МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ СИСТЕМ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

Направление 09.03.01 – Информатика и вычислительная техника
(код и наименование направления)
Профиль 02 – Программное обеспечение вычислительной техники и автоматизированных систем

Допускаю к защите
Заведующий кафедрой ЭВМ
_____/Страбыкин Д. А./
(подпись) (Ф.И.О)

Разработка программы моделирования работы планировщиков памяти и процессов в операционных системах

Пояснительная записка выпускной квалификационной работы
ТПЖА 090301.02.066 ПЗ

Разработал: студент гр.ИВТб-4302-02-00 _____ /Рзаев А. Э./ _____

Руководитель: д.т.н., профессор _____ /Страбыкин Д.А./ _____

Нормоконтролер: к.т.н., доцент _____ /Скворцов А. А./ _____
(подпись) *(Ф.И.О.)* *(дата)*

Киров 2020

Содержание

Введение

В настоящее время в области образования все больше производится автоматизация контроля знаний учащихся и освоения нового материала.

Одним из способов автоматизации являются специальные программные модели, эмулирующие работу какой-либо системы. С их помощью студенты имеют возможность достаточно подробно изучить ее работу, принципы и особенности. В совокупности с теоретическим материалом это позволяет увеличить степень освоения новых знаний по данной дисциплине и повысить качество обучения в целом.

Такие программные модели достаточно широко используются при выполнении лабораторных работ по дисциплине «Операционные системы». К сожалению, качество некоторых приложений оставляет желать лучшего, что затрудняет изучение нового материала. Поэтому было принято решение выполнить анализ и доработку наиболее проблемной модели – диспетчера задач операционной системы.

1 Анализ предметной области

На данном этапе работы необходимо рассмотреть функции планировщиков процессов и оперативной памяти операционной системы, провести обзор текущей программной модели диспетчера задач, ее возможностей по изучению функций ОС, выяснить ее недостатки и обосновать актуальность разработки новой.

1.1 Планировщик памяти

Функциями планировщика оперативной памяти (подсистемы управления памятью) в многозадачных ОС являются:

– отслеживание свободной и занятой памяти;

– первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе и освобождение памяти по завершении процессов;

– настройка адресов программы на конкретную область физической памяти;

– полное или частичное вытеснение кодов и данных процессов из оперативной памяти (ОП) на диск, когда размеры ОП недостаточны для размещения всех процессов, и возвращение их в ОП;

– защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;

– дефрагментация памяти [1].

1.2 Планировщик процессов

Функциями планировщика процессов (подсистемы управления

процессами) являются:

– планирование выполнения процессов и потоков (задач);

– создание и уничтожение процессов;

– обеспечение процессов необходимыми системными ресурсами;

– поддержание взаимодействия между процессами.

При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление ресурсами:

– подсистема управления памятью;

– подсистема ввода/вывода;

– файловая подсистема.

Планирование выполнения процессов включает в себя решение следующих задач:

– определение момента времени для смены текущего активного потока;

– выбор потока для выполнения из очереди готовых потоков.

В большинстве ОС универсального назначения планирование осуществляется динамически, то есть решения принимаются во время работы системы на основе анализа текущей ситуации. За счет этого динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации.

Статическое планирование может быть использовано в специализированных системах (например, в системах реального времени), в которых весь набор одновременно выполняемых задач определен заранее.

Статический планировщик принимает решение о планировании не во время работы системы, а заранее. Результатом работы такого планировщика является таблица, в которой указывается к какому потоку или процессу, когда и на какое время должен быть предоставлен процессор.

Для выполнения задач планирования процессов ОС должна получить

управление при наступлении следующих событий:

– прерывание от таймера (истечение кванта времени);

– запрос на ввод/вывод или на доступ к ресурсу, который сейчас занят. Задача переходит в состояние ожидания;

– освобождение ресурса. Планировщик проверяет, не ожидает ли этот ресурс какая-либо задача. Если да, то эта задача переводится из состояния ожидания в состояние готовности;

– аппаратное прерывание, которое сигнализирует о завершении периферийным устройством операции ввода/вывода, соответствующая задача переводится в очередь готовых и выполняется перепланирование;

– внутреннее прерывание, сигнализирующее об ошибке, которая произошла в результате выполнения активной задачи. Планировщик снимает задачу и выполняет перепланирование;

– запросы приложений и пользователей на создание новой задачи или повышения приоритета существующей задачи.

1.3 Обзор текущей программной модели

Текущая программная модель была разработана в 2002 году студентами Вятского государственного университета А. С. Гордиенко и М. Н. Томчуком; доработана в 2006 году студентами А. Ю. Соколовым и И. А. Брызгаловым.

В программной модели рассматривается работа двух подсистем операционной системы: планировщика (диспетчера) процессов и планировщика памяти. В данной модели подразумевается только оперативная память.

1.3.1Модель планировщика памяти

В данной модели планировщика памяти адресное пространство разбито на 256 страниц памяти по 4096 байт каждая. Наименьшая единица адресного пространства, доступная для выделения процессу – страница. Выделять можно только целое число страниц.

Процессам в данной модели могут присваиваться идентификаторы (PID) от 0 до 255 включительно.

Непрерывная область памяти, состоящая из одной и более страниц и имеющая начальный адрес и размер, называется блоком памяти.

Каждый блок памяти имеет следующие параметры:

– адрес начала блока;

– размер блока в страницах;

– идентификатор процесса, которому принадлежит данный блок.

Память процессам при каждом запросе выделяется в виде одного блока памяти.

Состояние памяти определяется совокупностью всех блоков памяти. Каждая страница адресного пространства должна находиться в одном и только одном блоке памяти, т. е. последовательность блоков памяти, упорядоченных по начальному адресу, должна полностью покрыть адресное пространство.

Под начальным состоянием памяти подразумевается такое состояние памяти, при котором все адресное пространство покрыто одним свободным блоком памяти с начальным адресом 0 и размером 256.

1.3.2Используемые в модели дисциплины планирования процессов

Используемые в программной модели дисциплины планирования можно разделить на две группы: вытесняющие и невытесняющие.

К первой группе относятся следующие дисциплины:

– FCFS (First Come First Served). Бесприоритетная дисциплина планирования, в которой задачи обслуживаются в порядке их поступления. Используется две очереди: одна – для новых потоков, другая – для потоков, уже использовавших процессорное время. Предпочтение отдается потокам из второй очереди [1];

– SJN (Shortest Job Next). Используется одна очередь, в которой потоки упорядочены по заявленному времени выполнения. Приоритет отдается потоку с наименьшим заявленным временем. Потоки, превысившие данное ограничение, остаются в конце очереди [1];

– SRT (Shortest Remaining Time). Аналогична SJN, за исключением того, что потоки в очереди упорядочены по оставшемуся времени выполнения;

Ко второй группе относятся [1]:

– RR (Round Robin). Используется одна очередь, которая заполняется потоками в порядке поступления. Каждому потоку выделяется фиксированный промежуток времени – квант, в течение которого он может выполняться на процессоре. По истечении кванта времени поток добавляется в конец очереди;

– WinNT. Используется 16 очередей (по количеству приоритетов). У каждого потока есть два приоритета: текущий и базовый. Базовый задается при создании потока (процесса) и не меняется, текущий меняется со временем, но не может быть ниже базового. При добавлении процесса в очередь (только в случае, если истек квант времени) приоритет потока уменьшается на единицу. По завершении операций ввода/вывода ожидающему потоку назначается прибавка к приоритету [1].

