Федеральное государственное бюджетное образовательное учреждение высшего образования

Национально исследовательский университет «Московский энергетический институт»

Кафедра ВМСС

Лабораторная работа №1

по курсу

«Программное обеспечение высокопроизводительных вычислительных систем»

Выполнили: студенты группы А-07м-23

Балашов С.А., Кретов Н.В., Михайлов С.А.

Проверил: Орлов Д.А., Хиль С.

Москва, 2024 г.

Вариант 2

**Задание**

Разработайте два процесса, из которых ПРОЦЕСС-ПРЕДОК порождает ПРОЦЕСС-ПОТОМОК, и через механизм взаимодействия процессов происходит передача содержимого некоторого файла поблочно от ПРЕДКА к ПОТОМКУ. Если предок не может войти в файл, он должен передать сообщение на вывод STDERR. Потомок получает блоки файла и их выводит на экран, используя STDOUT. По окончании файла предок передает потомку сигнал завершения и после окончания работы потомка (о чем получает сигнал) прекращает свою работу.

1. **Использовать сегмент общей памяти и семафоры.**

// Подключение библиотек

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/wait.h>

#include <signal.h>

#include <string.h>

#include<sys/shm.h>

#include<sys/sem.h>

// Размер блока передаваемых данных

#define BLOCK\_SIZE 32

// Процедура потомка:

// принимает структуру семафора, идентификаторы семафора и памяти,

// указатель на память и размер данных

void child(struct sembuf sb, int sem\_id, int shm\_id, char \*mem, ssize\_t nbytes)

{

// Приветствие

printf("I AM CHILD. MY PID IS %d\n", getpid());

// Цикл до конца данных в памяти

while (1)

{

// Семафор +2: позволяет потомку начать работу только после работы предка (внутри циклов)

sb.sem\_num = 0;

sb.sem\_op = 2;

sb.sem\_flg = 0;

semop(sem\_id, &sb, 1);

// Семафор в режиме ожидания 0 (пока родитель не сделает свою работу в цикле)

sb.sem\_num = 0;

sb.sem\_op = 0;

sb.sem\_flg = 0;

semop(sem\_id, &sb, 1);

// Берем разделяемую память и смотрим в конец блока

mem = shmat(shm\_id, NULL, 0);

nbytes = mem[BLOCK\_SIZE\*sizeof(char)];

// Если в конце блока размер считанных данных не 0, то выводим данные

if (nbytes > 0)

{

write(1, mem, nbytes);

}

// Иначе отсоединяем память и завершаем цикл

else

{

shmdt(mem);

break;

}

}

}

// Процедура родителя:

// принимает структуру семафора, идентификаторы семафора и памяти,

// указатель на память, размер данных и номер процесса

void parent(struct sembuf sb, int sem\_id, int shm\_id, char \*mem, ssize\_t nbytes, pid\_t pid)

{

// Приветствие

printf("I AM PARENT. MY PID IS %d\n", getpid());

// Файловый дескриптор

int file;

// Открытие файла с проверкой на ошибку

file = open("../file.txt", O\_RDONLY);

if (file == -1) {

perror("open");

kill(pid, SIGTERM);

wait(NULL);

exit(1);

}

// Цикл до конца данных в памяти

while (1)

{

// Семафор -1: позволяет потомку начать работу только после работы предка (внутри циклов)

sb.sem\_num = 0;

sb.sem\_op = -1;

sb.sem\_flg = 0;

semop(sem\_id, &sb, 1);

// Берем разделяемую память и кладём в неё данные из файла

mem = shmat(shm\_id, NULL, 0);

nbytes = read(file, mem, BLOCK\_SIZE);

mem[BLOCK\_SIZE\*sizeof(char)] = nbytes;

// Семафор -1: позволяет потомку начать работу только после работы предка (внутри циклов)

sb.sem\_num = 0;

sb.sem\_op = -1;

sb.sem\_flg = 0;

semop(sem\_id, &sb, 1);

// Если данные в файле закончились, то закрываем файл и завершаем, очищаем память,

// завершаем процесс-потомок и завершаем процесс-родитель

if (nbytes == 0)

{

close(file);

shmdt(mem);

if (shmctl(shm\_id, IPC\_RMID, NULL) == -1) {

perror("shmctl");

kill(pid, SIGTERM);

wait(NULL);

exit(1);

