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: Антыгин В.Е.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 5.11.25

Москва, 2025

Постановка задачи

Вариант 11.

Необходимо переделать первую лабораторную работу с использованием shared memory (разделяемой памяти) и memory mapping (отображения памяти). Вместо каналов (pipes) для передачи данных между процессами используется механизм разделяемой памяти.

Родительский процесс создает два дочерних процесса. Родительский процесс принимает от пользователя строки произвольной длины и передает их через shared memory первому дочернему процессу. Процессы child1 и child2 производят обработку строк. Child2 передает результат своей работы родительскому процессу через shared memory. Родительский процесс полученный результат выводит в стандартный поток вывода.

Child1 переводит строки в верхний регистр.

Child2 превращает все пробельные символы в символ «_».

Для синхронизации доступа к разделяемой памяти используются именованные семафоры POSIX.

Общий метод и алгоритм решения

Использованные системные вызовы и функции:

- shm_open – создание или открытие объекта разделяемой памяти
- shm_unlink – удаление объекта разделяемой памяти из системы
- ftruncate – установка размера объекта разделяемой памяти
- mmap – отображение разделяемой памяти в адресное пространство процесса
- munmap – отмена отображения памяти
- sem_open – создание или открытие именованного семафора
- sem_close – закрытие семафора в процессе
- sem_unlink – удаление именованного семафора из системы
- sem_wait – блокирующее ожидание (захват семафора)
- sem_post – освобождение семафора (увеличение счетчика)
- fork – создание дочернего процесса
- execl – замена текущего процесса новой программой
- waitpid – ожидание завершения дочернего процесса
- read/write – чтение и запись данных

Алгоритм работы программы:

Программа реализует конвейерную обработку данных с использованием трех областей разделяемой памяти и шести семафоров для синхронизации.

Родительский процесс:

1. Создает три объекта shared memory (shm1, shm2, shm3) размером 4096 байт каждый
2. Для каждого shared memory создает пару семафоров:
    - sem_data (начальное значение = 0) – сигнализирует о готовности данных
    - sem_free (начальное значение = 1) – сигнализирует о свободном буфере
3. Запускает два дочерних процесса (child1 и child2)
4. В цикле считывает строки из stdin
5. Захватывает семафор sem1_free, записывает данные в shm1, освобождает sem1_data
6. Захватывает sem3_data, читает результат из shm3, освобождает sem3_free
7. Выводит результат в stdout
8. При получении EOF отправляет сигнал завершения (length = UINT32_MAX)
9. Ожидает завершения дочерних процессов и очищает ресурсы

Первый дочерний процесс (child1):

1) Открывает shm1 (входной) и shm2 (выходной)
2) Открывает соответствующие семафоры
3) В цикле:
    - Захватывает sem1_data, читает данные из shm1
    - Преобразует все символы в верхний регистр
    - Захватывает sem2_free, записывает результат в shm2, освобождает sem2_data
    - Освобождает sem1_free
4) При получении сигнала завершения (length = UINT32_MAX) передает его в shm2 и завершается

Второй дочерний процесс (child2):

1. Открывает shm2 (входной) и shm3 (выходной)

2. Открывает соответствующие семафоры
3. В цикле:
    - Захватывает sem2_data, читает данные из shm2
    - Заменяет все пробельные символы на символ '_'
    - Захватывает sem3_free, записывает результат в shm3, освобождает sem3_data
    - Освобождает sem2_free
4. При получении сигнала завершения передает его в shm3 и завершается

Код программы

parrent.c

```
#include <fcntl.h>
#include <semaphore.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>



#define SHM_SIZE 4096
#define DATA_SIZE (SHM_SIZE - sizeof(uint32_t))

const char SHM1_NAME[] = "/shm1";
const char SHM2_NAME[] = "/shm2";
const char SHM3_NAME[] = "/shm3";

const char SEM1_DATA[] = "/sem1_data";
const char SEM1_FREE[] = "/sem1_free";
const char SEM2_DATA[] = "/sem2_data";
const char SEM2_FREE[] = "/sem2_free";
const char SEM3_DATA[] = "/sem3_data";
const char SEM3_FREE[] = "/sem3_free";

typedef struct {
    uint32_t length;
    char data[DATA_SIZE];
} shm_buffer_t;
```

