Московский авиационный институт
(Национальный исследовательский университет)
Факультет "Информационные технологии и прикладная математика"
Кафедра "Вычислительная математика и программирование"

Лабораторная работа №2 по курсу “Операционные системы”

Студент: Кочкожаров Иван Вячеславович

Группа: М8О-208Б-22

Преподаватель: Миронов Евгений Сергеевич

Вариант: 13

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2023

Содержание

1 Репозиторий **3**

2 Цель работы **3**

3 Задание **3**

4 Описание работы программы **3**

5 Исходный код **4**

6 Тесты **10**

7 Консоль **11**

8 Запуск тестов **12**

9 Выводы **13**

1 Репозиторий

https://github.com/kochkozharov/os-labs

2 Цель работы

Приобретение практических навыков в:

- Управлении потоками в ОС
- Обеспечении синхронизации между потоками

3 Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

4 Описание работы программы

Необходимо было написать программу для наложения матрицы свертки на изображение. Изображение разбивается на массивы равной длинны и каждый процесс применяет матрицу свертки к своему участку. После каждой итерации происходит синхронизация по барьеру во избежание проблем с граничными пикселями В ходе выполнения лабораторной работы использованы следующие системные вызовы:

- pthread_create() - создание потока
- pthread_join() - ожидание завершения потока

5 Исходный код

blur.c

```
#include "blur.h"

#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static void Convolution(const Image* img, size_t idx, const Kernel* ker,
                        uc (*out)[]) {
    uc(*imgMat)[img->width] = img->matrix;
    const int(*kerMat)[ker->order] = ker->matrix;
    uc(*outMat)[img->width] = out;
    const size_t absI = idx / img->width * img->channels;
    const size_t absJ = idx % img->width * img->channels;
    size_t sum[MAX_CHANNELS] = {0};

    for (int i = -(ker->order / 2); i <= ker->order / 2; ++i) {
        for (int j = -ker->order / 2; j <= ker->order / 2; ++j) {
            const int kerI = i + ker->order / 2;
            const int kerJ = j + ker->order / 2;

            size_t imgI;
            if (i < 0 && (size_t)-i * img->channels > absI) {
                imgI = 0;
            } else {
                imgI = (size_t)i * img->channels + absI;
                if (imgI > (img->height - 1) * img->channels) {
                    imgI = (img->height - 1) * img->channels;
                }
            }

            size_t imgJ;
            if (j < 0 && (size_t)-j * img->channels > absJ) {
                imgJ = 0;
            } else {
                imgJ = (size_t)j * img->channels + absJ;
                if (imgJ > (img->width - 1) * img->channels) {
                    imgJ = (img->width - 1) * img->channels;
                }
            }

            int coef = kerMat[kerI][kerJ];
            for (int k = 0; k < img->channels; ++k) {
                sum[k] += (size_t)coef * imgMat[imgI][imgJ + k];
            }
        }
    }
    for (int k = 0; k < img->channels; ++k) {
        outMat[absI][absJ + k] = sum[k] / ker->divCoef;
```

```
52          }
53      }
54      static void* ChunkConvolution(void* ptr) {
55          ThreadArgs* arg = ptr;
56          uc(*buf)[] = arg->buf;
57          for (int iteration = 0; iteration < arg->times; ++
    iteration) {
58              for (size_t i = arg->begin; i < arg->end; ++i) {
59                  Convolution(arg->img, i, arg->ker, buf);
60              }
61              int status = pthread_barrier_wait(arg->barrier);
62              if (status != 0 && status !=
    PTHREAD_BARRIER_SERIAL_THREAD) {
63                  perror("pthread_barrier_wait");
64                  exit(EXIT_FAILURE);
65              }
66              uc(*temp)[] = arg->img->matrix;
67              arg->img->matrix = buf;
68              buf = temp;
69          }
70          return NULL;
71      }
72
73      const uc* ApplyKernel(Image* img, const Kernel* kernel, int k,
        uc (*buffer)[], unsigned long threadsNum) {
74          assert(kernel->order % 2 == 1 && img->channels <=
    MAX_CHANNELS);
75          int status;
76          size_t matrixSize = img->height * img->width;
77          pthread_barrier_t barrier;
78          status = pthread_barrier_init(&barrier, NULL, threadsNum);
79          if (status != 0) {
80              perror("pthread_barrier_init");
81              exit(status);
82          }
83
84          pthread_t *threads=malloc(threadsNum*sizeof(pthread_t));
85          ThreadArgs *args=malloc(threadsNum*sizeof(ThreadArgs));
86          size_t pixelsPerThread = matrixSize / threadsNum;
87
88          for (unsigned long i = 0; i < threadsNum; ++i) {
89              size_t begin = i * pixelsPerThread;
90              size_t end = i == threadsNum - 1 ? matrixSize : begin
    + pixelsPerThread;
91              args[i] = (ThreadArgs){.img = img,
92                                     .begin = begin,
93                                     .end = end,
94                                     .ker = kernel,
95                                     .buf = buffer,
96                                     .times = k,
97                                     .barrier = &barrier};
98              status = pthread_create(&threads[i], NULL,
    ChunkConvolution, &args[i]);
99              if (status != 0) {
100                 perror("pthread_create");
101                 exit(status);
102             }
103         }
104         for (unsigned long i = 0; i < threadsNum; ++i) {
```

```
105             pthread_join(threads[i], NULL);
106         }
107 
108         pthread_barrier_destroy(&barrier);
109         free(threads);
110         free(args);
111         return (uc*)(k % 2 == 1 ? buffer : img->matrix);
112     }
```

