МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Отчет по лабораторной работе №2 дисциплины «Операционные системы»

Выполнили студенты группы ИВТ-42 _____/Рзаев А. Э./
Проверил преподаватель_____/Карабааева О. В./

Киров 2019

1 Задание

Вариант 27

1) С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток записывает в файл введенные пользователем символы, кроме первого и последнего. Второй поток записывает в файл текущее время и количество символов в файле.

2) Задачу о курильщиках решить с помощью мьютекса.

2 Исходный код

2.1 Задание 1

main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QtConcurrent>

namespace Ui {
class Widget;
}

class Widget : public QWidget {
  Q_OBJECT

public:
  explicit Widget(QWidget *parent = nullptr);
  ~Widget();

private:
  Ui::Widget *ui;

  QFuture<void> t1;

  QFuture<void> t2;
};

#endif // WIDGET_H
```

widget.cpp

```
#include <algorithm>
#include <atomic>
#include <ctime>
#include <fstream>
#include <random>
#include <string>
#include <type_traits>

#include <QDateTime>
#include <QDebug>
#include <QFile>
#include <QString>
#include <QtConcurrent>

#include "ui_widget.h"
#include "widget.h"

template <class I, typename = std::enable_if_t<std::is_integral_v<I>>>
inline I randRange(I a, I b) {
  if (a > b) {
    std::swap(a, b);
  }
  static std::mt19937 e(static_cast<unsigned int>(std::time(nullptr)));
  std::uniform_int_distribution<I> dist(a, b);
  return dist(e);
}

class PetersonMutex {
private:
  std::atomic_bool wants[2];

  std::atomic_ullong waiting;

public:
  PetersonMutex() {
    wants[0] = wants[1] = false;
    waiting = 0;
  }

  void lock(size_t id) {
    wants[id] = true;
    size_t other = 1u - id;
    waiting = id;

    while (wants[other].load() && waiting.load() == id) {
    }
  }

  void unlock(size_t id) { wants[id] = false; }
};

static PetersonMutex mutex;

static unsigned int timeout{2000}, timeout2{2000};

static bool needstostop;

static QString path("output.txt");
```

```
static void proc1() {
  while (!needstostop) {
    size_t len = randRange(40u, 50u);
    std::string str;
    str.reserve(len);

    for (size_t i = 0; i < len; ++i) {
      char c = static_cast<char>(randRange(65, 122));
      str.push_back(c);
    }

    mutex.lock(0);

    std::ofstream out(path.toStdString(),
                      std::ios_base::out | std::ios_base::app);
    qDebug() << QString("Writing: %1").arg(QString::fromStdString(str));
    out << str << std::endl;
    out.flush();
    out.close();

    mutex.unlock(0);

    QThread::msleep(timeout);
  }
}

static void proc2() {
  while (!needstostop) {
    QFile file(path);
    auto size = file.size();

    mutex.lock(1);

    std::ofstream out(path.toStdString(),
                      std::ios_base::out | std::ios_base::app);
    auto str =
        QString("%1; %2 bytes")
            .arg(QDateTime::currentDateTime().toString("dd.MM.yyyy, HH:mm:ss"))
            .arg(size);
    qDebug() << QString("Writing: %1").arg(str);
    out << str.toStdString() << std::endl;
    out.flush();
    out.close();

    mutex.unlock(1);

    QThread::msleep(timeout2);
  }
}

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget) {
  ui->setupUi(this);

  connect(ui->slider, &QSlider::valueChanged, this, [](int value) {
    auto value_u = static_cast<unsigned int>(value);
    unsigned int time = 300 + 2700 * value_u / 100;
    timeout = time;
  });
  connect(ui->slider2, &QSlider::valueChanged, this, [](int value) {
    auto value_u = static_cast<unsigned int>(value);
    unsigned int time = 300 + 2700 * value_u / 100;
```

