TECHNICKÁ UNIVERZITA KOŠICE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

OPERAČNÉ SYSTÉMY

(ZADANIE UNIX1)

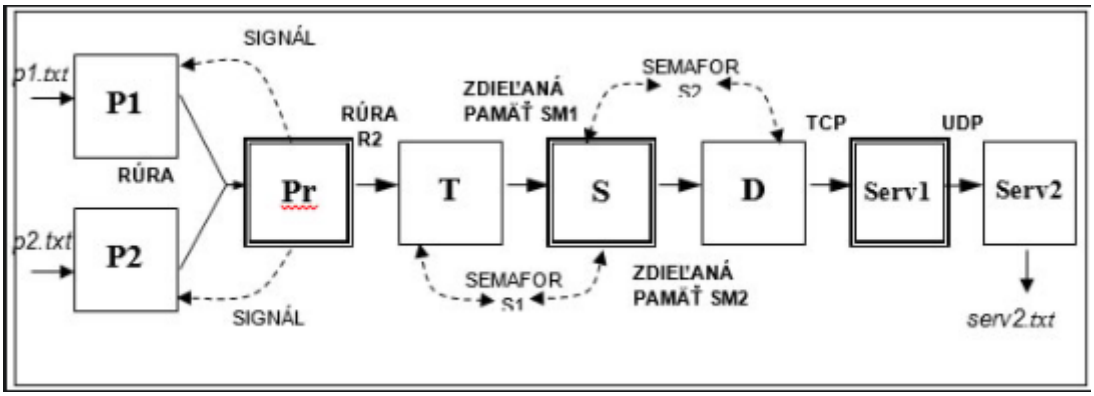
Vladyslav Todosiuk
Andrii Ivashkov
Ishtvan Danov
Šk. rok 2021

Obsah

Prilohy

1. Text zadania

1.1 Schema



- 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
```

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`

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!

„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.

1.4 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*.

1.5 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.

3. Popis relevantných štruktúr, algoritmov, dátových typov, konštant

struct sembuf sem_b[1] - štruktúra slúži na prácu so semaforom pri sluzbe jadra semop().

struct sockaddr_in server_address - štruktúra je potrená pre služby s prácou so servermi

struct hostent *server - štruktúra je potrebná na získanie IP adresy podľa mena uzla.

3.1 Process 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.

Dátové typy:

3.2 Process p1

Spúštanie procesu : proc_p1 <identifikačné číslo rúry 1 na zápis >

Algoritmus :

- Proces otvorí súbor p1.txt a nacita slovo do rury 1, ak mu proces PR pošle signál.

Dátové typy:

3.3 Process p2

Spúštanie 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.

Dátové typy:

3.4 Process T

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

Algoritmus :

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

Dátové typy:

3.5 Process D

Spúštanie procesu proc_d<identifikačné číslo zdieľanej pamäte 2, identifikačné číslo semaforu 2, číslo tcp portu>

Algoritmus :

- Proces pomocou semaforu zakáže zápis do zdielanej pamate 2, načíta zo zdieľanej pamate 2 slovo po znak noveho riadka a posle ho cez tcp port na server.

Dátové typy:

3.6 Process Serv2

Spúšťanie procesu proc_serv2< číslo udp portu>

Algoritmus :

- Proces vytvorí udp server, prijme slovo od tcp servera a zapise ho do suboru serv2.txt

Dátové typy:

```
struct sockaddr_in server_addr;  //obsahuje adresu servera

int sockfd, portno, i = 0; //pomocne premenne

char buffer[256]; //buffer pre ulozenie znakov zo socketu

FILE *outfile; //file
```

4. Analýza problematiky

Program:

- Vyhradí všetky zdroje pre medzi procesorovú komunikáciu
  - rúry R1 a R2
  - zdieľané pamäte SM1 a SM2
  - semafory S1 a S2

- Spúšťa procesy proc_p1, proc_p2, proc_pr, proc_t, proc_s, proc_serv2, proc_serv1 a proc_d.
  Po ich spustení počká istý čas aby sa vykonali procesy korektne.

- Výnimkou sú procesy proc_p1 a proc_p2, a proc_pr, kde zadanie čaká na ukončenie proces proc_pr pomocou waitpid(pid, -*status, options). Ak proces proc_pr skončí, skončí zapisovanie do rúr a ukončia sa aj proces proc_p1 a proc_p2 použitím procesu kill(pid,SIGINT) pre ukončenie.

Nakoniec program čaká aj na ukončenie procesu proc_serv2, ten skončí po zapísaní slovo do súboru serv2.txt. Po ukončený procesu program ukončí aj zvyšné procesy proc_t, proc_s, proc_d a proc_serv1. Nevýhodou je , že ak sa program Serv2 neukončí správne, tak sa procesy neukončia a ostanú visieť tcp a aj udp spojenie.

5. Popis navrhovaného riešenia

5.1 Návrh riešenia

Proces zadanie čaká, kým sa ukončí proces Serv2, pretože niekedy sa nestihli vykonať procesy a proces zadanie vymazal semafory a zdieľané pamäte, čo viedlo ku chybe výsledku.

Niekedy nastaval problém so synchronizáciou, lebo procesy sa nespustili v správnom poradí, tak je v procese zadanie dodefinovaná globálna premenná pripravený, ktorá je po poslaní signálu, že je proces pripravený nastavená na 0. Po každom spustení procesu sa spusti cyklus while(pripraveny), ktorý zastaví ďalšie vykonávanie.

5.2 Dátové štruktúry

5.2.1 Rúry

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.

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 <0,max (intiger)>, 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 nepreruš iteľné operácie P a V (buď sa vykoná celá naraz, alebo sa nevykoná vôbec).

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.

5.2.4 Zdiesana pamet

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.

5.3 Spôsob synchronizácie

Semafor S1

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

Semafor S2

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

1. Záver a zhodnotenie
2. Použitá literatúra a informačné zdroje

5.4 Algoritmy

5.4.1 Process p1

```
Begin
{
    if(file == NULL){
       zlý počet argumentov;
       koniec;
   }
   pipe1 <- prvý argument funkcie;

   kill(getppid(), SIGUSR1); - poslanie signálu hlavnej funkcii, že je proces pripravený
signal(SIGUSR1, Signal); - definovanie funkcie, ktorá sa zavolá , keď proces
dostane signál

}End
```

```
void sig_handler(int signum)
```

{
   Buffer <- fgets(1024, fd);
```
int size = strlen(buff);
```
   while(size > 0)
```
        written = write(write_pipe, buf, size);
```
}

5.4.2 Process p2

Begin
if(file == NULL){
      zlý počet argumentov;
      koniec;
   }

   pipe1 <- prvý argument funkcie;

   kill(getppid(), SIGUSR1); - poslanie signálu hlavnej funkcii, že je proces pripravený
signal(SIGUSR1, Signal); - definovanie funkcie, ktorá sa zavolá , keď proces
dostane signál
}End
```
void sig_handler(int signum)
```
{
   Buffer <- fgets(1024, fd);
```
int size = strlen(buff);
```
   while(size > 0)
```
        written = write(write_pipe, buf, size);
```
}

5.4.3 Process T

Begin{
while(1)
{
  Semafor zákaze čítanie zo zdieľanej pamäte;
```
  While(1)
```
  {
    Write ("T: znizenie hodnoty semaforu S1[0] zlyhalo!\n");
    Koniec;
  }
 \\ načítanie slova po znak nového riadku
  a = 0;
  while(data[a-1] != '\n')
  {
    read(pipe2,&data[a++],1);
  }
// zkopirovanie do zdieľanej pamäte
strncpy(share,data,i);
odblokovanie čítania zo zdieľanej pamäte procesu S ...

}
}End

5.4.4 Process D

Begin
{
Vytvorenie socketu;
Vytvorenie spojenia;

while(1)
{
Semafor zákaze zápis do zdieľanej pamäte;

Vynulovanie premennej data;
strcpy(data,share); - zkopirovanie zdieľanej pamäte do premennej data;

}
}End

5.4.5 Process Serv2

Begin
{
Vytvorenie socketu;
Pripojenie socketu k serveru;

