МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное учреждение высшего образования
«Чувашский государственный университет И.Н. Ульянова»
Факультет информатики и вычислительной техники
Кафедра вычислительной техники

Системное операционные системы
Лабораторная работа 2
Вариант 5

Выполнил:

Студент группы ИВТ-41-20
Галкин Д.С.

Проверил:

Старший преподаватель
Первов С.Г.

Чебоксары, 2024

Цель работы:

Получение навыков многопроцессорного программирования.

Задание:

Задания к этой лабораторной работе такие же, как и в предыдущей. Только на этот раз необходимо написать не многопоточную, а многопроцессную программу. Отличие состоит в том, что вместо создания потока для каждой подзадачи создается отдельный процесс (с помощью системного вызова fork). Для того, чтобы процессы могли обмениваться информацией друг с другом, необходимо использовать разделяемую память. При этом необходимо синхронизировать доступ процессов к разделяемой памяти с помощью семафоров или мьютексов. Семафоры или мьютексы должны быть или именованными или размещаемыми в разделяемой памяти.

Ход работы:

Функция потока: reader

```
void *reader_process(int arg) {
    while (1) {
        sem_wait(priority_mutex);
        while (phm->write_count > 0 || (phm->num_writers_waiting > 0 && phm-
>read_count >= PATIENCE_THRESHOLD_READ)) {
            sem_post(priority_mutex);
            sem_wait(read_priority);
            sem_wait(priority_mutex);
        }
        phm->read_count++;
        sem_post(priority_mutex);

        sem_wait(read_mutex);
        phm->num_readers++;
        if (phm->num_readers == 1) {
            sem_wait(write_mutex);
        }
        sem_post(read_mutex);

        // Чтение
        printf("[READ] -> Читатель %d читает в библиотеке\n", arg);
        usleep(rand() % 1000000);
        printf("[READ] -> Читатель %d прочитал книгу в библиотеке\n", arg);

        sem_wait(read_mutex);
        phm->num_readers--;
        if (phm->num_readers == 0) {
            sem_post(write_mutex);
        }
        sem_post(read_mutex);

        sem_wait(priority_mutex);
        phm->read_count--;
        // if (shm->read_count == 0) {
        //     sem_post(write_mutex);        // }        if (phm->num_writers_waiting
> 0 && phm->read_count < PATIENCE_THRESHOLD_WRITE) {
            sem_post(write_priority);
        } else {
            sem_post(read_priority); // Позволяет другим читателям продолжить
        }
        sem_post(priority_mutex);

        usleep(rand() % 1000000);
    }
}
```

Функция потока: writer

```
void *writer_process(int arg) {
    while (1) {
        sem_wait(priority_mutex);
        phm->num_writers_waiting++;
        while (phm->num_readers > 0 || phm->write_count > 0) {
            sem_post(priority_mutex);
            sem_wait(write_priority);
            sem_wait(priority_mutex);
        }
        phm->num_writers_waiting--;
        phm->write_count++;
        sem_post(priority_mutex);

        sem_wait(write_mutex);

        // sem_wait(priority_mutex);
        // shm->num_writers_waiting--;        // shm->write_count++;        //
sem_post(priority_mutex);
        // Запись        printf("[WRITE] -> Писатель %d пишет в библиотеке\n", arg);
        usleep(rand() % 3000000);
        printf("[WRITE] -> Писатель %d закончил писать в библиотеке\n", arg);
        sem_post(write_mutex);

        sem_wait(priority_mutex);
        phm->write_count--;
        if (phm->write_count == 0 && phm->num_writers_waiting > 0 && phm->read_count
>= PATIENCE_THRESHOLD_READ) {
            sem_post(write_priority);
        } else if (phm->write_count == 0) {
            for (int i = 0; i < PATIENCE_THRESHOLD_WRITE; i++) { // Разрешаем максимум
трём читателям
                sem_post(read_priority);
            }
        }
        sem_post(priority_mutex);

        usleep(rand() % 3000000);
    }
}
```

