ИДЗ-1

Клычков Максим Дмитриевич. БПИ237. Вариант 10.

Текст задания

> Разработать программу, которая меняет на обратный порядок следования символов **каждого слова** в ASCII–строке символов. Порядок слов остается неизменным. Слова состоят только из букв. Разделителями слов являются все прочие символы.

4 балла

Общая схема решения

Задача решается с помощью трех родственных процессов — читателя, писателя и обработчика. Между собой они взаимодействуют при помощи двух *неименованных* каналов: `raw` и `processed`, по которым передаются сырые (необработанные) и обработанные данные соответственно. В каждом из процессов сразу же закрываются файловые дескрипторы, которые не будут использоваться в этом процессе. Предоставим код каждого из взаимодействующих них:

1. Обработчик — процесс, который занимается обработкой данных согласно условию

```
// ------------------------Child process - processor------------------

// Close unused fd
if (close(fd_raw[1]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[0]) < 0) {
  perror("close");
  return 1;
}

// Main logic
char buffer[BUFFER_SIZE] = {};
int res;
if ((res = read(fd_raw[0], buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  // Process data
  Process(buffer, res);

  if (write(fd_processed[1], buffer, res) < 0) {
    perror("write");
    return 1;
```

```
  }
}

// Close used fd
if (close(fd_raw[0]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[1]) < 0) {
  perror("close");
  return 1;
}
```

2. Читатель — процесс, который занимается считыванием данных из файла

```
// -----------------Parent process - reader from file--------------------

// Close unused fd
if (close(fd_raw[0]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[0]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[1]) < 0) {
  perror("close");
  return 1;
}

// Read logic
char buffer[BUFFER_SIZE] = {};
int fd_from = open(argv[1], O_RDONLY);
if (fd_from < 0) {
  perror("open");
  return 1;
}

int res;
if ((res = read(fd_from, buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_raw[1], buffer, res) < 0) {
    perror("write");
    return 1;
  }
```

```
}

// Close used fd
if (close(fd_from) < 0) {
  perror("close");
  return 1;
}
if (close(fd_raw[1]) < 0) {
  perror("close");
  return 1;
}
```

3. Писатель — процесс, который записью обработанных данных в файл.

```
//  -----------------Child process - writer to file--------------------

// Close unused fd
if (close(fd_raw[0]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_raw[1]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[1]) < 0) {
  perror("close");
  return 1;
}

// Write logic
char buffer[BUFFER_SIZE] = {};
int fd_to = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd_to < 0) {
  perror("open");
  return 1;
}

int res;
if ((res = read(fd_processed[0], buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_to, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}
```

```
// Close used fd
if (close(fd_to) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[0]) < 0) {
  perror("close");
  return 1;
}
```

Подробнее об логике обработки файла

Алгоритм обработки буфера достаточно простой. Он заключается в линейном просмотре всех символов и нахождении одного слова. Затем полученное слово перезаписывается в обратном порядке в тот же буфер.

```
void Reverse(char *arr, int from, int to) {
  while (from < to) {
    int tmp = arr[from];
    arr[from] = arr[to];
    arr[to] = tmp;
    from++;
    to--;
  }
}

void Process(char *arr, int size) {
  int start = 0;
  for (int i = 0; i < size; i++) {
    if (!('a' <= arr[i] && arr[i] <= 'z') &&
        !('A' <= arr[i] && arr[i] <= 'Z')) {
      Reverse(arr, start, i - 1);
      start = i + 1;
    }
  }
  Reverse(arr, start, size - 1);
}
```

Запуск программы

Непосредственно перед запуском необходимо скомпилировать программу. Пусть бинарный файл называется `reverser`. Тогда для запуска программы необходимо указать следующие аргументы:

`reverser <in> <out>`

где `<in>` и `<out>` имена входного и выходного файлов соответственно.

Тесты

Было разработано 5 тестовых файлов, все они находятся в директории `tests/`. Результаты можно найти в директориях, соответствующих определенной оценке.

Формально требуется продублировать тестовые данные и результаты для них.

