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»

Выполнил: М. Ефимовых
Группа: М8О-208БВ-24
Вариант: 11
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Составить и отладить программу, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решения задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Цель работы

Изучение механизмов межпроцессного взаимодействия через отображаемые файлы (memory-mapped files) и синхронизации процессов с использованием семафоров.

Задание

Реализовать программу, состоящую из родительского и двух дочерних процессов, взаимодействующих через отображаемый файл (memory-mapped file).

Родительский процесс считывает строку от пользователя и записывает её в отображаемый файл. Затем создаются два дочерних процесса, которые последовательно обрабатывают данные в этом файле:

- **Первый дочерний процесс** преобразует все буквы строки в верхний регистр
- **Второй дочерний процесс** заменяет все пробелы на символы подчеркивания

После завершения работы обоих дочерних процессов родительский процесс выводит финальный результат.

Для синхронизации доступа к общему файлу используются именованные семафоры POSIX.

Вариант

11

Архитектура программы

Общая структура

Программа реализует многопроцессную архитектуру с использованием memory-mapped файлов для межпроцессного взаимодействия. Основные компоненты системы:

- **parent.cpp** - родительский процесс, управляет созданием дочерних процессов и координирует их работу
- **child1.cpp** - первый дочерний процесс, преобразует строку в верхний регистр
- **child2.cpp** - второй дочерний процесс, заменяет пробелы на подчеркивания

- **os.cpp/os.hpp** - библиотека с функциями для работы с процессами, семафорами и отображаемыми файлами
- **shared_data.hpp** - структура данных для разделяемой памяти

Механизм взаимодействия процессов

Memory-mapped файл

Для обмена данными между процессами используется файл `data_to_process.txt`, который отображается в адресное пространство всех процессов с помощью системного вызова `mmap()`. Это позволяет всем процессам работать с одними и теми же данными в памяти.

Синхронизация с помощью семафоров

Для координации доступа к отображаемому файлу и правильного порядка выполнения используются три именованных POSIX-семафора:

- **sem_child** (начальное значение: 0) - сигнализирует дочерним процессам о готовности данных и координирует их последовательную работу
- **sem_parent** (начальное значение: 0) - сигнализирует родительскому процессу о завершении обработки
- **sem_file** (начальное значение: 1) - мьютекс для защиты доступа к файлу от одновременной записи

Алгоритм работы программы

Родительский процесс (parent.cpp)

1. Создает разделяемую память для управляющих данных с помощью `shm_open()` и инициализирует семафоры
2. Создает и отображает файл `data_to_process.txt` для обмена данными
3. Считывает строку от пользователя и записывает её в отображаемый файл
4. Создает два дочерних процесса с помощью `fork()` и `execl()`
5. Освобождает семафор `sem_child`, разрешая первому дочернему процессу начать работу
6. Ожидает сигнала от второго дочернего процесса на семафоре `sem_parent`
7. Выводит итоговый результат и очищает ресурсы

Первый дочерний процесс (child1.cpp)

1. Открывает разделяемую память и получает доступ к семафорам
2. Ожидает разрешения на семафоре `sem_child`
3. Захватывает мьютекс `sem_file` для эксклюзивного доступа к файлу
4. Отображает файл в память и преобразует все символы в верхний регистр
5. Освобождает мьютекс `sem_file`
6. Сигнализирует второму дочернему процессу через `sem_child`
7. Завершает работу

Второй дочерний процесс (child2.cpp)

1. Открывает разделяемую память и получает доступ к семафорам
2. Ожидает сигнала от первого дочернего процесса на семафоре `sem_child`
3. Захватывает мьютекс `sem_file`
4. Отображает файл и заменяет все пробелы на символы подчеркивания
5. Освобождает мьютекс `sem_file`
6. Сигнализирует родительскому процессу через `sem_parent`
7. Завершает работу

Использованные системные вызовы

- `shm_open/shm_unlink` - создание/удаление объекта разделяемой памяти POSIX
- `mmap/munmap` - отображение файла или памяти в адресное пространство процесса
- `sem_open/sem_close/sem_unlink` - работа с именованными семафорами POSIX
- `sem_wait/sem_post` - операции ожидания и освобождения семафора
- `fork` - создание дочернего процесса
- `execl` - замена образа текущего процесса новой программой
- `waitpid` - ожидание завершения дочернего процесса
- `open/close` - открытие/закрытие файла
- `ftruncate` - установка размера файла