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <child1> <child2>\n", argv[0]);
        return 1;
    }

    shm_unlink(SHM1_NAME);
    shm_unlink(SHM2_NAME);



    int shm1_fd = shm_open(SHM1_NAME, O_RDWR | O_CREAT | O_EXCL, 0600);
    if (shm1_fd == -1) {
        perror("shm_open shm1");
        return 1;
    }

    if (ftruncate(shm1_fd, SHM_SIZE) == -1) {
        perror("ftruncate shm1");
        return 1;
    }

    shm_buffer_t* shm1 = mmap(NULL, SHM_SIZE, PROT_WRITE | PROT_READ, MAP_SHARED,
shm1_fd, 0);
    if (shm1 == MAP_FAILED) {
        perror("mmap shm1");
        return 1;
    }
    shm1->length = 0;



    sem_unlink(SEM1_DATA);
    sem_unlink(SEM1_FREE);

    sem_t* sem1_data = sem_open(SEM1_DATA, O_CREAT | O_EXCL, 0600, 0);
    sem_t* sem1_free = sem_open(SEM1_FREE, O_CREAT | O_EXCL, 0600, 1);

    if (sem1_data == SEM_FAILED || sem1_free == SEM_FAILED) {
        perror("sem_open sem1");
        return 1;
    }



    shm_unlink(SHM2_NAME);
    int shm2_fd = shm_open(SHM2_NAME, O_RDWR | O_CREAT | O_EXCL, 0600);
    if (shm2_fd == -1) {
```

```
        perror("shm_open shm2");
        return 1;
    }
    if (ftruncate(shm2_fd, SHM_SIZE) == -1) {
        perror("ftruncate shm2");
        return 1;
    }
    shm_buffer_t* shm2 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm2_fd, 0);
    if (shm2 == MAP_FAILED) {
        perror("mmap shm2");
        return 1;
    }
    shm2->length = 0;

    sem_unlink(SEM2_DATA);
    sem_unlink(SEM2_FREE);
    sem_t* sem2_data = sem_open(SEM2_DATA, O_CREAT | O_EXCL, 0600, 0);
    sem_t* sem2_free = sem_open(SEM2_FREE, O_CREAT | O_EXCL, 0600, 1);
    if (sem2_data == SEM_FAILED || sem2_free == SEM_FAILED) {
        perror("sem_open sem2");
        return 1;
    }



    shm_unlink(SHM3_NAME);
    int shm3_fd = shm_open(SHM3_NAME, O_RDWR | O_CREAT | O_EXCL, 0600);
    if (shm3_fd == -1) {
        perror("shm_open shm3");
        return 1;
    }
    if (ftruncate(shm3_fd, SHM_SIZE) == -1) {
        perror("ftruncate shm3");
        return 1;
    }
    shm_buffer_t* shm3 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm3_fd, 0);
    if (shm3 == MAP_FAILED) {
        perror("mmap shm3");
        return 1;
    }
    shm3->length = 0;

    sem_unlink(SEM3_DATA);
    sem_unlink(SEM3_FREE);
    sem_t* sem3_data = sem_open(SEM3_DATA, O_CREAT | O_EXCL, 0600, 0);
    sem_t* sem3_free = sem_open(SEM3_FREE, O_CREAT | O_EXCL, 0600, 1);
    if (sem3_data == SEM_FAILED || sem3_free == SEM_FAILED) {
```

```
    perror("sem_open sem3");
    return 1;
}

pid_t pid1 = fork();
if (pid1 == -1) {
    perror("fork child1");
    return 1;
}
if (pid1 == 0) {
    execl(argv[1], argv[1], NULL);
    perror("execl child1");
    _exit(2);
}

pid_t pid2 = fork();
if (pid1 == -1) {
    perror("fork child2");
    return 1;
}
if (pid2 == 0) {
    execl(argv[2], argv[2], NULL);
    perror("execl child2");
    _exit(2);
}



char input_buff[DATA_SIZE];
bool running = true;

while (running) {
    ssize_t bytes_cnt = read(STDIN_FILENO, input_buff, sizeof(input_buff));

    if ((bytes_cnt <= 0) || (input_buff[0] == '\n')) {
        sem_wait(sem1_free);
        shm1->length = UINT32_MAX;
        sem_post(sem1_data);
        running = false;
        break;
    }

    sem_wait(sem1_free);
    shm1->length = bytes_cnt;
    memcpy(shm1->data, input_buff, bytes_cnt);
    sem_post(sem1_data);
```

