|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра теоретической и прикладной информатики | | |
| Лабораторная работа № 5 | | |
| по дисциплине «Управление ресурсами в вычислительных системах» | | |
| **МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ ПРОГРАММ** | | |
|  | | |
|  |  |  |
| Группа ПМ-12 | Лойченко данила |
| Вариант 3 |  |
|  |  |
|  |  |
|  |  |
| Преподаватели | Сивак М.А. |
|  | Стасышин В.М. |
| Новосибирск, 2024 | | |

1. **Задание**

Четыре дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередную строку некоторого стихотворения, при этом первый процесс передает 1-ю, 5-ю, 9-ю и т.д. строки, второй - 2-ю, 6-ю, 10-ю и т.д. строки, третий - 3-ю, 7-ю, 11-ю и т.д. строки, четвертый - 4-ю, 8-ю, 12- ю и т.д. строки. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы всех процессов. Решить задачу с использованием аппарата семафоров.

1. **Описание использованных структур**

**pid\_t fork(void)**

Порождение процесса-потомка, точной копии процесса-предка. Процесс-потомок в качестве возвращаемого значения системного вызова fork() получает 0, а процесс-предок - идентификатор процесса-потомка.

**void (\*signal (int signal, void (\*sigfunc) (int func)))(int)**

Дает указание выполнить функцию, на которую указывает sigfunc, в случае получения сигнала signal.

**int kill(pid\_t pid, int sig)**

Системный вызов kill может быть использован для посылки какого-либо сигнала

какому-либо процессу или группе процесса. Если значение pid является положительным, сигнал sig посылается процессу с идентификатором pid. Если pid равен 0, то sig посылается каждому процессу, который входит в группу текущего процесса. Если pid равен -1, то sig посылается каждому процессу, за исключением процесса с номером 1 (init), но есть нюансы, которые описываются ниже. Если pid меньше чем -1, то sig посылается каждому процессу, который входит в группу процесса -pid. Если sig равен 0, то никакой сигнал не посылается, а только выполняется проверка на ошибку.

**int semget(key\_t *key*, int *nsems*, int *semflg*)**

Системный вызов secret() возвращает набор семафоров System V идентификатор, связанный с аргументом key. Его можно использовать либо для получения идентификатора ранее созданного семафора set или создать новый набор. Новый набор семафоров nsems создаётся, если значение key равно IPC\_PRIVATE или с ключом key не связано ни одного существующего набора семафоров, а в semflg задано IPC\_CREAT.

**int semctl(int** *semid***, int** *semnum***, int** *cmd***, ...)**

Функция semctl позволяет выполнять операции, определенные в cmd над набором семафоров, указанным в semid или над семафором с номером semnum из этого набора. (Семафоры нумеруются, начиная с 0.)

**int semop(int semid, struct sembuf \*sops, unsigned nsops)**

Функция производит операции над выбранными элементами из набора семафоров semid. Каждый из элементов nsops в массиве sops определяет операцию, производимую над семафором в структуре struct sembuf, состоящей из полей: short sem\_num; short sem\_op,   short sem\_flg.

**int shm\_open(const char \*name, int oflag, mode\_t mode)**

shm\_open создает и открывает новый (или открывает уже существующий) объект разделяемой памяти POSIX. Операция shm\_open аналогична [open](https://www.opennet.ru/cgi-bin/opennet/man.cgi?topic=open&category=2)(2). name определяет собственно создаваемый объект разделяемой памяти для создания или открытия. oflag является маской битов, созданной через логическое сложение OR одного из флагов O\_RDONLY или O\_RWDR и любых других флагов. Поле mode задает права доступа к семафорам.

**int shm\_unlink(const char \*name)**

Функция shm\_unlink выполняет обратную операцию, удаляя объект, предварительно созданный с помощью shm\_open. При операции shm\_unlink удаляется имя объекта разделяемой памяти и, как только все процессы завершили работу с объектом и отменили его распределение, очищают пространство и уничтожают связанную с ним область памяти.

**int ftruncate(int fd, off\_t length)**

Функции ftruncate устанавливают длину обычного файла с файловым дескриптором fd в length байт. Если файл до этой операции был длиннее, то отсеченные данные теряются. Если файл был короче, то он увеличивается, а добавленная часть заполняется нулевыми байтами.