Результаты тестирования

Пример работы программы

Программа была скомпилирована и протестирована на ОС Linux. Ниже приведен пример её работы:

```
$ ./parent
Enter a string: hello world from test
Final output: HELLO_WORLD_FROM_TEST
```

Пошаговое описание работы

1. **Инициализация:** Родительский процесс создает разделяемую память и семафоры, создает и отображает файл `data_to_process.txt`
2. **Ввод данных:** Пользователь вводит строку "hello world from test"
3. **Создание дочерних процессов:** Родительский процесс создает два дочерних процесса с помощью `fork()` и `execl()`
4. **Запуск первого процесса:** Родительский процесс освобождает `sem_child`, разрешая первому дочернему процессу начать работу
5. **Обработка первым процессом:** Первый дочерний процесс (`child1`):
   - Захватывает семафоры `sem_child` и `sem_file`
   - Отображает файл в память
   - Преобразует строку в верхний регистр: "hello world from test"→ "HELLO WORLD FROM TEST"
   - Освобождает семафоры и сигнализирует второму процессу
6. **Обработка вторым процессом:** Второй дочерний процесс (`child2`):
   - Захватывает семафоры `sem_child` и `sem_file`
   - Отображает файл в память
   - Заменяет пробелы на подчеркивания: "HELLO WORLD FROM TEST"→ "HELLO_WORLD_F
   - Освобождает семафоры и сигнализирует родительскому процессу
7. **Вывод результата:** Родительский процесс выводит итоговый результат обработки
8. **Очистка ресурсов:** Родительский процесс удаляет файл, семафоры и разделяемую память

Тестовые примеры

Тест 1: Простая строка

```
Вход: "hello world"
Выход: "HELLO_WORLD"
```

Тест 2: Строка с несколькими пробелами

```
Вход: "this   is   a   test"
Выход: "THIS___IS___A___TEST"
```

Тест 3: Смешанный регистр

```
Вход: "HeLLo WoRLd"
Выход: "HELLO_WORLD"
```

Тест 4: Строка с цифрами и символами

```
Вход: "test123 program!"
Выход: "TEST123_PROGRAM!"
```

Анализ использованных системных вызовов

В программе используются следующие системные вызовы POSIX:
Работа с разделяемой памятью:

- `shm_open()` – создание/открытие объекта разделяемой памяти
- `shm_unlink()` – удаление объекта разделяемой памяти
- `ftruncate()` – установка размера объекта
- `mmap()` – отображение памяти в адресное пространство процесса
- `munmap()` – отмена отображения памяти

Работа с семафорами:

- `sem_open()` – создание/открытие именованного семафора
- `sem_close()` – закрытие семафора
- `sem_unlink()` – удаление именованного семафора
- `sem_wait()` – блокировка на семафоре
- `sem_post()` – разблокировка семафора

Работа с процессами:

- `fork()` – создание дочернего процесса
- `execl()` – запуск новой программы в процессе
- `waitpid()` – ожидание завершения дочернего процесса

Работа с файлами:

- `open()` – открытие файла
- `close()` – закрытие файлового дескриптора
- `unlink()` – удаление файла
- `read()` – чтение из stdin
- `write()` – запись в stdout

Системные вызовы, используемые в программе:

- `shm_open()`, `shm_unlink()` - создание/удаление объекта разделяемой памяти "/shared"
- `mmap()`, `munmap()` - отображение/отмена отображения памяти и файлов
- `sem_open()`, `sem_close()`, `sem_unlink()` - работа с именованными семафорами
- `sem_wait()`, `sem_post()` - операции синхронизации
- `fork()` - создание дочернего процесса
- `execl()` - замена образа процесса
- `open()`, `close()`, `ftruncate()` - работа с файлом data_to_process.txt
- `waitpid()` - ожидание завершения дочерних процессов
- `read()`, `write()` - ввод/вывод данных
- `unlink()` - удаление файла