– Unix. Используется 16 очередей (по количеству приоритетов). Имеется два класса задач: со статическими приоритетами (8-15) и динамическими (0-7). Чем дольше процесс с динамическим приоритетом

работает без выполнения операций ввода/вывода, тем меньше становится его приоритет. По завершении операций ввода/вывода у такого процесса увеличивается приоритет на 1. При создании процесса или при завершении ввода/вывода процесса с более высоким приоритетом, чем у активного, переключение на данный процесс не происходит. При истечении кванта времени или передаче управления ОС процессорное время выделяется потоку с наивысшим приоритетом.

1.3.3Интерфейс пользователя

Основное окно программы, представленное на рисунке 1, можно разделить на следующие области:

- строка меню;
- панель инструментов;
- боковая панель для переключения между заданиями;
- основная область с отдельным окном для каждого задания.

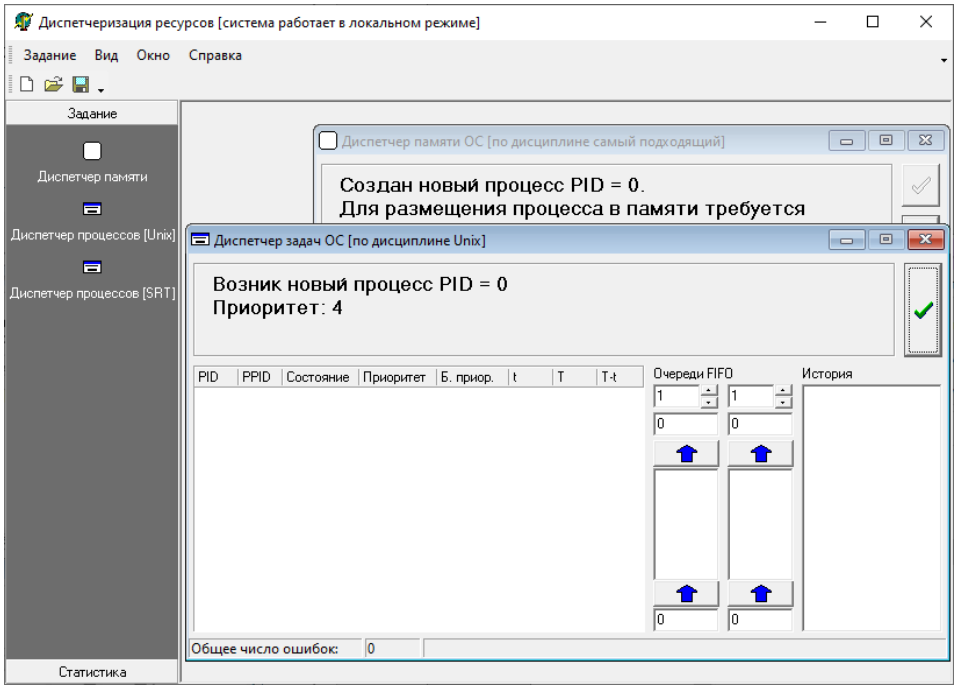


Рисунок 1 – Основное окно программы

Окно с заданием по планировщику памяти (рисунок 2) разделено на следующие области:

– заголовок с названием дисциплины;
– описание текущей заявки;
– список блоков памяти ОС;
– список свободных блоков памяти;
– блок кнопок: «Подтвердить» и «Отклонить»;
– список обработанных пользователем заявок;
– строка статуса.

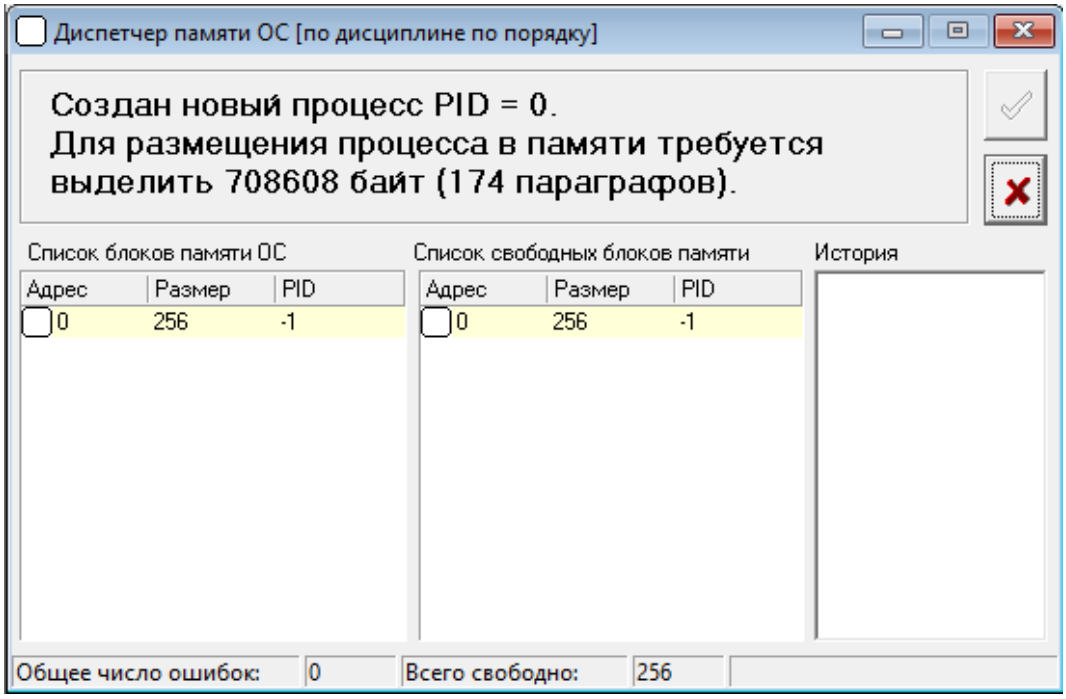


Рисунок 2 – Окно с заданием по планировщику памяти

В левой части окна находится список блоков памяти. Для каждого блока указан его начальный адрес, размер в страницах, а также идентификатор процесса, владеющего данным блоком («-1» – нет владельца).

