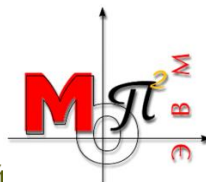


МИНОБРНАУКИ РОССИИ
Федеральное государственное образовательное
учреждение высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт компьютерных технологий и информационной безопасности

Кафедра Математического обеспечения и применения ЭВМ



ОТЧЕТ

по лабораторной работе №1
по курсу «Операционные системы реального времени»
«Разработка клиент-серверного приложения для ОСРВ QNX».

Выполнили:

студенты группы КТмо1-3
Бондаренко Д.С.
Клюйко А.И.

Проверила:

Пирская Л.В.

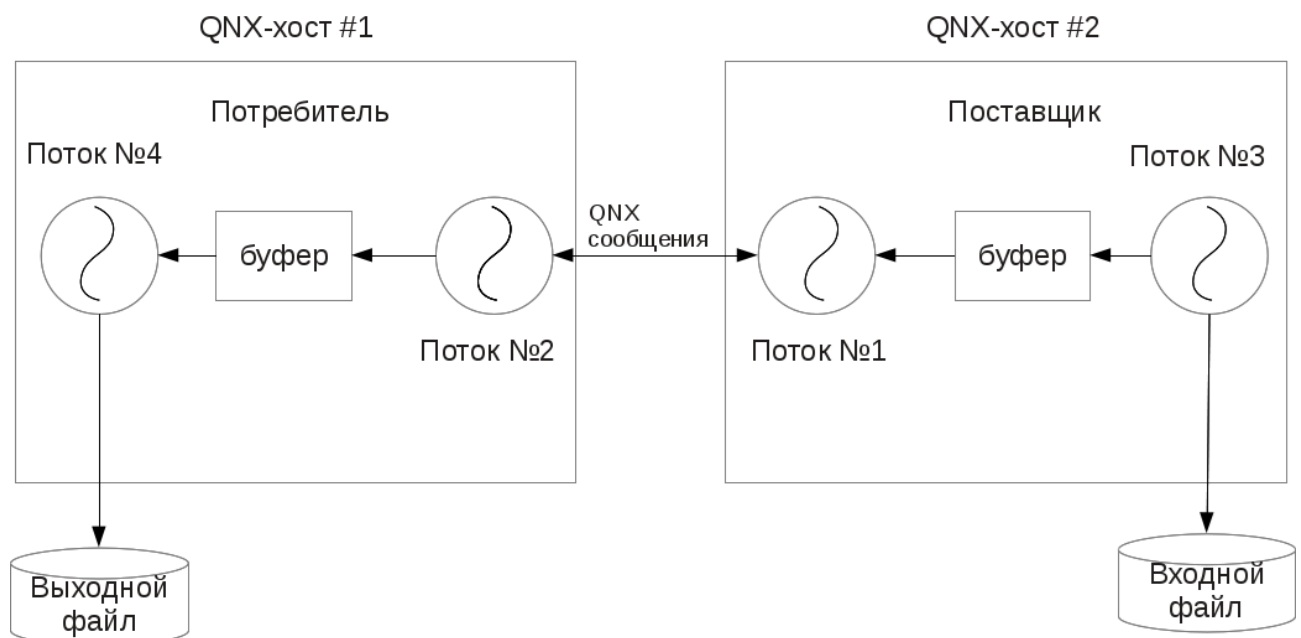
Оценка

« ____ » _____ 2017 г.

Таганрог 2017

Задание

Необходимо разработать сетевое многопоточное приложение, реализующее классическую задачу из области синхронизации процессов «поставщики-потребители» (<http://cs.mipt.ru/docs/courses/osstud/06/ch6.htm>) в распределенной среде QNX. Поставщик должен прочитать файл с любым изображением в формате BMP или TGA, повернуть его на заранее заданный угол и отправить его потребителю. При этом поставщик должен отправлять файл по частям через равные промежутки времени (100 мс). Размер части в байтах должен задаваться в параметрах приложения при запуске. Части файла должны отправляться не последовательно, а в случайном порядке (перемешиваться), но так, чтобы их последовательность можно было восстановить на принимающей стороне (можно, например, присвоить целочисленные идентификаторы). Потребитель, получив все части, должен масштабировать изображение (увеличить или уменьшить, фактор задается в параметрах приложения при запуске). Приложение должно быть организовано следующим образом:



- поток №1 — периодически выбирает из буфера часть и отправляет по сети потоку №2;
- поток №3 — считывает входной файл, поворачивает изображение и разбивает его на части, далее помещая их в буфер;

- поток №2 — принимает части изображения по сети и кладет их в буфер;
- поток №4 — берет из буфера все части, составляет изображение, масштабирует его и сохраняет в файл;
- примечание: а) размер буферов задается константой, б) в случае переполнения буфера поток №3 блокируется, возобновляя работу при наличии места в буфере; в) алгоритм(алгоритмы) обработки изображений может быть любым (например, аффинные преобразования);
- т. к. буфер потребителя также конечного размера, то поток №4 должен успевать обрабатывать части из буфера так, чтобы буфер не переполнялся, иначе необходимо генерировать исключительную ситуацию, с ее последующей обработкой как на стороне Потребителя, так и на стороне Поставщика;
- в случае успешной работы приложения, выходной файл должен содержать первоначальное изображение повернутое на заданный угол и корректно масштабированное (увеличенное или уменьшенное).

Описание используемых при разработке средств для создания, взаимодействия и синхронизации сетевых процессов и потоков

Для синхронизации потоков внутри одной программы используется общая память.

Для взаимодействия сетевых процессов используются сообщения:

Механизм обмена сообщениями в QNX также называют SRR-механизм, по первым буквам трёх основных функций, применяемых при обмене сообщениями. `MsgSend()` служит для отправки сообщения, `MsgReceive()` — для приёма сообщения, и `MsgReply()` — для передачи ответа вызывающей стороне.

