**Zadanie IPC**

## Text zadania

## SCHÉMA

*p2.txt*

*serv2.txt*

**TCP**

**UDP**

**Serv2**

**Serv1**

**D**

**ZDIEĽANÁ PAMÄŤ SM2**

SEMAFOR S2

SEMAFOR

S1

**ZDIEĽANÁ PAMÄŤ SM1**

**S**

**RÚRA**

**R2**

**T**

**Pr**

**RÚRA R1**

*p1.txt*

SIGNÁL

SIGNÁL

**P2**

**P1**

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

## 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“!).

## 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.

## Dodefinovanie zadania

K zadaniu sú dodané procesy *proc\_pr*, *proc\_s* a *proc\_serv1*, plus súbory *p1.txt* a *p2.txt*. Oba tieto súbory obsahujú po 10 slov rozdelených po 10 riadkoch. V zadaní treba vytvoriť procesy P1, P2, T, D a Serv2.

## Popis relevantných štruktúr, algoritmov, dátových typov, konštánt

1. **struct sembuf:**
   1. Táto štruktúra sa používa pri práci so semaformi. Slúži na definovanie operácií, ktoré sa majú vykonať na semafore.
   2. Príklad použitia: struct sembuf sem\_b sa používa na kontrolu semaforov prostredníctvom služby semop(). Táto štruktúra obsahuje tri členské premenné: sem\_num (identifikátor semaforu v sade), sem\_op (operácia na vykonanie - inkrementácia alebo dekrementácia) a sem\_flg (vlajky na riadenie operácie).
2. **struct sockaddr\_in:**
   1. Štruktúra používaná pri práci so sieťovými soketmi, najmä pri nastavovaní IP adresy a portu pre TCP/UDP komunikáciu.
   2. Obsahuje členské premenné ako sin\_family (rodina adries, typicky AF\_INET pre IPv4), sin\_port (portové číslo) a sin\_addr (IP adresa), pričom sin\_addr je ďalšia štruktúra obsahujúca s\_addr (32-bitová IP adresa).
3. **struct hostent:**
   1. Táto štruktúra sa používa pri prevode názvu hostiteľa na jeho IP adresu, čo je užitočné pri nastavovaní sieťových soketov.
   2. Obsahuje informácie ako meno hostiteľa, typ adresy, dĺžku adresy a zoznam adries. Pomocou tejto štruktúry môže proces získať IP adresu servera na základe jeho názvu.

## Proces Zadanie

* **Pipes (Rúry):**
  + Rúry slúžia na medziprocesovú komunikáciu. Sú implementované ako pole dvoch celočíselných hodnôt, kde pipe1[2] a pipe2[2] predstavujú dva kanály pre prenos dát medzi procesmi.
* **Semafory:**
  + Semafory sem1 a sem2 sú použité na synchronizáciu procesov a zabezpečenie konzistentného prístupu k zdieľaným zdrojom. Sú inicializované pomocou systémového volania semget a ovládané funkciou semctl.
* **Zdieľaná Pamäť:**
  + Dve zdieľané pamäťové segmenty shm1 a shm2 slúžia na ukladanie dát, ktoré sú prístupné viacerým procesom. Tieto segmenty sú vytvorené pomocou shmget.
* **Signály:**
  + Na ovládanie a koordináciu procesov sa využívajú signály, najmä SIGUSR1 a SIGUSR2. Tieto signály sú spracovávané pomocou funkcií create\_signals, check\_program\_status a kill\_processors..

1. **Proces P1**
   * **Spustenie**: proc\_p1 <id zapisovacieho konca rúry 1>
   * **Funkcie**:
     + Čítanie zo súboru p1.txt a zápis slova do rúry 1, keď dostane signál od procesu Pr.
     + Využíva signal handling a manipuláciu súborov.
2. **Proces P2**
   * **Spustenie**: proc\_p2 <id zapisovacieho konca rúry 1>
   * **Funkcie**:
     + Čítanie zo súboru p2.txt a zápis slova do rúry 1 po prijatí signálu od procesu Pr.
     + Používa metódy pre prácu so súbormi a signal handling.
3. **Proces T**
   * **Spustenie**: proc\_t <id čítacieho konca rúry 2> <id zdieľanej pamäte 1> <id semafora S1>
   * **Funkcie**:
     + Čítanie z rúry 2, zápis do zdieľanej pamäte 1 s využitím semafora pre koordináciu.
     + Využíva semafory, prácu so zdieľanou pamäťou a potrubím.
4. **Proces D**
   * **Spustenie**: proc\_d <id zdieľanej pamäte 2> <id semafora S2> <číslo portu 1 (tcp)>
   * **Funkcie**:
     + Čítanie zo zdieľanej pamäte 2, odoslanie dát cez TCP port na server.
     + Používa semafory, sokety a prácu so zdieľanou pamäťou.
5. **Proces Serv2**
   * **Spustenie**: proc\_serv2 <číslo portu 2 (udp)>
   * **Funkcie**:
     + Prijímanie dát cez UDP, zápis do súboru serv2.txt.
     + Práca s UDP soketmi, súbormi.

.