В правой части экрана находится список выполненных шагов, которые хранят историю действий пользователя. Элемент в списке

помечается красным цветом, если действия пользователя на данном шаге были некорректны. При наведении указателя мыши на элемент списка отображается информация о событии и действиях пользователя на данном шаге. Для отмены произвольного числа шагов необходимо щелкнуть мышью на элемент списка, до которого необходимо очистить историю. Отменённые шаги помечаются серым цветом. До тех пор, пока не выполнено какое-либо действие в окне диспетчера, можно восстановить действия пользователя, щелкнув на записи, соответствующей отмененному шагу.

В нижней части окна расположена строка статуса, в которой отображается общее число ошибок, которые допустил пользователь в процессе выполнения задания, а также количество свободных страниц оперативной памяти.

Окно с заданием по планировщику процессов (рисунок3) разделено на следующие области:

- заголовок с названием дисциплины;
- описание текущей заявки;
- список процессов;
- блок очередей процессов;
- кнопка «Подтвердить»;
- список обработанных пользователем заявок;
- строка статуса.

В верхней части окна присутствует описание заявки и количество выполненных заявок.

В центральной части окна находится список процессов, который содержит следующую информацию:

- состояние процесса; отображается цветом строки, значком, а также буквой в столбце «Состояние»:
  - «E» – процесс выполняется;

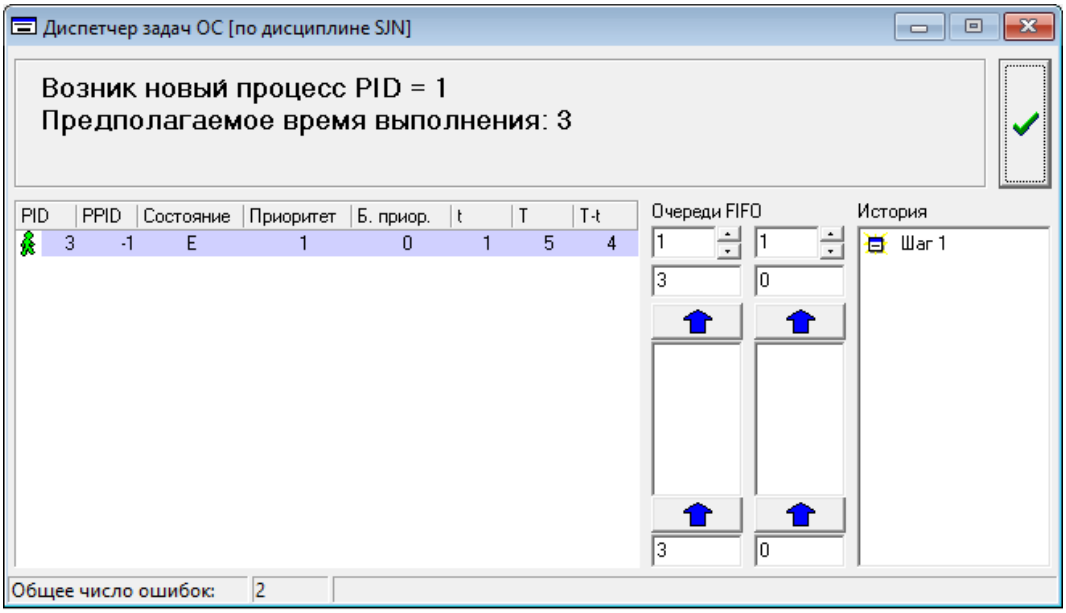


Рисунок 3 – Окно с заданием по планировщику процессов

- «A» – процесс готов к выполнению;
- «W» – процесс обратился к устройствам ввода/вывода и ожидает завершения обмена.

– идентификатор процесса – расположен в столбце «PID»;

– идентификатор родительского процесса – расположен в столбце «PPID»;

– текущий приоритет процесса (в системе Windows NT) – расположен в столбце «Приоритет»;

– базовый приоритет процесса (в системе Windows NT) – расположен в столбце «Приоритет»;

– время выполнения процесса (инкрементируется только когда процесс выполняется) – находится в столбце «t»;

– предполагаемое время выполнения процесса – находится в столбце «T»;

– разность между предполагаемым временем и реальным временем выполнения процесса (по сути – время, оставшееся до завершения процесса) – находится в столбце «T-t»; если значение отрицательное (процесс превысил

лимит времени), вместо числа выводится «-».

Справа от списка процессов находятся два окна очередей. В любом из них можно отобразить любую очередь (с номерами от 1 до 15). Для добавления значения в конец очереди, необходимо ввести его в нижнее окно ввода и нажать на нижнюю кнопку со стрелкой. Для извлечения значения из очереди необходимо нажать на верхнюю кнопку со стрелкой. Извлеченное значение помещается в окно ввода над кнопкой. В левом окне отображения очереди имеется возможность изменять порядок элементов в очереди путем перетаскивания мышью.

Правая и нижняя части окна имеют тот же интерфейс, что и в окне задания по планировщику памяти.

1.3.4Описание выполняемого пользователем задания

В обоих заданиях задача пользователя – корректно обработать поступающие заявки.

В задании по планировщику памяти заявки бывают следующих типов:

- создание нового процесса;
- завершение процесса;
- выделение памяти существующему процессу;
- освобождение памяти, которой владеет процесс.

Пользователь может выполнить следующие действия:

- ответить на запрос отказом, сразу нажав на кнопку «Отклонить»;
- выделить память процессу. Для этого необходимо щелкнуть правой кнопкой мыши на свободном блоке в списке блоков памяти ОС и выбрать пункт «Выделить приложению». После этого в появившемся окне нужно ввести идентификатор процесса, которому выделяется память, а также количество выделяемых страниц;
- освободить память. Для этого необходимо щелкнуть правой

кнопкой мыши на занятом блоке в списке блоков памяти ОС и выбрать пункт «Освободить»;

– объединить два свободных блока в один. Для этого необходимо щелкнуть правой кнопкой мыши на свободном блоке в списке блоков памяти ОС и выбрать пункт «Объединить со следующим», Данное действие необходимо выполнять каждый раз, когда образуются соседние свободные блоки;

– уплотнить (дефрагментировать память). Для этого необходимо щелкнуть правой кнопкой мыши на списке блоков памяти ОС и выбрать пункт «Уплотнение памяти»;

– подтвердить действия, нажав на кнопку «Подтвердить».

Основные дисциплины выбора свободного блока памяти:

– выбор первого подходящего блока. В этом случае список свободных блоков сортируется по адресу и выбирается первый блок, размер которого больше или равен требуемому размеру;

– выбор самого подходящего блока. Список свободных блоков сортируется по размеру (от меньшего к большему) и выбирается первый подходящий по размеру блок;

– выбор самого неподходящего блока. Самый неподходящий блок – это блок с максимальным размером. Список свободных блоков необходимо отсортировать в порядке убывания размера и выбрать первый блок.