```
    timeout2 = time;
  });
  ui->slider->setValue(60);
  ui->slider2->setValue(60);

  needstostop = false;

  // Clean up file
  std::ofstream out(path.toStdString());
  out.flush();
  out.close();

  t1 = QtConcurrent::run(&proc1);
  t2 = QtConcurrent::run(&proc2);
}

Widget::~Widget() {
  delete ui;
  needstostop = true;
  if (t1.isRunning()) {
    t1.waitForFinished();
  }
  if (t2.isRunning()) {
    t2.waitForFinished();
  }
}
```

2.2 Задание 2

main.cpp

```
#include <cstdlib>
#include <ctime>

#include <QApplication>

#include "widget.h"

int main(int argc, char *argv[]) {
  srand(0);

  QApplication a(argc, argv);
  Widget w;
  w.show();

  return a.exec();
}
```

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <cstdint>
#include <map>

#include <QPixmap>
#include <QString>
#include <QTimer>
#include <QWidget>
#include <QtConcurrent>
```

```
#define SCENE_HEIGHT 300
#define SCENE_WIDTH 500
#define TEXT_HEIGHT 25
#define TEXT_WIDTH 85
#define PIXMAP_SIZE 32
#define HAT_SIZE 50
#define OFFSET 10
#define MAX_TIMESLICE 4000
#define SMOKING_TIMESLICE 800
#define EMITTER_TIMESLICE 600
#define WAITING_TIMESLICE 247
#define RENDER_TIMEOUT 50

#define SMOKING 1
#define WAITING 0
#define NONE 2

namespace Ui {
class Widget;
}

enum class SmokeItemTypes : size_t {
  Match = 0,
  Paper = 1,
  Tobacco = 2,
  None = 3
};

class Widget : public QWidget {
  Q_OBJECT

public:
  explicit Widget(QWidget *parent = nullptr);

  ~Widget();

signals:
  void stateChanged(int, int);

  void workFinished();

private:
  Ui::Widget *ui;

  QThreadPool pool;

  QTimer timer;

  QMetaObject::Connection connection;

  std::map<SmokeItemTypes, QPixmap> pixmaps = {
      {SmokeItemTypes::Match,
       QPixmap(":/rc/pictures/match.png").scaled(PIXMAP_SIZE, PIXMAP_SIZE)},
      {SmokeItemTypes::Paper,
       QPixmap(":/rc/pictures/paper.png").scaled(PIXMAP_SIZE, PIXMAP_SIZE)},
      {SmokeItemTypes::Tobacco,
       QPixmap(":/rc/pictures/tobacco.png").scaled(PIXMAP_SIZE, PIXMAP_SIZE)}};

  std::map<uint8_t, QString> states = {
      {WAITING, "Жду"}, {SMOKING, "Курю"}, {NONE, "Не хочу"}};
```

```
  QPixmap hatPixmap =
      QPixmap(":/rc/pictures/top-hat.png").scaled(HAT_SIZE, HAT_SIZE);

  void setupScene();

  void setupSignals();

  void startWork();

  void onWorkFinished();

  void startSmoker(int smoker);

  void updateState();
};

#endif // WIDGET_H
```

