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: Бабицкий И.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 26.10.25

Москва, 2025

Постановка задачи

Вариант 22.

Переделать лабораторную работу №1 с использованием `shared memory` и `memory mapping`.

Так как блокирующего чтения из каналов больше не будет, то для синхронизации чтения и записи из `shared memory` нужно использовать семафор.

Системные функции для **Windows**: `CreateFileMapping`, `MapViewOfFile`, `UnmapViewOfFile`, `OpenSemaphore`, `WaitForSingleObject`, `ReleaseSemaphore`.

Системные функции для **Linux**: `shm_open`, `shm_unlink`, `ftruncate`, `mmap`, `munmap`, `sem_open`, `sem_wait`, `sem_post`, `sem_unlink`, `sem_close`.

`shared memory` и `memory mapping` — это не одно и то же. В этой лабораторной работе нужно создать именованный `shared memory` объект, который будет находиться в оперативной памяти, а не на диске в виде файлика.

Поскольку нужно создавать именованные `shared memory` и семафоры, то их название должно отличаться для экземпляров пар программ `server` и `client`. Не нужно делать название именованных `shared memory` и семафоров константным.

Общий метод и алгоритм решения

Использованные системные вызовы:

- `int shm_open(const char *name, int oflag, mode_t mode)` — открывает или создаёт объект разделяемой памяти, возвращает файловый дескриптор при успехе или `-1` при ошибке;
- `int shm_unlink(const char *name)` — удаляет объект разделяемой памяти, возвращает `0` при успехе или `-1` при ошибке;
- `int ftruncate(int fd, off_t length)` — устанавливает размер файла или объекта разделяемой памяти, возвращает `0` при успехе или `-1` при ошибке;
- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)` — отображает файл или объект разделяемой памяти в виртуальную память процесса, возвращает адрес области отображения при успехе или `MAP_FAILED` при ошибке;
- `int munmap(void *addr, size_t length)` — удаляет отображение памяти, возвращает `0` при успехе или `-1` при ошибке;
- `sem_t *sem_open(const char *name, int oflag, ...)` — открывает или создаёт именованный семафор, возвращает адрес семафора при успехе или `SEM_FAILED` при ошибке;
- `int sem_wait(sem_t *sem)` — ожидает семафор (уменьшает значение на `1`), возвращает `0` при успехе или `-1` при ошибке;
- `int sem_post(sem_t *sem)` — освобождает семафор (увеличивает значение на `1`), возвращает `0` при успехе или `-1` при ошибке;
- `int sem_unlink(const char *name)` — удаляет именованный семафор, возвращает `0` при успехе или `-1` при ошибке;
- `int sem_close(sem_t *sem)` — закрывает семафор, возвращает `0` при успехе или `-1` при ошибке;

- `pid_t fork()` — создаёт копию текущего процесса, возвращает `0` в дочернем или `PID` потока в родительском процессе при успехе или `-1` при ошибке;
- `int execv(const char *filename, char *const argv[])` — заменяет текущий процесс новым, управление не возвращается при успехе или возвращает `-1` при ошибке;
- `void exit(int status)` — завершает выполнение текущего процесса;
- `int pipe(int pipefd[2])` — создать неименованый канал для передачи данных между процессами, возвращает `0` при успехе или `-1` при ошибке;
- `pid_t waitpid(pid_t pid, int *status, int options)` — ожидает завершения конкретного дочернего процесса, возвращает его `PID` при успехе или `-1` при ошибке;
- `int open(const char *pathname, int flags, mode_t mode)` — открывает или создаёт файл, возвращает файловый дескриптор при успехе или `-1` при ошибке;
- `int close(int fd)` — закрывает файловый дескриптор, возвращает `0` при успехе или `-1` при ошибке;
- `ssize_t write(int fd, const void *buf, size_t count)` — записывает данные в файловый дескриптор, возвращает количество записанных байт при успехе или `-1` при ошибке;
- `ssize_t read(int fd, void *buf, size_t count)` — читает данные из файлового дескриптора, возвращает количество прочитанных байт при успехе или `-1` при ошибке.

Я изменил программу лабораторной работы №1 на языке **C**: для передачи данных между процессами вместо не именованных каналов использовал `shared memory` и `memory mapping`, а также семафор для синхронизации чтения из `shared memory` и записи в неё.

1. Родительский процесс создаёт два канала для хранения информации об именованной разделяемой памяти и связанных с ней семафоров: файловый дескриптор разделяемой памяти и её имя, адрес отображаемого в адресном пространстве процесса объекта разделяемой памяти, семафоры и их имена.
2. Родительский процесс создаёт два дочерних процесса и запускает в них программу для обработки данных из разделяемой памяти, передавая им имена объектов разделяемой памяти и семафоров.
3. Родительский процесс считывает данные из стандартного потока ввода, записывает их в разделяемую память и помощью семафоров сигнализирует дочерним процессам о их готовности.
4. Дочерние процессы ожидают появления данных в разделяемой памяти, обрабатывают их, записывают результаты в выходные файлы и с помощью семафоров сигнализируют родительскому процессу о выполнении обработки.
5. По окончании ввода данных родительский процесс сигнализирует дочерним процессам об этом, ожидает их завершения и освобождает все захваченные ресурсы, включая объекты разделяемой памяти и семафоры. Дочерние процессы со своей стороны также освобождают все захваченные ресурсы.

