Отчёт по лабораторной работе

Взаимодействие клиента и сервера через разделяемую память POSIX

Студент

18 марта 2025 г.

Содержание

1 Введение

1.1 Цель работы

Целью данной лабораторной работы является изучение механизма разделяемой памяти POSIX для организации взаимодействия между процессами, а также реализация различных способов корректного завершения работы процессов.

1.2 Постановка задачи

Разработать программы клиента и сервера, взаимодействующих через разделяемую память с использованием функций POSIX. Клиент генерирует случайные числа, а сервер осуществляет их вывод. Необходимо обеспечить корректное завершение работы для одного клиента и одного сервера, при котором удаляется сегмент разделяемой памяти. Предложить и реализовать несколько вариантов корректного завершения.

2 Теоретическая часть

2.1 Разделяемая память POSIX

Разделяемая память POSIX представляет собой механизм межпроцессного взаимодействия (IPC), который позволяет различным процессам иметь доступ к общему участку памяти. В отличие от других средств IPC (таких как сокеты или каналы), разделяемая память обеспечивает наиболее быстрый обмен данными, так как данные не копируются между процессами. Основные функции для работы с разделяемой памятью POSIX:

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

2.2 Семафоры POSIX

Для синхронизации доступа к разделяемой памяти используются семафоры POSIX. Семафор представляет собой счетчик, с помощью которого можно контролировать доступ к общим ресурсам. Основные функции для работы с семафорами POSIX:

- `sem_open()` — создание/открытие семафора
- `sem_wait()` — захват семафора (блокирующая операция)
- `sem_post()` — освобождение семафора
- `sem_close()` — закрытие семафора
- `sem_unlink()` — удаление семафора
- `sem_timedwait()` — ожидание семафора с таймаутом
- `sem_getvalue()` — получение значения семафора

3 Практическая часть

3.1 Общая структура программы

Программное решение состоит из следующих компонентов:

- Заголовочный файл `shared_memory.h` с общими определениями
- Реализация клиента для трех вариантов завершения
- Реализация сервера для трех вариантов завершения
- `Makefile` для удобства сборки и запуска

Для обмена данными между клиентом и сервером используется следующая структура:

```
typedef struct {
    int number;          // Сгенерированное случайное число
    int ready;           // Флаг готовности (1 - число готово, 0 - нет)
    int terminate;       // Флаг завершения (1 - завершить работу, 0 - п
   родолжать)
    pid_t client_pid;    // PID клиента для варианта с сигналами
} SharedData;
```

Листинг 1: Структура данных для обмена

Общий алгоритм работы системы:

1. Сервер и клиент запускаются независимо.
2. Оба процесса подключаются к разделяемой памяти и инициализируют семафоры.
3. Клиент генерирует случайное число, записывает его в разделяемую память и устанавливает флаг `ready`.
4. Сервер считывает число из разделяемой памяти, выводит его и сбрасывает флаг `ready`.
5. Процесс повторяется до тех пор, пока не будет получен сигнал завершения.
6. При завершении работы одного из процессов освобождаются все ресурсы.

3.2 Варианты корректного завершения работы

3.2.1 Вариант 1: Завершение с использованием сигналов

В этом варианте для завершения используются сигналы операционной системы. Клиент записывает свой PID в разделяемую память. Когда сервер решает завершить работу, он отправляет клиенту сигнал `SIGTERM`. Преимущества:

- Простота реализации — использование стандартных механизмов ОС.
- Прямое уведомление процесса о необходимости завершения.

Недостатки:

- Требуется знать PID процесса для отправки сигнала.
- При неожиданном завершении одного процесса другой может не получить сигнал.

```c
// Отправляем сигнал клиенту для завершения
if (shared_data->client_pid > 0) {
    printf("Сервер: отправка сигнала завершения клиенту (PID:
           shared_data->client_pid);
    kill(shared_data->client_pid, SIGTERM);
}
```

Листинг 2: Фрагмент кода сервера (отправка сигнала клиенту)

```c
void signal_handler(int sig) {
    done = 1;
}
// ...
// Установка обработчика сигнала
struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sa.sa_handler = signal_handler;
sigaction(SIGTERM, &sa, NULL);
```

Листинг 3: Обработчик сигнала в клиенте

3.2.2 Вариант 2: Завершение с использованием флага в разделяемой памяти

В этом варианте в разделяемой памяти предусмотрен флаг `terminate`, который устанавливается процессом, который решает завершить работу. Другой процесс периодически проверяет этот флаг и также завершает работу при его установке. Преимущества:

- Не требуется знать PID процесса для уведомления о завершении.
- Работает даже если процессы запускаются не одновременно.