При вызове функции `MsgSend()` клиент блокируется в одном из двух состояний: `SEND` или `REPLY`. Состояние `SEND` означает, что клиент отправил сообщение, а сервер его ещё не принял. После того, как сервер принимает сообщение, клиент переходит в состояние `REPLY`. Когда сервер вернёт сообщение с ответом, клиент разблокируется.

`MsgReceive()` служит для приёма сообщений от клиентов. Сервер вызывает `MsgReceive()` и блокируется в состоянии `RECEIVE`, если ни один из клиентов ещё не послал ему сообщение, т.е. не вызвал функцию `MsgSend()`. После того как это произошло (было передано сообщение серверу). Сервер разблокируется и продолжает своё выполнение. Серверу обычно требуется выполнить какие-то действия по обработке принятого сообщения и подготовке к приёму нового. Если сервер работает в несколько потоков, то обработку сообщения и ответ клиенту может выполнить другой поток. Чаще всего, поток принимающий сообщения работает в «вечном» цикле и после обработки принятого сообщения опять вызывает `MsgReceive()`.

`MsgReply()` используется для передачи сообщения с ответом клиенту. При вызове функции `MsgReply()` блокировка не происходит, т.е. сервер будет продолжать работать дальше. Это сделано потому, что клиент уже находится в заблокированном состоянии (`REPLY`) и не требуется какая-либо дополнительная синхронизация.

Описание структуры и алгоритмов разработанной программы

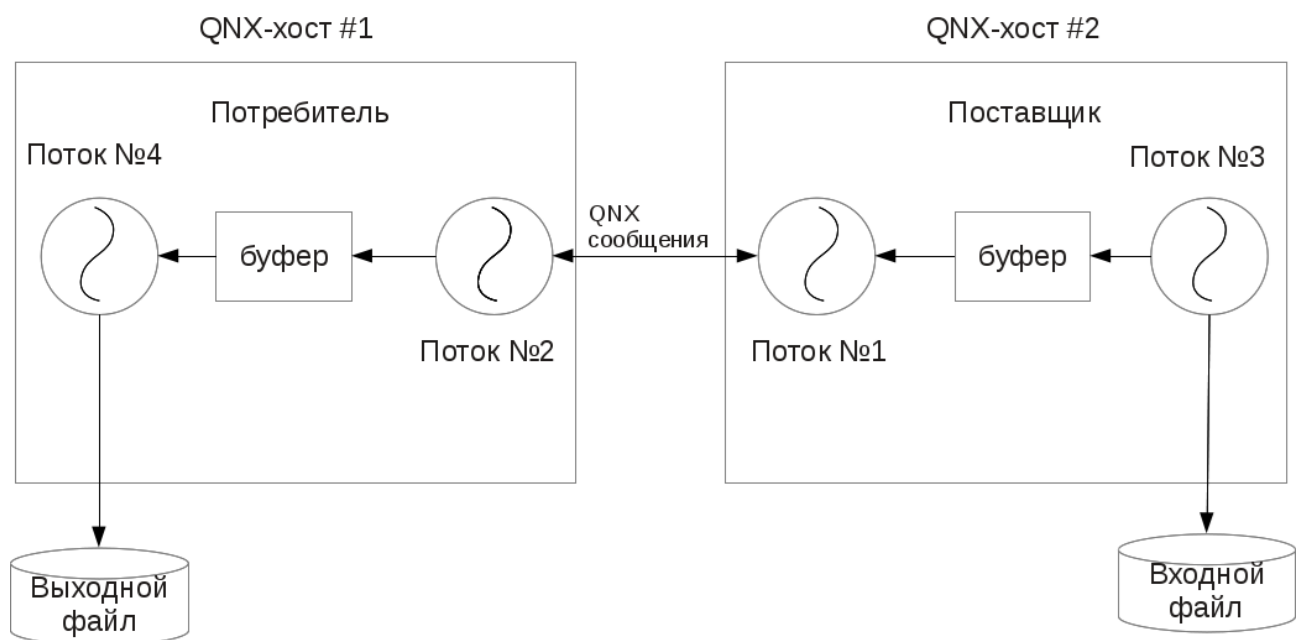
Структура передаваемого пакета через сообщения

```
1. typedef struct
2. {
3.     int id;
4.     char data[LENGTH_WORD];
5.     int length;
6.     int status;
7. } packet;
```

где `id` – идентификатор пакета, `data` – массив байт, с содержимым передаваемого файла, `length` – длина байт, значимых данных, `status` – пуст или полон пакет.

В лабораторной работе реализована двойная буферизация. Поочередно работающие для передачи данных и считывания или записи данных на диск.

Алгоритм разработанной программы полностью соответствует поставленной задаче на данной схеме:



Для обработки изображений использовалась стандартная библиотека `qnx img.h`. Были разработаны функции для поворота изображения и изменения масштаба.

Изображение хранится в структуре `img_t`:

```
typedef struct {
    union {
        struct {
            uint8      *data;
            unsigned    stride;
        } direct;

        struct {
            img_access_f *access_f;
            Uintptrt     data;
        } indirect;
    } access;
    unsigned w, h;
    img_format_t format;
    unsigned npalette;
    img_color_t *palette;
    unsigned flags;
    union {
        uint8      index;
        uint16     rgb16;
        img_color_t rgb32;
    }
    unsigned transparency;
    unsigned quality;
} img_t;
```

Цвета `img_color_t *palette` хранятся в зависимости от типа файла. Значение будет: индексный номер (0-255) для палитры на основе или `IMG_FMT_G8`, 16-битное значение для форматов 16 битном (кодирование совпадает с форматом изображения), 32-битное значение для 24 или 32 форматов BPP (кодирование `IMG_FMT_PKNE_ARGB8888`).