Особенности синхронизации

Программа демонстрирует корректную работу механизма синхронизации с помощью семафоров:

- Семафор `sem_file` (инициализирован единицей) работает как мьютекс, предотвращая одновременный доступ к файлу
- Семафоры `sem_child` и `sem_parent` (инициализированы нулем) обеспечивают правильный порядок выполнения: сначала child1, затем child2, затем parent
- Использование memory-mapped файла позволяет дочерним процессам видеть изменения, сделанные друг другом, в реальном времени

Выводы

В ходе выполнения данной лабораторной работы я получил практический опыт работы с механизмами межпроцессного взаимодействия в POSIX-совместимых операционных системах.

Основные результаты

Межпроцессное взаимодействие через memory-mapped файлы

Я освоил технологию отображаемых в память файлов (memory-mapped files), которая позволяет нескольким процессам эффективно обмениваться данными через общий файл, отображенный в их адресные пространства. Этот механизм обеспечивает высокую производительность, так как данные не требуется явно копировать между процессами - все изменения в отображенной области памяти автоматически становятся видны другим процессам.

Преимущества данного подхода:

- Высокая скорость обмена данными по сравнению с каналами (pipes) и сокетами
- Простота программирования - работа с файлом как с обычной памятью
- Эффективное использование памяти за счет механизма страниц ОС

Синхронизация процессов с помощью семафоров POSIX

Изучил и применил на практике именованные семафоры POSIX для организации синхронизации между процессами:

- **Мьютекс** - семафор `sem_file` использовался для защиты критической секции (доступа к файлу), предотвращая одновременную модификацию данных несколькими процессами
- **Барьеры синхронизации** - семафоры `sem_child` и `sem_parent` обеспечивали строгий порядок выполнения операций в дочерних и родительском процессах

Особое внимание было уделено избеганию ситуаций взаимной блокировки (deadlock) через правильный порядок захвата и освобождения семафоров.

Создание и управление процессами

Получил практические навыки работы с системными вызовами для управления процессами:

- `fork()` - создание дочерних процессов путем дублирования родительского
- `execl()` - замена образа процесса новой программой
- `waitpid()` - ожидание завершения дочернего процесса для предотвращения зомби-процессов

Приобретенные навыки

1. **Работа с POSIX API** - освоил функции для работы с разделяемой памятью (`shm_open`, `shm_unlink`), отображением файлов (`mmap`, `munmap`) и семафорами (`sem_open`, `sem_wait`, `sem_post`)
2. **Проектирование многопроцессных приложений** - научился разделять задачу на независимые процессы и организовывать их корректное взаимодействие
3. **Отладка системных программ** - использовал утилиты `strace` для трассировки системных вызовов и анализа поведения программы
4. **Управление ресурсами** - изучил важность правильного освобождения системных ресурсов (семафоров, разделяемой памяти, файловых дескрипторов) для предотвращения утечек

Практическое применение

Полученные знания и навыки применимы в следующих областях:

- Разработка высокопроизводительных серверных приложений с использованием модели многопроцессной обработки
- Создание систем с разделением привилегий (privilege separation) для повышения безопасности
- Проектирование распределенных систем обработки данных
- Оптимизация производительности через параллельную обработку на многоядерных процессорах

Заключение

Лабораторная работа позволила получить глубокое понимание механизмов межпроцессного взаимодействия и синхронизации в Unix-подобных ОС. Практический опыт работы с memory-mapped файлами и семафорами POSIX является фундаментальной основой для разработки эффективных многопроцессных приложений.

Особую ценность представляет понимание того, как различные механизмы IPC взаимодействуют с ядром операционной системы, что критически важно для написания надежного и производительного системного кода.

