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-215Б-23

Студент: Лапенко К.А.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 27.02.24

Москва, 2024

**Постановка задачи**

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение 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.

**Вариант 14.**

Родительский процесс создает два дочерних процесса. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Child1 и Child2 можно «соединить» между собой дополнительным каналом. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child1 и child2 производят работу над строками. Child2 пересылает результат своей работы родительскому процессу. Родительский процесс полученный результат выводит в стандартный поток вывода.

14 вариант) Child1 переводит строки в нижний регистр. Child2 убирает все задвоенные пробелы.

**Общий метод и алгоритм решения**

**Использованные системные вызовы:**

* pid\_t fork(void); – Создает новый процесс путем копирования текущего процесса. Новый процесс называется дочерним, а исходный процесс называется родительским.
* \*\*int execl(const char *path, const char arg, ...);* – Заменяет текущий образ процесса новым образом, загружаемым из исполняемого файла, указанного в path. В вашем проекте используется для запуска дочерних процессов child1 и child2.
* \**int sem\_open(const char name, int oflag, ...);* – Открывает именованный семафор. Используется для создания и открытия семафоров, которые синхронизируют доступ к разделяемой памяти.
* \**int sem\_wait(sem\_t sem);* – Ожидает, пока значение семафора не станет больше нуля, и затем уменьшает его на единицу. Используется для синхронизации доступа к разделяемой памяти.
* \**int sem\_post(sem\_t sem);* – Увеличивает значение семафора на единицу. Используется для сигнализации о завершении работы с разделяемой памятью.
* \**int sem\_close(sem\_t sem);* – Закрывает семафор. Используется для освобождения ресурсов, связанных с семафором.
* \**int sem\_unlink(const char name);* – Удаляет именованный семафор. Используется для очистки семафоров после завершения работы программы.
* \**int open(const char pathname, int flags, ...);* – Открывает файл и возвращает файловый дескриптор. Используется для открытия файлов, которые будут использоваться как разделяемая память.
* \*\*void *mmap(void addr, size\_t length, int prot, int flags, int fd, off\_t offset);* – Отображает файл или устройство в память. Используется для создания отображения файлов в память, чтобы процессы могли обмениваться данными через разделяемую память.
* \**int munmap(void addr, size\_t length);* – Удаляет отображение памяти, созданное с помощью mmap. Используется для освобождения ресурсов, связанных с отображением памяти.
* \**int msync(void addr, size\_t length, int flags);* – Синхронизирует отображение памяти с файлом на диске. Используется для обеспечения согласованности данных между памятью и файлом.
* int close(int fd); – Закрывает файловый дескриптор. Используется для освобождения ресурсов, связанных с открытыми файлами.
* int ftruncate(int fd, off\_t length); – Изменяет размер файла до указанной длины. Используется для подготовки файлов для отображения в память.
* \**pid\_t waitpid(pid\_t pid, int status, int options);* – Ожидает завершения дочернего процесса с указанным идентификатором
* \**int unlink(const char pathname);* – Удаляет файл из файловой системы. Используется для удаления временных файлов, созданных для разделяемой памяти.

**Проект состоит из трех основных компонентов:**

1. Родительский процесс (parent.cpp) – Управляет созданием дочерних процессов, передает данные между процессами и собирает результаты.
2. Дочерний процесс 1 (child1.cpp) – Получает данные от родительского процесса, преобразует их (в нижний регистр) и передает дальше в дочерний процесс 2.
3. Дочерний процесс 2 (child2.cpp) – Получает данные от дочернего процесса 1, нормализует пробелы и передает результаты обратно в родительский процесс.

#### 1. Родительский процесс (parent.cpp)

Управляет всей работой программы. Создает дочерние процессы, передает данные и собирает результаты.

