Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**Потоки в операционных системах**

| Студент | Гришин Алексей Юрьевич |
| --- | --- |
| Группа | М8О-208Б-21 |
| Вариант | 17 |
| Преподаватель | Соколов Андрей Алексеевич |
| Оценка | 5 |
| Дата | 07.11.2022 |
| Подпись |  |

Москва, 2022.

## **Постановка задачи**

### **Цель работы**

Приобретение практических навыков в:

* Освоение принципов работы с файловыми системами
* Обеспечение обмена данных между процессами посредством технологии “File mapping”

### **Задание**

Составить и отладить программу на языке C, 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.

**Вариант 17:** Правило фильтрации – строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы удаляют все гласные из строк.

## **Общие сведения о программе**

Программа компилируется из файла scheduler.c. Основная логика родительского и дочернего процессов содержится в файлах manager.c, worker.c. Файл scheduler.c связан с файлами manager.c и worker.c через контракты, организованные в виде заголовочных файлов manager.h и worker.h соответственно.

Также, используются вспомогательные модули buffer, io\_line, remove\_vowels. Файл manager.c использует заголовочные файлы io\_line/io\_line.h, а файл worker.c – io\_line/io\_line.h и remove\_vowels/remove\_vowels.h.

Из системных библиотек, программа использует заголовочные файлы unistd.h, string.h, stdlib.h, fcntl.h, sys/wait.h, sys/types.h, sys/stat.h, sys/mman.h, semaphore.h, stdbool.h, errno.h.

### **Системные вызовы**

В программе используются следующие системные вызовы:

1. mmap(addr, length, prot, flags, fd, offset) – создает отображению на область в логической памяти. В параметре addr указывается желаемый адрес расположения отображения. Если область по данному адресу уже занята, то операционная система ищет другое. Параметр length обозначает размер занимаемой области. В качестве параметра prot передаются специальные флаги защиты, обозначающие права доступа к файлу: на запись, на чтение, на исполнение. В качестве параметра flags указываются параметры для отображения, среди которых я использовал MAP\_SHARED и MAP\_ANONYMOUS. Первый параметр обозначает, что создаваемое отображение будет общее между процессами, второй – что отображаемый файл анонимный. В случае, если не был поставлен флаг MAP\_ANONYMOUS, в параметр fd передается файловый дескриптор отображаемого файла, а параметр offset обозначает смещение в файле, после которого вся последующая часть файла отображается в логическую память.
2. munmap(addr, length) – освобождает занятую для отображения область логической памяти, находящуюся по адресу и размером length. В случае успешного выполнения возвращает 0, в случае неуспеха – MAP\_FAILED.
3. fork() – создает копию родительского процесса (который вызывает fork), которая называется дочерним процессом. После создания нового дочернего процесса оба процесса будут выполнять следующую инструкцию после системного вызова fork(). Для дочернего процесса возвращаемым значением будет 0, для родительского – id дочернего процесса. В случае ошибки возвращается отрицательное число.
4. waitpid(pid, status, options) – ожидает завершения дочернего процесса. В параметр pid передается id дочернего процесса, в status запишется статус, с которым завершился дочерний процесс. В параметр options передаются опции для ожидания такие, как WNOHANG (возвращает управление сразу же, если дочерний процесс пока не завершился).
5. exit(status) – завершает текущую программу. В параметр status передается статус, с которым завершается программа (обычно, при успешном выполнении записывается значение 0)
6. sem\_init(sem, pshared, value) – инициализирует анонимный семафор. Параметр pshared является флагом, определяющим, будет ли семафор общим для дочерних процессов при клонировании. В параметр value передается начальное значение семафора.
7. sem\_wait(sem) – уменьшает значение в семафоре на 1. Если при выполнении операции в семафоре лежит нулевое значение, то процесс “встает в очередь” до тех пор, пока в семафоре не будет положительное значение.
8. sem\_post(sem) – увеличивает значение в семафоре на 1
9. sem\_destroy(sem) – удаляет анонимный семафор в текущем процессе.
10. write(fd, addr, length) – записывает в файл, которому соответствует файловый дескриптор fd, данные, хранящиеся в логической памяти по адресу addr и имеющую размеры в length байт.
11. read(fd, addr, length) – считывает из файла, которому соответствует файловый дескриптор fd, length байт. Считанные данные заносятся в область, находящуюся по адресу addr.

