Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №3 по курсу

«Операционные системы»

Группа: М8О-211БВ-24

Студент: Бахшалиев М.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.11.25

Москва, 2025

Постановка задачи

Вариант 19.

Нужно взять задание «Правило фильтрации: с вероятностью 80% строки отправляются в pipe1, иначе в pipe2. Дочерние процессы удаляют все гласные из строк» и переделать её с использованием shared memory и memory mapping. Варианты остаются те же, что и у первой лабораторной. Так как блокирующего чтения из каналов у вас больше не будет, то для синхронизации чтения и записи из shared memory будем использовать семафор.

Общий метод и алгоритм решения

Архитектурный подход

- Используется модель «клиент-сервер» с параллельной обработкой данных
- Межпроцессное взаимодействие организовано через разделяемую память (shared memory)
- Синхронизация доступа реализована с помощью семафоров POSIX

Алгоритм работы клиента:

1. **Инициализация ресурсов:**
    - Создание двух сегментов разделяемой памяти
    - Инициализация двух бинарных семафоров
    - Запуск двух серверных процессов через fork() + execv()
2. **Обработка входных данных:**
    - Чтение строк из стандартного ввода (STDIN_FILENO)
    - Удаление символа новой строки (\n) из конца строки
    - Вероятностное распределение: 80% - первому серверу, 20% - второму
3. **Передача данных:**
    - Захват семафора соответствующего сервера
    - Запись в разделяемую память:
        - Длина данных (uint32_t)
        - Текст строки (char[])
    - Освобождение семафора
4. **Завершение работы:**
    - Отправка сигнала завершения (UINT32_MAX) обоим серверам
    - Ожидание завершения серверных процессов (waitpid)
    - Корректное освобождение всех ресурсов

Алгоритм работы сервера:

1. **Инициализация:**
    - Открытие существующих сегментов разделяемой памяти
    - Подключение к существующим семафорам
    - Открытие выходного файла для записи результатов
2. **Цикл обработки:**
    - Ожидание сигнала от семафора
    - Проверка поля длины в разделяемой памяти:
        - UINT32_MAX → завершение работы
        - 0 → продолжение ожидания
        - 0 → обработка данных
    - Фильтрация строки (удаление гласных)
    - Запись результата в файл
    - Сброс поля длины в 0
    - Освобождение семафора

Ключевые особенности:

- Некритическая секция защищена семафорами
- Разделяемая память используется как кольцевой буфер
- Сигнал UINT32_MAX служит индикатором завершения
- Вероятностное распределение нагрузки между серверами

Код программы

client.c:

```
#include <fcntl.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <unistd.h>
#include <sys/fcntl.h>
#include <wait.h>
```

```c
#include <semaphore.h>
#include <sys/mman.h>

#define SHM_SIZE 4096

int main(int argc, char *argv[])
{
    if (argc != 3) {
        const char msg[] = "usage: client filename1 filename2\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    srand(time(NULL));

    char shm_name1[256], shm_name2[256];
    snprintf(shm_name1, sizeof(shm_name1), "/shm_%d_1", getpid());
    snprintf(shm_name2, sizeof(shm_name2), "/shm_%d_2", getpid());

    int shm1 = shm_open(shm_name1, O_RDWR | O_CREAT | O_TRUNC, 0600);
    int shm2 = shm_open(shm_name2, O_RDWR | O_CREAT | O_TRUNC, 0600);

    if (shm1 == -1 || shm2 == -1) {
        const char msg[] = "error: failed to create SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    if (ftruncate(shm1, SHM_SIZE) == -1 || ftruncate(shm2, SHM_SIZE) == -1) {
        const char msg[] = "error: failed to resize SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    char *shm_buf1 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm1, 0);
    char *shm_buf2 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm2, 0);

    if (shm_buf1 == MAP_FAILED || shm_buf2 == MAP_FAILED) {
        const char msg[] = "error: failed to map SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
```

```
char sem_name1[256], sem_name2[256];
snprintf(sem_name1, sizeof(sem_name1), "/sem_%d_1", getpid());
snprintf(sem_name2, sizeof(sem_name2), "/sem_%d_2", getpid());

sem_t *sem1 = sem_open(sem_name1, O_CREAT, 0600, 1);
sem_t *sem2 = sem_open(sem_name2, O_CREAT, 0600, 1);

if (sem1 == SEM_FAILED || sem2 == SEM_FAILED) {
    const char msg[] = "error: failed to create semaphore\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    _exit(EXIT_FAILURE);
}

pid_t server1 = fork();
if (server1 == 0) {
    char *args[] = {"server", argv[1], shm_name1, sem_name1, NULL};
    execv("./server", args);

    const char msg[] = "error: failed to exec server 1\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    _exit(EXIT_FAILURE);
} else if (server1 == -1) {
    const char msg[] = "error: failed to fork server 1\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    _exit(EXIT_FAILURE);
}

pid_t server2 = fork();
if (server2 == 0) {
    char *args[] = {"server", argv[2], shm_name2, sem_name2, NULL};
    execv("./server", args);

    const char msg[] = "error: failed to exec server 2\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    _exit(EXIT_FAILURE);
} else if (server2 == -1) {
    const char msg[] = "error: failed to fork server 2\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    _exit(EXIT_FAILURE);
}

bool running = true;
while (running) {
```

```
    char buf[SHM_SIZE - sizeof(uint32_t)];
    ssize_t bytes = read(STDIN_FILENO, buf, sizeof(buf));

    if (bytes == -1) {
        const char msg[] = "error: failed to read from standard input\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    if (bytes > 0) {
        if (buf[bytes - 1] == '\n') {
            bytes--;
        }

        if (bytes == 0) {
            running = false;
            continue;
        }

        int r = rand() % 100;
        if (r < 80) {
            sem_wait(sem1);
            uint32_t *length1 = (uint32_t *)shm_buf1;
            char *text1 = shm_buf1 + sizeof(uint32_t);
            *length1 = bytes;
            memcpy(text1, buf, bytes);
            sem_post(sem1);
        } else {
            sem_wait(sem2);
            uint32_t *length2 = (uint32_t *)shm_buf2;
            char *text2 = shm_buf2 + sizeof(uint32_t);
            *length2 = bytes;
            memcpy(text2, buf, bytes);
            sem_post(sem2);
        }
    } else {
        running = false;
    }
}

sem_wait(sem1);
uint32_t *length1 = (uint32_t *)shm_buf1;
*length1 = UINT32_MAX;
```

```
    sem_post(sem1);

    sem_wait(sem2);
    uint32_t *length2 = (uint32_t *)shm_buf2;
    *length2 = UINT32_MAX;
    sem_post(sem2);

    waitpid(server1, NULL, 0);
    waitpid(server2, NULL, 0);

    sem_unlink(sem_name1);
    sem_close(sem1);
    sem_unlink(sem_name2);
    sem_close(sem2);
    munmap(shm_buf1, SHM_SIZE);
    munmap(shm_buf2, SHM_SIZE);
    shm_unlink(shm_name1);
    shm_unlink(shm_name2);
    close(shm1);
    close(shm2);
}
```