```
for (letter = 0; letter < 10; letter++) {
    recv(sockfd, buff, 200, 0);
    write(outfile, buff, strlen(buff));
    write(outfile, "\n", 1);
    printf("Serv2: %s\n", buff);
}
```

}End

6. Zaver a hodnotenie

V tomto projekte s mohli sme s kolegami vytvoriť zadanie v grupe. Nemali sme žiadne komunikačné problémy, ale mali sme nejaké problémy s úlohou. Pretože veľa vedomostí bolo potrebné získať samoštúdiom. V súbore unix1 bylo malo informácií o niektorých procesov takých ak proc_D a proc_T. Ale vo všeobecnosti bolo informácií dosť a smohli sme vyriešiť všetky problémy a urobiť správny projekt.

7. Použitá literatúra a informačné zdroje

- https://man7.org/linux/man-pages/man2/kill.2.html
- https://www.geeksforgeeks.org/udp-server-client-implementation-c/
- https://man7.org/linux/man-pages/man2/getpid.2.html
- man7.org
- 2016 – Sofia – Pipes – nepomenovane, pomenované (neautorizovany material)
- 2016 – Sofia – Signaly (neautorizovany material)
- 2016 – Sofia – Zdieľaná pamäť (neautorizovany material)
- 2016 – Sofia – Semafory (neautorizovany material)
- 2016 – Sofia – Sockety (Networking) (neautorizovany material)

Prilohy

8.Systémová príručka

8.1 Popis funkcií a štruktúr

- **shmat**() zdieľanú pamäť pripojí k adresnému priestoru daného procesu
- **shmdt**()odpojí zdieľanú pamäť od aktuálneho procesu, ale ju nevymaže.
- **shmctl**() slúži na ovládanie vytvorených zdieľaných pamätí.
- **semget**() slúži na vytvorenie sady semaforov a vráti ich identifikátor
- **semctl**() inicializuje a prečíta hodnoty semaforov zo sady semaforov alebo
- prípadne sadu semaforov odstráni zo systému.
- **socket**() vytvorí socket pre komunikáciu.
- **bind**() pripojí socketu IP adresu a port
- **read**() vracia počet načítaných bajtov alebo 0 keď dosiahne koniec súboru alebo -1pri chybe.
- **write**() vracia počet zapísaných bajtov alebo -1 pri chybe
- **open**() vracia file deskriptor súboru alebo -1 pri chybe
- **close**() vracia: 0 keď OK alebo -1 pri chybe
- **getpid**() vracia ID procesu
- **fork**() vytvorí (takmer) identický proces
- **waitpid**() čaká kým sa neskončí proces s id
- **pipe**() vytvorí rúru a sprístupni ju prostredníctvom dvoch deskriptorov súborov
- **signal**() čaká na špecifický signál
- **kill**() umožňuje zaslať signál danému procesu.
- **shmget**() získa identifikátor a vytvorí zdieľanú pamäť.

9. Príručka používateľa

9.1 Účel program

Hlavným účelom tohto programu je získať poznatky z oblasti medzi procesorovej komunikácie a synchronizácie medzi nimi, čo nie je jednoduché dosiahnuť.

9.2 Popis spustenia

Program sa spúšťa z terminálu zadaním príkazu ./zadanie <číslo tcp portu> <číslo udp portu>. Tento program je spustiteľný na každom počítači s operačným systémom Unix, pretože využíva služby jadra tohto systému. Pred spustením je nutná kompilácia programu zadaním príkazu ./makefile , 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 program v jazyku C

10.1 Process zadanie

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <errno.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdbool.h>
bool ends = false;
int a = 0;

void launch(int signum) {
    if (a == 1)
        printf("Launch process!\n");
}
void end(int signum) {
    ends = true;
}

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

    if (argc < 3) {
        exit(EXIT_FAILURE);
    } else {
        signal(SIGUSR1, launch);
   }

    int stat, stat2;
    key_t fkey1, fkey2;
    int sem1, sem2;
    int res = 0;
    int shm1, shm2;
    int p1[2], p2[2];
    pid_t pid_1, pid_2 , pid_pr, pid_pt, pid_ps, pid_pd, pid_ps1, pid_ps2;
    char pr1[32], pw1[32], pr2[32], pw2[32], pid_p1[32], pid_p2[32],
pid_sem1[32], pid_sem2[32], pid_shm1[32], pid_shm2[32];

    if (pipe(p1) == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }
    if (pipe(p2) == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }

    sprintf(pr1, "%d", p1[0]);
    sprintf(pw1, "%d", p1[1]);
```

```
    sprintf(pw2, "%d", p2[1]);

    //fork 1
    pid_1 = fork();
    if (pid_1 < 0) {
        fprintf(stderr, "Fork p2 is failed!");
        return 1;
    }
    else if (pid_1 == 0) {
        execl("./proc_p1", "proc_p1", pw1, "p1.txt", NULL);
        return 0;
    }
    pause();
    pid_2 = fork();
    if (pid_2 < 0) {
        fprintf(stderr, "Fork p2 is failed!");
        return 1;
    }
    else if (pid_2 == 0) {
        execl("./proc_p2", "proc_p2", pw1, "p2.txt", NULL);
        return 0;
    }
    pause();
    sprintf(pid_p1, "%d", (int)pid_1);
    sprintf(pid_p2, "%d", (int)pid_2);
    pid_pr = fork();

    if (pid_pr < 0) {
        fprintf(stderr, "Fork pr is failed!");
        return 1;
    }
    else if (pid_pr == 0) {
        execl("./proc_pr", "proc_pr", pid_p1, pid_p2, pr1, pw2, NULL);
        return 0;
    }
    waitpid(pid_pr, &stat, WUNTRACED);
    kill(pid_1, SIGTERM);
    kill(pid_2, SIGTERM);
    signal(SIGTERM, end);
   fkey1 = ftok(".", 1);
    fkey2 = ftok(".", 2);
    shm1 = shmget(fkey1, 1024, 0660 | IPC_CREAT);
    if (shm1 == -1) {
      exit(EXIT_FAILURE);
    }
    shm2 = shmget(fkey2, 1024, 0660 | IPC_CREAT);
    if (shm2 == -1) {
      exit(EXIT_FAILURE);
    }

    sem1 = semget(fkey1, 2, 0660 | IPC_CREAT);
    if (sem1 == -1) {
      exit(EXIT_FAILURE);
    }
    sem2 = semget(fkey2, 2, 0660 | IPC_CREAT);
    if (sem2 == -1) {
      exit(EXIT_FAILURE);
    }


   //Inicialization semafor
    res = semctl(sem1, 0, SETVAL, 1 );
   if(res == -1){
      exit(EXIT_FAILURE);
```

```
    }
     res = semctl(sem1, 1, SETVAL, 0 );
     if(res == -1){
       exit(EXIT_FAILURE);
    }
     res = semctl(sem2, 0, SETVAL, 1 );
    if(res == -1){
       exit(EXIT_FAILURE);
    }
     res = semctl(sem2, 1, SETVAL, 0 );
     if(res == -1){
       exit(EXIT_FAILURE);
    }

     sprintf(pr2, "%d", p2[0]);
     sprintf(pid_sem1, "%d", sem1);
     sprintf(pid_shm1, "%d", shm1);
     sprintf(pid_sem2, "%d", sem2);
     sprintf(pid_shm2, "%d", shm2);

     pid_ps1 = fork();

     if (pid_ps1 < 0) {
         fprintf(stderr, "Fork proc_serv1 is failed!");
         return 1;
     }
     else if (pid_ps1 == 0) {
         execl("./proc_serv1", "proc_serv1", argv[1], argv[2], NULL);
         return 0;
     }
     sleep(1);
     //pause();
     pid_ps2 = fork();

     if (pid_ps2 < 0) {
         fprintf(stderr, "Fork proc_serv2 is failed!");
         return 1;
     }
     else if (pid_ps2 == 0) {
         execl("./proc_serv2", "proc_serv2", argv[2], NULL);
         return 0;
     }
     sleep(1);
     pid_pt = fork();
     if (pid_pt < 0) {
         fprintf(stderr, "Fork proc_t is failed!");
         return 1;
     }
     else if (pid_pt == 0) {
         execl("./proc_t", "proc_t", pr2, pid_shm1, pid_sem1, NULL);
         return 0;
     }
     sleep(1);


     pid_ps = fork();

     if (pid_ps < 0) {
         fprintf(stderr, "Fork proc_s is failed!");
         return 1;
     }
     else if (pid_ps == 0) {
         execl("./proc_s", "proc_s", pid_shm1, pid_sem1, pid_shm2, pid_sem2,
NULL);
```

```
        return 0;
    }
    sleep(1);

    pid_pd = fork();
    a++;
    if (pid_pd < 0) {
        fprintf(stderr, "Fork proc_s is failed!");
        return 1;
    }
    else if (pid_pd == 0) {
        execl("./proc_d", "proc_d", pid_shm2, pid_sem2, argv[1], NULL);
        return 0;
    }
    sleep(1);
    while (!ends) {
        sleep(2);
    }

    kill(pid_pt, SIGTERM);
   kill(pid_ps, SIGTERM);
    kill(pid_pd, SIGTERM);
   sleep(1);
   semctl(sem1, 0, IPC_RMID);
   semctl(sem2, 0, IPC_RMID);

   shmctl(shm1, IPC_RMID, NULL);
   shmctl(shm2, IPC_RMID, NULL);

    close(atoi(argv[2]));
   close(atoi(argv[1]));
    printf("End of program.\n");
    exit(EXIT_SUCCESS);

    exit(EXIT_SUCCESS);
}
```