## Dátové typy

**Proces Zadanie**

* **pid procesov**
  + Typ: int
  + Popis: Identifikátory procesov ako Serv1, Serv2, D, P1, P2, Pr, T, S.
* **Reťazce na uloženie id procesov P1 a P2**
  + Typ: char[]
  + Popis: Slúžia na uloženie identifikátorov procesov P1 a P2.
* **id zdieľaných pamätí**
  + Typ: int
  + Popis: Identifikátory pre zdieľané pamäťové segmenty (shm1, shm2).
* **Reťazce na uloženie id zdieľaných pamätí**
  + Typ: char[]
  + Popis: Uchovávajú reťazcové reprezentácie identifikátorov zdieľaných pamätí.
* **id semaforov**
  + Typ: int
  + Popis: Identifikátory pre semafory (sem1, sem2).
* **Reťazce na uloženie id semaforov**
  + Typ: char[]
  + Popis: Uchovávajú reťazcové reprezentácie identifikátorov semaforov.
* **Deklarovanie rúr**
  + Typ: int[]
  + Popis: Deklaruje rúry pre medziprocesovú komunikáciu (pipe1, pipe2).
* **Reťazce na uloženie id čítacích a zapisovacích koncov rúr 1 a 2**
  + Typ: char[]
  + Popis: Uchovávajú reťazcové reprezentácie identifikátorov čítacích a zapisovacích koncov rúr.

 **Proces P1:**

* **Ukazovateľ na deskriptor súboru p1.txt:**
  + Typ: FILE\*
  + Popis: Súborový deskriptor pre čítanie z p1.txt.
* **Buffer:**
  + Typ: char[]
  + Popis: Dočasné úložisko pre dáta čítané zo súboru.

 **Proces P2:**

* **Ukazovateľ na deskriptor súboru p2.txt:**
  + Typ: FILE\*
  + Popis: Súborový deskriptor pre čítanie z p2.txt.
* **Buffer:**
  + Typ: char[]
  + Popis: Dočasné úložisko pre dáta čítané zo súboru.

 **Proces T:**

* **ID semafora 1:**
  + Typ: int
  + Popis: Identifikátor pre semafor 1.
* **Reťazec na uloženie ID zdieľanej pamäte 1:**
  + Typ: char\*
  + Popis: Uchováva identifikátor pre zdieľanú pamäť 1.
* **Rúra 2:**
  + Typ: int
  + Popis: Identifikátor pre rúru 2 pre medziprocesovú komunikáciu.
* **Ďalšie premenné (line, buffer):**
  + Typ: char[]
  + Popis: Buffer pre dočasné uchovávanie dát.
* **Počítadlo:**
  + Typ: int
  + Popis: Počítadlo pre interné účely procesu.

 **Proces D:**

* **Deklarovanie semafora 2:**
  + Typ: int
  + Popis: Identifikátor pre semafor 2.
* **Deskriptor socketu:**
  + Typ: int
  + Popis: Deskriptor socketu pre TCP komunikáciu.
* **Číslo TCP portu:**
  + Typ: int
  + Popis: Číslo portu pre TCP komunikáciu.
* **Buffer:**
  + Typ: char[]
  + Popis: Buffer pre dočasné uchovávanie a odosielanie dát.
* **ID zdieľanej pamäte 2:**
  + Typ: int
  + Popis: Identifikátor pre zdieľanú pamäť 2.
* **Reťazec na uloženie ID zdieľanej pamäte 2:**
  + Typ: char\*
  + Popis: Uchováva identifikátor pre zdieľanú pamäť 2.

 **Proces Serv2:**

* **Deskriptor súboru serv2.txt:**
  + Typ: int
  + Popis: Deskriptor súboru pre zápis dát prijatých cez UDP.
* **Deskriptor socketu:**
  + Typ: int
  + Popis: Deskriptor socketu pre UDP komunikáciu.
* **Číslo UDP portu:**
  + Typ: int
  + Popis: Číslo portu pre UDP komunikáciu.
* **Buffer:**
  + Typ: char[]
  + Popis: Buffer pre dočasné uchovávanie a odosielanie dát.
* **Počítadlo slov:**
  + Typ: int
  + Popis: Počítadlo slov pre interné účely procesu.

### Všeobecné Dátové Typy

* **int**
  + Použitie: Pre číselné hodnoty, identifikátory a porty.
* **char[]**
  + Použitie: Pre textové reťazce, buffery a dočasné úložisko dát.
* **FILE**\*
  + Použitie: Pre ukazovatele na súborové deskriptory.

## Analýza problematiky

Program Zadanie demonštruje komplexnú medziprocesovú komunikáciu a synchronizáciu v UNIXovom prostredí. Základnou výzvou je správne riadiť a synchronizovať rôzne procesy tak, aby nedochádzalo k chybám alebo konfliktom pri prístupe k zdieľaným zdrojom. Použitie semaforov a zdieľanej pamäte je efektívnym spôsobom, ako dosiahnuť potrebnú synchronizáciu, avšak vyžaduje si pozorné ošetrenie všetkých možných stavov. Správa signálov je tiež kľúčová pre koordináciu procesov a správnu reakciu na rôzne udalosti.

* **Proces P1 a P2:** Tieto procesy sú zodpovedné za čítanie dát zo súborov p1.txt a p2.txt. Pri prijatí signálu od procesu Pr tieto procesy načítajú slovo a posielajú ho do rúry 1.
* **Proces T:** Proces T číta dáta z rúry 2 a zapisuje ich do zdieľanej pamäte 1. Na zápis do zdieľanej pamäte používa semafor S1, čím zabezpečuje, že zápis bude synchronizovaný a nebude dochádzať ku konfliktom.
* **Proces D:** Tento proces číta dáta zo zdieľanej pamäte 2, pričom na zápis a čítanie používa semafor S2. Následne tieto dáta odosiela cez TCP port na server.
* **Proces Serv2:** Tento proces vytvára UDP server, ktorý prijíma dáta od TCP servera a zapisuje ich do súboru serv2.txt.
* **Dátové typy a štruktúry:** Kód využíva rôzne dátové typy a štruktúry na spracovanie a ukladanie dát. Tieto zahŕňajú štruktúry ako sembuf, sockaddr\_in, hostent a rôzne základné dátové typy ako int, char\* atď.
* **Komunikácia a synchronizácia:** Kľúčovým aspektom kódu je efektívna komunikácia a synchronizácia medzi procesmi. Semaforové operácie a signály zabezpečujú, že procesy nebudú pristupovať k zdrojom súbežne, čo by mohlo spôsobiť chyby alebo konflikty.