widget.cpp

```
#include <algorithm>
#include <atomic>
#include <mutex>
#include <vector>

#include <QDebug>
#include <QGraphicsPixmapItem>
#include <QGraphicsScene>
#include <QGraphicsTextItem>
#include <QPixmap>
#include <QPoint>
#include <QString>
#include <QThread>
#include <QTimer>
#include <QtConcurrent>

#include "ui_widget.h"
#include "widget.h"

struct SmokeItem {
  QPointF pos;
  SmokeItemTypes type;
};

struct SmokerInfo {
  std::pair<SmokeItem, SmokeItem> items;
  SmokeItem missed;
  QPoint hatPos;
  size_t num;
};

static const QPointF SCENE_CENTER{SCENE_WIDTH / 2 - PIXMAP_SIZE / 2,
                                  SCENE_HEIGHT / 2 - PIXMAP_SIZE / 2};

static std::mutex mutex;

static bool needsToStop;

static std::atomic_uint stopped{0xF};

static unsigned int timeslices[3];
```

```
static SmokeItemTypes currentSmokeItem;

static uint8_t isSmoking[3] = {WAITING, WAITING, WAITING};

static QGraphicsItem *tablePixmapItem = nullptr;

static QGraphicsItem *smokersPixmapItems[3];

static QGraphicsItem *statePixmapItems[3];

static int to_int(SmokeItemTypes item) { return static_cast<int>(item); }

static unsigned int computeTimeslice(int value) {
  return static_cast<unsigned int>(300 + MAX_TIMESLICE * (100 - value) / 100);
}

static std::vector<SmokerInfo> smokers = {
    {{SmokeItem{{OFFSET, OFFSET + HAT_SIZE}, SmokeItemTypes::Match},
      SmokeItem{{OFFSET + PIXMAP_SIZE, OFFSET + HAT_SIZE},
                SmokeItemTypes::Paper}},
     SmokeItem{{OFFSET + 2 * PIXMAP_SIZE, OFFSET + HAT_SIZE},
               SmokeItemTypes::Tobacco},
     QPoint{OFFSET, OFFSET},
     0},
    {{SmokeItem{{OFFSET + HAT_SIZE, SCENE_HEIGHT - (OFFSET + 2 * PIXMAP_SIZE)},
                SmokeItemTypes::Paper},
      SmokeItem{{OFFSET + HAT_SIZE, SCENE_HEIGHT - (OFFSET + PIXMAP_SIZE)},
                SmokeItemTypes::Tobacco}},
     SmokeItem{{OFFSET + HAT_SIZE, SCENE_HEIGHT - (OFFSET + 3 * PIXMAP_SIZE)},
               SmokeItemTypes::Match},
     QPoint{OFFSET, SCENE_HEIGHT - (OFFSET + 2 * PIXMAP_SIZE)},
     1},
    {{SmokeItem{{SCENE_WIDTH - (OFFSET + 2 * PIXMAP_SIZE),
                 SCENE_HEIGHT / 2 + HAT_SIZE / 2},
                SmokeItemTypes::Match},
      SmokeItem{{SCENE_WIDTH - (OFFSET + PIXMAP_SIZE),
                 SCENE_HEIGHT / 2 + HAT_SIZE / 2},
                SmokeItemTypes::Tobacco}},
     SmokeItem{{SCENE_WIDTH - (OFFSET + 3 * PIXMAP_SIZE),
                SCENE_HEIGHT / 2 + HAT_SIZE / 2},
               SmokeItemTypes::Paper},
     QPoint{SCENE_WIDTH - (OFFSET + 2 * PIXMAP_SIZE),
            SCENE_HEIGHT / 2 - HAT_SIZE / 2},
     2}};

Widget::Widget(QWidget *parent)
    : QWidget(parent), ui(new Ui::Widget), pool(this) {
  ui->setupUi(this);

  setupSignals();
  setupScene();

  pool.setMaxThreadCount(4);
  timer.setInterval(RENDER_TIMEOUT);
}

Widget::~Widget() {
  needsToStop = true;
  timer.stop();
  while (stopped != 0xF) {
  }
```

```
  delete ui;
}

void Widget::setupScene() {
  auto *view = ui->graphicsView;

  auto *scene = new QGraphicsScene();

  view->setScene(scene);

  scene->addRect(0, -20, SCENE_WIDTH, SCENE_HEIGHT + 20);
  scene->addEllipse(180, 80, 140, 140);

  for (const auto &smoker : smokers) {
    // set hat
    auto *hat = scene->addPixmap(hatPixmap);
    hat->setPos(smoker.hatPos);
    // set first item
    auto first = smoker.items.first;
    auto *firstItem = scene->addPixmap(pixmaps[first.type]);
    firstItem->setPos(first.pos);
    // set second item
    auto second = smoker.items.second;
    auto *secondItem = scene->addPixmap(pixmaps[second.type]);
    secondItem->setPos(second.pos);
  }
}

void Widget::setupSignals() {
  connect(this, &Widget::stateChanged, this, &Widget::updateState,
          Qt::QueuedConnection);
  connect(ui->startButton, &QPushButton::clicked, this, &Widget::startWork);
  connect(this, &Widget::workFinished, this, &Widget::onWorkFinished);
  connect(&timer, &QTimer::timeout, this, &Widget::updateState);
  connect(ui->stopButton, &QPushButton::clicked, this, [this]() {
    needsToStop = true;
    ui->stopButton->setDisabled(true);
    this->timer.stop();

    QtConcurrent::run([this]() {
      while (stopped != 0xF) {
      }
      emit this->workFinished();
    });
  });
  connect(ui->speedSlider, &QSlider::valueChanged, this, [](int value) {
    timeslices[0] = computeTimeslice(value);