В задании по планировщику процессов заявки бывают следующих типов:

– создание нового процесса;

– создание дочернего процесса;

– завершение процесса;

– запрос на ввод/вывод;

– завершение ввода/вывода;

– передача управления операционной системе;

– истечение кванта времени.

Пользователь может выполнить следующие действия:

– ответить на запрос отказом, сразу нажав на кнопку «Подтвердить»;

– добавить процесс в список процессов – щелкнуть правой кнопкой мыши на списке процессов и выбрать пункт «Добавить» в контекстном меню, после чего ввести параметры создаваемого процесса: идентификатор, идентификатор родителя (если родителя нет – -1), значение базового и текущего приоритетов, предполагаемое и текущее время выполнения, состояние процесса, и нажать кнопку «ОК»;

– добавить идентификатор процесса в очередь;

– поменять порядок элементов в очереди;

– удалить процесс из списка процессов и из очереди – щелкнуть правой кнопкой мыши на нужном элементе в списке процессов и в проявившемся меню выбрать пункт «Удалить»;

– переключить процесс в состояние ожидания – щелкнуть правой кнопкой мыши на нужном элементе в списке процессов и в проявившемся меню выбрать пункт «Переключить в состояние ожидания»;

– переключить процесс в состояние готовности – щелкнуть правой кнопкой мыши на нужном элементе в списке процессов и в проявившемся меню выбрать пункт «Переключить в состояние готовности»;

– выбрать процесс для выполнения – щелкнуть правой кнопкой мыши на нужном элементе в списке процессов и в проявившемся меню выбрать пункт «Переключиться»;

– подтвердить действия, нажав на кнопку «Подтвердить».

1.3.5Недостатки

В текущей программной модели были найдены следующие недостатки:

– приложение доступно только для ОС Windows. Пользователи других операционных систем должны запускать Windows в виртуальной машине либо использовать другие средства по запуску Windows-приложений в других ОС;

– нестабильность. В ходе выполнения лабораторной работы программная модель несколько раз аварийно завершалась, из-за чего результаты выполненной работы безвозвратно терялись;

– ошибки при проверке пользовательских действий. В некоторых случаях было замечено, что программная модель принимала правильную последовательность действий как ошибочную. Это в совокупности с нестабильностью программы усложняет изучение студентами программной модели и, как следствие, увеличивает время на выполнение лабораторной работы;

– нечеткость, «размытость» интерфейса на дисплеях со сверхвысоким разрешением (HiDPI);

– в связи с утерей исходного кода программы и сложностью дизассемблирования определить формат файла задания не представляется возможным, что препятствует разработке новых вариантов заданий.

1.4 Актуальность разработки

Лабораторная работа по данной теме (управление памятью и процессами в ОС) имеет важную роль в закреплении лекционного материала и предоставляет студентам возможность изучить более подробно устройство ОС.

Из-за того, что исходный код программной модели утерян, исправить ошибки и недостатки в текущей модели не представляется возможным. Поэтому было принято решение разработать новую программную модель, повторяющую функционал текущей, в которой будут исправлены вышеописанные ошибки и недостатки.

Выводы по разделу

Проведя анализ предметной области, можно сделать следующие выводы:

– текущая программная модель имеет недостатки, препятствующие изучению материала по данной теме;

– лабораторные работы по данной теме с использованием программной модели имеет важную роль в закреплении лекционного материала;

– из-за того, что исходный код программной модели утерян, исправить ошибки и недостатки в текущей модели не представляется возможным, а процесс обратной разработки становится крайне трудоемким.

2 Техническое задание

В данном разделе представлено техническое задание на разработку программы моделирования работы планировщиков памяти и процессов в операционных системах «Диспетчер задач ОС».

2.1 Краткая характеристика области применения

Программа предназначена для закрепления студентами лекционного материала по дисциплине «Операционные системы», а именно: управление процессами и памятью в операционных системах.

2.2 Назначение разработки

Функциональным назначением программы является предоставление студентам возможности изучить работу планировщиков процессов и памяти во время выполнения лабораторных работ.

Программа должна эксплуатироваться на ПК студентов, преподавателей и на ПК, установленных в учебных аудиториях Вятского государственного университета. Особые требования к конечному пользователю не предъявляются.

2.3 Требования к программе

Программа должна обладать следующими функциональными возможностями:

1) генерация уникальных заданий;
2) сохранение текущей сессии пользователя в файл с возможностью

восстановления;

3) шифрование сохраняемых файлов;
4) подсчет количества ошибок, сделанных в ходе выполнения задания;
5) просмотр действий пользователя, выполненных в ходе прохождения задания.

2.4 Требования к надежности

Надежное (устойчивое) выполнение программы должно быть обеспечено выполнением пользователем совокупности организационно-технических мероприятий, перечень которых приведен ниже:

1) организацией бесперебойного питания технических средств;
2) использованием лицензионного программного обеспечения.

Отказы программы возможны вследствие некорректных действий пользователя при взаимодействии с операционной системой. Во избежание возникновения отказов программы по указанной выше причине следует обеспечить работу конечного пользователя без предоставления ему административных привилегий.

2.5 Требования к составу и параметрам технических средств

В состав технических средств должен входить IBM-совместимый персональный компьютер, включающий в себя:

1) x86-совместимый процессор с тактовой частотой не меньше 1.0 ГГц;
2) дисплей с разрешением не меньше, чем 1024x768;
3) не менее 1 гигабайта оперативной памяти;
4) не менее 100 мегабайт свободного дискового пространства;

5) клавиатуру, мышь.

Системные программные средства, используемые программой, должны быть представлены следующими операционными системами:

– 64-разрядная ОС Windows 7 или более поздней версии;

– macOS: 10.13 High Sierra или более поздней версии

– Linux: Ubuntu 18.04 или старше, Debian 10 или старше, Fedora 31 или старше.

2.6 Требования к интерфейсу

Программа должна обеспечивать взаимодействие с пользователем посредством графического пользовательского интерфейса и предоставлять возможность выполнять наиболее частые операции с помощью сочетаний клавиш на клавиатуре.

Предполагается использовать организацию графического интерфейса, аналогичную той, что используется в текущей программной модели.

Также необходимо обеспечить поддержку экранов со сверхвысоким разрешением.

2.7 Требования к программной документации

Состав программной документации должен включать в себя:

1) техническое задание;
2) руководство пользователя;
3) техническую документацию;
4) исходный код.

3 Разработка структуры приложения

В данном разделе в соответствии с требованиями, поставленными в техническом задании, описана модульная структура приложения, алгоритмы функционирования, а также определен состав содержимого файлов пользовательских сессий и средства защиты от несанкционированного доступа к нему.

3.1 Разработка модульной структуры приложения

Для обеспечения функционирования системы разработана обобщенная структура программного продукта, представляющая собой набор взаимосвязанных модулей, в которых будут реализованы алгоритмы функционирования приложения. Модульная структура приложения представлена на рисунке 4.