**Алгоритм работы:**

1. Удаляет старые семафоры (если они существуют) с помощью sem\_unlink.
2. Создает три семафора (sem1, sem2, sem3) для синхронизации доступа к разделяемой памяти.
3. Подготавливает три файла для отображения в память:
   * MAPPED\_FILE1 – для передачи данных от родителя к дочернему процессу 1.
   * MAPPED\_FILE2 – для передачи данных от дочернего процесса 1 к дочернему процессу 2.
   * MAPPED\_FILE3 – для передачи результатов от дочернего процесса 2 к родителю.
4. Открывает файлы и отображает их в память с помощью mmap.
5. Инициализирует разделяемую память:
   * Устанавливает начальные значения для структур SharedData и BufferedSharedData.
6. Создает два дочерних процесса с помощью fork:
   * Первый дочерний процесс запускает child1.
   * Второй дочерний процесс запускает child2.
7. Родительский процесс читает строки с ввода пользователя и передает их в дочерний процесс 1 через MAPPED\_FILE1.
8. После завершения ввода (Ctrl+D) родительский процесс сигнализирует child1 о завершении, устанавливая shared1->done = true.
9. Родительский процесс читает результаты из циклического буфера (MAPPED\_FILE3) с помощью функции readBufferedOutput.
10. Ожидает завершения дочерних процессов с помощью waitpid.
11. Выводит результаты на экран.
12. Освобождает ресурсы:
    * Удаляет отображение памяти с помощью munmap.
    * Закрывает файловые дескрипторы.
    * Закрывает и удаляет семафоры.

#### 2. Дочерний процесс 1 (child1.cpp):

Child1 получает данные от родительского процесса, преобразует их в нижний регистр и передает в child2.

**Алгоритм работы:**

1. Открывает семафоры sem1 и sem2.
2. Открывает файлы MAPPED\_FILE1 и MAPPED\_FILE2 и отображает их в память с помощью mmap.
3. Инициализирует разделяемую память shared2 (для передачи данных в дочерний процесс 2).
4. Входит в бесконечный цикл, где:
   * Проверяет наличие новых данных от родительского процесса через shared1.
   * Если данные есть, копирует их в локальный буфер и преобразует в нижний регистр.
   * Передает преобразованные данные в дочерний процесс 2 через shared2.
   * Если родительский процесс сигнализирует о завершении (shared1->done = true), передает сигнал в дочерний процесс 2 и завершает работу.
5. Освобождает ресурсы:
   * Удаляет отображение памяти.
   * Закрывает файловые дескрипторы.
   * Закрывает семафоры.

#### 3. Дочерний процесс 2 (child2.cpp):

Дочерний процесс 2 получает данные от дочернего процесса 1, удаляет лишние пробелы и табуляции и передает результаты обратно в родительский процесс.

**Алгоритм работы:**

1. Открывает семафоры sem2 и sem3.
2. Открывает файлы MAPPED\_FILE2 и MAPPED\_FILE3 и отображает их в память с помощью mmap.
3. Инициализирует разделяемую память shared3 (для передачи данных в родительский процесс).
4. Входит в бесконечный цикл, где:
   * Проверяет наличие новых данных от дочернего процесса 1 через shared2.
   * Если данные есть, копирует их в локальный буфер и удаляет лишние пробелы и табуляции.
   * Передает преобразованные данные в родительский процесс через циклический буфер в shared3.
   * Если дочерний процесс 1 сигнализирует о завершении (shared2->done = true), передает сигнал в родительский процесс и завершает работу.
5. Освобождает ресурсы:
   * Удаляет отображение памяти.
   * Закрывает файловые дескрипторы.
   * Закрывает семафоры.

### Общий алгоритм работы программы:

1. Родительский процесс создает семафоры и файлы для разделяемой памяти.
2. Родительский процесс создает два дочерних процесса:
   * child1 – преобразует строки в нижний регистр.
   * child2 – нормализует пробелы.
3. Родительский процесс передает строки в child1 через MAPPED\_FILE1.
4. child1 преобразует строки и передает их в child2 через MAPPED\_FILE2.
5. child2 нормализует пробелы и передает результаты в родительский процесс через MAPPED\_FILE3.
6. Родительский процесс собирает результаты и выводит их на экран.
7. Все процессы завершают работу, освобождая ресурсы.