1 тест — разное

```
444hello5556world      hse

studend 2025 test


eof
```

Результат:

```
444olleh5556dlrow      esh

dneduts 2025 tset


foe
```

2 тест — разное количество разделителей между словами

```
0digits1separated2example3with4single5digit6
00more11digits22separators33
000one111more222without333end444digits
0000im1111tired
```

Результат:

```
0stigid1detarapes2elpmaxe3htiw4elgnis5tigid6
00erom11stigid22srotarapes33
000eno111erom222tuohtiw333dne444stigid
0000mi1111derit
```

3 тест — только небуквенные символы

```
12345678910
10987654321
[[[]]] ((())) {{{}}}
2+2=5
5/0=0
52


```

Результат:

```
12345678910
10987654321
```

```
[[[]]] ((())) {{{}}}
2+2=5
5/0=0
52
```

4 тест — только буквенные символы

```
onemoretestonlyonelinenoseparatorsonlyhardcoreandlettersmorelettersandlettersiloveasciisymbolsyes
```

Результат:

```
koonononseyseyseyslobmysiicsaevolisretteldnasretteleromsretteldnaerocdrahylnosrotarapesonenilenoy
```

5 тест — разделитель в разных частях слова

```
_sepbefore
sep_inside
sepafter_
```

Результат:

```
_erofebpes
pes_edisni
retfapes_
```

5 баллов

В целом поменялся только вид каналов, все остальные пункты остаются актуальными. Тестовые данные не изменились, результаты тестов в директории, соответствующей оценке.

Создание именованных каналов

Для общения между процессами используются два именованных канала `raw.fifo`, отвечающий за передачу сырых данных, и `processed.fifo`, отвечающий за передачу обработанных данных. Были написаны функции для создания каналов, если они еще не созданы. В каждом процессе нужный канал (или несколько) открывается, данные передаются, а затем каналы закрываются.

Пример функций создания:

```c
const char FIFO_RAW_PATHNAME[] = "raw.fifo";
const char FIFO_PROCESSED_PATHNAME[] = "processed.fifo";

void MakeRawFifo() {
  int res = mknod(FIFO_RAW_PATHNAME, S_IFIFO | 0666, 0);
  if (res == -1 && errno != EEXIST) {
    perror("mknod");
    _exit(1);
  }
}
```

```
void MakeProcessedFifo() {
  int res = mknod(FIFO_PROCESSED_PATHNAME, S_IFIFO | 0666, 0);
  if (res == -1 && errno != EEXIST) {
    perror("mknod");
    _exit(1);
  }
}
```

Пример измененного процесса, отвечающего за обработку (в остальных аналогично поменялось только открытие/закрытие каналов):

```
// ------------------------Child process - processor------------------

// Open(create) FIFOs
MakeRawFifo();
MakeProcessedFifo();
int fd_raw, fd_processed;
if ((fd_raw = open(FIFO_RAW_PATHNAME, O_RDONLY)) < 0) {
  perror("open");
  return 1;
}
if ((fd_processed = open(FIFO_PROCESSED_PATHNAME, O_WRONLY)) < 0) {
  perror("open");
  return 1;
}

// Main logic
char buffer[BUFFER_SIZE] = {};
int res;
if ((res = read(fd_raw, buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  // Process data
  Process(buffer, res);

  if (write(fd_processed, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}

// Close used fd
if (close(fd_raw) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed) < 0) {
```

```
  perror("close");
  return 1;
}
```

Тесты

Все тесты аналогичны программе на 4 балла. Результаты расположены в директории, соответствующей оценке, здесь и далее они дублироваться не будут, так как остаются точно такими же, как и для программы на меньшую оценку.

6 баллов

В целом эта программа незначительно отличается от программы на 4 балла: лишь были объединены два процесса, отвечающие за работу с файлами.