### **Описание**

Так как задание лабораторной работы №4 основывается на задании лабораторной работы №2, то решение, соответственно, тоже основано на коде из 2-й лабораторной работы. Изменился только способ общения между родительским и дочерними процессами. Для обеспечения общения между процессами с использованием memory mapped файла я реализовал модуль buffer, который добавляет тип данных MappedBuffer и методы, для работы с буфером: init\_buffer, send\_data, receive\_data, close\_buffer. По своему описанию буфер практически ничем не отличается от анонимного канала, однако, он основывается на других технологиях. Метод init\_buffer создает анонимный memory mapped файл, который является общим для родительского и дочерних процессов. Для синхронизации операций используется 2 семафора: для обозначения заполненности буфера и его пустоты. Если при вызове метода send\_data буфер оказался полным вызывающий процесс блокируется до тех пор, пока буфер не очистится. Аналогично, если при вызове операции receive\_data буфер оказался пустым, вызывающий процесс блокируется до тех пор, пока буфер не заполнится.

### **Исходный код**

#### **buffer/buffer.c**

| #include "buffer.h"  #include <string.h>  #include <sys/mman.h>  #include <semaphore.h>  typedef struct \_\_BufferMetadata {  sem\_t empty;  sem\_t full;  void \* buff;  } BufferMetadata;  int init\_buffer(MappedBuffer \* buffer, size\_t size) {  BufferMetadata \* md = mmap(  NULL, sizeof(BufferMetadata),  PROT\_READ | PROT\_WRITE,  MAP\_SHARED | MAP\_ANONYMOUS,  0, 0  );  md -> buff = mmap(  NULL, size,  PROT\_READ | PROT\_WRITE,  MAP\_SHARED | MAP\_ANONYMOUS,  0, 0  );  buffer -> pointer = md;  buffer -> size = size;  sem\_init( & md -> full, 1, 0);  sem\_init( & md -> empty, 1, 1);  return BUFFER\_OK;  }  int send\_data(MappedBuffer \* buffer, void \* data, size\_t size) {  BufferMetadata \* md = buffer -> pointer;  size\_t copy\_size;  while (size > 0) {  copy\_size = (size < buffer -> size) ? size : buffer -> size;  sem\_wait( & md -> empty);  memcpy(md -> buff, data, copy\_size);  sem\_post( & md -> full);  size -= copy\_size;  data += copy\_size;  }  return BUFFER\_OK;  }  int receive\_data(MappedBuffer \* buffer, void \* buff, size\_t size) {  BufferMetadata \* md = buffer -> pointer;  size\_t copy\_size;  while (size > 0) {  copy\_size = (size < buffer -> size) ? size : buffer -> size;  sem\_wait( & md -> full);  memcpy(buff, md -> buff, copy\_size);  sem\_post( & md -> empty);  size -= copy\_size;  buff += copy\_size;  }  return BUFFER\_OK;  }  void close\_buffer(MappedBuffer \* buffer) {  BufferMetadata \* md = buffer -> pointer;  munmap(md -> buff, buffer -> size);  sem\_destroy( & md -> full);  sem\_destroy( & md -> empty);  munmap(buffer -> pointer, sizeof(BufferMetadata));  } |
| --- |

#### **buffer/buffer.h**

| #ifndef BUFFER\_H  #define BUFFER\_H  #include <stdlib.h>  #define BUFFER\_OK 0  #define BUFFER\_ERROR\_INIT 1  #define BUFFER\_ERROR\_RECEIVE 2  #define BUFFER\_ERROR\_SEND 3  struct \_\_MappedBuffer {  void \* pointer;  size\_t size;  };  typedef struct \_\_MappedBuffer MappedBuffer;  int init\_buffer(MappedBuffer \* buffer, size\_t buf\_size);  int send\_data(MappedBuffer \* buffer, void \* data, size\_t size);  int receive\_data(MappedBuffer \* buffer, void \* buff, size\_t size);  void close\_buffer(MappedBuffer \* buffer);  #endif |
| --- |