10.2 Process p1

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

FILE* fd = NULL;
FILE* fderr = NULL;
int write_pipe = 0;

void sig_handler(int signum) {
    if (signum == SIGUSR1) {
        char buff[1024];
        if (fgets(buff, 1024, fd) == NULL) {
            exit(EXIT_SUCCESS);
        }
```

```
        int size = strlen(buff);
        int written = 0;
        char* buf = buff;
        while(size > 0) {
            written = write(write_pipe, buf, size);
            if (written == -1) {
                break;
            }
            buf += written;
            size -= written;
        }
    }
}

void end_of_proc(int signum){
    exit(EXIT_SUCCESS);
}

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

    write_pipe = atoi(argv[1]);
    fd = fopen("p1.txt", "r");

    if (fd == NULL) {
        printf("Error: file is not exist!");
        return -1;
    }

    signal(SIGUSR1, sig_handler);
    signal(SIGTERM, end_of_proc);
   if(kill(getppid(), SIGUSR1) < 0){
        exit(EXIT_FAILURE);
   }

    for (;;) {
        sleep(1);
    }

    fclose(fd);
    return 0;
}
```

10.3 Process p2

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

FILE* fd = NULL;
FILE* fderr = NULL;
int write_pipe = 0;

void sig_handler(int signum) {
    if (signum == SIGUSR1) {
```

```
        char buff[1024];
        if (fgets(buff, 1024, fd) == NULL) {
            exit(EXIT_SUCCESS);
        }

        int size = strlen(buff);
        int written = 0;
        char* buf = buff;
        while(size > 0) {
            written = write(write_pipe, buf, size);
            if (written == -1) {
                break;
            }
            buf += written;
            size -= written;
        }
    }
}

void end_of_proc(int signum){
    exit(EXIT_SUCCESS);
}

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

    write_pipe = atoi(argv[1]);
    fd = fopen("p2.txt", "r");

    if (fd == NULL) {
        printf("Error: file is not exist!");
        return -1;
    }

    signal(SIGUSR1, sig_handler);
    signal(SIGTERM, end_of_proc);
   if(kill(getppid(), SIGUSR1) < 0){
        exit(EXIT_FAILURE);
   }

    for (;;) {
        sleep(1);
    }

    fclose(fd);
    return 0;
}
```

