> //for atoi()
#include <fcntl.h>  //for open()
#include <sys/socket.h> //for socket(), bind()
#include <netinet/in.h> //for struct sockaddr_in
#include <string.h>
#include <unistd.h> //for read(), write()
#include <arpa/inet.h>
#include <signal.h>
#include <termios.h>


int getch(void);
void error(char *errorMessage); //a method for fprintf error mistake

//the main function
int main(int argc, char* argv[]){
    //checks if input arguments are less, or more than 2 => returns the error
message
    if(argc != 2){
        error("Error: Incorrect number of input arguments\n");
    }

    int firstPort = atoi(argv[1]);  //converts the string argument argv[1] to
an integer (firstPort)
    //if firstPort is 0, than we will have a mistake (error message)
    if(firstPort == 0){ //checks, if converting has a mistake
        error("Error: Input argument isn't number\n");  //return the mistake
message
    }

    int firstSocket = socket(AF_INET, SOCK_DGRAM, 0);   // function shall
create an unbound socket in a communications domain
    //AF_INET (AF - Address Family) -  communication domain, refers to
addresses from the internet, IP addresses specifically
    //SOCK_DGRAM - Provides datagrams, which are connectionless-mode,
unreliable messages of fixed maximum length.

    if(firstSocket == -1){ //if we can't create a socket, than returns error
message
        error("Error in creating a socket\n");
    }

    //open the file "serv2.txt", with flags and 0777 mod
    int file = open("serv2.txt", O_CREAT | O_WRONLY | O_TRUNC, 0777);
    if(file == -1){ //if we can't open the file, than it returns error
message
        error("Error in opening the file\n");
    }


    struct sockaddr_in client;  //the basic structure for all syscalls and
functions that deal with internet addresses
    // Filling server information

    /*      Information about "struct sockaddr_in" and "struct in_addr"
    struct sockaddr_in {
        short            sin_family;   // e.g. AF_INET
        unsigned short   sin_port;     // e.g. htons(3490)
        struct in_addr   sin_addr;     // see struct in_addr, below
        char             sin_zero[8];  // zero this if you want to
```

```
    };

    struct in_addr {
        unsigned long s_addr;  // load with inet_aton()
    };
    */

    client.sin_family = AF_INET; //IPv4 protocol
    // inet_aton("127.0.0.1", &client.sin_addr.s_addr); or we can write like
this:
    //function shall convert the string pointed to by cp, in the standard
IPv4 dotted decimal notation, to an integer value suitable for use as an
Internet address
    client.sin_addr.s_addr = INADDR_ANY;
    client.sin_port = htons(firstPort); //translates a short integer from
host byte order to network byte order

    //assigns the address specified by &client to the socket referred to by
the file descriptor firstSocket.
    // sizeOfSocket specifies the size, in bytes, of the address structure
pointed to by client
    socklen_t sizeOfSocket = sizeof(client);
    if(bind(firstSocket, (struct sockaddr*) &client, sizeOfSocket) < 0){
        error("Error in bind\n");   //if it has an error, than it returns the
error message
    }
    //The kill() function sends a signal to a process or process group
specified by pid (getppid()).
    //The kill() function is successful if the process has permission to send
the signal sig(SIGUSR1) to any of the processes specified by pid (getppid()).
    //If kill() is not successful, no signal is sent.
    kill(getppid(), SIGUSR1);

    char buffer[200];  //create a buffer for read and write the file's
content

    int element, input;

        for (int i = 0; i < 10; i++){
                recv(firstSocket, buffer, 200, 0);
                write(file, buffer, strlen(buffer));
                write(file, "\n", 1);
        }

    //The kill() function sends a signal to a process or process group
specified by pid (getppid()).
    //The kill() function is successful if the process has permission to send
the signal sig(SIGUSR2) to any of the processes specified by pid (getppid()).
    //If kill() is not successful, no signal is sent.
    kill(getppid(), SIGUSR2);

    //the exit of the program
    // close(firstSocket);
    exit(EXIT_SUCCESS);

    return 0;
}

// a method for an error message and exit with failure
void error(char *errorMessage){
    fprintf(stderr,"[./proc_serv2] %s", errorMessage);