main.c

```c
#include "model_thread.h"
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>



// Объявления для разделяемой памяти
struct SharedMemory {
    int num_readers;
    int num_writers_waiting;
    int read_count;
    int write_count;
};

int PATIENCE_THRESHOLD_READ = 3;
int PATIENCE_THRESHOLD_WRITE = 5;

// Инициализация именованных семафоров
sem_t *write_mutex;
sem_t *read_mutex;
sem_t *priority_mutex;
sem_t *write_priority;
sem_t *read_priority;

struct SharedMemory *shm, *phm;

void main() {
    // Создание разделяемой памяти
int shm_fd = shm_open("/my_shm", O_CREAT | O_RDWR | O_TRUNC, S_IRUSR | S_IWUSR);
ftruncate(shm_fd, sizeof(struct SharedMemory));
shm = mmap(0, sizeof(struct SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd,
0);

// Инициализация семафоров
write_mutex = sem_open("/write_mutex", O_CREAT, 0644, 1);
read_mutex = sem_open("/read_mutex", O_CREAT, 0644, 1);
priority_mutex = sem_open("/priority_mutex", O_CREAT, 0644, 1);
write_priority = sem_open("/write_priority", O_CREAT, 0644, 0);
read_priority = sem_open("/read_priority", O_CREAT, 0644, 0);

// Инициализация разделяемой памяти
shm->num_readers = 0;
shm->num_writers_waiting = 0;
shm->read_count = 0;
shm->write_count = 0;
// shm->patience_threshold_read = 3;
// shm->patience_threshold_write = 5;

pid_t pid;
char queues[] = "wwrrwwrrrrrrrrrrwwwwrrrrwwwwrrrrwwwwrrrrwwwwrrrrwww";
int queues_length = strlen(queues);
```

```
for (int i = 0; i < queues_length; i++) {
    pid = fork();
    phm = mmap(0, sizeof(struct SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
    if (pid > 0) { // Child process
        if (queues[i] == 'r') {
            reader_process(i);
        } else {
            writer_process(i);
        }
        exit(0); // Завершение дочернего процесса
    }
}

for (int i = 0; i < queues_length; i++) {
    wait(NULL); // Ожидание завершения всех дочерних процессов
}

// Очистка
sem_close(write_mutex);
sem_close(read_mutex);
sem_close(priority_mutex);
sem_close(write_priority);
sem_close(read_priority);
sem_unlink("/write_mutex");
sem_unlink("/read_mutex");
sem_unlink("/priority_mutex");
sem_unlink("/write_priority");
sem_unlink("/read_priority");
munmap(shm, sizeof(struct SharedMemory));
shm_unlink("/my_shm");
```

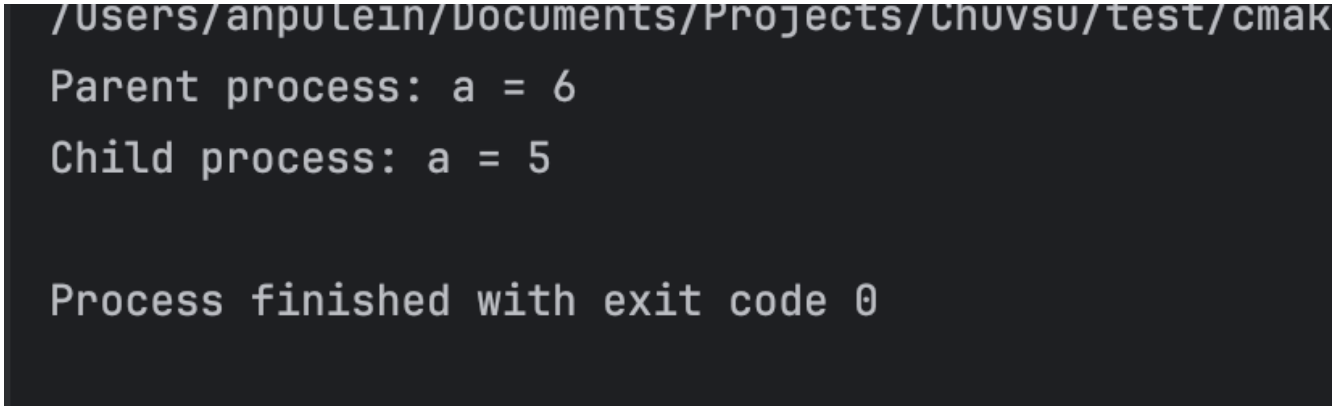
Пример работы:

```
/Users/anpulein/Documents/Projects/Chuvsu/Chuvsu_4Kurs_NetworkOperatingSystems/Ne
[WRITE] -> Писатель 0 пишет в библиотеке
[WRITE] -> Писатель 0 закончил писать в библиотеке
[READ] -> Читатель 2 читает в библиотеке
[READ] -> Читатель 3 читает в библиотеке
[READ] -> Читатель 6 читает в библиотеке
[READ] -> Читатель 2 прочитал книгу в библиотеке
[READ] -> Читатель 6 прочитал книгу в библиотеке
[READ] -> Читатель 3 прочитал книгу в библиотеке
[WRITE] -> Писатель 5 пишет в библиотеке
[WRITE] -> Писатель 5 закончил писать в библиотеке
[READ] -> Читатель 9 читает в библиотеке
[READ] -> Читатель 10 читает в библиотеке
[READ] -> Читатель 12 читает в библиотеке
[READ] -> Читатель 9 прочитал книгу в библиотеке
```

Задание:

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

int a = 5;

void *increment_function(void *arg) {
    for(int i = 0; i < 10; ++i) {
        a++;
        sleep(2); // Чтобы изменения были заметнее
    }
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread_id;

    // Создание поток
    if (pthread_create(&thread_id, NULL, increment_function, NULL)) {
        return 1;
    }

    // Вызываем fork()
    pid_t pid = fork();

    if (pid == -1) {
        return 1;
    } else if (pid == 0) {
        // Дочерний процесс
        printf("Child process: a = %d\n", a);
    } else {
        // Родительский процесс
        printf("Parent process: a = %d\n", a);
    }

    pthread_join(thread_id, NULL);

    return 0;
}
```

Родительский и дочерний процесс имеют разные значения переменной a



Вывод:

В ходе выполнения лабораторной работы я получил навыки многопроцессорного программирования.