Для поворота изображения, была разработана функция `char* GetImageAndRotate(char* fileName, int degree)`, которая принимает на вход имя файла с изображением и угол поворота, выходные данные – это имя файла, сохраненного на диск, повернутого на определенный угол. Для поворота изображения использовалась функция:

```
#include <img.h>

int img_rotate_ortho( const img_t *src,
                    img_t *dst,
                    img_fixed_t angle );
```

Для изменения масштаба изображения, была разработана функция `int GetImageAndScale(char* file_name, int value)`, которая принимает на вход имя файла с изображением и значением с коэффициентом масштаба, выходные данные – это имя файла, сохраненного на диск, измененного масштаба. Для изменения масштаба изображения использовалась функция:

```
#include <img.h>

int img_resize_fs( const img_t *src,
                  img_t *dst );
```

Результат работы программы

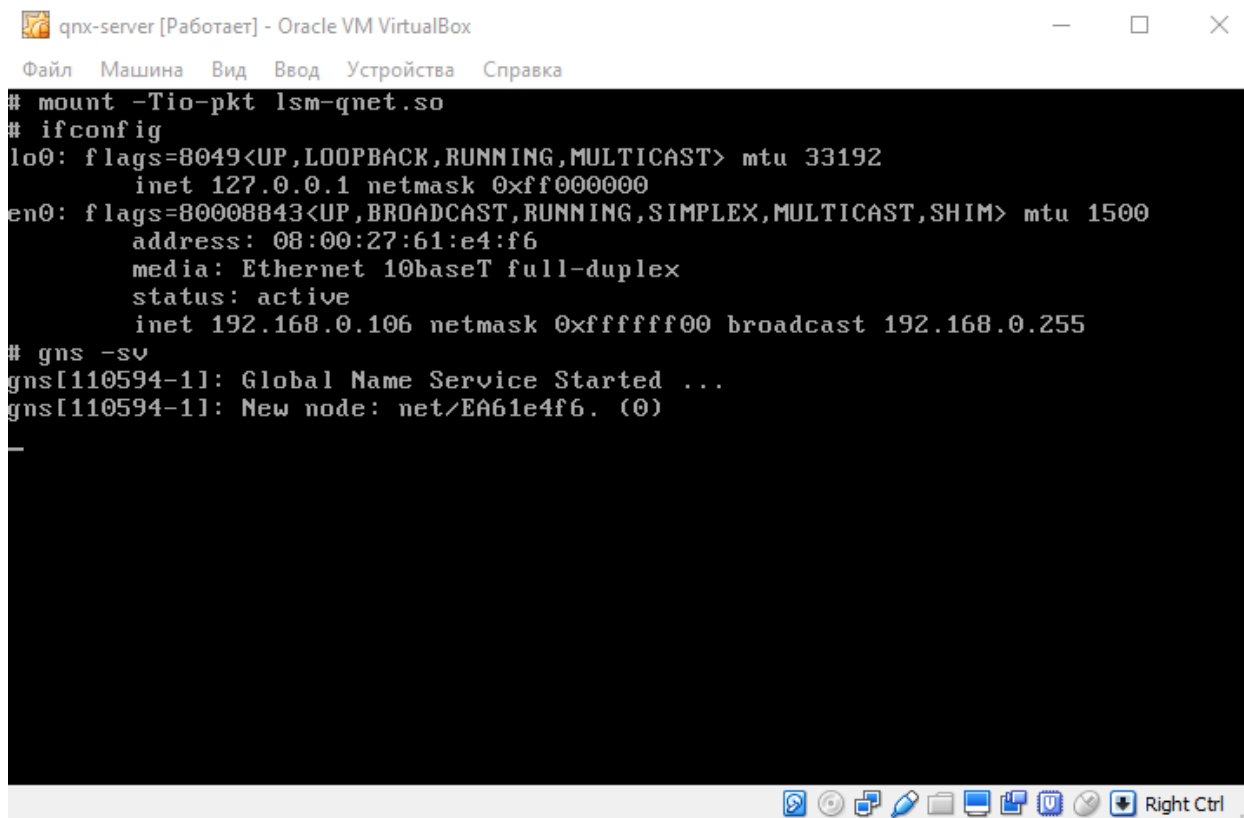
Исходное изображение (160x265, формат bmp). Угол поворота 90 градусов.



Изображение на выходе (1600x966, формат bmp). Масштаб x10.



Описание запуска QNX GNS SERVER



```
# mount -Tio-pkt lsm-qnet.so
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
    inet 127.0.0.1 netmask 0xff000000
en0: flags=80008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,SHIM> mtu 1500
    address: 08:00:27:61:e4:f6
    media: Ethernet 10baseT full-duplex
    status: active
    inet 192.168.0.106 netmask 0xffffffff broadcast 192.168.0.255
# gns -sv
gns[110594-1]: Global Name Service Started ...
gns[110594-1]: New node: net/EA61e4f6. (0)
```