Процессы и их взаимодействие

Логика нового родительского процесса — чтение данных из файла, передача их по каналу `raw`, затем чтение данных из канала `processed` и последующая запись их в выходной файл.

```
//  ----------Parent process - reader/writer from/to file--------------

// Close unused fd
if (close(fd_raw[0]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[1]) < 0) {
  perror("close");
  return 1;
}

// Read logic
char buffer[BUFFER_SIZE] = {};
int fd_from = open(argv[1], O_RDONLY);
if (fd_from < 0) {
  perror("open");
  return 1;
}

int res;
if ((res = read(fd_from, buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_raw[1], buffer, res) < 0) {
    perror("write");
    return 1;
```

```
  }
}

// Write logic
int fd_to = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd_to < 0) {
  perror("open");
  return 1;
}

if ((res = read(fd_processed[0], buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_to, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}

// Close used fd
if (close(fd_to) < 0) {
  perror("close");
  return 1;
}
if (close(fd_from) < 0) {
  perror("close");
  return 1;
}
if (close(fd_raw[1]) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed[0]) < 0) {
  perror("close");
  return 1;
}
```

Тесты

Все тесты аналогичны программе на 4-5 баллов. Результаты расположены в директории, соответствующей оценке, здесь и далее они дублироваться не будут, так как остаются точно такими же, как и для программы на меньшую оценку.

7 баллов

В целом эта программа незначительно отличается от программы на 5 баллов: лишь были объединены два процесса, отвечающие за работу с файлами.

Процессы и их взаимодействие

Логика нового родительского процесса — чтение данных из файла, передача их по именованному каналу `raw`, затем чтение данных из именованного канала `processed` и последующая запись их в выходной файл.

```
//  ----------Parent process - reader/writer from/to file--------------

// Open(create) FIFOs
MakeRawFifo();
MakeProcessedFifo();
int fd_raw, fd_processed;
if ((fd_raw = open(FIFO_RAW_PATHNAME, O_WRONLY)) < 0) {
  perror("open");
  return 1;
}
if ((fd_processed = open(FIFO_PROCESSED_PATHNAME, O_RDONLY)) < 0) {
  perror("open");
  return 1;
}

// Read logic
char buffer[BUFFER_SIZE] = {};
int fd_from = open(argv[1], O_RDONLY);
if (fd_from < 0) {
  perror("open");
  return 1;
}

int res;
if ((res = read(fd_from, buffer, sizeof(buffer))) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_raw, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}

// Write logic
int fd_to = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd_to < 0) {
  perror("open");
  return 1;
}

if ((res = read(fd_processed, buffer, sizeof(buffer))) != 0) {
```

```
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_to, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}

// Close used fd
if (close(fd_from) < 0) {
  perror("close");
  return 1;
}
if (close(fd_to) < 0) {
  perror("close");
  return 1;
}
if (close(fd_raw) < 0) {
  perror("close");
  return 1;
}
if (close(fd_processed) < 0) {
  perror("close");
  return 1;
}
```

Тесты

Все тесты аналогичны программе на 4-6 баллов. Результаты расположены в директории, соответствующей оценке, здесь и далее они дублироваться не будут, так как остаются точно такими же, как и для программы на меньшую оценку.

8 баллов

В этом пункте требуется разработать уже 2 программы, то есть разделить логику обработчика данных и ввода/вывода.

common

Общие константы и функции были вынесены в `common.c` и соответствующий ему `common.h`. Это сделано во избежание дублирования кода. Приведем содержание этих файлов:

`common.h`:

```
#ifndef COMMON_H
#define COMMON_H

#include <errno.h>
```

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/unistd.h>
#include <sys/wait.h>
#include <unistd.h>

#define BUFFER_SIZE 5000

extern const char FIFO_RAW_PATHNAME[];
extern const char FIFO_PROCESSED_PATHNAME[];

void MakeRawFifo();
void MakeProcessedFifo();

#endif
```

common.c:

```c
#include "common.h"

const char FIFO_RAW_PATHNAME[] = "raw.fifo";
const char FIFO_PROCESSED_PATHNAME[] = "processed.fifo";

void MakeRawFifo() {
  int res = mknod(FIFO_RAW_PATHNAME, S_IFIFO | 0666, 0);
  if (res == -1 && errno != EEXIST) {
    perror("mknod");
    _exit(1);
  }
}

void MakeProcessedFifo() {
  int res = mknod(FIFO_PROCESSED_PATHNAME, S_IFIFO | 0666, 0);
  if (res == -1 && errno != EEXIST) {
    perror("mknod");
    _exit(1);
  }
}
```

Обработчик processor

Здесь объявлены все функции связанные только с обработкой (бизнес-логика).