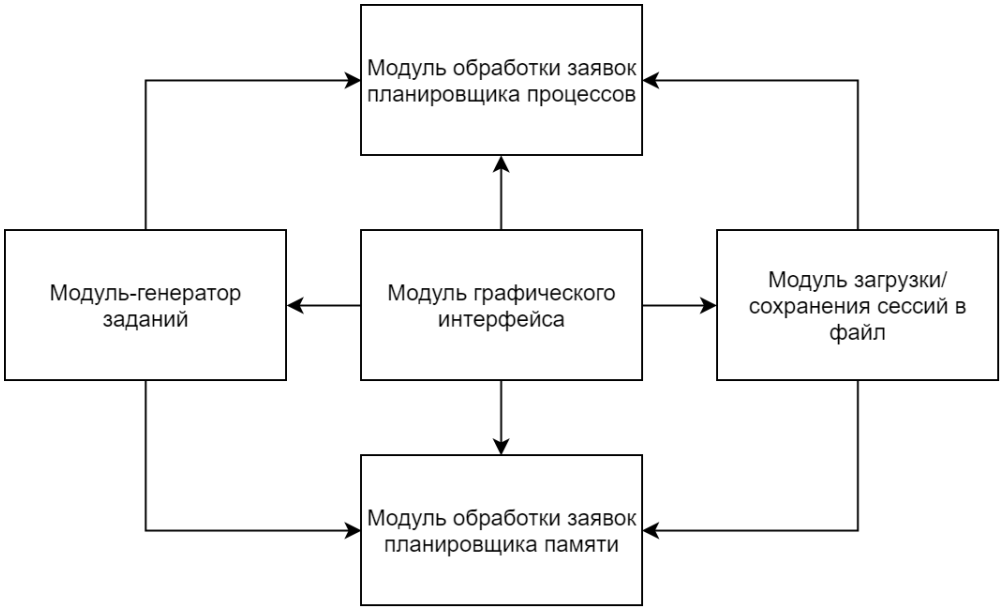


Рисунок 4 – Модульная структура приложения

Каждый из модулей имеет следующее назначение и функционал:

– модули обработки заявок планировщиков памяти и процессов. Определяют алгоритмы обработки заявок и проверки выполненных пользователем действий, а также определяют необходимые типы данных для представления заявок, состояний планировщиков и собственно заданий на лабораторную работу;

– модуль-генератор заданий. Определяет алгоритмы генерации заданий. Обеспечивает достаточную степень уникальности задания для каждого пользователя без необходимости разработки заданий вручную;

– модуль загрузки и сохранения сессий (пользовательских) в файл. Предоставляет возможность сохранять текущее состояние выполняемого задания между запусками приложения, а также обеспечивает защиту от непредвиденного изменения структуры файла;

– модуль графического интерфейса. Является связующим звеном между приложением и пользователем; отображает данные о ходе выполнения задания в текстовом и графическом виде.

3.2 Алгоритмы функционирования планировщика памяти

В данной программной модели состояние памяти определяется списком всех блоков памяти, упорядоченных по начальному адресу, а также списком свободных блоков памяти.

Во время обработки заявки пользователь выполняет некоторую последовательность действий из шагов определенного типа. Данные шаги можно представить как операции, применяемые к состоянию системы.

Для проверки действий, выполненных пользователем над состоянием системы, были разработаны алгоритмы обработки поступающих заявок.

Алгоритм обработки заявки «Создан новый процесс»:

1) сначала нужно удостовериться в том, что процессу не выделены какие-либо блоки памяти:
    1.1) если процессу выделены блоки памяти, то считается, что процесс уже создан, а следовательно, заявка некорректная. Никаких действий над памятью не выполнять; алгоритм завершен;
    1.2) в противном случае перейти к пункту 2;
2) найти в соответствии с дисциплиной свободный блок памяти с размером не меньше, чем запрашивается;
3) если такой блок есть, то выделить в нем память процессу и перейти к пункту 6;
4) если такого блока нет, но суммарно свободной памяти достаточно, то выполнить дефрагментацию памяти, в новом свободном блоке выделить память процессу и перейти к пункту 6;
5) если свободной памяти недостаточно, то никаких действий над памятью не выполнять; алгоритм завершен;
6) упорядочить список свободных блоков памяти в соответствии с дисциплиной.

Схема алгоритма обработки заявки «Создан новый процесс» представлена на рисунке 5.

Алгоритм обработки заявки «Завершение процесса»:

1) проверить, выделены ли данному процессу какие-либо блоки памяти:
    1.1) если процессу не выделено никаких блоков памяти, то считается, что процесс не существует, а следовательно, заявка некорректная. Никаких действий над памятью не выполнять; алгоритм завершен;
    1.2) в противном случае перейти к пункту 2;