```
            sem_wait(sem3_data);
            if (shm3->length == UINT32_MAX) {
                running = false;

            } else if (shm3->length > 0) {
                write(STDOUT_FILENO, shm3->data, shm3->length);
                fputc('\n', stdout);
            }

            sem_post(sem3_free);
        }


        waitpid(pid1, NULL, 0);
        waitpid(pid2, NULL, 0);

        sem_close(sem1_data);
        sem_close(sem1_free);
        sem_close(sem2_data);
        sem_close(sem2_free);
        sem_close(sem3_data);
        sem_close(sem3_free);

        sem_unlink(SEM1_DATA);
        sem_unlink(SEM1_FREE);
        sem_unlink(SEM2_DATA);
        sem_unlink(SEM2_FREE);
        sem_unlink(SEM3_DATA);
        sem_unlink(SEM3_FREE);

        munmap(shm1, SHM_SIZE);
        munmap(shm2, SHM_SIZE);
        munmap(shm3, SHM_SIZE);

        shm_unlink(SHM1_NAME);
        shm_unlink(SHM2_NAME);
        shm_unlink(SHM3_NAME);

        close(shm1_fd);
        close(shm2_fd);
        close(shm3_fd);

        return 0;
    }
```

child1.c

```
// child1.c
#include <ctype.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_SIZE 4096
#define DATA_SIZE (SHM_SIZE - sizeof(uint32_t))

const char SHM1_NAME[] = "/shm1";
const char SHM2_NAME[] = "/shm2";
const char SEM1_DATA[] = "/sem1_data";
const char SEM1_FREE[] = "/sem1_free";
const char SEM2_DATA[] = "/sem2_data";
const char SEM2_FREE[] = "/sem2_free";

typedef struct {
    uint32_t length;
    char data[DATA_SIZE];
} shm_buffer_t;

int main() {
    int shm1_fd = shm_open(SHM1_NAME, O_RDWR, 0600);
    if (shm1_fd == -1) {
        perror("child1: shm_open shm1");
        return 1;
    }

    shm_buffer_t* shm1 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm1_fd, 0);
    if (shm1 == MAP_FAILED) {
        perror("child1: mmap shm1");
        return 1;
    }

    sem_t* sem1_data = sem_open(SEM1_DATA, 0);
    sem_t* sem1_free = sem_open(SEM1_FREE, 0);
    if (sem1_data == SEM_FAILED || sem1_free == SEM_FAILED) {
        perror("child1: sem_open sem1");
        return 1;
```

```
    }

    int shm2_fd = shm_open(SHM2_NAME, O_RDWR, 0600);
    if (shm2_fd == -1) {
        perror("child1: shm_open shm2");
        return 1;
    }

    shm_buffer_t* shm2 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm2_fd, 0);
    if (shm2 == MAP_FAILED) {
        perror("child1: mmap shm2");
        return 1;
    }

    sem_t* sem2_data = sem_open(SEM2_DATA, 0);
    sem_t* sem2_free = sem_open(SEM2_FREE, 0);
    if (sem2_data == SEM_FAILED || sem2_free == SEM_FAILED) {
        perror("child1: sem_open sem2");
        return 1;
    }

    bool running = true;
    while (running) {
        sem_wait(sem1_data);

        if (shm1->length == UINT32_MAX) {
            sem_wait(sem2_free);
            shm2->length = UINT32_MAX;
            sem_post(sem2_data);
            sem_post(sem2_free);
            sem_post(sem1_free);
            break;
        }

        for (uint32_t i = 0; i < shm1->length; i++) {
            shm1->data[i] = toupper((unsigned char)shm1->data[i]);
        }

        sem_wait(sem2_free);
        shm2->length = shm1->length;
        memcpy(shm2->data, shm1->data, shm1->length);
        sem_post(sem2_data);

        sem_post(sem1_free);
    }
```

```
    sem_close(sem1_data);
    sem_close(sem1_free);
    sem_close(sem2_data);
    sem_close(sem2_free);
    munmap(shm1, SHM_SIZE);
    munmap(shm2, SHM_SIZE);
    close(shm1_fd);
    close(shm2_fd);