    qDebug() << "timeslice[0]: " << timeslices[0];
  });
  connect(ui->speedSlider2, &QSlider::valueChanged, this, [](int value) {
    timeslices[1] = computeTimeslice(value);
    qDebug() << "timeslice[1]: " << timeslices[1];
  });
  connect(ui->speedSlider3, &QSlider::valueChanged, this, [](int value) {
    timeslices[2] = computeTimeslice(value);
    qDebug() << "timeslice[2]: " << timeslices[2];
  });
  ui->speedSlider->setValue(50);
  ui->speedSlider2->setValue(50);
  ui->speedSlider3->setValue(50);
}
```

```
void Widget::onWorkFinished() {
  ui->stopButton->setDisabled(true);
  ui->startButton->setEnabled(true);

  if (tablePixmapItem &&
      tablePixmapItem->scene() == ui->graphicsView->scene()) {
    ui->graphicsView->scene()->removeItem(tablePixmapItem);
    tablePixmapItem = nullptr;
  }
}

void Widget::startWork() {
  timeslices[0] = computeTimeslice(ui->speedSlider->value());
  timeslices[1] = computeTimeslice(ui->speedSlider2->value());
  timeslices[2] = computeTimeslice(ui->speedSlider3->value());
  qDebug() << timeslices[0] << timeslices[1] << timeslices[2];

  ui->stopButton->setEnabled(true);
  ui->startButton->setDisabled(true);

  needsToStop = false;
  stopped = 0x0;
  currentSmokeItem = SmokeItemTypes::None;

  QtConcurrent::run(&this->pool, []() {
    std::vector<SmokeItemTypes> items = {
        SmokeItemTypes::Tobacco, SmokeItemTypes::Match, SmokeItemTypes::Paper};
    QString names[] = {"Tobacco", "Match", "Paper"};

    while (!needsToStop) {
      mutex.lock();
      currentSmokeItem = items[rand() % 3];
      qDebug() << "emitted item" << names[to_int(currentSmokeItem)];
      mutex.unlock();

      QThread::msleep(EMITTER_TIMESLICE);
    }

    stopped |= 1 << 3;
    qDebug() << "emitter finished";
  });

  for (int i = 0; i < 3; ++i) {
    QtConcurrent::run(&this->pool, [this, i]() { this->startSmoker(i); });
  }

  timer.start();
}

void Widget::startSmoker(int smoker) {
  auto index = static_cast<unsigned int>(smoker);
  auto missedItem = smokers[index].missed.type;

  while (!needsToStop) {
    mutex.lock();

    isSmoking[index] = WAITING;

    mutex.unlock();
```

```
    QThread::msleep(WAITING_TIMESLICE);

    mutex.lock();

    if (currentSmokeItem == missedItem) {
      isSmoking[index] = SMOKING;
      currentSmokeItem = SmokeItemTypes::None;
      qDebug() << "smoker" << index << "took item";

      mutex.unlock();

      QThread::msleep(SMOKING_TIMESLICE);

      mutex.lock();

      isSmoking[index] = NONE;

      mutex.unlock();

      QThread::msleep(timeslices[index]);
    } else {
      mutex.unlock();
    }
  }

  stopped |= 1u << index;
  qDebug() << "smoker" << index << "finished";
}

void Widget::updateState() {
  if (needsToStop) {
    return;
  }

  mutex.lock();

  auto item = currentSmokeItem;
  uint8_t localIsSmoking[3] = {isSmoking[0], isSmoking[1], isSmoking[2]};

  mutex.unlock();

  auto *scene = ui->graphicsView->scene();

  // Update item on the table
  if (tablePixmapItem && tablePixmapItem->scene() == scene) {
    scene->removeItem(tablePixmapItem);
  }
  tablePixmapItem = new QGraphicsPixmapItem(pixmaps[item]);
  scene->addItem(tablePixmapItem);
  tablePixmapItem->setPos(SCENE_CENTER);

  // Update smokers' items
  for (size_t i = 0; i < 3; ++i) {
    if (smokersPixmapItems[i] && smokersPixmapItems[i]->scene() == scene) {
      scene->removeItem(smokersPixmapItems[i]);
      delete smokersPixmapItems[i];
      smokersPixmapItems[i] = nullptr;
    }
  }
  // Update state's items
  for (size_t i = 0; i < 3; ++i) {
```

```
    auto smokerInfo = smokers[i];
    if (statePixmapItems[i] && statePixmapItems[i]->scene() == scene) {
      scene->removeItem(statePixmapItems[i]);
      delete statePixmapItems[i];
    }
    statePixmapItems[i] = new QGraphicsTextItem(
        QString("%1: %2").arg(i + 1).arg(states[localIsSmoking[i]]));
    scene->addItem(statePixmapItems[i]);
    auto pos = smokerInfo.hatPos;
    pos.setY(pos.y() - 20);
    statePixmapItems[i]->setPos(pos);
  }

  for (size_t i = 0; i < 3; ++i) {
    if (localIsSmoking[i] == SMOKING) {
      auto smokerInfo = smokers[i];
      smokersPixmapItems[i] =
          new QGraphicsPixmapItem(pixmaps[smokerInfo.missed.type]);
      scene->addItem(smokersPixmapItems[i]);
      smokersPixmapItems[i]->setPos(smokerInfo.missed.pos);
    }
  }
}
```