server.c:

```
#include <fcntl.h>
#include <stdint.h>
#include <stdbool.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/fcntl.h>
#include <wait.h>
#include <semaphore.h>
#include <sys/mman.h>

#define SHM_SIZE 4096

int is_vowel(char c) {
    c = tolower(c);
    return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c == 'y');
}
```

```
void remove_vowels(const char *input, char *output, size_t length) {
    size_t j = 0;
    for (size_t i = 0; i < length; i++) {
        if (!is_vowel(input[i])) {
            output[j++] = input[i];
        }
    }
    output[j] = '\0';
}

int main(int argc, char *argv[])
{
    if (argc != 4) {
        const char msg[] = "usage: server filename shm_name sem_name\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    int shm = shm_open(argv[2], O_RDWR, 0600);
    if (shm == -1) {
        const char msg[] = "error: failed to open SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    char *shm_buf = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);
    if (shm_buf == MAP_FAILED) {
        const char msg[] = "error: failed to map SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    sem_t *sem = sem_open(argv[3], O_RDWR);
    if (sem == SEM_FAILED) {
        const char msg[] = "error: failed to open semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    int file = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0600);
    if (file == -1) {
        const char msg[] = "error: failed to open file\n";
        write(STDERR_FILENO, msg, sizeof(msg));
```

```
        _exit(EXIT_FAILURE);
    }

    bool running = true;
    while (running) {
        sem_wait(sem);

        uint32_t *length = (uint32_t *)shm_buf;
        char *text = shm_buf + sizeof(uint32_t);

        if (*length == UINT32_MAX) {
            running = false;
        } else if (*length > 0) {
            char result[SHM_SIZE - sizeof(uint32_t)];
            remove_vowels(text, result, *length);

            if (strlen(result) > 0) {
                dprintf(file, "%s\n", result);
            }

            *length = 0;
        }

        sem_post(sem);
    }

    close(file);
    sem_close(sem);
    munmap(shm_buf, SHM_SIZE);
    close(shm);
}
```

Протокол работы программы

Тестирование:

Ввод:

aaaaa

bbbbb

ccccc

ddddd

eeeee

fffff

1.txt:

bbbbb

ddddd

fffff

2.txt:

ccccc

Вывод

Данная программа реализует задачу фильтрации строк с использованием механизмов разделяемой памяти и семафоров для межпроцессного взаимодействия. Клиентский процесс распределяет строки, считанные из стандартного ввода, между двумя серверными процессами с вероятностью 80% и 20%. Для передачи данных используются сегменты разделяемой памяти, а синхронизация доступа к ним обеспечивается семафорами, что исключает состояние гонки.

Серверные процессы выполняют фильтрацию строк путем удаления гласных букв и сохраняют результаты в отдельные файлы. Корректное завершение работы обеспечивается отправкой специального сигнала через разделяемую память. Применение данных механизмов позволяет эффективно организовать параллельную обработку данных без использования блокирующих операций ввода-вывода.