Код программы

utilities.h

```
#ifndef UTILITIES_H
#define UTILITIES_H

#define GET_FILE_NAME_SUCCESS  0
#define GET_FILE_NAME_FAILURE -1
#define MIN_FILE_NAME_LENGTH   2

void printError(const char *message);

void print(const char *message);

int getFileName(char *fileName, int fileNumber);

#endif
```

utilities.c

```
#include "../include/utilities.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void printError(const char *message) {
  write(STDERR_FILENO, message, strlen(message));
}

void print(const char *message) {
  write(STDOUT_FILENO, message, strlen(message));
}

int getFileName(char *fileName, int fileNumber) {
  char message[BUFSIZ];
  int messageLen = snprintf(message, BUFSIZ, "Enter the name of the child file #%d: ",
                            fileNumber);
  write(STDOUT_FILENO, message, messageLen);

  ssize_t fileNameLen = read(STDIN_FILENO, fileName, BUFSIZ - 1);
  if (fileNameLen < MIN_FILE_NAME_LENGTH) {
    printError("Invalid name of the file.");
    return GET_FILE_NAME_FAILURE;
  }
  fileName[fileNameLen - 1] = '\0';
  return GET_FILE_NAME_SUCCESS;
}
```

shm.h

```
#ifndef SHM_H
```

```
#define SHM_H

#include <unistd.h>
#include <stdio.h>

typedef struct {
  ssize_t len;
  char data[BUFSIZ];
  int finished;
} shm_buffer_t;

#endif
```

parent.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <string.h>
#include <time.h>
#include "../include/shm.h"
#include "../include/utilities.h"

#define FORK_FAILURE   -1
#define FORK_CHILD      0
#define IPC_NAME_LEN   64
#define CHILDREN_COUNT  2

typedef struct {
  int shm_fd;
  shm_buffer_t *buf;
  sem_t *sem_w;
  sem_t *sem_r;
  char shm_name[IPC_NAME_LEN];
  char sem_w_name[IPC_NAME_LEN];
  char sem_r_name[IPC_NAME_LEN];
} channel_t;

void initChannel(channel_t *ch, size_t i, pid_t pid) {
  snprintf(ch->shm_name, IPC_NAME_LEN, "/shm_%d_%zu", pid, i);
  snprintf(ch->sem_w_name, IPC_NAME_LEN, "/sem_w_%d_%zu", pid, i);
  snprintf(ch->sem_r_name, IPC_NAME_LEN, "/sem_r_%d_%zu", pid, i);

  ch->shm_fd = shm_open(ch->shm_name, O_CREAT | O_RDWR, 0600);

  ftruncate(ch->shm_fd, sizeof(shm_buffer_t));
```

```
  ch->buf = mmap(NULL, sizeof(shm_buffer_t), PROT_READ | PROT_WRITE, MAP_SHARED,
                 ch->shm_fd, 0);

  ch->sem_w = sem_open(ch->sem_w_name, O_CREAT, 0600, 1);
  ch->sem_r = sem_open(ch->sem_r_name, O_CREAT, 0600, 0);

  ch->buf->finished = 0;
}

int main() {
  srand(time(NULL));
  pid_t pid = getpid();

  channel_t channels[CHILDREN_COUNT];
  char fileNames[CHILDREN_COUNT][BUFSIZ];
  for (size_t i = 0; i != CHILDREN_COUNT; ++i) {
    if (getFileName(fileNames[i], i + 1) == GET_FILE_NAME_FAILURE) {
      printError("Invalid file name.");
      exit(EXIT_FAILURE);
    }
    initChannel(&channels[i], i, pid);
  }

  const char *path = "./child";
  const char *program = "child";
  for (size_t i = 0; i != CHILDREN_COUNT; ++i) {
    switch (fork()) {
      case FORK_FAILURE:
        printError("Fork failed.");
        exit(EXIT_FAILURE);
      case FORK_CHILD:
        execl(path, program, channels[i].shm_name, channels[i].sem_w_name,
              channels[i].sem_r_name, fileNames[i], NULL);
        exit(EXIT_FAILURE);
    }
  }

  char buf[BUFSIZ];
  ssize_t len;
  while ((len = read(STDIN_FILENO, buf, BUFSIZ)) > 0) {
    size_t i = (rand() % 100 < 80) ? 0 : 1;
    channel_t *ch = &channels[i];

    sem_wait(ch->sem_w);

    memcpy(ch->buf->data, buf, len);
    ch->buf->len = len;
```

```
    sem_post(ch->sem_r);
  }

  for (size_t i = 0; i != CHILDREN_COUNT; ++i) {
    sem_wait(channels[i].sem_w);
    channels[i].buf->finished = 1;
    sem_post(channels[i].sem_r);
  }