10.4 Process T

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

```c
#include <sys/wait.h>

void end_of_proc(int signum){
    exit(EXIT_SUCCESS);
}

int main (int argc, char* argv[]) {
    struct sembuf sem_b[1];
    int pipe, portno, semafor, a = 0;
    char *buffer;
    char string;
    char letters[128];

    kill(getppid(), SIGUSR1);
    signal(SIGTERM, end_of_proc);

    pipe = atoi(argv[1]);
    portno = atoi(argv[2]);
    semafor = atoi(argv[3]);
    buffer = shmat(portno, NULL, 0);
    if (buffer == NULL)
    {
        exit(EXIT_FAILURE);
    }

    do
    {
        sem_b[0].sem_num = 0;
        sem_b[0].sem_op = -1;
        sem_b[0].sem_flg = 0;

        if (semop(semafor, sem_b, 1))
        {
            return (0);
        }

        do {
            read(pipe, &letters[a], 1);
            a++;
        } while(letters[a - 1] != '\n');


        letters[a - 1] = '\0';

        strcpy(buffer, letters);

        a = 0;

        sem_b[0].sem_num = 1;
        sem_b[0].sem_op = 1;
        sem_b[0].sem_flg = 0;

        if (semop(semafor, sem_b, 1))
        {
            return (0);
        }
    } while (1);

    return 0;
}
```

10.5 Process D

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

//funkcia pre uspesne ukoncenie programu
void end_of_proc(int signum){
    exit(EXIT_SUCCESS);
}

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


   char *buffer;
   struct sembuf sembuf_struct[1];

   if ((buffer = shmat(atoi(argv[1]), NULL, 0)) == NULL){
      perror("ERROR pamat");
      exit(1);
   }

   kill(getppid(), SIGUSR1);
   signal(SIGTERM, end_of_proc);
   int s = socket(AF_INET, SOCK_STREAM, 0);
   struct sockaddr_in addr;
   if (s == -1) {
        perror("ERROR socket"); //vytvorenie socketu
        exit(1);
    }

   addr.sin_family = AF_INET;
   addr.sin_port = htons(atoi(argv[3]));
   addr.sin_addr.s_addr = inet_addr("127.0.0.1");
   bzero(&(addr.sin_zero), 8);

   if (connect(s, (struct sockaddr*) &addr, sizeof (addr)) == -1) {
        perror("ERROR! connection");
        exit(1);
    }
   //pomocny txt file pre kontrolu ake slova program prijal a odoslal
    struct sembuf sembuffer[1];
   for (;;) {
      sembuffer[0].sem_num = 1;
      sembuffer[0].sem_op = -1;
      sembuffer[0].sem_flg = 0;
      semop(atoi(argv[2]),sembuffer,1);
        int len = strlen(buffer);
        int byte = write(s, buffer, len + 1);
      if (byte == len + 1) {
```

```
            sleep(2);
        } else if (byte != len + 1) {
            perror("ERROR write");
              exit(1);
        }

        sembuffer[0].sem_num = 0;
        sembuffer[0].sem_op = 1;
        sembuffer[0].sem_flg = 0;
        semop(atoi(argv[2]),sembuffer,1);
    }


}
```

10.6 Process Serv2

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

#include <sys/socket.h>
#include <netinet/in.h>


void error(char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char* argv[]) {
    struct sockaddr_in server_addr;
    char buff[256];
    int portno, sockfd, outfile;

    if (argc < 2) {
        fprintf(stderr, "ERROR, no port provided\n");
        exit(1);
    }

    outfile = open("serv2.txt", O_CREAT | O_WRONLY, 0666);

    if (outfile < 0) {
        perror("Error with opening file\n");
    }

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    if (sockfd < 0)
    {
        error("ERROR opening socket");
    }

    portno = atoi(argv[1]);
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(portno);
```

```
    if (bind(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) <
0) {
        error("ERROR on binding");
    }

    kill(getppid(), SIGUSR1);

    int a;
    for (a = 0; a < 10; a++) {
        recv(sockfd, buff, 200, 0);
        write(outfile, buff, strlen(buff));
        write(outfile, "\n", 1);
    }

    kill(getppid(), SIGTERM);

    printf("Turn Off server2\n");
    return 0;
}
```