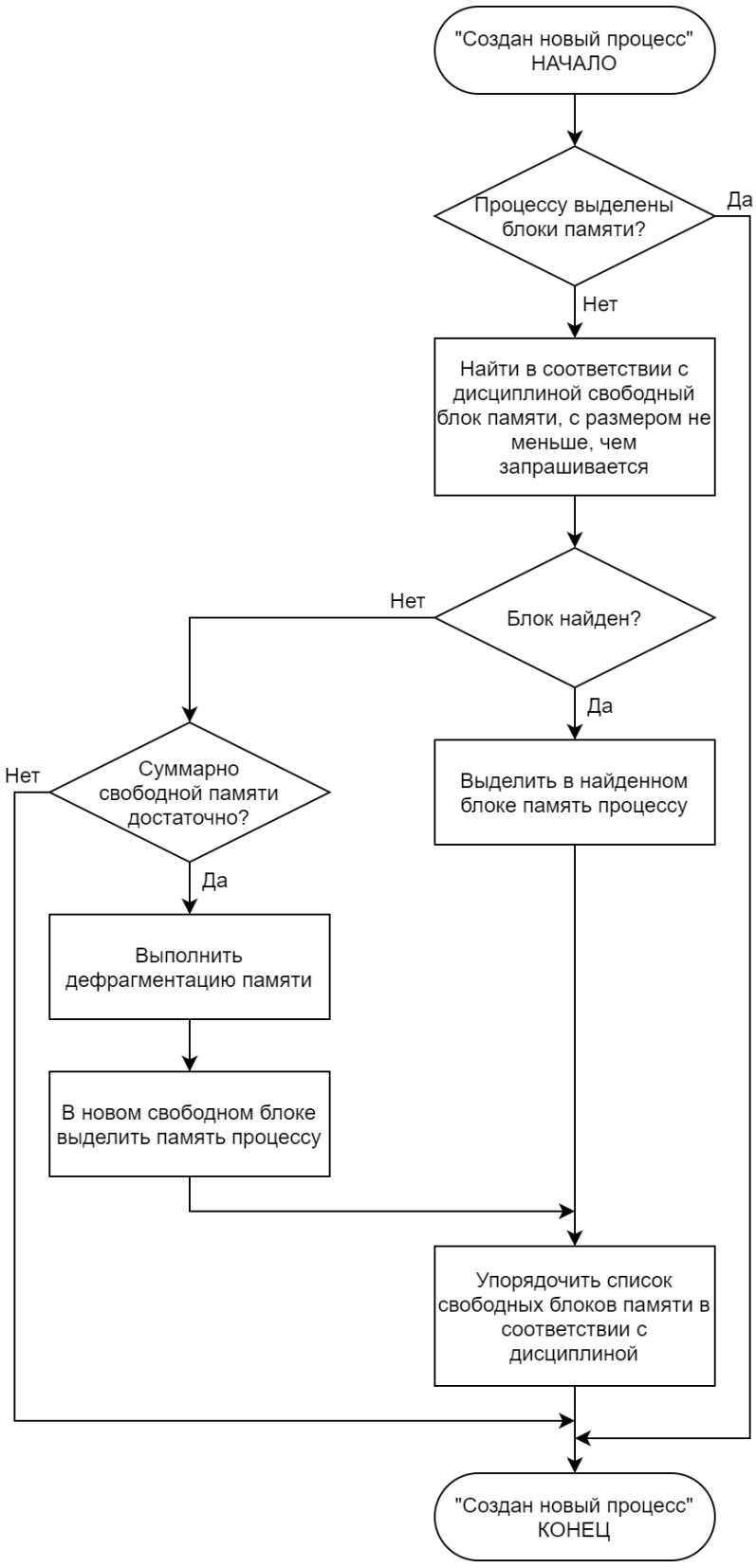


Рисунок 5 – Схема алгоритма обработки заявки «Создан новый процесс»

2) освободить все блоки памяти, принадлежащие данному процессу;
3) объединить соседние свободные блоки памяти;
4) упорядочить список свободных блоков памяти в соответствии с дисциплиной.

Алгоритм обработки заявки «Выделение памяти существующему процессу»:

1) сначала нужно удостовериться в том, что процессу выделены какие-либо блоки памяти:
   1.1) если процессу не выделены блоки памяти, то считается, что процесс не существует, а следовательно, заявка некорректная. Никаких действий над памятью не выполнять; алгоритм завершен;
   1.2) в противном случае перейти к пункту 2;
2) найти в соответствии с дисциплиной свободный блок памяти с размером не меньше, чем запрашивается;
3) если такой блок есть, то выделить в нем память процессу и перейти к пункту 6;
4) если такого блока нет, но суммарно свободной памяти достаточно, то выполнить дефрагментацию памяти, в новом свободном блоке выделить память процессу и перейти к пункту 6;
5) если свободной памяти недостаточно, то никаких действий над памятью не выполнять; алгоритм завершен;
6) упорядочить список свободных блоков памяти в соответствии с дисциплиной.

Схема алгоритма обработки заявки «Выделение памяти существующему процессу» представлен на рисунке 6.

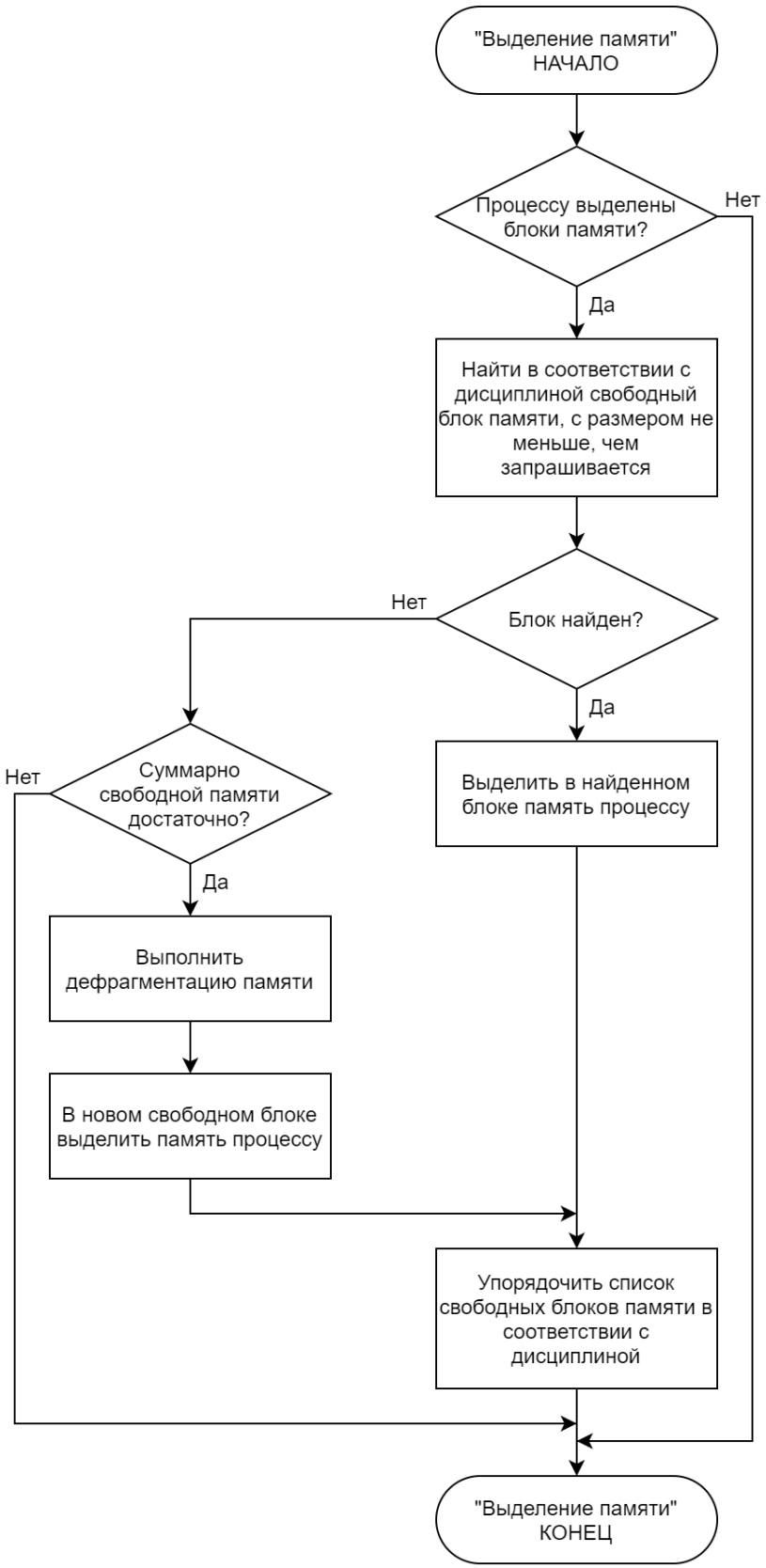


Рисунок 6 – Схема алгоритма обработки заявки «Выделение памяти существующему процессу»

Алгоритм обработки заявки «Освобождение памяти, которой владеет процесс»:

1) найти блок с заданным начальным адресом;
2) обработать следующие исключительные ситуации:
   2.1) блок не найден; алгоритм завершен;
   2.2) блок выделен другому процессу; алгоритм завершен;
3) освободить найденный блок памяти;
4) объединить соседние свободные блоки памяти;
5) упорядочить список свободных блоков памяти в соответствии с дисциплиной.