#### **io\_line/io\_line.c**

| #include "io\_line.h"  #include <stdio.h>  #include <unistd.h>  #include <string.h>  #include <errno.h>  #include <stdbool.h>  bool \_\_is\_endline\_symbol(char symbol) {  return (symbol == EOF || symbol == '\n');  }  bool \_\_try\_to\_write(Descriptor d, void \* data, size\_t data\_size) {  return write(d, data, data\_size) == data\_size;  }  bool \_\_try\_to\_read(Descriptor d, void \* data, size\_t data\_size) {  return read(d, data, data\_size) == data\_size;  }  int \_\_handle\_file\_error(int error) {  if (error == EACCES)  return IO\_LINE\_ACCESS\_DENIED;  if (error == EBADF)  return IO\_LINE\_INVALID\_DESCRIPTOR;  }  int \_\_get\_char(Descriptor d, char \* symbol) {  if (!\_\_try\_to\_read(d, symbol, 1 \* sizeof(char)))  return \_\_handle\_file\_error(errno);  return IO\_LINE\_OK;  }  int write\_line(Descriptor d, String line) {  StringSize line\_length = strlen(line);  if (!\_\_try\_to\_write(d, line, line\_length \* sizeof(char)))  return \_\_handle\_file\_error(errno);  if (!\_\_try\_to\_write(d, "\n", 1 \* sizeof(char)))  return \_\_handle\_file\_error(errno);  return IO\_LINE\_OK;  }  int read\_line(Descriptor d, String buffer, StringSize buffer\_size) {  char symbol;  size\_t index = 0;  int result;  while (1) {  result = \_\_get\_char(d, & symbol);  if (result != IO\_LINE\_OK)  return result;  if (\_\_is\_endline\_symbol(symbol))  break;  if (symbol == '\r')  continue;  if (index >= buffer\_size - 1)  return IO\_LINE\_NOT\_ENOUGH\_BUFFER\_SIZE;  buffer[index++] = symbol;  }  buffer[index] = '\0';  return IO\_LINE\_OK;  } |
| --- |

#### **io\_line/io\_line.h**

| #ifndef IO\_LINE\_H  #define IO\_LINE\_H  #include <stdlib.h>  #define IO\_LINE\_OK 0  #define IO\_LINE\_INVALID\_DESCRIPTOR 11  #define IO\_LINE\_ACCESS\_DENIED 12  #define IO\_LINE\_NOT\_ENOUGH\_BUFFER\_SIZE 13  typedef int Descriptor;  typedef char \* String;  typedef size\_t StringSize;  int write\_line(Descriptor d, String line);  int read\_line(Descriptor d, String buffer, StringSize buffer\_size);  #endif |
| --- |

#### **remove\_vowels/remove\_vowels.c**

| #include "remove\_vowels.h"  #include <string.h>  #include <stdbool.h>  char \* \_\_vowels = "eyuioaEYUIOA";  bool \_\_is\_vowel(char symbol) {  for (char \* c = \_\_vowels;\* c != '\0'; c++)  if ( \* c == symbol)  return true;  return false;  }  int remove\_vowels(char \* s, char \* buff, size\_t buff\_size) {  int index = 0;  size\_t s\_length = strlen(s);  int ret\_value = REMOVE\_VOWELS\_OK;  for (int j = 0;  (j < s\_length) && (index < buff\_size - 1); j++) {  if (\_\_is\_vowel(s[j]))  continue;  buff[index++] = s[j];  }  buff[index] = '\0';  if (index == buff\_size - 1)  return REMOVE\_VOWELS\_ERROR\_NOT\_ENOUGH\_BUFFER\_LEN;  return REMOVE\_VOWELS\_OK;  } |
| --- |