**void \* mmap(void \*start, size\_t length, int prot , int flags, int fd, off\_t offset)**

Функция mmap отражает length байтов, начиная со смещения offset файла (или другого объекта), определенного файловым описателем fd, в память, начиная с адреса start. Последний параметр (адрес) необязателен, и обычно бывает равен 0. Настоящее местоположение отраженных данных возвращается самой функцией mmap, и никогда не бывает равным 0.

**void perror(const char \*s)**

Процедура perror() выводит в стандартный поток ошибки сообщения, описывая ошибку, произошедшую при последнем системном вызове или вызове библиотечной функции. Сначала (если s не равно NULL и \*s не равно NULL) выводится строка s, затем двоеточие, пробел и сообщение, завершающееся переводом строки. Для большего удобства параметы строки должны содержать имя функции, вызвавшей ошибку. Номер ошибки извлекается из внешней переменной errno, которая устанавливается в случае ошибки, но не "очищается" в случае нормального завершения работы.

**pid\_t wait(int \*status)**

Приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился, то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

**void exit(int value)**

Выполняет немедленное завершение программы. Аргумент параметра value возвращается принимающей стороной (ОС или другой программой) в родительский процесс. Как правило, возвращается значение 0 или EXIT\_SUCCESS указывает на успешное завершение программы, и любое другое значение или значение макроса EXIT\_FAILURE используется для указания об аварийном завершении программы.

1. **Спецификация**

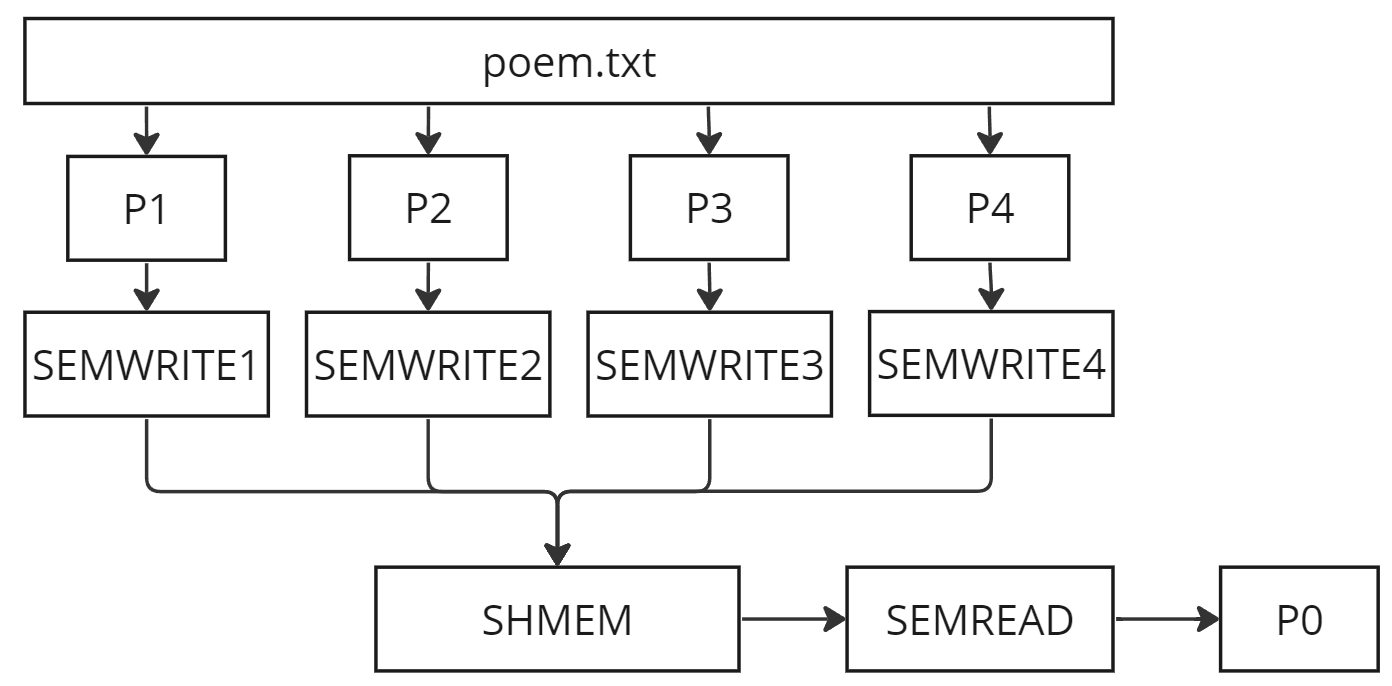
Программа разработана и протестирована на компьютере с операционной системой Linux. В качестве компилятора используется GCC версии 11.4.0.