3 Выводы

В ходе выполнения лабораторной работы узнали, что алгоритм Петерсона простым в реализации алгоритмом, а значит для его работы потребуется меньше ресурсов. Так же из достоинств данного алгоритма можно отметить:

1) Взаимное исключение - оба потока никогда не попадут в критическую секцию одновременно.
2) Отсутствие взаимной блокировки – для того, чтобы оба процесса находились в ожидании, необходимы противоположные значения переменной turn, что невозможно.
3) Отсутствие бесконечного ожидания - в алгоритме Петерсона процесс не будет ждать дольше, чем один вход в критическую секцию: после выполнения *leaveRegion* и повторного захода в *enterRegion* процесс установит себя как ждущего и попадёт в цикл, который не завершится, пока не отработает другой процесс.

Но у данного алгоритма иметься также весомый недостаток: наличие активного ожидания – так как ожидание производится циклом, это приводит к бесцельному расходованию ресурсов процессора, а также может иметь некоторые неожиданные последствия.

При реализации задачи о курильщиках, мы на практике закрепили, важность взаимоисключения по отношению к критическому ресурсу, в данном случаи столу с необходимым предметом для курения.

В данном задании для синхронизации использовался `std::mutex` из стандартной библиотеки C++. Из достоинств данного способа синхронизация можно отметить:

1) Простота.
2) Эффективность.