3.3 Алгоритмы функционирования планировщика процессов

В данной программной модели состояние системы определяется списком процессов и состоянием очередей процессов.

Во время обработки заявки пользователь выполняет некоторую последовательность действий из шагов определенного типа. Данные шаги можно представить как операции, применяемые к состоянию системы.

Для проверки действий, выполненных пользователем над состоянием системы, были разработаны алгоритмы обработки поступающих заявок.

Алгоритм обработки заявки «Создание нового процесса»:

1) сначала нужно удостовериться в том, что процесс с заданным PID не существует. Если такой процесс уже есть, то завершить алгоритм;
2) создать процесс с заданными значениями;
3) добавить процесс в соответствующую очередь;
4) переключить процесс в состояние «Готов к исполнению»;
5) выбрать процесс для исполнения согласно дисциплине;

6) если процесса для выполнения нет, то завершить алгоритм; в противном случае перейти к следующему пункту;
7) если на данный момент исполняющегося процесса нет, то извлечь выбранный для исполнения процесс из очереди и переключить его в состояние «Исполняется» и завершить алгоритм; в противном случае перейти к следующему пункту;
8) если используются относительные приоритеты, то завершить алгоритм; в противном случае перейти к следующему пункту;
9) обработать следующие случаи:
    9.1) у текущего исполняющегося процесса приоритет ниже, чем у выбранного. Исполняющийся процесс добавить в соответствующую очередь и переключить в состояние «Готов к исполнению». Извлечь выбранный для исполнения процесс из очереди и переключить его в состояние «Исполняется»;
    9.2) у текущего исполняющегося процесса приоритет не меньше, чем у выбранного. Переключать процессы не нужно.

Схема алгоритма обработки заявки «Создание нового процесса» представлена на рисунке 7.

Алгоритм обработки заявки «Завершение процесса»:

1) сначала нужно удостовериться в том, что процесс с заданным PID существует. Если такого процесса нет, то завершить алгоритм;
2) завершить данный процесс и все его дочерние процессы;
3) если есть исполняющийся процесс, то завершить алгоритм; в противном случае перейти к следующему пункту;
4) выбрать процесс для исполнения согласно дисциплине;
5) если процесса для выполнения нет, то завершить алгоритм; в противном случае перейти к следующему пункту;

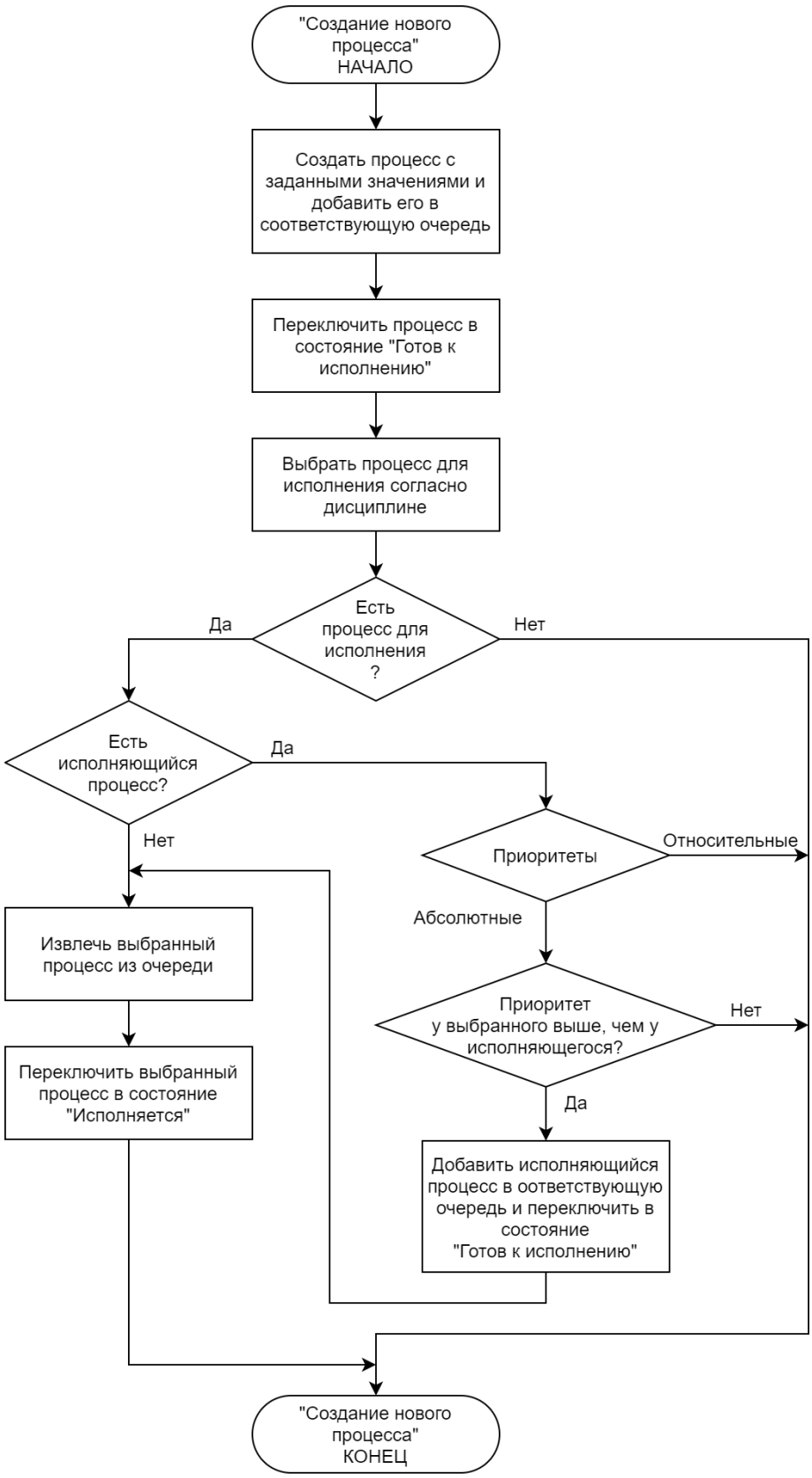


Рисунок 7 – Схема алгоритма обработки заявки «Создание нового процесса»

6) извлечь выбранный процесс из очереди и переключить его в состояние «Исполняется».