Листинг программы

Заголовочные файлы

shared_data.hpp - структура данных для разделяемой памяти

```cpp
#pragma once

#include <semaphore.h>

struct SharedData {
    int number;
    int signal;
    char file[256];
    sem_t* sem_parent;
    sem_t* sem_child;
    sem_t* sem_file;
};
```

os.hpp - интерфейсы функций для работы с ОС

```cpp
#pragma once

#include <semaphore.h>
#include <sys/types.h>
#include <cstddef>

#include "shared_data.hpp"

enum ProcessRole { IS_PARENT, IS_CHILD };

SharedData* CreateSharedMemory();
SharedData* OpenSharedMemory();
void DestroySharedMemory(SharedData* data);

ProcessRole ProcessCreate();
void ProcessExecute(const char* program, const char* arg);
pid_t ProcessWait(pid_t pid);

void SemaphorePost(sem_t* sem);
void SemaphoreWait(sem_t* sem);

char* MapFileContent(const char* filename, size_t size, int flags);
```

Реализация вспомогательных функций

os.cpp - реализация функций для работы с ОС

```cpp
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```cpp
#include <semaphore.h>
#include <cstring>
#include <sys/wait.h>

#include "os.hpp"

SharedData* CreateSharedMemory() {
    shm_unlink("/shared");
    int shm_fd = shm_open("/shared", O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) { perror("shm_open"); exit(1); }
    ftruncate(shm_fd, sizeof(SharedData));
    SharedData* data = (SharedData*)mmap(NULL, sizeof(SharedData),
                    PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (data == MAP_FAILED) { perror("mmap"); exit(1); }
    close(shm_fd);

    sem_unlink("/sem_file");
    sem_unlink("/sem_parent");
    sem_unlink("/sem_child");

    data->sem_file = sem_open("/sem_file", O_CREAT, 0666, 1);
    data->sem_parent = sem_open("/sem_parent", O_CREAT, 0666, 0);
    data->sem_child = sem_open("/sem_child", O_CREAT, 0666, 0);

    if (data->sem_file == SEM_FAILED ||
        data->sem_parent == SEM_FAILED ||
        data->sem_child == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }

    return data;
}

SharedData* OpenSharedMemory() {
    int shm_fd = shm_open("/shared", O_RDWR, 0);
    if (shm_fd == -1) { perror("shm_open"); exit(1); }
    SharedData* data = (SharedData*)mmap(NULL, sizeof(SharedData),
                    PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (data == MAP_FAILED) { perror("mmap"); exit(1); }
    close(shm_fd);

    data->sem_file = sem_open("/sem_file", 0);
    data->sem_parent = sem_open("/sem_parent", 0);
    data->sem_child = sem_open("/sem_child", 0);

    if (data->sem_file == SEM_FAILED ||
        data->sem_parent == SEM_FAILED ||
        data->sem_child == SEM_FAILED) {
        perror("sem_open in OpenSharedMemory");
        exit(1);
    }

    return data;
}

void DestroySharedMemory(SharedData* data) {
    sem_close(data->sem_file);
```

```
66      sem_close(data->sem_parent);
67      sem_close(data->sem_child);
68
69      sem_unlink("/sem_file");
70      sem_unlink("/sem_parent");
71      sem_unlink("/sem_child");
72
73      munmap(data, sizeof(SharedData));
74      shm_unlink("/shared");
75  }
76
77  ProcessRole ProcessCreate() {
78      pid_t pid = fork();
79      if (pid == -1) {
80          std::cout << "Ошибка созданияprocess" << std::endl;
81          exit(-1);
82      }
83      if (pid == 0) { return IS_CHILD; }
84      return IS_PARENT;
85  }
86
87  void ProcessExecute(const char* program, const char* arg) {
88      execl(program, arg, NULL);
89      perror("execl");
90      exit(1);
91  }
92
93  pid_t ProcessWait(pid_t pid) {
94      int status;
95      return waitpid(pid, &status, 0);
96  }
97
98  void SemaphorePost(sem_t* sem) {
99      if (sem_post(sem) == -1) { perror("sem_post"); exit(1); }
100 }
101
102 void SemaphoreWait(sem_t* sem) {
103     if (sem_wait(sem) == -1) { perror("sem_wait"); exit(1); }
104 }
105
106 char* MapFileContent(const char* filename, size_t size, int flags) {
107     int fd = open(filename, flags, 0666);
108     if (fd == -1) {
109         perror("open for file mapping");
110         return nullptr;
111     }
112
113     if (ftruncate(fd, size) == -1 && (flags & O_CREAT)) {
114         close(fd);
115         perror("ftruncate");
116         return nullptr;
117     }
118
119     char* mapped_addr = (char*)mmap(NULL, size,
120                         PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
121     close(fd);
122
123     if (mapped_addr == MAP_FAILED) {
```

```
        perror("mmap file content");
        return nullptr;
    }
    return mapped_addr;
}
```