**Код программы**

**parent.cpp**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

#include <sys/wait.h>

#include <semaphore.h>

#include <vector>

#include <string>

#include "common.h"

void prepareFileForMapping(const char\* filename, size\_t size) {

int fd = open(filename, O\_RDWR | O\_CREAT | O\_TRUNC, 0666);

if (fd == -1) {

perror("Error creating mapped file");

exit(EXIT\_FAILURE);

}

if (ftruncate(fd, size) == -1) {

perror("Error setting file size");

close(fd);

exit(EXIT\_FAILURE);

}

close(fd);

}

// Function to check and read from circular buffer

void readBufferedOutput(struct BufferedSharedData\* shared3, std::vector<std::string>& results, sem\_t\* sem3) {

sem\_wait(sem3);

// Read all available entries

while (shared3->entry\_count > 0) {

// Copy data from the current read position

shared3->buffers[shared3->read\_index].data[shared3->buffers[shared3->read\_index].size] = '\0';

results.push\_back(std::string(shared3->buffers[shared3->read\_index].data));

// Update read index and count

shared3->read\_index = (shared3->read\_index + 1) % BUFFER\_COUNT;

shared3->entry\_count--;

}

msync(shared3, sizeof(struct BufferedSharedData), MS\_SYNC);

sem\_post(sem3);

}

int main() {

// Clean up any existing semaphores

sem\_unlink(SEM\_NAME1);

sem\_unlink(SEM\_NAME2);

sem\_unlink(SEM\_NAME3);

// Create semaphores

sem\_t \*sem1 = sem\_open(SEM\_NAME1, O\_CREAT | O\_EXCL, 0666, 1);

sem\_t \*sem2 = sem\_open(SEM\_NAME2, O\_CREAT | O\_EXCL, 0666, 1);

sem\_t \*sem3 = sem\_open(SEM\_NAME3, O\_CREAT | O\_EXCL, 0666, 1);

if (sem1 == SEM\_FAILED || sem2 == SEM\_FAILED || sem3 == SEM\_FAILED) {

perror("Error creating semaphores");

exit(EXIT\_FAILURE);

}

// Prepare memory mapped files

prepareFileForMapping(MAPPED\_FILE1, sizeof(struct SharedData));

prepareFileForMapping(MAPPED\_FILE2, sizeof(struct SharedData));

prepareFileForMapping(MAPPED\_FILE3, sizeof(struct BufferedSharedData));

// Open the memory mapped files

int fd1 = open(MAPPED\_FILE1, O\_RDWR);

int fd3 = open(MAPPED\_FILE3, O\_RDWR);

if (fd1 == -1 || fd3 == -1) {

perror("Error opening mapped files");

exit(EXIT\_FAILURE);

}

// Map the files into memory

struct SharedData\* shared1 = (struct SharedData\*)mmap(NULL, sizeof(struct SharedData),

PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd1, 0);

struct BufferedSharedData\* shared3 = (struct BufferedSharedData\*)mmap(NULL, sizeof(struct BufferedSharedData),

PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd3, 0);

if (shared1 == MAP\_FAILED || shared3 == MAP\_FAILED) {

perror("Error mapping files");

exit(EXIT\_FAILURE);

}

// Initialize shared memory

shared1->size = 0;

shared1->done = false;

shared3->write\_index = 0;

shared3->read\_index = 0;

shared3->entry\_count = 0;

shared3->done = false;

// Create first child process

pid\_t child1 = fork();

if (child1 == -1) {

perror("Error creating first child process");

exit(EXIT\_FAILURE);

}

if (child1 == 0) {

// First child process

execl("./child1", "child1", NULL);

perror("Error executing child1");

exit(EXIT\_FAILURE);

}

// Create second child process

pid\_t child2 = fork();

if (child2 == -1) {

perror("Error creating second child process");

exit(EXIT\_FAILURE);

}

if (child2 == 0) {

// Second child process

execl("./child2", "child2", NULL);

perror("Error executing child2");

exit(EXIT\_FAILURE);