    exit(EXIT_FAILURE);
```

```
}
```

### 10.5 Proces Zadanie (súbor zadanie.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <stdbool.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <sys/wait.h>

void p1synch();
void p2synch();
void p4synch();
void p5synch();
void p6synch();
void p7synch();
void p8synch();
void last();


union semun
{
        int val;
        struct semid_ds *buf;
        unsigned short *array;
}semafor_struct;



int Server1, Server2, D, P1, P2, PR, T, S;

int progress = 0, ended = 0;

int s1_synch = 0;

void finished() {
    sleep(5);
    kill(P1, SIGKILL);
    kill(P2, SIGKILL);
    kill(PR, SIGKILL);
    kill(T, SIGKILL);
    kill(S, SIGKILL);
    kill(D, SIGKILL);
    kill(Server1, SIGKILL);
    kill(Server2, SIGKILL);
}

int main(int argc, char * argv[]) {
    if (argc != 3) {
        printf("Wrong number of arguments\n");
        exit(0);
    }
```

```
int serv1, serv2, shmem1, shmem2;
int ppr1[2], ppr2[2];

char pipe1_read_char[10];
char pipe2_read_char[10];
char pipe2_write_char[10];
char pipe1_write_char[10];

char semaphore1_char[10];
char semaphore2_char[10];

char sharedmem1_char[10];
char sharedmem2_char[10];

char P1_descriptor[10];
char P2_descriptor[10];

char server1_char[10];
char server2_char[10];

if (pipe(ppr1) == -1)
    printf("Couldn't create pipe r1\n");
if (pipe(ppr2) == -1)
    printf("Couldn't create pipe r2\n");

int pipe1_read = dup(ppr1[0]);
int pipe1_write = dup(ppr1[1]);

int pipe2_read = dup(ppr2[0]);
int pipe2_write = dup(ppr2[1]);

int sem1 = semget(4001, 2, 0666 | IPC_CREAT);
int sem2 = semget(4010, 2, 0666 | IPC_CREAT);

if (sem1 == -1) {
    printf("Couldn't create sem1\n");
    fprintf(stderr, "semget failed\n");
           exit(EXIT_FAILURE);
}
if (sem2 == -1){
    printf("Couldn't create sem2\n");
    fprintf(stderr, "semget failed\n");
           exit(EXIT_FAILURE);
}

semctl(sem1, 0, SETVAL, 0);
semctl(sem2, 0, SETVAL, 0);
semctl(sem1, 1, SETVAL, 0);
semctl(sem2, 1, SETVAL, 0);

    semafor_struct.val = 1;
    semctl (sem1, 0, SETVAL, semafor_struct);
    semafor_struct.val = 0;
    semctl (sem1, 1, SETVAL, semafor_struct);

    semafor_struct.val = 1;
    semctl(sem2, 0, SETVAL, semafor_struct);
    semafor_struct.val = 0;
    semctl(sem2, 1, SETVAL, semafor_struct);

//shared memory

if ((shmem1 = shmget(2005, 200*sizeof(char), 0666 | IPC_CREAT)) == -1){
```

```
        printf("Couldn't create shared memory 1\n");
               fprintf(stderr, "shmget failed\n");
               exit(EXIT_FAILURE);
    }
    if ((shmem2 = shmget(2007, 200*sizeof(char), 0666 | IPC_CREAT)) == -1){
        printf("Couldn't create shared memory 2\n");
               fprintf(stderr, "shmget failed\n");
               exit(EXIT_FAILURE);
    }


    memset(pipe1_read_char, '\0', sizeof(pipe1_read));
    memset(pipe2_read_char, '\0', sizeof(pipe2_read));
    memset(pipe1_write_char, '\0', sizeof(pipe1_write));
    memset(pipe2_write_char, '\0', sizeof(pipe2_write));
    memset(semaphore1_char, '\0', sizeof(sem1));
    memset(semaphore2_char, '\0', sizeof(sem2));
    memset(sharedmem1_char, '\0', sizeof(shmem1));
    memset(sharedmem2_char, '\0', sizeof(shmem2));

    sprintf(pipe1_read_char, "%d", pipe1_read);
    sprintf(pipe2_read_char, "%d", pipe2_read);
    sprintf(pipe1_write_char, "%d", pipe1_write);
    sprintf(pipe2_write_char, "%d", pipe2_write);
    sprintf(semaphore1_char, "%d", sem1);
    sprintf(semaphore2_char, "%d", sem2);
    sprintf(sharedmem1_char, "%d", shmem1);
    sprintf(sharedmem2_char, "%d", shmem2);

    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p1synch);
    P1 = fork();

    switch (P1) {
        case 0:
            printf("P1 is starting...\n");
            execl("proc_p1", "proc_p1", pipe1_write_char, pipe1_read_char,
(char * ) NULL);
            exit(0);
            break;
        case -1:
            printf("P1 wasn't started\n");
            break;
            exit(1);
    }

    while (progress != 1) {
        printf("Waiting for P1 to start...\n");
        sleep(3);
    }

    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p2synch);

    P2 = fork();

    switch (P2) {
        case 0:
            printf("P2 was started successfully\n");
            execl("proc_p2", "proc_p2", pipe1_write_char, pipe1_read_char,
(char * ) NULL);
```

```
            exit(0);
            break;
        case -1:
            printf("P2 wasn't started\n");
            exit(1);
            break;
    }

    while (progress != 3) {
        printf("Waiting for P2 to start...\n");
        sleep(3);
    }