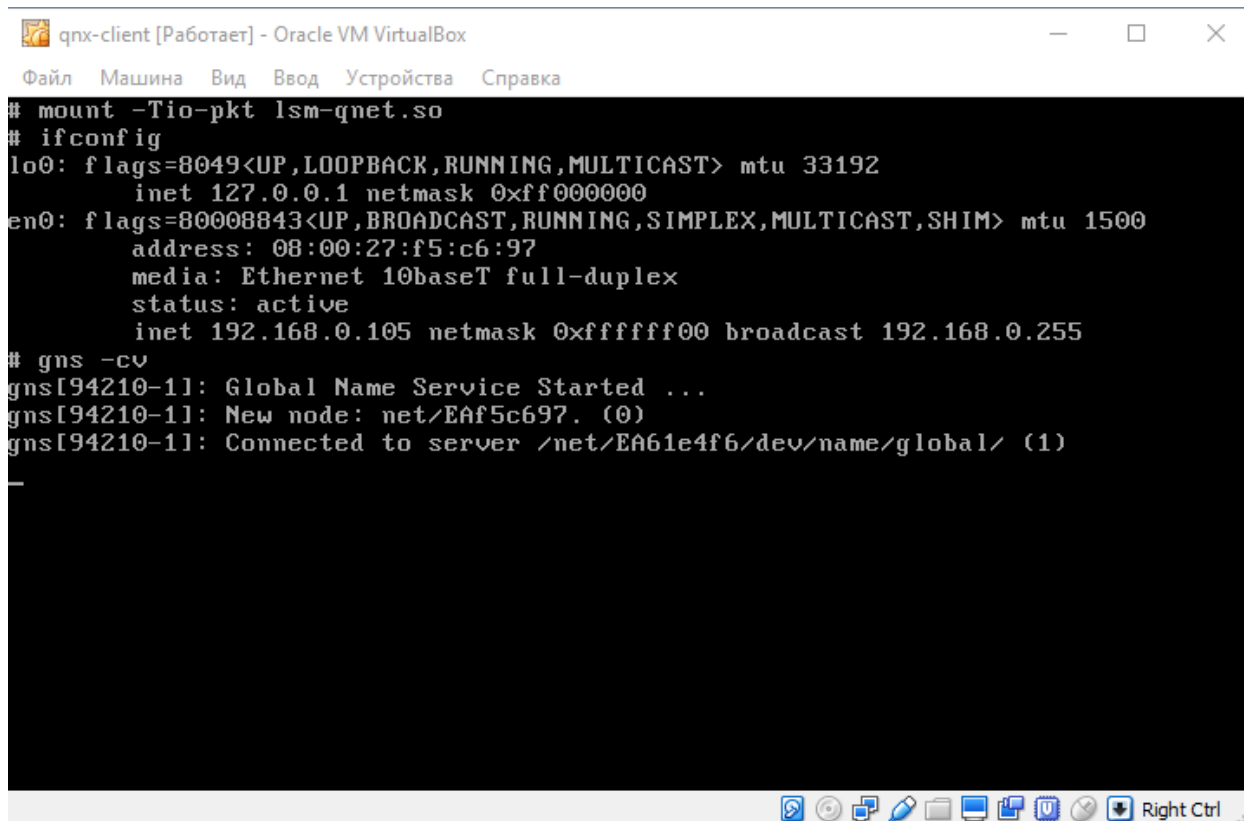
Запуск io-pkt с поддержкой протокола Qnet:

```
mount -Tio-pkt lsm-qnet.so
```

Запуск GNS (Global Name Service) в роли сервера:

```
gns -s
```

Описание запуска QNX GNS CLIENT



```
# mount -Tio-pkt lsm-qnet.so
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
    inet 127.0.0.1 netmask 0xff000000
en0: flags=80008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,SHIM> mtu 1500
    address: 08:00:27:f5:c6:97
    media: Ethernet 10baseT full-duplex
    status: active
    inet 192.168.0.105 netmask 0xfffff00 broadcast 192.168.0.255

# gns -c
gns[94210-1]: Global Name Service Started ...
gns[94210-1]: New node: net/Eaf5c697. (0)
gns[94210-1]: Connected to server /net/EA61e4f6/dev/name/global/ (1)
```

Запуск io-pkt с поддержкой протокола Qnet:

```
mount -Tio-pkt lsm-qnet.so
```

Запуск GNS (Global Name Service) в роли клиента:

```
gns -c
```

Выводы

Благодаря данной лабораторной работе, мы познакомились с операционной системой реального времени QNX, средой разработки программного обеспечения QNX Momentics IDE и средствами передачи данных по сети.