}

// Parent process continues here

printf("Введите строки (Ctrl+D для завершения):\n");

// Store all results for later printing

std::vector<std::string> results;

char line[MAX\_LINE];

while (fgets(line, MAX\_LINE, stdin) != NULL) {

// Send data to child1 through first shared memory

sem\_wait(sem1);

size\_t len = strlen(line);

strncpy(shared1->data, line, len);

shared1->size = len;

msync(shared1, sizeof(struct SharedData), MS\_SYNC);

sem\_post(sem1);

// readBufferedOutput(shared3, results, sem3);

}

// Signal end of input

sem\_wait(sem1);

shared1->done = true;

msync(shared1, sizeof(struct SharedData), MS\_SYNC);

sem\_post(sem1);

// Check for remaining output

readBufferedOutput(shared3, results, sem3);

// Wait for child processes to complete

waitpid(child1, NULL, 0);

waitpid(child2, NULL, 0);

// Print all collected results

printf("Преобразованный текст:\n");

for (const auto& result : results) {

printf("%s", result.c\_str());

}

// Clean up

munmap(shared1, sizeof(struct SharedData));

munmap(shared3, sizeof(struct BufferedSharedData));

close(fd1);

close(fd3);

sem\_close(sem1);

sem\_close(sem2);

sem\_close(sem3);

sem\_unlink(SEM\_NAME1);

sem\_unlink(SEM\_NAME2);

sem\_unlink(SEM\_NAME3);

unlink(MAPPED\_FILE1);

unlink(MAPPED\_FILE2);

unlink(MAPPED\_FILE3);

printf("\nВсе процессы завершены.\n");

return 0;

}

**child1.cpp**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

#include <semaphore.h>

#include "common.h"

int main() {

// Open semaphores

sem\_t \*sem1 = sem\_open(SEM\_NAME1, 0);

sem\_t \*sem2 = sem\_open(SEM\_NAME2, 0);

if (sem1 == SEM\_FAILED || sem2 == SEM\_FAILED) {

perror("Error opening semaphores");

return EXIT\_FAILURE;

}

// Open first mapped file (for input from parent)

int fd1 = open(MAPPED\_FILE1, O\_RDWR);

if (fd1 == -1) {

perror("Error opening first mapped file");

return EXIT\_FAILURE;

}

// Open second mapped file (for output to child2)

int fd2 = open(MAPPED\_FILE2, O\_RDWR);

if (fd2 == -1) {

perror("Error opening second mapped file");

close(fd1);

return EXIT\_FAILURE;

}

// Map the files into memory

struct SharedData\* shared1 = (struct SharedData\*)mmap(NULL, sizeof(struct SharedData),

PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd1, 0);

struct SharedData\* shared2 = (struct SharedData\*)mmap(NULL, sizeof(struct SharedData),

PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd2, 0);

if (shared1 == MAP\_FAILED || shared2 == MAP\_FAILED) {

perror("Error mapping files");

close(fd1);

close(fd2);

return EXIT\_FAILURE;

}

// Initialize the second shared memory

shared2->size = 0;

shared2->done = false;

msync(shared2, sizeof(struct SharedData), MS\_SYNC);

char buffer[MAX\_LINE];

// Process data until parent signals completion

while (true) {

bool is\_done = false;

bool has\_data = false;

// Check for new input data

sem\_wait(sem1);

if (shared1->size > 0) {

// Read data from parent

strncpy(buffer, shared1->data, shared1->size);

buffer[shared1->size] = '\0';

shared1->size = 0; // Mark as read

msync(shared1, sizeof(struct SharedData), MS\_SYNC);

has\_data = true;

}

is\_done = shared1->done;

sem\_post(sem1);

// Process the data if we got any

if (has\_data) {

// Convert to lowercase

for (size\_t i = 0; buffer[i]; i++) {

buffer[i] = tolower(buffer[i]);

}

// Send processed data to child2

sem\_wait(sem2);

size\_t len = strlen(buffer);

strncpy(shared2->data, buffer, len);

shared2->size = len;

msync(shared2, sizeof(struct SharedData), MS\_SYNC);

sem\_post(sem2);