    return 0;}
```

child2.c

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

#define SHM_SIZE 4096
#define DATA_SIZE (SHM_SIZE - sizeof(uint32_t))

const char SHM2_NAME[] = "/shm2";
const char SHM3_NAME[] = "/shm3";
const char SEM2_DATA[] = "/sem2_data";
const char SEM2_FREE[] = "/sem2_free";
const char SEM3_DATA[] = "/sem3_data";
const char SEM3_FREE[] = "/sem3_free";

typedef struct {
    uint32_t length;
    char data[DATA_SIZE];
} shm_buffer_t;

int main(void) {
    int shm2_fd = shm_open(SHM2_NAME, O_RDWR, 0600);
    if (shm2_fd == -1) {
        perror("child2: shm_open shm2");
        return 1;
    }
    shm_buffer_t* shm2 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm2_fd, 0);
    if (shm2 == MAP_FAILED) {
        perror("child2: mmap shm2");
```

```
        return 1;
    }

    sem_t* sem2_data = sem_open(SEM2_DATA, 0);
    sem_t* sem2_free = sem_open(SEM2_FREE, 0);
    if (sem2_data == SEM_FAILED || sem2_free == SEM_FAILED) {
        perror("child2: sem_open sem2");
        return 1;
    }

    int shm3_fd = shm_open(SHM3_NAME, O_RDWR, 0600);
    if (shm3_fd == -1) {
        perror("child2: shm_open shm3");
        return 1;
    }
    shm_buffer_t* shm3 = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm3_fd, 0);
    if (shm3 == MAP_FAILED) {
        perror("child2: mmap shm3");
        return 1;
    }

    sem_t* sem3_data = sem_open(SEM3_DATA, 0);
    sem_t* sem3_free = sem_open(SEM3_FREE, 0);
    if (sem3_data == SEM_FAILED || sem3_free == SEM_FAILED) {
        perror("child2: sem_open sem3");
        return 1;
    }

    bool running = true;
    while (running) {
        sem_wait(sem2_data);

        if (shm2->length == UINT32_MAX) {
            sem_wait(sem3_free);
            shm3->length = UINT32_MAX;
            sem_post(sem3_data);
            sem_post(sem2_free);
            running = false;
            break;
        }

        for (uint32_t i = 0; i < shm2->length; i++) {
            if (isspace((unsigned char)shm2->data[i])) {
                shm2->data[i] = '_';
            }
        }
```

```
            sem_wait(sem3_free);
            shm3->length = shm2->length;
            memcpy(shm3->data, shm2->data, shm2->length);
            sem_post(sem3_data);

            sem_post(sem2_free);
        }

        sem_close(sem2_data);
        sem_close(sem2_free);
        sem_close(sem3_data);
        sem_close(sem3_free);
        munmap(shm2, SHM_SIZE);
        munmap(shm3, SHM_SIZE);
        close(shm2_fd);
        close(shm3_fd);

        return 0;
    }
```

Протокол работы программы

```
…/build/bin ✕ ./parent.out child1.out child2.out
daskdmnaklsmd qd l      kn lknq  mw;` d '.a.
DASKDMNAKLSMD_QD_L_KN_LKNQ__MW;`_D_'.A.
sald;m a;d m    ; md a
SALD;M_A;D_M_;_MD_A
asdk;m a;sd'    d; mwq1231szsd
ASDK;M_A;SD'_D;_MWQ1231SZSD
m;wmf;dm;fÁSMD:A
M;WMF;DM;FÁSMD:A

______

___

____
                a       .
__A_.
                a       .
_________________A_______.

…/build/bin ❯
```

Вывод

Выполнив эту работу я приобрел навыки работы с разделяемой памятью (shared memory) и механизмами синхронизации процессов с помощью семафоров POSIX. Освоил использование функций для работы с shared memory (shm_open, mmap, munmap, shm_unlink) и семафорами (sem_open, sem_wait, sem_post, sem_unlink).

Основное отличие от первой лабораторной работы заключается в механизме передачи данных: вместо каналов (pipes) с блокирующим чтением используется разделяемая память с явной синхронизацией через семафоры. Это обеспечивает более высокую скорость передачи данных за счет отсутствия копирования через буферы ядра ОС.

Столкнулся с необходимостью тщательной синхронизации доступа к разделяемым ресурсам для избежания состояний гонки (race conditions).