Название файла: **PZ5.c**

Компиляция программы на языке Си: **gcc PZ5.c -o <результат компиляции>**

Запуск программы на языке Си: **./<результат компиляции>**

1. **Алгоритм работы на языке C**



1. Будем обозначать за P0 - родительский процесс, а P1, P2, P3 и P4 - дочерние процессы.
2. Создаем четыре семафора (SEMWRITE) на запись и один семафор (SEMREAD) на чтение и установим их как закрытые.
3. При помощи fork() создаем P1, P2, P3, P4.
4. В P0 открываем SEMWRITE1 для P1.
5. После разблокировки P1 он «закрывает за собой SEMWRITE1», записывает данные в SHMEM, открывает SEMREAD и встает на ожидание пока другой процесс не разблокирует SEMWRITE1.
6. После разблокировки P0 он «закрывает за собой SEMREAD», выводит в stdout данные, разблокирует SEMWRITE2 и встает на ожидание пока другой процесс не разблокирует SEMREAD1.
7. Повторяем 4-5 пункт меняя SEMWRITE(i), пока не закончится файл poem.txt.
8. Из дочерних процессов посылаем сигнал родительском об окончании отсылки данных в SHMEM.
9. Родительский процесс ожидает завершение всех дочерних процессов и заканчивает работу.

**Тесты программы на языке C**#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <sys/mman.h>

#include <sys/wait.h>

#include <sys/ipc.h>

#include <fcntl.h>

#include <signal.h>

#define SEM\_KEY\_1 123

#define SEM\_KEY\_2 456

#define SEM\_KEY\_3 768

#define SEM\_KEY\_4 910

#define SEM\_KEY\_5 1112

#define NAME "/tmp"

#define STRLEN 128

#define SIZE\_SHMEM (STRLEN \* sizeof(char))

#define NUM\_CHILDPROCS 4

FILE \*log;

volatile int stop = 0;

// Операция сигнала SIGUSR1 для отсановки считывания род процессом shmem

void inthand(int signum)

{

    stop = 1;

}

// P-операция для закрытия семафора

void P(int semid)

{

    struct sembuf buf;

    buf.sem\_num = 0;

    buf.sem\_op = -1;

    buf.sem\_flg = SEM\_UNDO;

    semop(semid, &buf, 1);

}

// V-операция для открытия семафора

void V(int semid)

{

    struct sembuf buf;

    buf.sem\_num = 0;

    buf.sem\_op = 1;

    buf.sem\_flg = SEM\_UNDO;

    semop(semid, &buf, 1);

}

void Child(int childNum, int semRead, int semWrite)

{

    // Открывает объект shmem

    int fd = shm\_open(NAME, O\_EXCL | O\_RDWR, 0600);

    if (fd < 0) {

        perror("Child   : shm\_open");

        return;

    }

    // Задание размера shmem

    ftruncate(fd, SIZE\_SHMEM);

    // Cоздает новое сопоставление в виртуальном адресном пространстве вызывающего процесса

    char \*data = (char \*)mmap(0, SIZE\_SHMEM, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

    fprintf(stdout, "pid: %d\t sender mapped addr: %p\n", getpid(), data);

    // Открытие файла на считывание для взятия k-ых строчек

    FILE \*fdPoem = fopen("poem.txt", "r");

    char strBuf[128];

    for (int i = 0; !feof(fdPoem); i++)

    {

        fgets(strBuf, sizeof(strBuf), fdPoem); // Считывание строки из файла

        if(i % NUM\_CHILDPROCS == childNum) // Если текующая строка должна использоваться доч. процессом, то...