}

// Check if we're done

if (is\_done && !has\_data) {

// Signal child2 that we're done

sem\_wait(sem2);

shared2->done = true;

msync(shared2, sizeof(struct SharedData), MS\_SYNC);

sem\_post(sem2);

break;

}

}

// Clean up

munmap(shared1, sizeof(struct SharedData));

munmap(shared2, sizeof(struct SharedData));

close(fd1);

close(fd2);

sem\_close(sem1);

sem\_close(sem2);

return 0;

}

**child2.cpp**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

#include <semaphore.h>

#include "common.h"

int main() {

// Open semaphores

sem\_t \*sem2 = sem\_open(SEM\_NAME2, 0);

sem\_t \*sem3 = sem\_open(SEM\_NAME3, 0);

if (sem2 == SEM\_FAILED || sem3 == SEM\_FAILED) {

perror("Error opening semaphores");

return EXIT\_FAILURE;

}

// Open second mapped file (for input from child1)

int fd2 = open(MAPPED\_FILE2, O\_RDWR);

if (fd2 == -1) {

perror("Error opening mapped file");

return EXIT\_FAILURE;

}

// Open third mapped file (for output to parent)

int fd3 = open(MAPPED\_FILE3, O\_RDWR);

if (fd3 == -1) {

perror("Error opening mapped file");

close(fd2);

return EXIT\_FAILURE;

}

// Map the files into memory

struct SharedData\* shared2 = (struct SharedData\*)mmap(NULL, sizeof(struct SharedData),

PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd2, 0);

struct BufferedSharedData\* shared3 = (struct BufferedSharedData\*)mmap(NULL, sizeof(struct BufferedSharedData),

PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd3, 0);

if (shared2 == MAP\_FAILED || shared3 == MAP\_FAILED) {

perror("Error mapping files");

close(fd2);

close(fd3);

return EXIT\_FAILURE;

}

// Initialize shared memory for output to parent

sem\_wait(sem3);

shared3->write\_index = 0;

shared3->read\_index = 0;

shared3->entry\_count = 0;

shared3->done = false;

msync(shared3, sizeof(struct BufferedSharedData), MS\_SYNC);

sem\_post(sem3);

char buffer[MAX\_LINE];

char output[MAX\_LINE];

// Process data until child1 signals completion

while (true) {

bool is\_done = false;

bool has\_data = false;

// Check for new input data from child1

sem\_wait(sem2);

if (shared2->size > 0) {

// Read data from child1

strncpy(buffer, shared2->data, shared2->size);

buffer[shared2->size] = '\0';

shared2->size = 0; // Mark as read

msync(shared2, sizeof(struct SharedData), MS\_SYNC);

has\_data = true;

}

is\_done = shared2->done;

sem\_post(sem2);

// Process the data if we got any

if (has\_data) {

// Normalize whitespace

bool last\_was\_space = false;

size\_t j = 0;

for (size\_t i = 0; buffer[i]; i++) {

if (buffer[i] == ' ' || buffer[i] == '\t') {

if (!last\_was\_space) {

output[j++] = ' ';

last\_was\_space = true;

}

} else {

output[j++] = buffer[i];

last\_was\_space = false;

}

}

output[j] = '\0';

// Send processed data to parent using circular buffer

sem\_wait(sem3);

// // Wait if buffer is full

// while (shared3->entry\_count >= BUFFER\_COUNT) {

// sem\_post(sem3);

// usleep(1000); // Brief wait to avoid spin-lock

// sem\_wait(sem3);

// }

// Add to circular buffer

size\_t len = strlen(output);

strncpy(shared3->buffers[shared3->write\_index].data, output, len);

shared3->buffers[shared3->write\_index].size = len;

// Update write index and count

shared3->write\_index = (shared3->write\_index + 1) % BUFFER\_COUNT;

shared3->entry\_count++;

msync(shared3, sizeof(struct BufferedSharedData), MS\_SYNC);

sem\_post(sem3);