}

printf("\nWaiting for child\n");

kill(pid, SIGUSR1);

int child\_status;

waitpid(pid, &child\_status, 0);

if (WIFEXITED(child\_status)) {

int exit\_code = WEXITSTATUS(child\_status);

printf("Child process exited with code %d\n", exit\_code);

} else {

printf("Child process terminated abnormally\n");

}

printf("Parent process finished\n");

exit(0);

}

}

}

// Главная функция

int main ()

{

// Переменные

ssize\_t nbytes = 0;

pid\_t pid;

struct sembuf sb = {0, 1, 0};

char \*mem = NULL;

// Разделяемая память, семафор и их идентификаторы

int shm\_id, sem\_id;

shm\_id = shmget(102|IPC\_PRIVATE, BLOCK\_SIZE, 0666|IPC\_CREAT);

if (shm\_id == -1)

{

perror("shmget");

return 1;

}

sem\_id = semget(IPC\_PRIVATE, 1, IPC\_CREAT | 0600);

if (sem\_id == -1)

{

perror("semget");

return 1;

}

// fork innit?

if ((pid = fork()) == -1) {

perror("fork");

return 1;

}

if (pid == 0)

{

// Потомок

child(sb, sem\_id, shm\_id, mem, nbytes);

} else

{

// Предок

parent(sb, sem\_id, shm\_id, mem, nbytes, pid);

}

}

1. **Использовать очередь сообщений.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/wait.h>

#include <signal.h>

#include<sys/msg.h>

#include <string.h>

#define BLOCK\_SIZE 32

struct send{

long mtype;

char buf[BLOCK\_SIZE];

};

void handleSignal(int signum) {

printf("Signal (%d) from pid=[%d] received\n", signum, getpid());

}

void handleFile(int Q\_id, const char\* filePath) {

struct send snd;

char buf[BLOCK\_SIZE];

ssize\_t nbytes;

int file;

if ((file = open(filePath, O\_RDONLY)) == -1) {

fprintf(stderr, "Invalid path to file\n");

return;

}

while ((nbytes = read(file, buf, BLOCK\_SIZE)) > 0) {

memcpy(snd.buf, buf, sizeof(char)\*BLOCK\_SIZE);

msgsnd(Q\_id, &snd, BLOCK\_SIZE, 0);

}

close(file);

exit(0);

}

void childTask(int Q\_id)

{

printf("Child process created\n");

printf("Pid: %d\n", getpid());

if (signal(SIGUSR1, handleSignal) == SIG\_ERR) {

printf("Fail to set signal function\n");

}

struct send snd;

char rbuf[BLOCK\_SIZE];

ssize\_t nbytes;

while ((nbytes = msgrcv(Q\_id, &snd, BLOCK\_SIZE, 0, IPC\_NOWAIT)) > 0) {

memcpy(rbuf, snd.buf, sizeof(char)\*BLOCK\_SIZE);

write(1, rbuf, nbytes);

}

}

void parentTask(int Q\_id, const char\* filePath)

{

printf("Parent process created\n");

printf("Pid: %d\n", getpid());

if (signal(SIGCHLD, handleSignal) == SIG\_ERR) {

printf("Failed to set signal function\n");

}

handleFile(Q\_id, filePath);

}

void parentJob(const char\* filePath) {

pid\_t child\_pid;

int Q\_id;

struct send snd;

Q\_id = msgget(1|IPC\_PRIVATE,IPC\_CREAT|0666);

if (Q\_id<=0)

{

printf("Error=%d\n",Q\_id);

return;

}

printf("Parent PID = %d\n", getpid());

if ((child\_pid = fork()) == 0) {

childTask(Q\_id);

} else {

parentTask(Q\_id, filePath);

}

printf("\nWaiting for child\n");

kill(child\_pid, SIGUSR1);

int child\_status;

waitpid(child\_pid, &child\_status, 0);

if (WIFEXITED(child\_status)) {

int exit\_code = WEXITSTATUS(child\_status);

printf("Child process exited with code %d\n", exit\_code);

} else {

printf("Child process terminated abnormally\n");

}

msgctl(Q\_id, IPC\_RMID, NULL);

printf("Parent process finished\n");

}

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

// if (argc = 2) {

// printf("Invalid number of arguments\n");

// return 1;

// }

parentJob("./reg.txt");

return 0;

}