Листинг

Программа 1:

```
1. #include <stdlib.h>
2. #include <stdio.h>
3. #include <pthread.h>
4. #include <time.h>
5. #include <sys/iofunc.h>
6. #include <sys/dispatch.h>
7. #include <string.h>
8. #include <errno.h>
9. #include <time.h>
10. #include <img/img.h>
11.
12. #define LENGTH_BUFFER 16
13. #define LENGTH_WORD 2048
14. #define NAME_IN_FILE "input_2.bmp"
15. #define DEGREE 90
16.
17.
18. typedef struct
19. {
20.     int id; // Идентификатор части.
21.     char data[LENGTH_WORD]; // Содержимое.
22.     int length; // Длина содержимого (т.к. последний пакет может содержать
        меньше данных чем массив data).
23.     int status; // Статус части (занят или свободен).
24. } packet;
25.
26. typedef struct
27. {
28.     char file_name[30];
29.     long int size;
30. } info_file;
31.
32. info_file info;
33. packet buffer[2][LENGTH_BUFFER];
34.
35. int buffer_full[2] = {0, 0};
36. int select_buf = 1;
37. int exit_code = 0;
38. int end_read_file = 0;
39. char file_to_send[30];
40.
41. // Периодически (100 мс) выбирает из буфера часть и отправляет её пакетами (один пакет =
42. // одно сообщение) по сети потоку №2.
43. void* ThreadOne( void* arg )
44. {
45.     int server_coid = -1;
46.
47.     while( server_coid == -1 )
48.     {
49.         server_coid = name_open("One", NAME_FLAG_ATTACH_GLOBAL);
50.     }
51.
52.     // Буфер с каким работаем.
53.     int num_buf = 1;
```

```

54.
55.     // Указывает на конец передачи.
56.     int end_transmitted = 0;
57.
58.     // Массив индексов необходим для смешивания пакетов.
59.     int mas_index[LENGTH_BUFFER];
60.
61.     int i;
62.
63.     for (i=0; i < LENGTH_BUFFER; i++)
64.         mas_index[i] = i;
65.
66.     SendInformationMsgToClient(server_coid);
67.
68.     while(end_transmitted != 1)
69.     {
70.         switch(num_buf)
71.         {
72.             case 1:
73.                 if(buffer_full[0] == 1)
74.                 {
75.                     RandomArrayPacket(mas_index); // Перемешиваем пакеты.
76.
77.                     SendPacketToClient(mas_index, server_coid, buffer[0]);
78.
79.                     buffer_full[0] = 0; num_buf = 2;
80.                 }
81.                 if(end_read_file == 1 && buffer_full[0] == 0 && buffer_full[1] == 0)
82.                 {
83.                     end_transmitted = SendLastPacketToClient(server_coid);
84.                 }
85.                 break;
86.             case 2:
87.                 if(buffer_full[1] == 1)
88.                 {
89.                     RandomArrayPacket(mas_index);
90.
91.                     SendPacketToClient(mas_index, server_coid, buffer[1]);
92.
93.                     buffer_full[1] = 0; num_buf = 1;
94.                 }
95.                 if(end_read_file == 1 && buffer_full[1] == 0 && buffer_full[0] == 0)
96.                 {
97.                     end_transmitted = SendLastPacketToClient(server_coid);
98.                 }
99.                 break;
100.            } // EndSwitch.
101.        } // EndWhile.
102.
103.        name_close(server_coid);
104.        exit_code = -1;
105.
106.        printf("End send file, thread 1 - CLOSE\n");
107.
108.        return( 0 );
109.    }

```

```

110.
111.     // Считывает входной файл, разбивает его на части и помещает их в буфер.
112.     void* ThreadThree( void* arg )
113.     {
114.         FILE* file = fopen( file_to_send, "rb" );
115.
116.         if( file != NULL )
117.         {
118.             int i, stop = 0, id = 0;
119.
120.             char* byte[1];
121.
122.             int result_read = 1;
123.
124.             while(result_read == 1 && exit_code == EOK)
125.             {
126.                 switch(select_buf)
127.                 {
128.                     case 1:
129.                         if(buffer_full[0] == 0)
130.                         {
131.                             int free_idbuf = GetFreeIdPlaceBuffer(buffer[0]);
132.
133.                             if(free_idbuf != -1)
134.                             {
135.                                 result_read = SetDataToBuffer(file, free_idbuf,
136.                                     buffer[0], &id, byte, 0);
137.                                 else buffer_full[0] = 1;
138.                             }
139.                             break;
140.                     case 2:
141.                         if(buffer_full[1] == 0)
142.                         {
143.                             int free_idbuf = GetFreeIdPlaceBuffer(buffer[1]);
144.
145.                             if(free_idbuf != -1)
146.                             {
147.                                 result_read = SetDataToBuffer(file, free_idbuf,
148.                                     buffer[1], &id, byte, 1);
149.                                 else buffer_full[1] = 1;
150.                             }
151.                             break;
152.                         }
153.                     }
154.
155.                     fclose(file); end_read_file = 1;
156.                 }
157.
158.                 printf("End read file, thread 3 - CLOSE\n");
159.
160.                 return( 0 );
161.             }
162.
163.     char* GetImageAndRotate(char* fileName, int degree);

```

```

164.
165.     int main(int argc, char *argv[]){
166.
167.         strcat(file_to_send, GetImageAndRotate(NAME_IN_FILE,DEGREE), 30);
168.
169.         SetInfoFile(file_to_send);
170.
171.         pthread_t thread1_tid, thread3_tid;
172.
173.         pthread_create(&thread3_tid, NULL, &ThreadThree, NULL );
174.
175.         pthread_create(&thread1_tid, NULL, &ThreadOne, NULL );
176.
177.         while(exit_code == EOK)
178.         {
179.             switch(select_buf)
180.             {
181.                 case 1:
182.                     select_buf = (buffer_full[0] == 1 && buffer_full[1] == 0) ? 2 : 1;
183.                     break;
184.                 case 2:
185.                     select_buf = (buffer_full[1] == 1 && buffer_full[0] == 0) ? 1 : 2;
186.                     break;
187.             }
188.         }
189.
190.         sleep(1);
191.         printf("File has been sent!");
192.         sleep(1);
193.
194.         return EXIT_SUCCESS;
195.     }
196.
197.     char* GetImageAndRotate(char* fileName, int degree)
198.     {
199.         img_lib_t ilib = NULL;
200.
201.         img_lib_attach(&ilib);
202.
203.         int rc;
204.
205.         img_t img, imgOut;
206.
207.         /* initialize an img_t by setting its flags to 0 */
208.         img.flags = 0;
209.         img.format = IMG_FMT_PKLE_ARGB1555;
210.         img.flags |= IMG_FORMAT;
211.
212.         imgOut.flags = 0;
213.         imgOut.format = IMG_FMT_PKLE_ARGB1555;
214.         imgOut.flags |= IMG_FORMAT;
215.
216.         /* char* type;
217.
218.         switch(fileType){
219.             case 1:

```

```

220.         type = ".bmp";
221.         break;
222.     case 2:
223.         type = ".tga";
224.         break;
225.     default:
226.         return fileName;
227.         break;
228.     }
229.     */
230.
231.     char filename[30];
232.
233.     strcpy(filename, fileName);
234.
235.     if ((rc = img_load_file(ilib, filename, NULL, &img)) != IMG_ERR_OK) {
236.         fprintf(stderr, "img_load_file(%s) failed: %d\n", filename, rc);
237.         return fileName;
238.     }
239.
240.     fprintf(stdout, "Uploaded Image is %dx%dx%d (%s)\n", img.w, img.h,
        IMG_FMT_BPP(img.format), filename);
241.
242.     img_fixed_t angel = (int)((degree * 3.141592 / 180) * 65536.0);
243.
244.     if((rc = img_rotate_ortho(&img, &imgOut, angel))!= IMG_ERR_OK){
245.         fprintf(stderr, "img_rotate_ortho failed: %d\n", rc);
246.         return fileName;
247.     }
248.
249.     printf("Rotated Image\n");
250.
251.     char outfile[60];
252.
253.     strcpy(outfile, "out_");
254.
255.     strncat(outfile, filename, 30);
256.
257.     if ((rc = img_write_file(ilib, outfile, NULL, &imgOut)) != IMG_ERR_OK) {
258.         fprintf(stderr, "img_write_file(%s) failed: %d\n", outfile, rc);
259.         return fileName;
260.     }
261.
262.     printf("Writed Image (%s)\n", outfile);
263.
264.     /* for our purposes we're done with the img lib */
265.     img_lib_detach(ilib);
266.
267.     return outfile;
268. }
269.
270. int SetDataToBuffer(FILE* file, int free_idbuf, packet* buffer, int*
    id_packet, char* byte, int num_buf){
271.
272.     int i, rc = 1, stop = 0;
273.

```



```

274.     for(i=0; i<LENGTH_WORD; i++)
275.     {
276.         rc = fread(byte, sizeof(char), 1, file);
277.
278.         if(rc == 1)
279.         {
280.             buffer[free_idbuf].length = i+1;
281.             buffer[free_idbuf].data[i] = byte[0];
282.         }
283.         else{
284.             buffer[free_idbuf].status = 1;
285.             buffer[free_idbuf].id = *id_packet;
286.             buffer_full[num_buf] = 1;
287.             stop = 1;
288.             break;
289.         }
290.
291.     }
292.
293.     if(stop != 1)
294.     {
295.         buffer[free_idbuf].status = 1;
296.         buffer[free_idbuf].id = *id_packet;
297.         *id_packet = *id_packet + 1;
298.     }
299.
300.     return rc;
301. }
302.
303. int SendInformationMsgToClient(int server_coid)
304. {
305.     int reply;
306.
307.     // Отправляем пакет.
308.     MsgSend(server_coid, &info, sizeof(info), &reply, sizeof(int));
309.
310.     while(reply != EOK)
311.     {
312.         MsgSend(server_coid, &info, sizeof(info), &reply, sizeof(int));
313.     }
314.
315.     return reply;
316. }
317.
318. void SendPacketToClient(int* mas_index, int server_coid, packet* buffer ){
319.
320.     int i, reply_client;
321.
322.     for(i=0; i<LENGTH_BUFFER;i++)
323.     {
324.         if(buffer[mas_index[i]].status != 0){
325.             // Отправляем пакет.
326.             MsgSend(server_coid, &buffer[mas_index[i]], sizeof(buffer[mas_index[i]]),
&reply_client, sizeof(int));
327.
328.             while(reply_client != EOK)

```

```

329.         {
330.             MsgSend(server_coid,
331.                 &buffer[mas_index[i]], sizeof(buffer[mas_index[i]]), &reply_client, sizeof(int));
332.         }
333.         buffer[mas_index[i]].status = 0;
334.     }
335. }
336. }
337.
338. int SendLastPacketToClient(int server_coid){
339.     int reply_client;
340.
341.     packet end_packet;
342.
343.     end_packet.status = 250494;
344.
345.     MsgSend(server_coid, &end_packet, sizeof(end_packet),
346.         &reply_client, sizeof(int));
347.     /* while(reply_client != EOK)
348.     {
349.         MsgSend(server_coid, &end_packet, sizeof(end_packet), &reply_client,
350.             sizeof(int));
351.     }
352.     */
353.     return 1;
354. }
355. void SetInfoFile(char* filename){
356.     FILE* file;
357.
358.     file = fopen(filename, "rb");
359.
360.     fseek(file, 0, SEEK_END);
361.
362.     info.size = ftell(file);
363.
364.     strcpy(info.file_name, NAME_IN_FILE);
365.
366.     fclose(file);
367. }
368.
369. int GetFreeIdPlaceBuffer(packet* buffer)
370. {
371.     int i;
372.
373.     for(i = 0; i<LENGTH_BUFFER; i++)
374.     {
375.         if(buffer[i].status==0)
376.         {
377.             return i;
378.         }
379.     }
380.     return -1;
381. }

```

```
382.  
383.     void RandomArrayPacket(int* arrayIndex)  
384.     {  
385.         int i; int j, temp;  
386.  
387.         for(i = 0; i < LENGTH_BUFFER; i++)  
388.         {  
389.             j = LENGTH_BUFFER - 1;  
390.  
391.             temp = arrayIndex[i];  
392.  
393.             arrayIndex[i] = arrayIndex[j];  
394.  
395.             arrayIndex[j] = temp;  
396.         }  
397.     }
```