  for (size_t i = 0; i != CHILDREN_COUNT; ++i) {
    wait(NULL);
  }

  for (size_t i = 0; i != CHILDREN_COUNT; ++i) {
    munmap(channels[i].buf, sizeof(shm_buffer_t));
    close(channels[i].shm_fd);

    sem_close(channels[i].sem_w);
    sem_close(channels[i].sem_r);

    shm_unlink(channels[i].shm_name);
    sem_unlink(channels[i].sem_w_name);
    sem_unlink(channels[i].sem_r_name);
  }

  return 0;
}
```

child.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <string.h>
#include "../include/shm.h"
#include "../include/utilities.h"

#define FD_FAILED  -1
#define SHM_FAILED -1

void freeBeforeMap(int shm_fd) {
  close(shm_fd);
}

void freeBeforeSemW(int shm_fd, shm_buffer_t *buf) {
  munmap(buf, sizeof(shm_buffer_t));
  freeBeforeMap(shm_fd);
```

```
}

void freeBeforeSemR(int shm_fd, shm_buffer_t *buf, sem_t *sem_w) {
  sem_close(sem_w);
  freeBeforeSemW(shm_fd, buf);
}

void freeBeforeFd(int shm_fd, shm_buffer_t *buf, sem_t *sem_w, sem_t *sem_r) {
  sem_close(sem_r);
  freeBeforeSemR(shm_fd, buf, sem_w);
}

void freeAll(int shm_fd, shm_buffer_t *buf, sem_t *sem_w, sem_t *sem_r, int fd) {
  close(fd);
  freeBeforeFd(shm_fd, buf, sem_w, sem_r);
}

void reverse(char *str, ssize_t len) {
  if (len > 0 && str[len - 1] == '\n') {
    len--;
  }

  for (ssize_t i = 0; i < len / 2; i++) {
    char tmp = str[i];
    str[i] = str[len - 1 - i];
    str[len - 1 - i] = tmp;
  }
}

int main(int argc, char *argv[]) {
  if (argc != 5) {
    printError("Invalid command. Usage: ./child <shm_name> <sem_w_name> <sem_r_name>
                <file_name>");
    exit(EXIT_FAILURE);
  }

  const char *shm_name = argv[1];
  const char *sem_w_name = argv[2];
  const char *sem_r_name = argv[3];
  const char *fileName = argv[4];

  int shm_fd = shm_open(shm_name, O_RDWR, 0600);
  if (shm_fd == SHM_FAILED) {
    printError("shm_open failed");
    exit(EXIT_FAILURE);
  }

  shm_buffer_t *buf = mmap(NULL, sizeof(shm_buffer_t), PROT_READ | PROT_WRITE,
                           MAP_SHARED, shm_fd, 0);
```

```
  if (buf == MAP_FAILED) {
    freeBeforeMap(shm_fd);
    printError("mmap failed");
    exit(EXIT_FAILURE);
  }

  sem_t *sem_w = sem_open(sem_w_name, 0);
  if (sem_w == SEM_FAILED) {
    freeBeforeSemW(shm_fd, buf);
    printError("sem_open for write failed");
    exit(EXIT_FAILURE);
  }

  sem_t *sem_r = sem_open(sem_r_name, 0);
  if (sem_r == SEM_FAILED) {
    freeBeforeSemR(shm_fd, buf, sem_w);
    printError("sem_open for read failed");
    exit(EXIT_FAILURE);
  }

  int fd = open(fileName, O_CREAT | O_WRONLY | O_TRUNC, 0600);
  if (fd == FD_FAILED) {
    freeBeforeFd(shm_fd, buf, sem_w, sem_r);
    char message[BUFSIZ];
    snprintf(message, BUFSIZ, "Can not open the file \"%s\"", fileName);
    printError(message);
    exit(EXIT_FAILURE);
  }

  do {
    sem_wait(sem_r);

    if (buf->finished) {
      sem_post(sem_w);
      break;
    }

    reverse(buf->data, buf->len);
    write(fd, buf->data, buf->len);

    sem_post(sem_w);
  } while (!buf->finished);

  freeAll(shm_fd, buf, sem_w, sem_r, fd);

  return 0;
}
```

Протокол работы программы

```
$ ./parent
Enter the name of the child file #1: output1.txt
Enter the name of the child file #2: output2.txt
apple
banana
potato
orange
yellow
pencil
Moscow
Russia
$ cat ./output1.txt
elppa
otatop
egnaro
aissuR
$ cat ./output2.txt
ananab
wolley
licnep
wocsoM
```

Вывод

В ходе выполнения лабораторной работы я научился работать с именованной разделяемой памятью процессов: создавать её объекты, отображать их в адресном пространстве процесса, читать данные из них и записывать данные в них, синхронизируя доступ с помощью семафоров, а также освобождать захваченные ресурсы. Помимо этого, я закрепил навыки работы с несколькими процессами одновременно и сравнил использование неименованных каналов и именованной разделяемой памяти.