        {

            P(semWrite); // Ожиание разблокировки семафора

            snprintf(data, 150, "pid: %d\t msg: %s", getpid(), strBuf); // записали данные в shmem

            V(semRead); // Разблокировали семафора для считывания родительским процессом

        }

    }

    usleep(1000);

    for (int i = 0; i < 5; i++) // Послание сигнала родительскому процессу об окончании считывания файла дочерними процессами

        kill(getppid(), SIGUSR1);

    V(semRead);

    close(fd);

    fclose(fdPoem);

    shm\_unlink(NAME);

}

void Parent(int semRead, int semWrite1, int semWrite2, int semWrite3, int semWrite4)

{

    signal(SIGUSR1, inthand);

    // Создает и открывает новый объект общей памяти

    int fd = shm\_open(NAME, O\_RDONLY, 0666);

    if (fd < 0)

    {

        perror("Parent  : shm\_open");

        return;

    }

    // Cоздает новое сопоставление в виртуальном адресном пространстве вызывающего процесса

    char \*data = (char \*)mmap(0, SIZE\_SHMEM, PROT\_READ, MAP\_SHARED, fd, 0);

    fprintf(stdout, "pid: %d\t receiver mapped addr: %p\n", getpid(), data);

    int sems[NUM\_CHILDPROCS];

    sems[0] =  semWrite1;

    sems[1] =  semWrite2;

    sems[2] =  semWrite3;

    sems[3] =  semWrite4;

    V(semWrite1); // Разблокировка семафора для P1

    for (int i = 1; ; i++)

    {

        P(semRead);

        if(stop != 1) // Ожиание разблокировки семафора

            fprintf(stdout, "%s", data);

        else

            break;

        V(sems[i % NUM\_CHILDPROCS]); // Разблокировали семафор указанного дочернего процесса процессом

    }

    fprintf(stdout, "\n");

    close(fd);

    shm\_unlink(NAME);

}

int main()

{

    log = fopen("log.txt", "w");

    // Создание семафора для считывания данных

    int semRead = semget(SEM\_KEY\_1, 1, IPC\_CREAT | 0666);

    if (semRead == -1) {

        perror("semget");

        exit(1);

    }

    // Создание семафора для записи данных

    int semWrite1 = semget(SEM\_KEY\_2, 1, IPC\_CREAT | 0666);

    if (semWrite1 == -1) {

        perror("semget");

        exit(1);

    }

    int semWrite2 = semget(SEM\_KEY\_3, 1, IPC\_CREAT | 0666);

    if (semWrite2 == -1) {

        perror("semget");

        exit(1);

    }

    int semWrite3 = semget(SEM\_KEY\_4, 1, IPC\_CREAT | 0666);

    if (semWrite3 == -1) {

        perror("semget");

        exit(1);

    }

    int semWrite4 = semget(SEM\_KEY\_5, 1, IPC\_CREAT | 0666);

    if (semWrite4 == -1) {

        perror("semget");

        exit(1);

    }

    // Установка начальный значений семафоров

    if (semctl(semRead, 0, SETVAL, 0) == -1 || semctl(semWrite1, 0, SETVAL, 0) == -1 ||

        semctl(semWrite2, 0, SETVAL, 0) == -1 || semctl(semWrite3, 0, SETVAL, 0) == -1 ||

        semctl(semWrite4, 0, SETVAL, 0) == -1)

    {

        perror("semctl");

        exit(1);

    }

    // Создает и открывает новый объект shmem

    int fdtest = shm\_open(NAME, O\_CREAT | O\_EXCL | O\_RDWR, 0600);

    pid\_t pidP1, pidP2, pidP3, pidP4;

    switch (pidP1 = fork())

    {

    case -1:

        perror("fork");

        exit(EXIT\_FAILURE);

    case 0:

        Child(0, semRead, semWrite1);

        exit(EXIT\_SUCCESS);

    default:

        switch (pidP2 = fork())

        {

        case -1:

            perror("fork");

            exit(EXIT\_FAILURE);

        case 0:

            Child(1, semRead, semWrite2);

            exit(EXIT\_SUCCESS);

        default:

            switch (pidP3 = fork())

            {

            case -1:

            perror("fork");

            exit(EXIT\_FAILURE);

            case 0:

                Child(2, semRead, semWrite3);

                exit(EXIT\_SUCCESS);

            default:

                switch (pidP4 = fork())

                {

                case -1:

                    perror("fork");

                    exit(EXIT\_FAILURE);

                case 0:

                    Child(3, semRead, semWrite4);

                    exit(EXIT\_SUCCESS);

                default:

                    Parent(semRead, semWrite1, semWrite2, semWrite3, semWrite4);

                    for (int i = 0; i < NUM\_CHILDPROCS; i++)

                        wait(NULL);

                    exit(EXIT\_SUCCESS);

                }

            }

        }

    }

    return 1;

}

1. **Тестирование программы**

|  |  |  |
| --- | --- | --- |
| **№** | **Назначение** | **Результат** |
| **1** | **Проверка работоспособности** |  |