## Popis navrhovaného riešenia

## Návrh riešenia

Navrhované riešenie sa zameriava na medziprocesovú komunikáciu a synchronizáciu v systéme UNIX/Linux pomocou rúr (pipes), semaforov, zdieľanej pamäte a soketov. Každý proces vykonáva špecifickú úlohu a je navrhnutý tak, aby efektívne spolupracoval s ostatnými procesmi v systéme.

## Dátové štruktúry

1. **Rúry (Pipes):** Sú využité na jednosmernú komunikáciu medzi procesmi. Rúra umožňuje procesu zapisovať dáta na jeden koniec a čítať ich z druhého konca.
2. **Semafory:** Použité na synchronizáciu prístupu k zdieľaným zdrojom medzi procesmi. Semafory v tomto projekte obmedzujú prístup k zdieľanej pamäti a zabezpečujú synchronizované čítanie a zápis dát.
3. **Zdieľaná pamäť:** Segment pamäte, ktorý je prístupný viacerým procesom. Procesy môžu do tejto pamäte zapisovať a z nej čítať, čo umožňuje efektívnu výmenu dát.

## **Spôsob synchronizácie**

Synchronizácia je dosiahnutá pomocou semaforov. Semafor S1 umožňuje synchronizáciu medzi procesmi T a S pri práci so zdieľanou pamäťou SM1. Semafor S2 zase reguluje prístup procesov S a D k zdieľanej pamäti SM2. Tieto semafory zabezpečujú, že procesy nebudú súčasne čítať a zapisovať do rovnakej oblasti pamäte.

## **Algoritmy**

## **Zadanie**

Hlavný algoritmus programu koordinuje vytvorenie a správu týchto procesov, zabezpečuje ich správnu inicializáciu a synchronizuje ich činnosť pomocou semaforov a signálov.

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <signal.h>

#include <errno.h>

#include <iostream>

#pragma region Configuration

*// descriptors for different processors*

int Serv1, Serv2, D, P1, P2, Pr, T, S;

int sem1, sem2, shm1, shm2;

*// local methods*

void create\_signals();

void initialize\_semaphores(int *sem1*, int *sem2*);

void kill\_processors(int *s*);

void check\_program\_status(int *s*);

#pragma endregion

#pragma region Main\_Method

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

    if(argc < 3) {

        printf("[Zadanie] Error in argument numbers. Please check your arguments!\n");

        exit(EXIT\_FAILURE);

    }

    int error; *// to check if "execl == -1"*

#pragma region Signals

*// create signals for re-read the first and the second configutration files*

    create\_signals();

#pragma endregion

#pragma region PIPES

*// ============== PIPES ==============*

    int pipe1[2];

    int pipe2[2];

    if (pipe(pipe1) < 0) {

        printf("[Zadanie] Error in creating pipe R1\n");

        perror("pipe");

    }

    if (pipe(pipe2) < 0) {

        printf("[Zadanie] Error in creating pipe R2\n");

        perror("pipe");

    }

    #pragma region INIT\_STR\_RESOURCES\_PIPE

        int pipe1Read = dup(pipe1[0]);

        int pipe1Write = dup(pipe1[1]);

        int pipe2Read = dup(pipe2[0]);

        int pipe2Write = dup(pipe2[1]);

        char pipe1StringR[10];

        char pipe1StringW[10];

        char pipe2StringR[10];

        char pipe2StringW[10];

*// Initialising string representations of resources*

        memset(pipe1StringR,'\0', sizeof(pipe1Read));

        memset(pipe1StringW,'\0', sizeof(pipe1Write));

        memset(pipe2StringR,'\0', sizeof(pipe2Read));

        memset(pipe2StringW,'\0', sizeof(pipe2Write));

        sprintf(pipe1StringR, "%d", pipe1Read);

        sprintf(pipe1StringW, "%d", pipe1Write);

        sprintf(pipe2StringR, "%d", pipe2Read);

        sprintf(pipe2StringW, "%d", pipe2Write);

    #pragma endregion

#pragma endregion

#pragma region SEMAPHORES

*// ============== SEMAPHORES ==============*

    printf("\n");

    sem1 = semget(IPC\_PRIVATE, 2, 0666 | IPC\_CREAT);

    if(sem1 < 0) {

        printf("[Zadanie] Error in creating SEM1\n");

        perror("semget"); *// check why we have that error*

        exit(EXIT\_FAILURE);

*//get\_exit(false, "Error in semget\n");*

    }

    sem2 = semget(IPC\_PRIVATE, 2, 0666 | IPC\_CREAT);

    if(sem2 < 0) {

        printf("[Zadanie] Error in creating SEM2\n");

        perror("semget"); *// check why we have that error*

        exit(EXIT\_FAILURE);

*//get\_exit(false, "Error in semget\n");*

    }

    initialize\_semaphores(sem1, sem2);