Недостатки:

- Требуется явная проверка флага завершения в цикле обработки.
- При неожиданном завершении одного процесса (например, при сбое) флаг может не быть установлен.

```c
// Проверка флага завершения
if (shared_data->terminate || done) {
    printf("Клиент: получен сигнал завершения\n");
    break;
}
```

Листинг 4: Проверка флага завершения

```c
// Установка флага завершения для другого процесса
sem_wait(sem_producer);
shared_data->terminate = 1;
sem_post(sem_consumer);
```

Листинг 5: Установка флага завершения

3.2.3 Вариант 3: Завершение с использованием дополнительного семафора

В этом варианте используется дополнительный семафор `sem_termination`. Когда один из процессов решает завершить работу, он увеличивает значение этого семафора. Другой процесс периодически проверяет значение семафора и завершает работу, когда оно становится больше нуля. Преимущества:

- Семафор сохраняется в системе даже после завершения процесса.
- Не требуется постоянного доступа к разделяемой памяти для проверки флага.
- Более надежный механизм при неожиданном завершении одного из процессов.

Недостатки:

- Требуется дополнительный семафор.
- Необходимость его явной проверки.

```
// Проверка семафора завершения без блокировки
int termination_value;
sem_getvalue(sem_termination, &termination_value);
if (termination_value > 0) {
    printf("Клиент: получен сигнал завершения через семафор\n");
    break;
}
```

Листинг 6: Проверка семафора завершения

```
// Сигнализируем о завершении через семафор
sem_post(sem_termination);
```

Листинг 7: Сигнализация о завершении через семафор

3.3 Сравнение вариантов завершения

Вариант	**Преимущества**	**Недостатки**
Сигналы	• Стандартный механизм ОС • Прямое уведомление	• Необходимость знать PID • Возможна потеря сигнала
Флаг в памяти	• Не требует PID • Универсальность	• Необходимость проверки • Проблемы при сбоях
Семафор	• Надежность • Сохраняется в системе	• Дополнительный ресурс • Явная проверка

Таблица 1: Сравнение вариантов завершения работы

4 Результаты работы

При запуске программы с использованием любого из трех вариантов мы получаем следующее поведение:

```
# Запуск сервера
$ ./server_signal
Сервер запущен. Нажмите Ctrl+C для завершения.
# Запуск клиента (в другом терминале)
$ ./client_signal
Клиент запущен (PID: 12345). Нажмите Ctrl+C для завершения.
Клиент: сгенерировано число 42
Сервер: получено число 42
Клиент: сгенерировано число 17
Сервер: получено число 17
...
# При нажатии Ctrl+C на сервере
Сервер: завершение работы...
Сервер: отправка сигнала завершения клиенту (PID: 12345)
Сервер: ресурсы освобождены
# В терминале клиента
Клиент: завершение работы...
```

Листинг 8: Пример вывода при выполнении (вариант с сигналами)

Аналогичное поведение наблюдается и при запуске с другими вариантами завершения, с небольшими отличиями в сообщениях.

5 Выводы

В ходе выполнения лабораторной работы были успешно разработаны программы клиента и сервера, взаимодействующие через разделяемую память POSIX. Основные результаты работы:

1. Реализован механизм обмена данными между процессами с использованием разделяемой памяти.
2. Обеспечена синхронизация доступа к разделяемой памяти с помощью семафоров.
3. Разработаны три различных варианта корректного завершения работы:
   - Завершение с использованием сигналов
   - Завершение с использованием флага в разделяемой памяти
   - Завершение с использованием дополнительного семафора
4. Проведено сравнение реализованных вариантов, выявлены их преимущества и недостатки.

Каждый из реализованных вариантов имеет свои особенности и может быть применен в зависимости от конкретных требований к системе. Вариант с сигналами является наиболее простым, но требует знания PID процесса. Вариант с флагом в разделяемой памяти более универсален, но требует постоянной проверки. Вариант с семафором является наиболее надежным, особенно в случае неожиданного завершения одного из процессов. В реальных системах часто используется комбинация этих подходов для обеспечения максимальной надежности и гибкости.

6 Приложение: Исходные коды программ

Полный исходный код программ, включая заголовочные файлы, реализации клиентов и серверов для всех трех вариантов, а также Makefile, доступен в отдельных файлах.