}

// Check if we're done

if (is\_done && !has\_data) {

// Signal parent that we're done

sem\_wait(sem3);

shared3->done = true;

msync(shared3, sizeof(struct BufferedSharedData), MS\_SYNC);

sem\_post(sem3);

break;

}

}

// Clean up

munmap(shared2, sizeof(struct SharedData));

munmap(shared3, sizeof(struct BufferedSharedData));

close(fd2);

close(fd3);

sem\_close(sem2);

sem\_close(sem3);

return 0;

}

**common.h**

#ifndef COMMON\_H

#define COMMON\_H

#define MAX\_LINE 1024

#define SHARED\_MEM\_SIZE (MAX\_LINE \* 100)

#define BUFFER\_COUNT 30 // Number of buffer entries in mmf3

#define MAPPED\_FILE1 "/tmp/mapped\_file1"

#define MAPPED\_FILE2 "/tmp/mapped\_file2"

#define MAPPED\_FILE3 "/tmp/mapped\_file3" // For output back to parent

#define SEM\_NAME1 "/sem1\_lab"

#define SEM\_NAME2 "/sem2\_lab"

#define SEM\_NAME3 "/sem3\_lab"

// Standard SharedData for mmf1 and mmf2

struct SharedData {

char data[SHARED\_MEM\_SIZE];

size\_t size;

bool done;

};

// Enhanced SharedData for mmf3 with circular buffer

struct BufferedSharedData {

struct {

char data[MAX\_LINE];

size\_t size;

} buffers[BUFFER\_COUNT];

int write\_index; // Where child2 will write next

int read\_index; // Where parent will read next

int entry\_count; // Current number of entries

bool done;

};

#endif

**Протокол работы программы**

**Strace:**

execve("./parent", ["./parent"], 0xfffff6fdd080 /\* 11 vars \*/) = 0

brk(NULL) = 0xaaaace4e0000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d93c000

faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=12067, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 12067, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xffff8d939000

close(3) = 0

openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libstdc++.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=2190752, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 2332704, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d6ce000

mmap(0xffff8d6d0000, 2267168, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xffff8d6d0000

munmap(0xffff8d6ce000, 8192) = 0

munmap(0xffff8d8fa000, 55328) = 0

mprotect(0xffff8d8da000, 61440, PROT\_NONE) = 0

mmap(0xffff8d8e9000, 57344, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x209000) = 0xffff8d8e9000

mmap(0xffff8d8f7000, 10272, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xffff8d8f7000

close(3) = 0

openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libgcc\_s.so.1", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=84296, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 213704, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d69b000

mmap(0xffff8d6a0000, 148168, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xffff8d6a0000

munmap(0xffff8d69b000, 20480) = 0

munmap(0xffff8d6c5000, 41672) = 0

mprotect(0xffff8d6b4000, 61440, PROT\_NONE) = 0

mmap(0xffff8d6c3000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x13000) = 0xffff8d6c3000

close(3) = 0

openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\340u\2\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0755, st\_size=1637400, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 1805928, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d4e7000

mmap(0xffff8d4f0000, 1740392, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xffff8d4f0000

munmap(0xffff8d4e7000, 36864) = 0

munmap(0xffff8d699000, 28264) = 0

mprotect(0xffff8d678000, 61440, PROT\_NONE) = 0

mmap(0xffff8d687000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x187000) = 0xffff8d687000

mmap(0xffff8d68d000, 48744, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xffff8d68d000

close(3) = 0

openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libm.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=551064, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 680048, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d449000

mmap(0xffff8d450000, 614512, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xffff8d450000

munmap(0xffff8d449000, 28672) = 0

munmap(0xffff8d4e7000, 32880) = 0

mprotect(0xffff8d4d6000, 61440, PROT\_NONE) = 0

mmap(0xffff8d4e5000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x85000) = 0xffff8d4e5000