    #pragma region INIT\_STR\_RESOURCES\_SEMAPHORES

        char semaphor1String[10];

        char semaphor2String[10];

*// Initialising string representations of resources*

        memset(semaphor1String,'\0', sizeof(sem1));

        memset(semaphor2String,'\0', sizeof(sem2));

        sprintf(semaphor1String,"%d", sem1);

        sprintf(semaphor2String,"%d", sem2);

    #pragma endregion

#pragma endregion

#pragma region SHARED\_MEMORY

*// ============== SHARED MEMORY ==============*

    shm1 = shmget(IPC\_PRIVATE, 256, 0666 | IPC\_CREAT);

    if (shm1 < 0) {

        printf("[Zadanie] Error in creating SHARED MEMORY 1\n");

        perror("shmget"); *// writes information about our error*

        fprintf(stderr, "shmget failed\n");

        exit(EXIT\_FAILURE);

    }

    shm2 = shmget(IPC\_PRIVATE, 256, 0666 | IPC\_CREAT);

    if (shm2 < 0) {

        printf("[Zadanie] Error in creating SHARED MEMORY 2\n");

        perror("shmget"); *// writes information about our error*

        fprintf(stderr, "shmget failed\n");

        exit(EXIT\_FAILURE);

    }

    #pragma region INIT\_STR\_RESOURCES\_SHARED\_MEMORY

        char shm1String[10];

        char shm2String[10];

*// Initialising string representations of resources*

        memset(shm1String,'\0', sizeof(shm1));

        memset(shm2String,'\0', sizeof(shm2));

        sprintf(shm1String,"%d", shm1);

        sprintf(shm2String,"%d", shm2);

    #pragma endregion

#pragma endregion

#pragma region P1

*// ========== P1 ==========*

    P1 = fork();

    switch (P1)

    {

        case 0:

        {

            printf("Executing proc\_p1...\n");

            error = execl("proc\_p1","proc\_p1", pipe1StringW, (char\*)NULL);

            if (error == -1) {

                printf("[Zadanie] Failed to execute proc\_p1. Exiting...\n");

                perror("execl"); *// writes the information about this error*

                exit(EXIT\_FAILURE);

            }

            printf("proc\_p1 execution completed.\n");

            exit(EXIT\_SUCCESS);

            break;

        }

        case -1:

            printf("[Zadanie] Failed to create process P1. Exiting...\n");

            perror("P1");

            exit(EXIT\_FAILURE);

            break;

        default:

            break;

    }

    pause();

    #pragma region INIT\_STR\_RESOURCES\_P1

        char P1\_d[10];

        memset(P1\_d, '\0', sizeof(P1));

        sprintf(P1\_d, "%d", P1);

    #pragma endregion

#pragma endregion

#pragma region P2

*// ========== P2 ==========*

    P2 = fork();

    switch (P2)

    {

        case 0:

        {

            printf("Executing proc\_p2...\n");

            error = execl("proc\_p2","proc\_p2",pipe1StringW,(char\*)NULL);

            if (error == -1) {

                printf("[Zadanie] Failed to execute proc\_p2. Exiting...\n");

                perror("execl"); *// writes the information about this error*

                exit(EXIT\_FAILURE);

            }

            exit(EXIT\_SUCCESS);

            break;

        }

        case -1:

            printf("[Zadanie] Failed to create process P2. Exiting...\n");

            perror("P2");

            exit(EXIT\_FAILURE);

            break;

        default:

            break;

    }

    pause();

    #pragma region INIT\_STR\_RESOURCES\_P2

        char P2\_d[10];

        memset(P2\_d, '\0', sizeof(P2));

        sprintf(P2\_d, "%d", P2);

    #pragma endregion

#pragma endregion

#pragma region Pr

*// ============ Pr ============*

    Pr = fork();

    switch (Pr)

    {

        case 0:

        {

            printf("Executing proc\_pr...\n");

            error = execl("proc\_pr", "proc\_pr",P1\_d,P2\_d,pipe1StringR,pipe2StringW,(char\*)NULL);

            if (error == -1) {

                printf("[Zadanie] Failed to execute proc\_pr. Exiting...\n");

                perror("execl"); *// writes the information about this error*

                exit(EXIT\_FAILURE);

            }

            printf("proc\_pr execution completed.\n");

            exit(EXIT\_SUCCESS);

        }

        case -1:

            printf("[Zadanie] Failed to create process Pr. Exiting...\n");

            perror("Pr");

            exit(EXIT\_FAILURE);

            break;

        default:

            break;

    }

    pause();

    int statusPr;

    waitpid(Pr, &statusPr, 0);

#pragma endregion

#pragma region SERVERS

    char string\_server1[10];

    char string\_server2[10];

    int serv1\_port = atoi(argv[1]);

    int serv2\_port = atoi(argv[2]);

*// Initialising string representations of resources*

    memset(string\_server1,'\0', sizeof(serv1\_port));

    memset(string\_server2,'\0', sizeof(serv2\_port));

    sprintf(string\_server1,"%d", serv1\_port);

    sprintf(string\_server2,"%d", serv2\_port);

#pragma endregion

#pragma region Serv1

*// ============ Serv1 ============*

    Serv1 = fork();

    switch (Serv1)

    {

        case 0:

        {

            printf("Executing proc\_serv1...\n");

            error = execl("proc\_serv1", "proc\_serv1", string\_server1, string\_server2, (char \* ) NULL);

            if (error == -1) {

*//fprintf (stderr, "-: Couldn't open file -; %s\n", strerror (errno));*

                printf("[Zadanie] Failed to execute proc\_serv1. Exiting...\n");

                perror("execl"); *// writes the information about this error*

                exit(EXIT\_FAILURE);

            }

            printf("proc\_serv1 execution completed.\n");

            exit(EXIT\_SUCCESS);

            break;

        }

        case -1:

            printf("[Zadanie] Failed to create process Serv1. Exiting...\n");

            break;

        default:

            break;

    }

    pause();

#pragma endregion

#pragma region Serv2

    // ============== Serv2 ==============

    Serv2 = fork();

    switch (Serv2)

    {

        case 0:

        {

            printf("Executing proc\_serv2...\n");

            error = execl("proc\_serv2", "proc\_serv2", string\_server2, (char \* ) NULL);

            if (error == -1) {

                printf("[Zadanie] Failed to execute proc\_serv2. Exiting...\n");

                perror("execl"); // writes the information about this error

                exit(EXIT\_FAILURE);

            }

            printf("proc\_serv2 execution completed.\n");

            exit(EXIT\_SUCCESS);

            break;

        }

        case -1:

            printf("[Zadanie] Failed to create process Serv2. Exiting...\n");

            perror("Serv2");

            break;

        default:

            break;

    }

    pause();

#pragma endregion

#pragma region proc\_T

    // ============ proc\_T ============

    T = fork();

    switch (T)

    {

        case 0:

        {

            printf("Executing proc\_t...\n");

            error = execl("proc\_t", "proc\_t",pipe2StringR, shm1String, semaphor1String, (char\*)NULL);

            if (error == -1) {

                printf("[Zadanie] Failed to execute proc\_t. Exiting...\n");

                perror("execl"); // writes the information about this error

                exit(EXIT\_FAILURE);

            }

            printf("proc\_t execution completed.\n");

            exit(EXIT\_SUCCESS);

        }

        case -1:

            printf("[Zadanie] Failed to create process T. Exiting...\n");

            perror("T");

            exit(EXIT\_FAILURE);

            break;

        default:

            break;

    }

    pause();

#pragma endregion

#pragma region D

    // ============ proc\_D ============

    D = fork();

    switch (D)

    {

        case 0:

        {

            printf("Executing proc\_d...\n");

            error = execl("proc\_d", "proc\_d", shm2String, semaphor2String, string\_server1, (char\*)NULL);

            if (error == -1) {

                printf("[Zadanie] Failed to execute proc\_d. Exiting...\n");

                perror("execl"); // writes the information about this error

                exit(EXIT\_FAILURE);

            }

            printf("proc\_d execution completed.\n");

            exit(EXIT\_SUCCESS);

            break;

        }

        case -1:

            printf("[Zadanie] Failed to create process D. Exiting...\n");

            perror("D");

            break;

        default:

            break;

    }

    pause();

#pragma endregion

#pragma region proc\_S

    // =========== proc\_S ===========

    S = fork();

    switch (S)

    {

        case 0:

        {

            printf("Executing proc\_s...\n");

            error = execl("proc\_s", "proc\_s",shm1String, semaphor1String, shm2String, semaphor2String, (char\*)NULL);

            if(error == -1){

                perror("execl");

                printf("[Zadanie] Failed to execute proc\_s. Exiting...\n");

                exit(EXIT\_FAILURE);

            }

            printf("proc\_s execution completed.\n");

            exit(EXIT\_SUCCESS);

        }

        case -1:

            printf("[Zadanie] Failed to create process T. Exiting...\n");

            perror("S");

            exit(EXIT\_FAILURE);

            break;

        default:

            break;

    }

    pause();

#pragma endregion

    pause();

    printf("The end of the program :)\n");

    return EXIT\_SUCCESS;

}

#pragma endregion

#pragma region Local\_Methods

void create\_signals() {

    // create a signal for re-read the first configuration file

    static\_cast<void>(signal(SIGUSR1, check\_program\_status));

    // create a signal for re-read the second configuration file

    static\_cast<void>(signal(SIGUSR2, kill\_processors));

}

void initialize\_semaphores(int sem1, int sem2) {

    // Set initial values of semaphores

    semctl(sem1, 0, SETVAL, 0); // 0, 0

    semctl(sem2, 0, SETVAL, 0); // 0, 0

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

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

}

void check\_program\_status(int s) {

    printf("Program still running...\n");

}

void kill\_processors(int s) {

    printf("Terminating all processes...\n");

    sleep(10);

    int buffer[8] = { P1, P2, Pr, T, S, D, Serv1, Serv2 };

    for (int i = 0; i < 8; i++)

    {

        if (buffer[i] != -1)

        {

            kill(buffer[i], SIGKILL);

        }

    }

    semctl(sem1, 0, IPC\_RMID, 0);

    semctl(sem2, 0, IPC\_RMID, 0);

    shmctl(shm1, 0, IPC\_RMID);

    shmctl(shm2, 0, IPC\_RMID);

}

#pragma endregion

## **Proces P1**

Po prijatí signálu SIGUSR1 začnú čítať dáta zo súborov p1.txt a p2.txt a zapisujú ich do rúry.