```c
void Reverse(char *arr, int from, int to) {
  while (from < to) {
    int tmp = arr[from];
    arr[from] = arr[to];
    arr[to] = tmp;
    from++;
    to--;
  }
```

```
}

void Process(char *arr, int size) {
  int start = 0;
  for (int i = 0; i < size; i++) {
    if (!('a' <= arr[i] && arr[i] <= 'z') &&
        !('A' <= arr[i] && arr[i] <= 'Z')) {
      Reverse(arr, start, i - 1);
      start = i + 1;
    }
  }
  Reverse(arr, start, size - 1);
}
```

В `main` создаются (если они еще не созданы) и открываются именованные каналы, далее происходит запись в них обработанных данных и последующее закрытие каналов.

```
int main(int argc, char *argv[]) {
  MakeRawFifo();
  MakeProcessedFifo();
  int fd_raw, fd_processed;
  if ((fd_raw = open(FIFO_RAW_PATHNAME, O_RDONLY)) < 0) {
    perror("open");
    return 1;
  }
  if ((fd_processed = open(FIFO_PROCESSED_PATHNAME, O_WRONLY)) < 0) {
    perror("open");
    return 1;
  }

  // Main logic
  char buffer[BUFFER_SIZE] = {};
  int res;
  if ((res = read(fd_raw, buffer, sizeof(buffer))) != 0) {
    if (res < 0) {
      perror("read");
      return 1;
    }

    // Process data
    Process(buffer, res);

    if (write(fd_processed, buffer, res) < 0) {
      perror("write");
      return 1;
    }
  }

  // Close used fd
  if (close(fd_raw) < 0) {
```

```
    perror("close");
    return 1;
  }
  if (close(fd_processed) < 0) {
    perror("close");
    return 1;
  }
}
```

Стоит отметить, что никаких параметров через аргументы командной строки эта программа не принимает.

Ввод и вывод `readwriter`

Все достаточно тривиально, просто копируем код, связанный с процессом ввода-вывода, из предыдущих программы на 7 баллов, никаких дополнительных функций не требуется.

```
int main(int argc, char *argv[]) {
  // Open(create) FIFOs
  MakeRawFifo();
  MakeProcessedFifo();
  int fd_raw, fd_processed;
  if ((fd_raw = open(FIFO_RAW_PATHNAME, O_WRONLY)) < 0) {
    perror("open");
    return 1;
  }
  if ((fd_processed = open(FIFO_PROCESSED_PATHNAME, O_RDONLY)) < 0) {
    perror("open");
    return 1;
  }

  // Read logic
  char buffer[BUFFER_SIZE] = {};
  int fd_from = open(argv[1], O_RDONLY);
  if (fd_from < 0) {
    perror("open");
    return 1;
  }

  int res;
  if ((res = read(fd_from, buffer, sizeof(buffer))) != 0) {
    if (res < 0) {
      perror("read");
      return 1;
    }

    if (write(fd_raw, buffer, res) < 0) {
      perror("write");
      return 1;
    }
```

```
  }

  // Write logic
  int fd_to = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
  if (fd_to < 0) {
    perror("open");
    return 1;
  }

  if ((res = read(fd_processed, buffer, sizeof(buffer))) != 0) {
    if (res < 0) {
      perror("read");
      return 1;
    }

    if (write(fd_to, buffer, res) < 0) {
      perror("write");
      return 1;
    }
  }