close(3) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d937000

set\_tid\_address(0xffff8d937af0) = 66

set\_robust\_list(0xffff8d937b00, 24) = 0

rseq(0xffff8d9381c0, 0x20, 0, 0xd428bc00) = 0

mprotect(0xffff8d687000, 16384, PROT\_READ) = 0

mprotect(0xffff8d4e5000, 4096, PROT\_READ) = 0

mprotect(0xffff8d6c3000, 4096, PROT\_READ) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8d935000

mprotect(0xffff8d8e9000, 45056, PROT\_READ) = 0

mprotect(0xaaaaaecc4000, 4096, PROT\_READ) = 0

mprotect(0xffff8d942000, 8192, PROT\_READ) = 0

prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0

munmap(0xffff8d939000, 12067) = 0

getrandom("\x97\x45\xa8\xd6\x01\xf0\x0a\x99", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0xaaaace4e0000

brk(0xaaaace501000) = 0xaaaace501000

unlinkat(AT\_FDCWD, "/dev/shm/sem.sem1\_lab", 0) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.sem2\_lab", 0) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.sem3\_lab", 0) = 0

getrandom("\x45\x8b\x89\x57\xdd\x45\x66\x0c", 8, GRND\_NONBLOCK) = 8

newfstatat(AT\_FDCWD, "/dev/shm/sem.vcjCPj", 0xffffc352fae8, AT\_SYMLINK\_NOFOLLOW) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/dev/shm/sem.vcjCPj", O\_RDWR|O\_CREAT|O\_EXCL, 0666) = 3

write(3, "\1\0\0\0\0\0\0\0\200\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 32) = 32

mmap(NULL, 32, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 3, 0) = 0xffff8d93b000

linkat(AT\_FDCWD, "/dev/shm/sem.vcjCPj", AT\_FDCWD, "/dev/shm/sem.sem1\_lab", 0) = 0

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=32, ...}, AT\_EMPTY\_PATH) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.vcjCPj", 0) = 0

close(3) = 0

getrandom("\x26\xfe\x41\x72\x41\x05\xf8\xd4", 8, GRND\_NONBLOCK) = 8

newfstatat(AT\_FDCWD, "/dev/shm/sem.GxZT9A", 0xffffc352fae8, AT\_SYMLINK\_NOFOLLOW) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/dev/shm/sem.GxZT9A", O\_RDWR|O\_CREAT|O\_EXCL, 0666) = 3

write(3, "\1\0\0\0\0\0\0\0\200\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 32) = 32

mmap(NULL, 32, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 3, 0) = 0xffff8d93a000

linkat(AT\_FDCWD, "/dev/shm/sem.GxZT9A", AT\_FDCWD, "/dev/shm/sem.sem2\_lab", 0) = 0

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=32, ...}, AT\_EMPTY\_PATH) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.GxZT9A", 0) = 0

close(3) = 0

getrandom("\x14\xea\x81\x30\x8e\xd5\xbf\x96", 8, GRND\_NONBLOCK) = 8

newfstatat(AT\_FDCWD, "/dev/shm/sem.Oc8k92", 0xffffc352fae8, AT\_SYMLINK\_NOFOLLOW) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/dev/shm/sem.Oc8k92", O\_RDWR|O\_CREAT|O\_EXCL, 0666) = 3

write(3, "\1\0\0\0\0\0\0\0\200\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 32) = 32

mmap(NULL, 32, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 3, 0) = 0xffff8d939000

linkat(AT\_FDCWD, "/dev/shm/sem.Oc8k92", AT\_FDCWD, "/dev/shm/sem.sem3\_lab", 0) = 0