#include <iostream>

#include <string>

#include <fcntl.h>

#include <unistd.h>

#include <csignal>

#include <cstring>

int fd, P1;

void handler(int *signum*);

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

    signal(SIGUSR1, handler);

    fd = open("p1.txt", O\_RDONLY);

    if (argc > 1) {

        P1 = std::atoi(argv[1]);

    } else {

        std::cerr << "Error: Missing argument" << std::endl;

        return 1;

    }

    std::cout << "P1.cpp SA HLASI KU SLUZBE!" << std::endl;

    kill(getppid(), SIGUSR1);

    while (true) {

        sleep(5);

    }

    return 0;

}

void handler(int *signum*) {

    char buffer;

    std::string input;

    input.reserve(151); *// Reserve memory to avoid frequent allocations*

    while (true) {

        read(fd, &buffer, 1); *// Read from the file*

        if (buffer != '\n') {

            input += buffer;

        } else {

            input += buffer;

            break;

        }

    }

    write(P1, input.c\_str(), input.length()); *// Write to the pipe*

}

## **Proces P2– analogicky k P1, mení sa iba otvorený textový súbor – p2.txt**

#include <iostream>

#include <string>

#include <fcntl.h>

#include <unistd.h>

#include <csignal>

#include <cstring>

int fd, P2;

void handler(int *signum*);

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

    signal(SIGUSR1, handler);

    fd = open("p2.txt", O\_RDONLY);

    if (argc > 1) {

        P2 = std::atoi(argv[1]);

    } else {

        std::cerr << "Error: Missing argument" << std::endl;

        return 1;

    }

    std::cout << "P2.cpp SA HLASI KU SLUZBE!" << std::endl;

    kill(getppid(), SIGUSR1);

    while (true) {

        sleep(5);

    }

    return 0;

}

void handler(int *signum*) {

    char buffer;

    std::string input;

    input.reserve(151); *// Reserve memory to avoid frequent allocations*

    while (true) {

        read(fd, &buffer, 1); *// Read from the file*

        if (buffer != '\n') {

            input += buffer;

        } else {

            input += buffer;

            break;

        }

    }

    write(P2, input.c\_str(), input.length()); *// Write to the pipe*

}

## **Proces T**

Číta dáta z rúry a zapisuje ich do zdieľanej pamäte SM1. Používa semafor S1 na synchronizáciu prístupu k pamäti.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <signal.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <fcntl.h>

void pipeToMemory(int *read\_pipe*, int *memory\_pid*, int *semaphore\_pid*);

void semaphoreWait(int *semaphore\_pid*);

void semaphoreSignal(int *semaphore\_pid*);

void handleExit(const char\* *msg*, int *exit\_status*);

void signalHandler(int *signum*);

*// Funkcia pre čítanie z potrubia a zápis do zdieľanej pamäte*

void pipeToMemory(int *read\_pipe*, int *memory\_pid*, int *semaphore\_pid*) {

  const int BUFFER\_SIZE = 256;

  char buffer[BUFFER\_SIZE];

  char \*shared\_memory = (char\*) shmat(memory\_pid, NULL, 0); *// Pripojenie zdieľanej pamäte*

  if (shared\_memory == (void\*) -1) handleExit("proc\_t: shmat failed", 1);

  while (1) {

    semaphoreWait(semaphore\_pid); *// Čakanie na povolenie zápisu*

*// Čítanie z potrubia*

    int i;

    for (i = 0; i < BUFFER\_SIZE -1; i++) {

        if (read(read\_pipe, &buffer[i], 1) < 1 || buffer[i] == '\n') {

          break;

        }

    }

    buffer[i] = '\0'; *// Nastavenie konca reťazca*

*// Zápis do zdieľanej pamäte*

    strcpy(shared\_memory, buffer);

*// Signalizácia konca zápisu*

    semaphoreSignal(semaphore\_pid);

  }

}

*// Funkcia pre riadenie ukončenia programu*

void handleExit(const char\* *msg*, int *exit\_status*) {

  if (exit\_status == EXIT\_SUCCESS) {

      printf("%s\n", msg); *// Výpis správy pri úspešnom ukončení*

      exit(EXIT\_SUCCESS);

  } else {

      fprintf(stderr,"%s\n",msg); *// Výpis chybovej správy*

      exit(EXIT\_FAILURE);

  }

}

*// Spracovanie signálu SIGUSR2*

void signalHandler(int *signum*) {

  if (signum == SIGUSR2) {

      handleExit("proc\_t: SUCCESS MESSAGE", EXIT\_SUCCESS); *// Spracovanie signálu SIGUSR2*

  }

}

*// Čakanie na semafor*

void semaphoreWait(int *semaphore\_pid*) {

*// Inicializácia štruktúry pre semafor*

  struct sembuf semaphore\_buffer;

  semaphore\_buffer.sem\_num = 1;

  semaphore\_buffer.sem\_op = 1;

  semaphore\_buffer.sem\_flg = SEM\_UNDO;

*// Čakanie na semafor*

  if (semop(semaphore\_pid, &semaphore\_buffer, 1) == -1) {

      handleExit("proc\_t: semaphoreWait failed", 1); *// Spracovanie chyby pri čakaní na semafor*

  }

}

*// Signalizácia semaforu*

void semaphoreSignal(int *semaphore\_pid*) {

*// Inicializácia štruktúry pre semafor*

  struct sembuf semaphore\_buffer;

  semaphore\_buffer.sem\_num = 0;

  semaphore\_buffer.sem\_op = -1;

  semaphore\_buffer.sem\_flg = SEM\_UNDO;

*// Signalizovanie semaforu*

  if (semop(semaphore\_pid, &semaphore\_buffer, 1) == -1) {

      handleExit("proc\_t: semaphoreSignal failed", 1); *// Spracovanie chyby pri signalizácii semaforu*