    ////////////////////////////////////////////////////////////////////////

    memset(P1_descriptor, '\0', sizeof(P1));
    memset(P2_descriptor, '\0', sizeof(P2));

    sprintf(P1_descriptor, "%d", P1);
    sprintf(P2_descriptor, "%d", P2);

    ////////////////////////////////////////////////////////////////////////

    // signal(SIGUSR1, p3synch);
    PR = fork();
    switch (PR) {
        case 0:
            printf("Process PR was forked\n");
            execl("proc_pr", "proc_pr", P1_descriptor, P2_descriptor,
pipe1_read_char, pipe2_write_char, NULL);
            exit(EXIT_SUCCESS);
            break;
        case -1:
            perror("Process D wasn't forked\n");
            exit(EXIT_FAILURE);
    }

    int state;
        waitpid(PR, &state, WUNTRACED);
        kill(P1,SIGUSR2);
        kill(P2,SIGUSR2);


    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p4synch);
    signal(SIGUSR2, last);
    Server2 = fork();

    switch (Server2) {
        case 0:
            printf("Server 2 was forked\n");
            execl("proc_serv2", "proc_serv2", argv[2], (char * ) NULL);
            exit(0);
            break;
        case -1:
            perror("Couldn't fork server 2");
            exit(1);
            break;
    }

    while (progress != 4) {
        printf("Waiting for serv2 to start... %d %d\n", progress, s1_synch);
```

```
        sleep(3);
    }

    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p5synch);
    Server1 = fork();

    switch (Server1) {
        case 0:
            printf("Server 1 was forked\n");
            execl("proc_serv1", "proc_serv1", argv[1], argv[2], (char * )
NULL);
            exit(0);
            break;
        case -1:
            perror("Couldn't fork server 1");
            exit(1);
            break;
    }

    while (progress != 5) {
        printf("Waiting for serv1 to start...\n");
        sleep(3);
    }

    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p6synch);
    T = fork();
    switch (T) {
        case 0:
            printf("Process T was forked\n");
            execl("proc_t", "proc_t", pipe2_read_char, sharedmem1_char,
semaphore1_char, NULL);
            exit(EXIT_SUCCESS);
            break;
        case -1:
            perror("Process T wasn't forked\n");
            exit(EXIT_FAILURE);
    }

    while (progress != 6) {
        printf("Waiting for T to start...\n");
        sleep(3);
    }

    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p7synch);

    D = fork();

    switch (D) {
        case 0:
            printf("Process D was forked\n");
            execl("proc_d", "proc_d", sharedmem2_char, semaphore2_char,
argv[1], (char * ) NULL);
            exit(0);
            break;
        case -1:
            perror("Process D wasn't forked");
            exit(0);
```

```
            break;
    }

    while (progress != 7) {
        printf("Waiting for D to start...\n");
        sleep(3);
    }

    ////////////////////////////////////////////////////////////////////////

    signal(SIGUSR1, p8synch);
    S = fork();
    switch (S) {
        case 0:
            printf("Process S was forked\n");
            execl("proc_s", "proc_s", sharedmem1_char, semaphore1_char,
sharedmem2_char, semaphore2_char, NULL);
            exit(EXIT_SUCCESS);
            break;
        case -1:
            perror("Process S wasn't forked\n");
            exit(EXIT_FAILURE);
    }

    while (progress != 8) {
        printf("Waiting for end...\n");
        sleep(3);
    }

    while (ended == 0 && progress == 8){
        printf("Almost done...\n");
        sleep(3);
    }

    finished();

    close(atoi(argv[1]));
    close(atoi(argv[2]));
    sleep(3);
    printf("\nEnd of zadanie!\n");

    return 0;
}

void last(){
    printf("wrapping up...\n");
    ended = 1;
}

void p1synch() {
    if (progress == 0) {
        printf("Process p1 was done\n");
        progress = 1;
    }
}

void p2synch() {
    if (progress == 1) {
        printf("Process p2 was done\n");
        progress = 3;
    }
}
```

```
void p4synch() {
    if (progress == 3) {
        printf("Process serv2 was done\n");
        progress = 4;
    }
}

void p5synch() {
     if (progress == 4) {
        printf("Process serv1 was done\n");
        progress = 5;
    }
}

void p6synch() {
    if (progress == 5) {
        printf("Process T was done\n");
        progress = 6;
    }
}

void p7synch() {
    if (progress == 6) {
        printf("Process D was done\n");
        progress = 7;
    }
}

void p8synch() {
    if (progress == 7) {
        printf("Process S was done\n");
        progress = 8;
    }
}
```