  // Close used fd
  if (close(fd_from) < 0) {
    perror("close");
    return 1;
  }
  if (close(fd_to) < 0) {
    perror("close");
    return 1;
  }
  if (close(fd_raw) < 0) {
    perror("close");
    return 1;
  }
  if (close(fd_processed) < 0) {
    perror("close");
    return 1;
  }

  return 0;
}
```

Стоит отметить, что эта эта программа принимает через аргументы командной строки имя файла для чтения и имя файла для записи аналогично приведенному примеру выше.

Тесты

Все тесты аналогичны программе на 4-7 баллов. Результаты расположены в директории, соответствующей оценке, здесь и далее они дублироваться не будут, так как остаются точно такими же, как и для программы на меньшую оценку.

9 баллов

В программе на эту оценку произошло много изменений...

Усложненная логика обработки текста

Задача осложняется тем, что фиксированный размер буфера может кончаться на середине слова, однако производит reverse слова мы можем только зная целое слово. Мы встаем перед задачей: необходимо хранить последнее считанное слово, так как оно может продолжиться в следующем чтении, тем самым дополнив текущее.

Пусть функция `Process`, которая выполняет просмотр буфера и его reverse его части при необходимости, возвращает индекс начала последнего слова.

```
int Process(char *arr, int size) {
  int start = 0;
  for (int i = 0; i < size; i++) {
    if (!('a' <= arr[i] && arr[i] <= 'z') &&
        !('A' <= arr[i] && arr[i] <= 'Z')) {
      Reverse(arr, start, i - 1);
      start = i + 1;
    }
  }
  return start; // return the start of the last word
}
```

Внутри основной программы будем производить чтение из канала до тех пор, пока он не будет закрыт (закрытие будет производиться из другой программы). Всегда будем читать текст кусками до 128 байт (по условию), обрабатывать его и писать только ту часть обработанного текста, в которой содержаться уже полные слова (то есть все без последнего). Затем производиться сдвиг буфера таким образом, чтобы оставшееся слово стояло в начале. Вновь производится чтение и обработка.

Также нельзя забывать про крайний случай: текст без разделителей. Обработка такого текста может быть закончена только после полного прочтения его в память, поэтому буферу необходимо уметь динамически увеличиваться.

Предоставим основную логику (без открытия-закрытия каналов для наглядности):

```
// Main logic
int buffer_size = BUFFER_SIZE * 2; // my own heuristic
char *buffer = (char *)malloc(buffer_size * sizeof(char));

int count = 0, count_read = 0;
while ((count_read = read(fd_raw, buffer + count, BUFFER_SIZE)) != 0) {
  if (count_read < 0) {
    perror("read");
    return 1;
  }
  count += count_read;

  // Process data
  int last_start = Process(buffer, count);
```

```
  if (last_start > 0) {
    // Write to channel
    int to_write = last_start, offset = 0;
    while (to_write > 0) {
      int written =
          write(fd_processed, buffer + offset, Min(to_write, BUFFER_SIZE));
      if (written != Min(to_write, BUFFER_SIZE)) {
        perror("write");
        return 1;
      }
      to_write -= written;
      offset += written;
    }

    // Move the last word to the beginning of the buffer
    for (int i = last_start; i < count; i++) {
      buffer[i - last_start] = buffer[i];
    }
    count -= last_start;
  }

  if (buffer_size - count < BUFFER_SIZE) {
    buffer_size += BUFFER_SIZE * 2; // my own heuristic
    buffer = (char *)realloc(buffer, buffer_size * sizeof(char));
  }
}

// Reverse and write the last word
Reverse(buffer, 0, count - 1);
int to_write = count, offset = 0;
while (to_write > 0) {
  int written =
      write(fd_processed, buffer + offset, Min(to_write, BUFFER_SIZE));
  if (written != Min(to_write, BUFFER_SIZE)) {
    perror("write");
    return 1;
  }
  to_write -= written;
  offset += written;
}
```

Изменения в `readwriter`

Практически ничего не поменялось, кроме того, что теперь мы читаем и пишем в цикле кусками по 128 байт. Также важное: теперь необходимо, чтобы `readwriter` закрывал за собой канал `raw` сразу же после окончания записи в него, так как `processor` будет в цикле ожидать новых данных через этот канал.