  }

}

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

  printf("proc\_t spusteny\n");

  int pipe\_read\_fd = atoi(argv[1]); *// Fd pre čítanie z potrubia*

  int memory\_segment\_pid = atoi(argv[2]); *// ID zdieľanej pamäte*

  int semaphore\_pid = atoi(argv[3]); *// ID semaforu*

  kill(getppid(), SIGUSR1); *// Poslanie signálu rodičovskému procesu*

  signal(SIGUSR2, signalHandler); *// Nastavenie správcu signálu SIGUSR2*

  pipeToMemory(pipe\_read\_fd, memory\_segment\_pid, semaphore\_pid); *// Spustenie funkcie na čítanie z potrubia a zápis do zdieľanej pamäte*

  return 0;

}

**Funkcie operácií semaforov**

## **Proces D**

Číta dáta zo zdieľanej pamäte SM2 a odosiela ich cez TCP soket na server. Používa semafor S2 na synchronizáciu.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <signal.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <fcntl.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

void handleExit(const char\* *msg*, int *exit\_status*);

void semaphoreWait(int *semaphore\_pid*);

void semaphoreSignal(int *semaphore\_pid*);

void sendData(int *memory\_pid*, int *semaphore\_pid*, int *port*);

int setupAndConnect(int\* *socket\_descriptor*, struct sockaddr\_in\* *server\_addr*, int *port*);

*// Funkcia na spracovanie ukončenia programu*

void handleExit(const char\* *msg*, int *exit\_status*) {

    if (exit\_status == EXIT\_SUCCESS) {

        printf("%s\n", msg); *// Vypíše správu o úspešnom ukončení*

        exit(EXIT\_SUCCESS);

    } else {

        fprintf(stderr,"%s\n",msg); *// Vypíše chybovú správu*

        exit(EXIT\_FAILURE);

    }

}

*// Operácia čakania pre semafor*

void semaphoreWait(int *semaphore\_pid*) {

*// Inicializácia štruktúry pre semafor*

    struct sembuf semaphore\_buffer;

    semaphore\_buffer.sem\_num = 0;

    semaphore\_buffer.sem\_op = 1;

    semaphore\_buffer.sem\_flg = SEM\_UNDO;

*// Čakanie na semafor*

    if (semop(semaphore\_pid, &semaphore\_buffer, 1) == -1) {

        handleExit("proc\_d: semaphoreWait failed", EXIT\_FAILURE);

    }

}

*// Signalizácia semaforu*

void semaphoreSignal(int *semaphore\_pid*) {

*// Inicializácia štruktúry pre semafor*

    struct sembuf semaphore\_buffer;

    semaphore\_buffer.sem\_num = 1;

    semaphore\_buffer.sem\_op = -1;

    semaphore\_buffer.sem\_flg = SEM\_UNDO;

*// Signalizovanie semaforu*

    if (semop(semaphore\_pid, &semaphore\_buffer, 1) == -1) {

        handleExit("proc\_d: semaphoreSignal failed", EXIT\_FAILURE);

    }

}

*// Nastavenie soketu a pripojenie sa k serveru*

int setupAndConnect(int\* *socket\_descriptor*, struct sockaddr\_in\* *server\_addr*, int *port*) {

*// Vytvorenie socketu*

    \*socket\_descriptor = socket(AF\_INET, SOCK\_STREAM, 0);

    if (\*socket\_descriptor < 0) {

        handleExit("proc\_d: socket error", EXIT\_FAILURE);

    }

    server\_addr->sin\_family = AF\_INET; *// Nastavenie typu adresy (IPv4)*

    server\_addr->sin\_port = htons(port); *// Nastavenie portu servera*

    server\_addr->sin\_addr.s\_addr = inet\_addr("127.0.0.1"); *// Nastavenie IP adresy servera*

    bzero(&(server\_addr->sin\_zero), 8); *// Vynulovanie dodatočných núl v štruktúre adresy*

*// Pripojenie k serveru*

    if (connect(\*socket\_descriptor, (struct sockaddr \*) server\_addr, sizeof(\*server\_addr)) < 0) {

        handleExit("proc\_d: connect error", EXIT\_FAILURE);

    }

    return \*socket\_descriptor;

}

*// Odoslanie údajov na server*

void sendData(int *memory\_pid*, int *semaphore\_pid*, int *port*) {

    char buffer[256];

*// Priradenie zdieľanej pamäte*

    char \*shared\_memory = (char\*) shmat(memory\_pid, NULL, 0);

    if (shared\_memory == (void\*) -1) {

        handleExit("proc\_d: shmat failed", EXIT\_FAILURE);

    }

    int socket\_descriptor;

    struct sockaddr\_in server\_addr;

    setupAndConnect(&socket\_descriptor, &server\_addr, port); *// Nastavenie soketu a pripojenie sa k serveru*

    while (1) {

        semaphoreWait(semaphore\_pid); *// Čakanie na povolenie zapisovania do zdieľanej pamäte*

        strcpy(buffer, shared\_memory); *// Kopírovanie dát zo zdieľanej pamäte do buffera*

        write(socket\_descriptor, buffer, strlen(buffer) + 1); *// Zápis dát na server*

        memset(buffer, '\0', sizeof(buffer)); *// Vyčistenie buffera*

        semaphoreSignal(semaphore\_pid); *// Signalizovanie konca zápisu*

        sleep(1);