#### **remove\_vowels/remove\_vowels.h**

| #ifndef REMOVE\_VOWELS\_H  #define REMOVE\_VOWELS\_H  #include <stdlib.h>  #define REMOVE\_VOWELS\_OK 0  #define REMOVE\_VOWELS\_ERROR\_NOT\_ENOUGH\_BUFFER\_LEN 31  int remove\_vowels(char \* s, char \* buff, size\_t buff\_size);  #endif |
| --- |

#### **manager.c**

| #include "manager.h"  #include "io\_line/io\_line.h"  #include <unistd.h>  #include <string.h>  #define MAX\_INPUT\_STRING\_LEN 256  void manager\_logic(MappedBuffer workers[2]) {  char input\_string[MAX\_INPUT\_STRING\_LEN];  while (1) {  int res = read\_line(  STDIN\_FILENO, input\_string, MAX\_INPUT\_STRING\_LEN);  if (res != IO\_LINE\_OK)  exit(res);  if (strlen(input\_string) == 0)  break;  int i = strlen(input\_string) > 10;  send\_data( & workers[i], input\_string, strlen(input\_string));  }  } |
| --- |

#### **manager.h**

| #ifndef MANAGER\_H  #define MANAGER\_H  #include "buffer/buffer.h"  void manager\_logic(MappedBuffer workers[2]);  #endif |
| --- |

#### **worker.c**

| #include "worker.h"  #include "io\_line/io\_line.h"  #include "remove\_vowels/remove\_vowels.h"  #include <string.h>  #include <stdlib.h>  #include <fcntl.h>  #include <unistd.h>  #define MAX\_INPUT\_LINE\_LEN 256  #define FAILED\_TO\_OPEN\_RESULT\_FILE 52  int protected\_open\_result\_file\_descriptor(char \* filepath) {  int descriptor;  if ((descriptor = open(filepath, O\_WRONLY | O\_TRUNC)) == -1)  exit(FAILED\_TO\_OPEN\_RESULT\_FILE);  return descriptor;  }  void protected\_get\_data(  MappedBuffer \* src,  char \* buff,  size\_t buff\_size)  {  receive\_data(src, buff, buff\_size);  }  void protected\_write\_line(int descriptor, char \* line) {  int result = write\_line(descriptor, line);  if (result != IO\_LINE\_OK)  exit(result);  }  void protected\_format\_line(char \* input\_line, char \* buff, size\_t buff\_size) {  int result = remove\_vowels(input\_line, buff, buff\_size);  if (result != REMOVE\_VOWELS\_OK)  exit(result);  }  void worker\_logic(MappedBuffer \* buff, char \* result\_filepath) {  int result\_file\_descriptor;  char input\_line[MAX\_INPUT\_LINE\_LEN];  char formatted\_line[MAX\_INPUT\_LINE\_LEN];  result\_file\_descriptor = protected\_open\_result\_file\_descriptor(result\_filepath);  protected\_get\_data(buff, input\_line, MAX\_INPUT\_LINE\_LEN);  while (strlen(input\_line) != 0) {  protected\_format\_line(  input\_line, formatted\_line, MAX\_INPUT\_LINE\_LEN);  protected\_write\_line(result\_file\_descriptor, formatted\_line);  protected\_get\_data(buff, input\_line, MAX\_INPUT\_LINE\_LEN);  }  close(result\_file\_descriptor);  exit(0);  } |
| --- |