Основные программы

parent.cpp - родительский процесс

```
#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <cstdlib>

#include "os.hpp"
#include "shared_data.hpp"

const size_t FILE_SIZE = 4096;
const char* DATA_FILENAME = "data_to_process.txt";

int main() {
    SharedData* control_data = CreateSharedMemory();
    strncpy(control_data->file, DATA_FILENAME,
            sizeof(control_data->file));
    control_data->file[sizeof(control_data->file) - 1] = '\0';

    char* file_content = MapFileContent(DATA_FILENAME, FILE_SIZE,
                                        O_CREAT | O_RDWR);
    if (!file_content) {
        DestroySharedMemory(control_data);
        return -1;
    }

    std::string input_string;
    std::cout << "Enter a string: ";
    std::getline(std::cin, input_string);

    strncpy(file_content, input_string.c_str(), FILE_SIZE - 1);
    file_content[FILE_SIZE - 1] = '\0';

    pid_t pid1 = 0;
    if (ProcessCreate() == IS_CHILD) {
        ProcessExecute("./child1", "child1");
    }
    pid1 = getpid();

    pid_t pid2 = 0;
    if (ProcessCreate() == IS_CHILD) {
        ProcessExecute("./child2", "child2");
    }
    pid2 = getpid();
```

```
47
48      SemaphorePost(control_data->sem_child);
49      SemaphoreWait(control_data->sem_parent);
50
51      std::cout << "Final output: " << file_content << std::endl;
52
53      munmap(file_content, FILE_SIZE);
54      unlink(DATA_FILENAME);
55
56      ProcessWait(pid1);
57      ProcessWait(pid2);
58
59      DestroySharedMemory(control_data);
60
61      return 0;
62  }
```

child1.cpp - первый дочерний процесс (преобразование в верхний регистр)

```
1   #include <cctype>
2   #include <cstring>
3   #include <unistd.h>
4   #include <sys/mman.h>
5   #include <fcntl.h>
6   #include <cstdlib>
7
8   #include "os.hpp"
9   #include "shared_data.hpp"
10
11  const size_t FILE_SIZE = 4096;
12
13  int main() {
14      SharedData* control_data = OpenSharedMemory();
15
16      SemaphoreWait(control_data->sem_child);
17      SemaphoreWait(control_data->sem_file);
18
19      char* file_content = MapFileContent(control_data->file,
20                                          FILE_SIZE, O_RDWR);
21      if (!file_content) {
22          SemaphorePost(control_data->sem_file);
23          return -1;
24      }
25
26      size_t len = strlen(file_content);
27      for (size_t i = 0; i < len; ++i) {
28          file_content[i] = std::toupper(
29              static_cast<unsigned char>(file_content[i]));
30      }
31
32      munmap(file_content, FILE_SIZE);
33      SemaphorePost(control_data->sem_file);
34      SemaphorePost(control_data->sem_child);
35
36      munmap(control_data, sizeof(SharedData));
37      return 0;
```

```
}
```

child2.cpp - второй дочерний процесс (замена пробелов на подчеркивания)

```
#include <cstring>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <cstdlib>

#include "os.hpp"
#include "shared_data.hpp"

const size_t FILE_SIZE = 4096;

int main() {
    SharedData* control_data = OpenSharedMemory();

    SemaphoreWait(control_data->sem_child);
    SemaphoreWait(control_data->sem_file);

    char* file_content = MapFileContent(control_data->file,
                                        FILE_SIZE, O_RDWR);
    if (!file_content) {
        SemaphorePost(control_data->sem_file);
        return -1;
    }

    size_t len = strlen(file_content);
    for (size_t i = 0; i < len; ++i) {
        if (file_content[i] == ' ') {
            file_content[i] = '_';
        }
    }

    munmap(file_content, FILE_SIZE);
    SemaphorePost(control_data->sem_file);
    SemaphorePost(control_data->sem_parent);

    munmap(control_data, sizeof(SharedData));
    return 0;
}
```