    }

}

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

    printf("proc\_d spustený\n"); *// Výpis o spustení programu*

    int memory\_segment\_pid = atoi(argv[1]); *// Získanie ID zdieľanej pamäte*

    int semaphore\_pid = atoi(argv[2]); *// Získanie ID semaforu*

    int port = atoi(argv[3]); *// Získanie čísla portu*

    kill(getppid(), SIGUSR1); *// Poslanie signálu rodičovskému procesu*

    sendData(memory\_segment\_pid, semaphore\_pid, port); *// Spustenie funkcie na odosielanie dát*

    return 0;

}

**Funkcie operácií semaforov**

**Proces Serv2**

Vytvára UDP server, prijíma dáta a ukladá ich do súboru serv2.txt.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <fcntl.h>
5. #include <signal.h>
6. #include <stdbool.h>
7. #include <netinet/in.h>
8. #include <sys/socket.h>
9. #include <unistd.h>
10. const int BUF\_S = 200; *// buffer's size for reading and writing file contents*
11. int get\_exit(bool *exit\_result*,const char \**error\_message*); *// method that returns the correct output*
12. int send\_signal\_to\_process(int *port\_number*, int *socket*); *// method for sending a process signal*
13. int main(int *argc*, char\* *argv*[]) {
14. printf("Server 2 is running...\n");
15. if(argc < 2) {
16. get\_exit(false, "Arguments number error\n");
17. *//printf("Arguments number error\n");*
18. *//exit(EXIT\_FAILURE);*
19. }
20. int port = atoi(argv[1]);
21. if (port == 0) {
22. get\_exit(false, "Port number error\n");
23. }
24. int socket\_d = socket(AF\_INET, SOCK\_DGRAM, 0);
25. if(socket\_d == -1) {
26. get\_exit(false, "Server 2 socket() error\n");
27. }
28. send\_signal\_to\_process(port, socket\_d);
29. printf("Openning file...\n");
30. int file = open("serv2.txt", O\_WRONLY | O\_TRUNC | O\_CREAT, 0777);
31. printf("Copying data...\n");
32. char buf[BUF\_S]; *// buffer for reading and writing file contents*
33. for (int i = 0; i < 10; i++) {
34. *// try to read and write data*
35. if(recv(socket\_d, buf, BUF\_S, 0) < 0 || write(file, buf, strlen(buf)) < 0){
36. get\_exit(false, "Copying data error\n");
37. }
38. write(file, "\n", 1);
39. }
40. *// end preparation*
41. printf("Sending signal to a process...\n");
42. kill(getppid(), SIGUSR2);
43. get\_exit(true, "");
44. return 0;
45. }
46. */\* methods realisation \*/*
47. int get\_exit(bool *exit\_result*, const char \**error\_message*) {
48. if (exit\_result == true){
49. exit(EXIT\_SUCCESS);
50. } else {
51. *// error message*
52. fprintf(stderr,"[./proc\_serv2] %s", error\_message);
53. exit(EXIT\_FAILURE);
54. }
55. }
56. int send\_signal\_to\_process(int *port\_number*, int *socket*)
57. {
58. *// fill in the server with relevant information*
59. struct sockaddr\_in client;
60. client.sin\_family = AF\_INET; *// set IPv4 protocol*
61. client.sin\_addr.s\_addr = INADDR\_ANY; *// binding to any available network interface*
62. client.sin\_port = htons(port\_number); *// set the port number and convert it to network byte order*
63. bzero(&(client.sin\_zero), 8);
64. if(bind(socket,(struct sockaddr \*)&client, sizeof(struct sockaddr)) < 0) {
65. get\_exit(false, "Bind work error\n");
66. }
67. printf("Sending signal to a process...\n");
68. kill(getppid(), SIGUSR1);
70. return 0;
71. }

## ****Záver a zhodnotenie****

Celková implementácia je funkčná a demonštruje základné koncepty medziprocesovej komunikácie v UNIX/Linux systémoch. Kód je navrhnutý s ohľadom na efektívnu komunikáciu a synchronizáciu medzi procesmi, avšak vyžaduje si ďalšie testovanie a možné optimalizácie na zlepšenie robustnosti a efektívnosti.

## Použitá literatúra a informačné zdroje

Sofia, Stack Overflow, Chat GPT.

## PRÍLOHY

## Systémová príručka

shmget() - vytvorí zdieľanú pamäť a vráti jej identifikátor

semget() vytvorí sadu semaforov a vráti jej identifikátor

semctl() – inicializuje hodnoty semaforov

fork() – vytvorí (takmer) identický proces

execl() – novému procesu priradí nový program so zoznamom argumentov

pause() – pozastaví proces až pokým nezachytí signál

pipe() – vytvorí rúru

read() – vráti počet načítaných bajtov, alebo 0 na konci súboru, alebo -1 pri chybe

write() – vráti počet zapísaných bajtov, alebo -1 pri chybe

signal() – čaká na daný signál

kill() – pošle signál danému procesu

sleep() – pozastaví proces na určitý čas

socket() – vytvorí socket pre komunikáciu

getppid() – vráti id rodičovského procesu

## Príručka používateľa

Účel programu: Nadobudnúť znalosť o medziprocesorovej komunikácii a synchronizácii.

Spustenie: ./zadanie <číslo tcp portu> <číslo udp portu>. Program je spustiteľný na operačnom systéme Unix. Pred spustením treba kompilovať program príkazom make all. Spustenie tiež potrebuje súbory p1.txt a p2.txt. Slová načítané zo súborov prechádzajú procesmi, ktoré k nim postupne pridávajú svoje značky a nakoniec sú zapísané do súboru serv2.txt.