Программа 2:

```
1. #include <stdlib.h>
2. #include <stdio.h>
3. #include <sys/iofunc.h>
4. #include <sys/dispatch.h>
5. #include <errno.h>
6. #include <time.h>
7. #include <img/img.h>
8.
9. #define LENGTH_BUFFER 16
10. #define LENGTH_WORD 2048
11. #define SCALE 10
12.
13. #define NAME_OUT_FILE "data.dat"
14.
15. /* We specify the header as being at least a pulse */
16. typedef struct _pulse msg_header_t;
17.
18. typedef struct
19. {
20.     int id;
21.     char data[LENGTH_WORD];
22.     int length;
23.     int status;
24. } packet;
25.
26. typedef struct
27. {
28.     char file_name[30];
29.     long int size;
30. } info_file;
31.
32. info_file info;
33.
34. int threadBuf = 1;
35. int buffer1Full = 0;
36. int buffer2Full = 0;
37. int EXITCODE = 0;
38.
39. packet buffer1[LENGTH_BUFFER];
40. packet buffer2[LENGTH_BUFFER];
41.
42. void* ThreadTwo( void* arg ){
43.     msg_header_t header;
44.
45.     name_attach_t* attach;
46.
47.     int rcvid, ENDPACKET = 0;
48.
49.     if (!( attach = name_attach(NULL, "One", NAME_FLAG_ATTACH_GLOBAL) ) )
50.     {
51.         EXITCODE = ECONNREFUSED;
52.         return EXIT_FAILURE;
53.     }
54.
55.     rcvid = MsgReceive(attach->chid, &header, sizeof(header), NULL);
```

```

56.
57.     if (rcvid == -1)
58.     {
59.         ConnectDetach(header.scoid);
60.         EXITCODE = ECONNREFUSED;
61.         return EXIT_FAILURE;
62.     }
63.
64.     // name_open() sends a connect message, must EOK this.
65.     if (header.type == _IO_CONNECT ) {
66.         MsgReply( rcvid, EOK, NULL, 0 );
67.     }
68.
69.     GetInformationMsgFromServer(&rcvid, attach->chid);
70.
71.     while(ENDPACKET == 0)
72.     {
73.         switch(threadBuf)
74.         {
75.             case 1:
76.                 if(buffer1Full == 0)
77.                 {
78.                     GetDataFromServer(attach->chid, &ENDPACKET, buffer1 );
79.
80.                     buffer1Full = 1;
81.                 }
82.                 break;
83.             case 2:
84.                 if(buffer2Full == 0)
85.                 {
86.                     GetDataFromServer(attach->chid, &ENDPACKET, buffer2 );
87.
88.                     buffer2Full = 1;
89.                 }
90.                 break;
91.         }
92.     }
93.
94.     /* Remove the name from the space */
95.     name_detach(attach, 0);
96.     return(0);
97. }
98.
99. void* ThreadFour( void* arg ){
100.     FILE *file;
101.     file = fopen(NAME_OUT_FILE, "wb");
102.     fclose(file);
103.
104.     clock_t ca, ce;
105.
106.     int numBuf = 1;
107.
108.     ca = clock();
109.
110.     while(EXITCODE == 0)
111.     {

```

```

112.         switch(numBuf)
113.         {
114.             case 1:
115.                 if(buffer1Full == 1)
116.                 {
117.                     int i;
118.
119.                     file = fopen(NAME_OUT_FILE,"ab");
120.
121.                     int x = ReturnIndexMinIdInBuffer(buffer1);
122.
123.                     if(x == -2){
124.                         EXITCODE = 1;
125.                         fclose(file);
126.                         buffer1Full = 0;
127.                         numBuf = 2;
128.                         break;
129.                     }
130.
131.                     fwrite(buffer1[x].data, sizeof(char),buffer1[x].length, file );
132.                     buffer1[x].status = 0;
133.
134.                     for(i=0; i<LENGTH_BUFFER - 1;i++)
135.                     {
136.                         x = FindNextIndexId(buffer1[x].id,buffer1);
137.
138.                         switch(x)
139.                         {
140.                             case -1:
141.                                 break;
142.                             case -2:
143.                                 EXITCODE = 1;
144.                                 break;
145.                             default:
146.                                 fwrite(buffer1[x].data, sizeof(char),buffer1[x].length,
147.                                     file );
148.                                 buffer1[x].status = 0;
149.                                 break;
150.                         }
151.
152.                         fseek(file,0,SEEK_END);
153.                         printf("%ld / %ld byte \n", ftell(file), info.size);
154.
155.                         fclose(file);
156.                         buffer1Full = 0;
157.                         numBuf = 2;
158.                     }
159.                     break;
160.             case 2:
161.                 if(buffer2Full == 1)
162.                 {
163.                     int i;
164.
165.                     file = fopen(NAME_OUT_FILE,"ab");
166.

```

```

167.             int x = ReturnIndexMinIdInBuffer(buffer2);
168.
169.             if(x==-2){
170.                 EXITCODE = 1;
171.                 fclose(file);
172.                 buffer2Full = 0;
173.                 numBuf = 1;
174.                 break;
175.             }
176.
177.             fwrite(buffer2[x].data, sizeof(char),buffer2[x].length, file );
178.             buffer2[x].status = 0;
179.
180.             for(i=0; i<LENGTH_BUFFER - 1;i++)
181.             {
182.                 x = FindNextIndexId(buffer2[x].id,buffer2);
183.
184.                 switch(x)
185.                 {
186.                     case -1:
187.                         break;
188.                     case -2:
189.                         EXITCODE = 1;
190.                         break;
191.                     default:
192.                         fwrite(buffer2[x].data, sizeof(char),buffer2[x].length, file );
193.                         buffer2[x].status = 0;
194.                         break;
195.                 }
196.             }
197.
198.             fseek(file,0,SEEK_END);
199.             printf("%ld / %ld byte \n", ftell(file), info.size);
200.
201.             fclose(file);
202.             buffer2Full = 0;
203.             numBuf = 1;
204.         }
205.         break;
206.     }
207. }
208.
209. ce = clock();
210.
211. printf("time spent on CPU: %lf sec\n", (ce - ca)/(double)CLOCKS_PER_SEC);
212.
213. GetImageAndScale(info.file_name, SCALE);
214.
215. EXITCODE = -1;
216. return(0);
217. }
218.
219. int main(int argc, char *argv[]) {
220.
221.     pthread_t thread2_tid, thread4_tid;