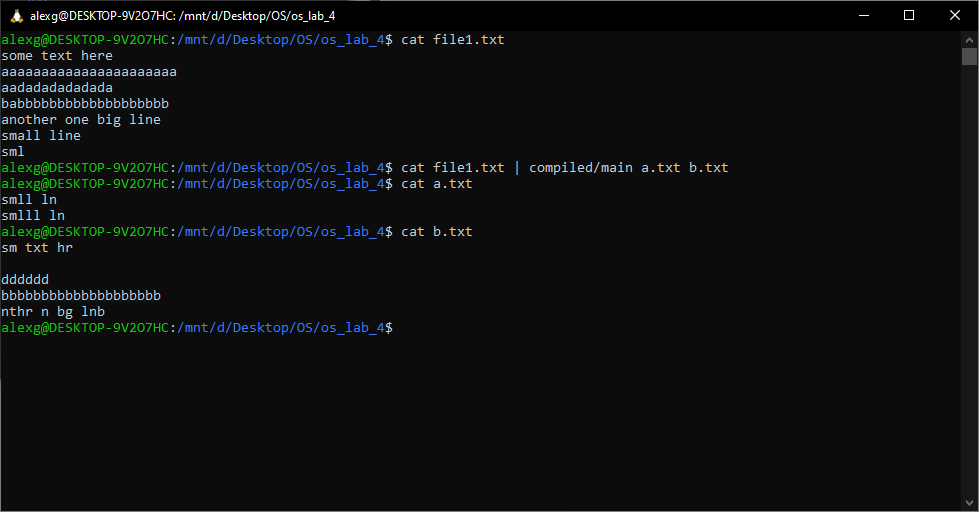
#### **worker.h**

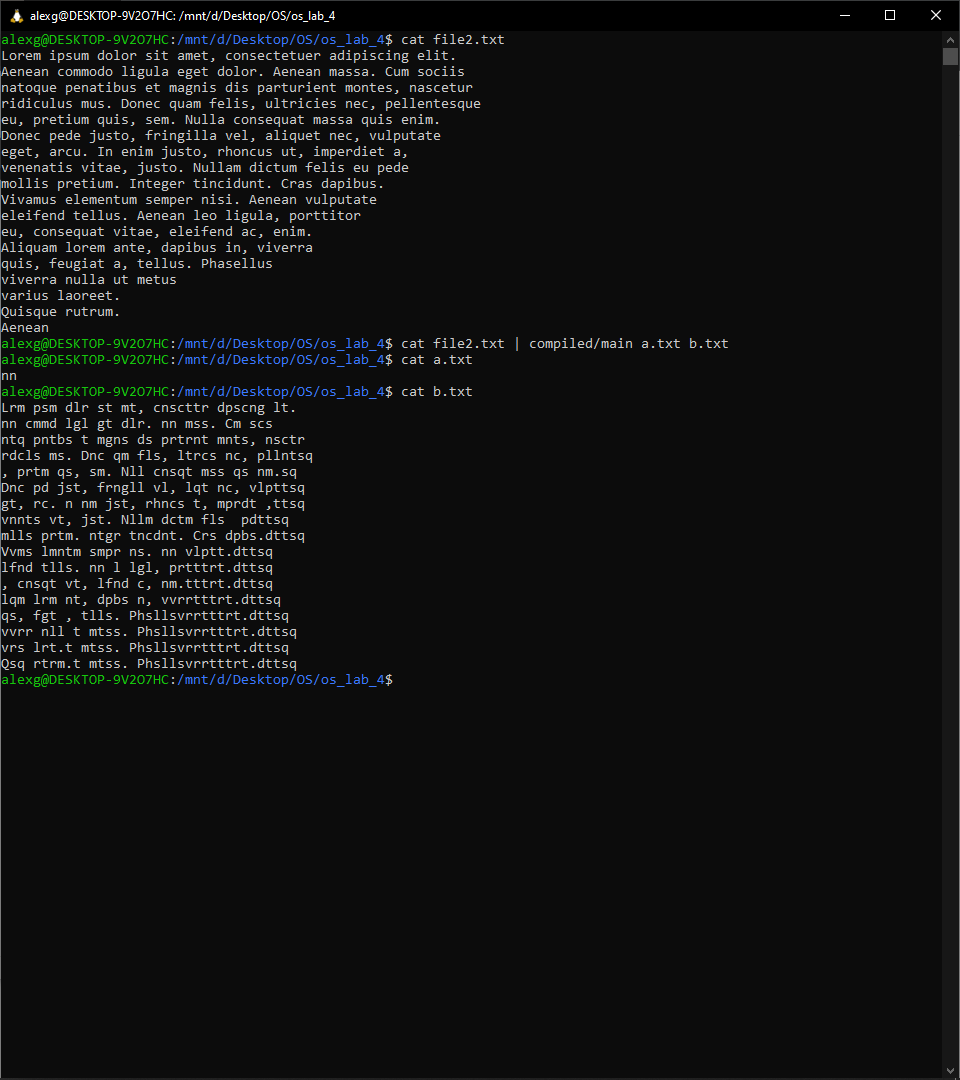
| #ifndef WORKER\_H  #define WORKER\_H  #include "buffer/buffer.h"  void worker\_logic(MappedBuffer\* buff, char\* result\_filepath);  #endif |
| --- |