main.c

```
1     #include <getopt.h>
2 #define STB_IMAGE_IMPLEMENTATION
3 #include <stb/stb_image.h>
4 #define STB_IMAGE_WRITE_IMPLEMENTATION
5 #include <stb/stb_image_write.h>
6 #include <time.h>
7 
8 #include "blur.h"
9 
10 //#define TEST
11 
12 typedef enum { gauss, box } TFilter;
13 
14 static const Kernel GAUSSIAN5 = {
15     .matrix =
16         (const int[5][5]){
17             {1, 4, 6, 4, 1},
18             {4, 16, 24, 16, 4},
19             {6, 24, 36, 24, 6},
20             {4, 16, 24, 16, 4},
21             {1, 4, 6, 4, 1},
22         },
23     .order = 5,
24     .divCoef = 256,
25 };
26 
27 static const Kernel BOX3 = {
28     .matrix =
29         (const int[3][3]){
30             {1, 1, 1},
31             {1, 1, 1},
32             {1, 1, 1},
33         },
34     .order = 3,
35     .divCoef = 9,
36 };
37 
38 int main(int argc, char *argv[]) {
39 #ifdef TEST
40     (void)BOX3;
41     if (argc < 2) {
42         fprintf(stderr,
43                 "Usage: blur INPUT_FNAME\n");
44         exit(EXIT_SUCCESS);
45     }
46 #else
47     if (argc < 3 || strcmp(argv[1], "--help") == 0) {
48         fprintf(stderr,
49                 "Usage: blur INPUT_FNAME OUTPUT_FNAME -f FILTER -k
```

```
    K (apply "
50                  "filter FILTER K times)\n");
51          exit(EXIT_SUCCESS);
52      }
53
54      long times = 1;
55      unsigned long threadsNum = DEF_THREAD_NUM;
56      TFilter filter = gauss;
57      const char *filterName;
58      for (int opt; opt = getopt(argc, argv, "f:k:t:r"), opt != -1;)
        {
59          switch (opt) {
60              case '?':
61                  perror("getopt");
62                  exit(EXIT_FAILURE);
63              case 'f':
64                  if (strcmp(optarg, "box") == 0) {
65                      filter = box;
66                  }
67                  break;
68              case 'k': {
69                  char *end;
70                  times = strtol(optarg, &end, 10);
71                  break;
72              }
73              case 't': {
74                  char *end;
75                  threadsNum = strtol(optarg, &end, 10);
76                  break;
77              }
78          }
79      }
80  #endif
81
82      int width;
83      int height;
84      int channels;
85
86  #ifndef TEST
87      stbi_uc *img = stbi_load(argv[argc - 2], &width, &height, &
        channels, 0);
88  #else
89      stbi_uc *img = stbi_load(argv[1], &width, &height, &channels,
        0);
90  #endif
91
92      if (img == NULL) {
93          perror("stbi_load");
94          exit(EXIT_FAILURE);
95      }
96      printf("Loaded. x: %dpx y: %dpx channels: %d.\n", width,
        height, channels);
97
98      size_t imgSize = (size_t)width * height * channels;
99      stbi_uc *buf = malloc(imgSize);
100     if (!buf) {
101         perror("malloc");
102         exit(EXIT_FAILURE);
103     }
```

```
104
105     const Kernel *ker = &GAUSSIAN5;
106
107 #ifdef TEST
108     const stbi_uc *weakPtr;
109     for (int threads = 1; threads < 21; ++threads) {
110         printf("Applying gaussian blur 20 times on %d threads\n",
    threads);
111 #else
112     if (filter == box) {
113         ker = &BOX3;
114         filterName = "box";
115     } else {
116         filterName = "gaussian";
117     }
118     printf("Applying %s blur %ld times on %ld threads\n",
    filterName, times,
119            threadsNum);
120 #endif
121
122         struct timespec start;
123         struct timespec finish;
124         clock_gettime(CLOCK_MONOTONIC, &start);
125 #ifdef TEST
126         weakPtr = ApplyKernel(&(Image){.matrix = (stbi_uc(*)[])img
    ,
127                                        .width = width,
128                                        .height = height,
129                                        .channels = channels},
130                               ker, 20, (stbi_uc(*)[])buf, threads)
    ;
131 #else
132     const stbi_uc *weakPtr =
133         ApplyKernel(&(Image){.matrix = (stbi_uc(*)[])img,
134                              .width = width,
135                              .height = height,
136                              .channels = channels},
137                     ker, (int)times, (stbi_uc(*)[])buf, threadsNum
    );
138 #endif
139         clock_gettime(CLOCK_MONOTONIC, &finish);
140         double elapsed;
141         elapsed = (finish.tv_sec - start.tv_sec);
142         elapsed += (finish.tv_nsec - start.tv_nsec) / 1.0E9;
143         printf("Function took %fs to execute\n", elapsed);
144
145 #ifdef TEST
146     }
147     printf("Test end\n");
148 #endif
149
150 #ifdef TEST
151     int status =
152         stbi_write_jpg("result.jpg", width, height, channels,
    weakPtr, 100);
153 #else
154     int status =
155         stbi_write_jpg(argv[argc - 1], width, height, channels,
    weakPtr, 100);
```

```
156 #endif
157
158     if (status == 1) {
159         printf("Successfully written %zu bytes\n",
160                (size_t)width * height * channels);
161     } else {
162         perror("stbi_write_jpg");
163         exit(EXIT_FAILURE);
164     }
165
166     stbi_image_free(img);
167     free(buf);
168     return 0;
169 }
```