```
// Read logic
char buffer[BUFFER_SIZE] = {};
int fd_from = open(argv[1], O_RDONLY);
if (fd_from < 0) {
  perror("open");
  return 1;
}

int res;
while ((res = read(fd_from, buffer, BUFFER_SIZE)) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_raw, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}
if (close(fd_raw) < 0) {
  perror("close");
  return 1;
}

// Write logic
int fd_to = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd_to < 0) {
  perror("open");
  return 1;
}

while ((res = read(fd_processed, buffer, BUFFER_SIZE)) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  if (write(fd_to, buffer, res) < 0) {
    perror("write");
    return 1;
  }
}
```

Однако есть большой недостаток: при обработке достаточно большого файла канал `processed` переполняется и больше не принимает в себя данные, необходимо из него прочитать, однако программа устроена так, что сначала отправляет все данные в `processor`, а затем уже пишет. Эту проблему можно было бы побороть использованием трех процессов: писатель, читатель и обработчик, как это делолось в программах на оценку ниже, но не

будем же мы противоречить условию...

Тесты

Некоторые тесты аналогичны программе на 4-8 баллов. Результаты расположены в директории, соответствующей оценке, здесь и далее они дублироваться не будут, так как остаются точно такими же, как и для программы на меньшую оценку.

Также были разработаны тесты с общим размером большим 128 Байт, они расположены в директории со всеми тестами, а результаты в директории с результатами на оценку 9.

10 баллов

Перепишем всю логику IPC на очереди сообщений. Оставшаяся часть — полная копия предыдущей программы.

IPC

Будем использовать две отдельных очереди по аналогии с каналами: `raw` и `processed`.

- Для обозначения конца записи в очередь сообщений (это нужно, чтобы читающий процесс заканчивал свое чтение) отправляется пустое сообщение.
- Для простоты ключи очередей будем хардкодить в программу (они указаны в `common`).
- Для получения/передачи сообщения в каждой из программ будем импользовать структуру `msgbuf`, причем для получения и передачи используются разные буферы: `sbuf` — для отправки, и `rbuf` — для получения.

`common.h`:

```
#ifndef COMMON_H
#define COMMON_H

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/unistd.h>
#include <sys/wait.h>
#include <unistd.h>

#define BUFFER_SIZE 128
#define KEY_RAW 52
#define KEY_PROCESSED 53

typedef struct msgbuf {
  long mtype;
  char mtext[BUFFER_SIZE];
} message_buf;

#endif
```

`common.c`:

```c
#include "common.h"

const char FIFO_RAW_PATHNAME[] = "raw.fifo";
const char FIFO_PROCESSED_PATHNAME[] = "processed.fifo";
```

Приведем примеры передачи данных между процессами.

Передача данных из файла в `processor`:

```c
// Read logic
char buffer[BUFFER_SIZE] = {};
int fd_from = open(argv[1], O_RDONLY);
if (fd_from < 0) {
  perror("open");
  return 1;
}

int res;
while ((res = read(fd_from, buffer, BUFFER_SIZE)) != 0) {
  if (res < 0) {
    perror("read");
    return 1;
  }

  // Prepare message
  memcpy(sbuf.mtext, buffer, res);

  if (msgsnd(msqid_raw, &sbuf, res, 0) < 0) {
    perror("msgsnd");
    return 1;
  }
}
msgsnd(msqid_raw, &sbuf, 0, 0);
```

Получение, обработка и дальнейшая отправка в `processor`'е:

```c
while ((count_read = msgrcv(msqid_raw, &rbuf, BUFFER_SIZE, 1, 0)) != 0) {
  if (count_read < 0) {
    perror("read");
    return 1;
  }

  // Copy data to buffer
  memcpy(buffer + count, rbuf.mtext, count_read);
  count += count_read;

  // Process data
  int last_start = Process(buffer, count);

  if (last_start > 0) {
    // Write to channel
```

```
    int to_write = last_start, offset = 0;
    while (to_write > 0) {
      // Prepare message
      int buf_length = Min(to_write, BUFFER_SIZE);
      memcpy(sbuf.mtext, buffer + offset, buf_length);

      if (msgsnd(msqid_processed, &sbuf, buf_length, 0) < 0) {
        perror("msgsnd");
        return 1;
      }
      to_write -= buf_length;
      offset += buf_length;
    }

    // Move the last word to the beginning of the buffer
    // ...
  }

  // Realloc buffer
  // ...
}
```

Тесты

Все тесты аналогичны предыдущим программам. Результаты в директории на соответствующую оценку.