#### **scheduler.c**

| #include "manager.h"  #include "worker.h"  #include "buffer/buffer.h"  #include <stdlib.h>  #include <unistd.h>  #include <fcntl.h>  #include <sys/wait.h>  #include <sys/types.h>  #include <sys/stat.h>  #define BUFFER\_SIZE 256  #define MAX\_INPUT\_STRING\_LEN 256  #define WORKER\_PATH "worker"  #define CREATE\_PROCESS\_ERROR 41  #define MANAGER\_ERROR\_CREATE\_RESULT\_FILE 44  #define MANAGER\_ERROR\_RESULT\_FILE\_NOT\_SPECIFIED 45  #define MANAGER\_ERROR\_OPEN\_RESULT\_FILE 46  void protected\_create\_file(char \* filename) {  int descriptor = open(filename, O\_CREAT, S\_IWRITE | S\_IREAD);  if (descriptor == -1)  exit(MANAGER\_ERROR\_CREATE\_RESULT\_FILE);  close(descriptor);  }  char \* protected\_get\_result\_file(int argc, char \* argv[], int index) {  int descriptor;  if (index >= argc)  exit(MANAGER\_ERROR\_RESULT\_FILE\_NOT\_SPECIFIED);  if (access(argv[index], F\_OK) != 0)  protected\_create\_file(argv[index]);  descriptor = open(argv[index], O\_WRONLY);  if (descriptor == -1)  exit(MANAGER\_ERROR\_OPEN\_RESULT\_FILE);  close(descriptor);  return argv[index];  }  pid\_t create\_worker(MappedBuffer \* buff, char \* result\_filepath) {  pid\_t pid = fork();  if (pid < 0)  exit(CREATE\_PROCESS\_ERROR);  if (pid == 0)  worker\_logic(buff, result\_filepath);  return pid;  }  void close\_worker(pid\_t worker\_id, MappedBuffer \* buff) {  send\_data(buff, "\0", 1);  waitpid(worker\_id, NULL, 0);  }  int main(int argc, char \* argv[]) {  char input\_string[MAX\_INPUT\_STRING\_LEN];  char \* result\_filepaths[2];  MappedBuffer worker\_buffers[2];  pid\_t worker\_ids[2];  for (int i = 0; i < 2; i++)  result\_filepaths[i] = protected\_get\_result\_file(argc, argv, i + 1);  for (int i = 0; i < 2; i++)  init\_buffer( & worker\_buffers[i], BUFFER\_SIZE);  for (int i = 0; i < 2; i++)  worker\_ids[i] = create\_worker( & worker\_buffers[i], result\_filepaths[i]);  manager\_logic(worker\_buffers);  for (int i = 0; i < 2; i++)  close\_worker(worker\_ids[i], & worker\_buffers[i]);  for (int i = 0; i < 2; i++)  close\_buffer( & worker\_buffers[i]);  exit(0);  } |
| --- |

### **Пример работы**





## **Вывод**

В ходе выполнения данной лабораторной работы я ознакомился с технология memory mapped файлов. Они гораздо удобнее анонимных каналов, так как работать с отображением мы можем, как с обычной областью памяти (например, через вызов malloc). Memory mapped файлы удобны в случаях, когда необходимо иметь общие данные между процессами. Однако, для задач вида Producer - Consumer, на мой взгляд, эта технология не подходит, так как работа с отображением происходит не синхронизировано, в то время как в pipe операции чтения и записи являются синхронизированными.