6 Тесты

```
#include <gtest/gtest.h>

extern "C" {
#include "blur.h"
}

TEST(SecondLabTests, SimpleTest) {
    const size_t width = 4;
    const size_t height = 2;
    const uc example[height][width] = {{1, 2, 3, 4}, {1, 2, 3, 4}};
    uc imgMat[height][width];
    memcpy(imgMat, example, width * height);
    Image img = {reinterpret_cast<uc(*)[]>(imgMat), width, height, 1};
    const int kerMat[3][3] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
    const Kernel ker = {reinterpret_cast<const int(*)[]>(kerMat), 3, 9};
    uc buf[height][width];
    const uc excpectedRes[height][width] = {{1, 2, 3, 3}, {1, 2, 3, 3}};
    const uc *weakPtr = ApplyKernel(&img, &ker, 1, reinterpret_cast<uc(*)[]>(buf), 1);
    for (unsigned int i = 0; i < height; ++i) {
        for (unsigned int j = 0; j < width; ++j) {
            uc c = *(weakPtr + i * width + j);
            uc rc = excpectedRes[i][j];
            EXPECT_EQ(c, rc);
        }
    }
    memcpy(imgMat, example, width * height);
    img.matrix = reinterpret_cast<uc(*)[]>(imgMat);
    weakPtr = ApplyKernel(&img, &ker, 1, reinterpret_cast<uc(*)[]>(buf), 4);
    for (unsigned int i = 0; i < height; ++i) {
        for (unsigned int j = 0; j < width; ++j) {
            uc c = *(weakPtr + i * width + j);
            uc rc = excpectedRes[i][j];
            EXPECT_EQ(c, rc);
        }
    }
}
```

7 Консоль

```
ivan@asus-vivobook ~/c/o/b/lab2 (reports)> ./blur 1024px-Lenna.png result.jpg -k 10
Loaded. x: 1024px y: 1024px channels: 3.
Applying gaussian blur 10 times on 12 threads
Function took 0.635936s to execute
Successfully written 3145728 bytes
```

8 Запуск тестов

```
Running main() from /var/tmp/portage/dev-cpp/gtest-1.13.0/work/googletest-1.13.0/g
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from SecondLabTests
[ RUN      ] SecondLabTests.SimpleTest
[       OK ] SecondLabTests.SimpleTest (1 ms)
[----------] 1 test from SecondLabTests (1 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (1 ms total)
[  PASSED  ] 1 test.
```

Ускорение:

$$S_4 = \frac{T_1}{T_4} < 4 \tag{1}$$

$$S_4 = \frac{7.323169}{1.848535} = 3.96 < 4 \tag{2}$$

Эффективность:

$$X_4 = \frac{S_4}{4} < 1 \tag{3}$$

$$X_4 = \frac{3.96}{4} = 0.99 < 1 \tag{4}$$

9 Выводы

В результате выполнения данной лабораторной работы была написана программа на языке C для наложения матричных фильтров на изображения, обрабатывающая данные в многопоточном режиме. Приобретены практические навыки в управлении потоками в ОС и обеспечении синхронизации между потоками.