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=32, ...}, AT\_EMPTY\_PATH) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.Oc8k92", 0) = 0

close(3) = 0

openat(AT\_FDCWD, "/tmp/mapped\_file1", O\_RDWR|O\_CREAT|O\_TRUNC, 0666) = 3

ftruncate(3, 102416) = 0

close(3) = 0

openat(AT\_FDCWD, "/tmp/mapped\_file2", O\_RDWR|O\_CREAT|O\_TRUNC, 0666) = 3

ftruncate(3, 102416) = 0

close(3) = 0

openat(AT\_FDCWD, "/tmp/mapped\_file3", O\_RDWR|O\_CREAT|O\_TRUNC, 0666) = 3

ftruncate(3, 30976) = 0mmap(NULL

close(3) = 0

openat(AT\_FDCWD, "/tmp/mapped\_file1", O\_RDWR) = 3

openat(AT\_FDCWD, "/tmp/mapped\_file3", O\_RDWR) = 4

mmap(NULL, 102416, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 3, 0) = 0xffff8d436000

mmap(NULL, 30976, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 4, 0) = 0xffff8d900000

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0xffff8d937af0) = 67

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0xffff8d937af0) = 68

newfstatat(1, "", {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}, AT\_EMPTY\_PATH) = 0

write(1, "\320\222\320\262\320\265\320\264\320\270\321\202\320\265 \321\201\321\202\321\200\320\276\320\272\320\270 (Ctr"..., 66) = 66

newfstatat(0, "", {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}, AT\_EMPTY\_PATH) = 0

read(0, "dkenwfjkJDWNJKQD djwqnd dwJK"..., 1024) = 63

msync(0xffff8d436000, 102416, MS\_SYNC) = 0

futex(0xffff8d93b000, FUTEX\_WAKE, 1) = 1

read(0, "dkawnd awJKNDJKDNJWKDND JDWANKD "..., 1024) = 55

futex(0xffff8d93b000, FUTEX\_WAIT\_BITSET|FUTEX\_CLOCK\_REALTIME, 0, NULL, FUTEX\_BITSET\_MATCH\_ANY) = 0

msync(0xffff8d436000, 102416, MS\_SYNC) = 0

futex(0xffff8d93b000, FUTEX\_WAKE, 1) = 1

read(0, "", 1024) = 0

msync(0xffff8d436000, 102416, MS\_SYNC) = 0

futex(0xffff8d93b000, FUTEX\_WAKE, 1) = 1

msync(0xffff8d900000, 30976, MS\_SYNC) = 0

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=67, si\_uid=0, si\_status=0, si\_utime=1800, si\_stime=0} ---

futex(0xffff8d939000, FUTEX\_WAKE, 1) = 1

wait4(67, NULL, 0, NULL) = 67

wait4(68, NULL, 0, NULL) = 68

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=68, si\_uid=0, si\_status=0, si\_utime=1801, si\_stime=0} ---

write(1, "\320\237\321\200\320\265\320\276\320\261\321\200\320\260\320\267\320\276\320\262\320\260\320\275\320\275\321\213\320\271 \321"..., 43) = 43

write(1, "dkenwfjkjdwnjkqd djwqnd dwjknwdq"..., 57) = 57

write(1, "dkawnd awjkndjkdnjwkdnd jdwankd "..., 49) = 49

munmap(0xffff8d436000, 102416) = 0

munmap(0xffff8d900000, 30976) = 0

close(3) = 0

close(4) = 0

munmap(0xffff8d93b000, 32) = 0

munmap(0xffff8d93a000, 32) = 0

munmap(0xffff8d939000, 32) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.sem1\_lab", 0) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.sem2\_lab", 0) = 0

unlinkat(AT\_FDCWD, "/dev/shm/sem.sem3\_lab", 0) = 0

unlinkat(AT\_FDCWD, "/tmp/mapped\_file1", 0) = 0

unlinkat(AT\_FDCWD, "/tmp/mapped\_file2", 0) = 0

unlinkat(AT\_FDCWD, "/tmp/mapped\_file3", 0) = 0

write(1, "\n", 1) = 1

write(1, "\320\222\321\201\320\265 \320\277\321\200\320\276\321\206\320\265\321\201\321\201\321\213 \320\267\320\260\320\262\320\265"..., 44) = 44

exit\_group(0) = ?

+++ exited with 0 +++

**Вывод**

Было интересно разобраться с memory-mapped files. Были получены практические навыки в освоение принципов работы с файловыми системами, а также в обеспечении обмена данных между процессами посредством технологии «File mapping».