```

```

222.
223.     pthread_create(&thread2_tid, NULL, &ThreadTwo, NULL );
224.
225.     pthread_create(&thread4_tid, NULL, &ThreadFour, NULL );
226.
227.     while(EXITCODE != -1)
228.     {
229.         switch(threadBuf)
230.         {
231.             case 1:
232.                 threadBuf = (buffer1Full == 1 && buffer2Full == 0) ? 2 : 1;
233.                 break;
234.             case 2:
235.                 threadBuf = (buffer2Full == 1 && buffer1Full == 0) ? 1 : 2;
236.                 break;
237.         }
238.     }
239.
240.     RenameInputFile(info.file_name);
241.
242.     sleep(1);
243.     printf("File is received successfully!\n");
244.
245.     return EXIT_SUCCESS;
246. }
247.
248. int GetImageAndScale(char* file_name, int value){
249.     img_lib_t ilib = NULL;
250.
251.     img_lib_attach(&ilib);
252.
253.     int rc;
254.
255.     img_t img, imgOut;
256.
257.     /* initialize an img_t by setting its flags to 0 */
258.     img.flags = 0;
259.     img.format = IMG_FMT_PKLE_ARGB1555;
260.     img.flags |= IMG_FORMAT;
261.
262.     imgOut.flags = IMG_W;
263.     imgOut.format = IMG_FMT_PKLE_ARGB1555;
264.
265.     char filename[30];
266.
267.     strcpy(filename, file_name);
268.
269.     if ((rc = img_load_file(ilib, filename, NULL, &img)) != IMG_ERR_OK) {
270.         fprintf(stderr, "img_load_file(%s) failed: %d\n", filename, rc);
271.         return -2;
272.     }
273.
274.     if(value != 0){
275.         if(value < 0){
276.             imgOut.h = img.h / (value * -1);
277.             imgOut.w = img.w / (value * -1);

```



```

278.         }
279.         else{
280.             imgOut.h = img.h * value;
281.             imgOut.w = img.h * value;
282.         }
283.     } else {
284.         imgOut.h = img.h;
285.         imgOut.w = img.w;
286.     }
287.
288.     if ((rc = img_resize_fs(&img, &imgOut)) != IMG_ERR_OK) {
289.         fprintf(stderr, "img_resize_fs failed: %d\n", rc);
290.         return -3;
291.     }
292.
293.     char outfile[60];
294.
295.     strcpy(outfile, "scale_");
296.
297.     strncat(outfile, file_name, 30);
298.
299.     if ((rc = img_write_file(ilib, outfile, NULL, &imgOut)) != IMG_ERR_OK) {
300.         fprintf(stderr, "img_write_file(%s) failed: %d\n", outfile, rc);
301.         return -4;
302.     }
303.
304.     printf("Writed Image (%s)\n", outfile);
305.
306.     /* for our purposes we're done with the img lib */
307.     img_lib_detach(ilib);
308.
309.     return 0;
310. }
311.
312. void GetDataFromServer(int chid, int* end_packet, packet* buffer_r ){
313.     int i, rcvid;
314.
315.     packet buffer;
316.
317.     for(i=0; i<LENGTH_BUFFER; i++)
318.     {
319.         rcvid = MsgReceive(chid, &buffer, sizeof(buffer), NULL);
320.
321.         if(buffer.status == 250494)
322.         {
323.             *end_packet      = 1;
324.             buffer_r[i].status = buffer.status;
325.             MsgReply( rcvid, EOK, NULL, 0 );
326.             break;
327.         }
328.
329.         if(buffer.status == 1)
330.         {
331.             buffer_r[i].id      = buffer.id;
332.             buffer_r[i].length = buffer.length;
333.             buffer_r[i].status = buffer.status;

```

```

334.         int j;
335.         for(j=0; j<LENGTH_WORD; j++){
336.             buffer_r[i].data[j] = buffer.data[j];
337.         }
338.     }
339.
340.     MsgReply( rcvid, EOK, NULL, 0 );
341. }
342. }
343.
344. void GetInformationMsgFromServer(int* rcvid, int chid){
345.     *rcvid = MsgReceive(chid, &info, sizeof(info), NULL);
346.     MsgReply( *rcvid, EOK, NULL, 0 );
347. }
348.
349. void RenameInputFile(char* file_name)
350. {
351.     rename(NAME_OUT_FILE, file_name);
352. }
353.
354. int ReturnIndexMinIdInBuffer(packet* src)
355. {
356.     int i, j = -1;
357.
358.     packet tmp;
359.
360.     for(i=0; i<LENGTH_BUFFER; i++){
361.         if(src[i].status == 1){
362.             tmp = src[i];
363.             j = i;
364.             break;
365.         }
366.     }
367.
368.     if(j == -1){
369.         for(i=0; i<LENGTH_BUFFER; i++){
370.             if(src[i].status == 250494) return -2;
371.         }
372.
373.         return j;
374.     }
375.
376.
377.     for (i = 0; i < LENGTH_BUFFER; i++)
378.     {
379.         if (src[i].status == 1 && tmp.id > src[i].id ){
380.             tmp=src[i];
381.             j=i;
382.         }
383.     }
384.
385.     return j;
386. }
387.
388. int FindNextIndexId(int id, packet* src)
389. {

```

```
390.     int i = -1;
391.
392.     for(i=0; i<LENGTH_BUFFER; i++){
393.         if(src[i].status == 1 && src[i].id == id + 1) return i;
394.     }
395.
396.     for(i=0; i<LENGTH_BUFFER; i++){
397.         if(src[i].status == 250494) return -2;
398.     }
399.     return i;
400. }
```