Схема алгоритма обработки заявки «Завершение процесса» представлена на рисунке 8.

Алгоритм обработки заявки «Запрос на ввод-вывод»:

1) сначала нужно удостовериться в том, что процесс с заданным PID существует и находится в состоянии «Исполняется». Если это не так, то завершить алгоритм;
2) переключить данный процесс в состояние «Ожидание»;
3) выбрать процесс для исполнения согласно дисциплине;
4) если процесса для выполнения нет, то завершить алгоритм; в противном случае перейти к следующему пункту;
5) извлечь выбранный процесс из очереди и переключить его в состояние «Исполняется».

Схема алгоритма обработки заявки «Запрос на ввод-вывод» представлен на рисунке 9.

Алгоритм обработки заявки «Завершение ввода-вывода»:

1) добавить процесс в соответствующую очередь;
2) переключить процесс в состояние «Готов к исполнению»;
3) выбрать процесс для исполнения согласно дисциплине;
4) если процесса для выполнения нет, то завершить алгоритм; в противном случае перейти к следующему пункту;
5) если на данный момент исполняющегося процесса нет, то извлечь выбранный для исполнения процесс из очереди и переключить его в состояние «Исполняется» и завершить алгоритм; в противном случае перейти к следующему пункту;

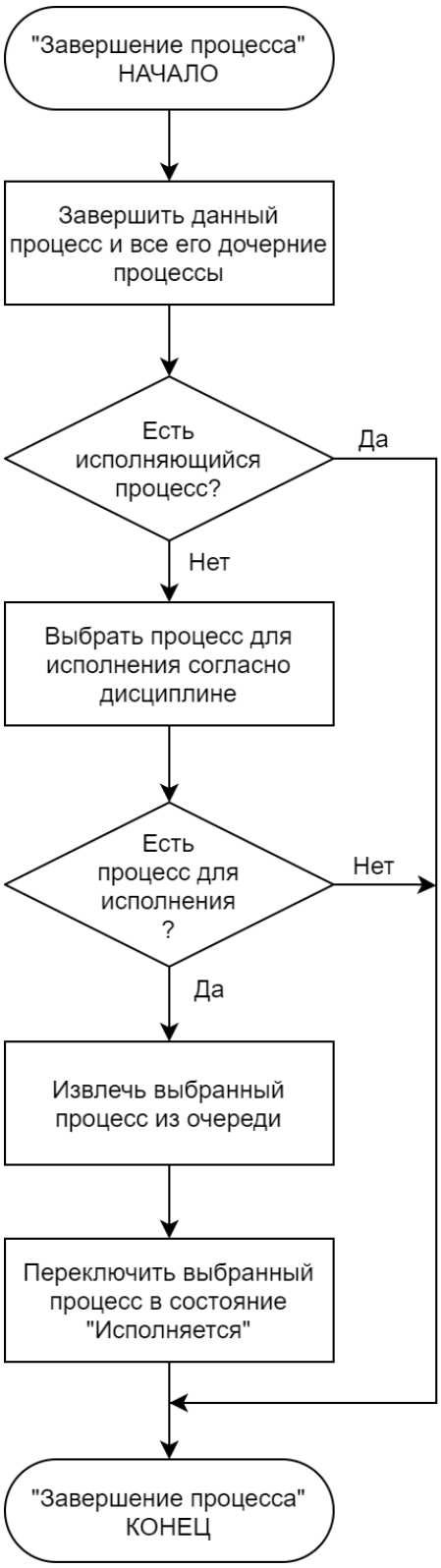


Рисунок 8 – Схема алгоритма обработки заявки «Завершение процесса»

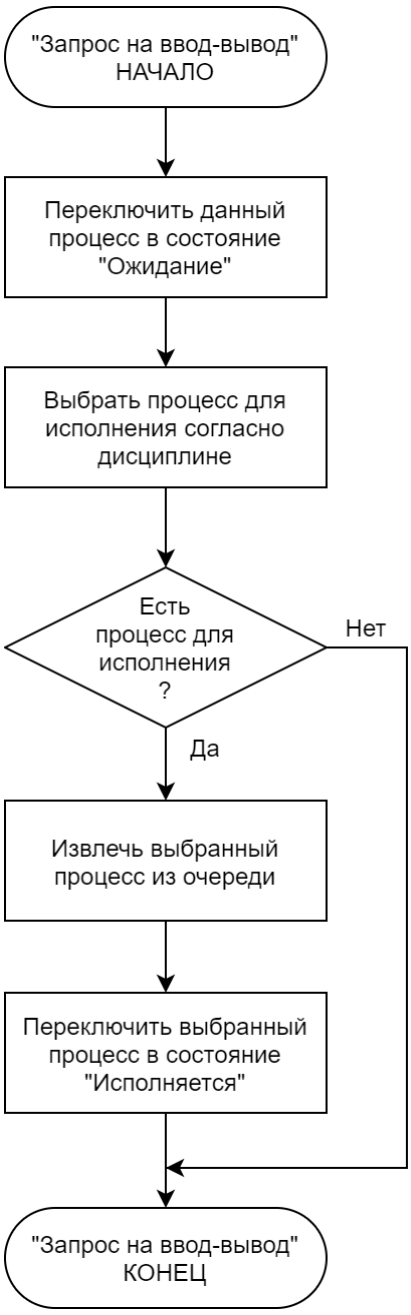


Рисунок 9 – Схема алгоритма обработки заявки «Запрос на ввод-вывод»

6) если используются относительные приоритеты, то завершить алгоритм; в противном случае перейти к следующему пункту;
7) обработать следующие случаи:
   7.1) у текущего исполняющегося процесса приоритет ниже, чем у выбранного. Исполняющийся процесс добавить в

соответствующую очередь и переключить в состояние «Готов к исполнению». Извлечь выбранный для исполнения процесс из очереди и переключить его в состояние «Исполняется»;

7.2) у текущего исполняющегося процесса приоритет не меньше, чем у выбранного. Переключать процессы не нужно.

Схема алгоритма обработки заявки «Завершение ввода-вывода» представлен на рисунке 10.

Алгоритм обработки заявки «Передача управления операционной системе»:

1) добавить процесс в соответствующую очередь;
2) переключить процесс в состояние «Готов к исполнению»;
3) выбрать процесс для исполнения согласно дисциплине;
4) извлечь выбранный процесс из очереди и переключить его в состояние «Исполняется».

Алгоритм обработки заявки «Истечение кванта времени»:

1) добавить процесс в соответствующую очередь;
2) переключить процесс в состояние «Готов к исполнению»;
3) выбрать процесс для исполнения согласно дисциплине;
4) извлечь выбранный процесс из очереди и переключить его в состояние «Исполняется».

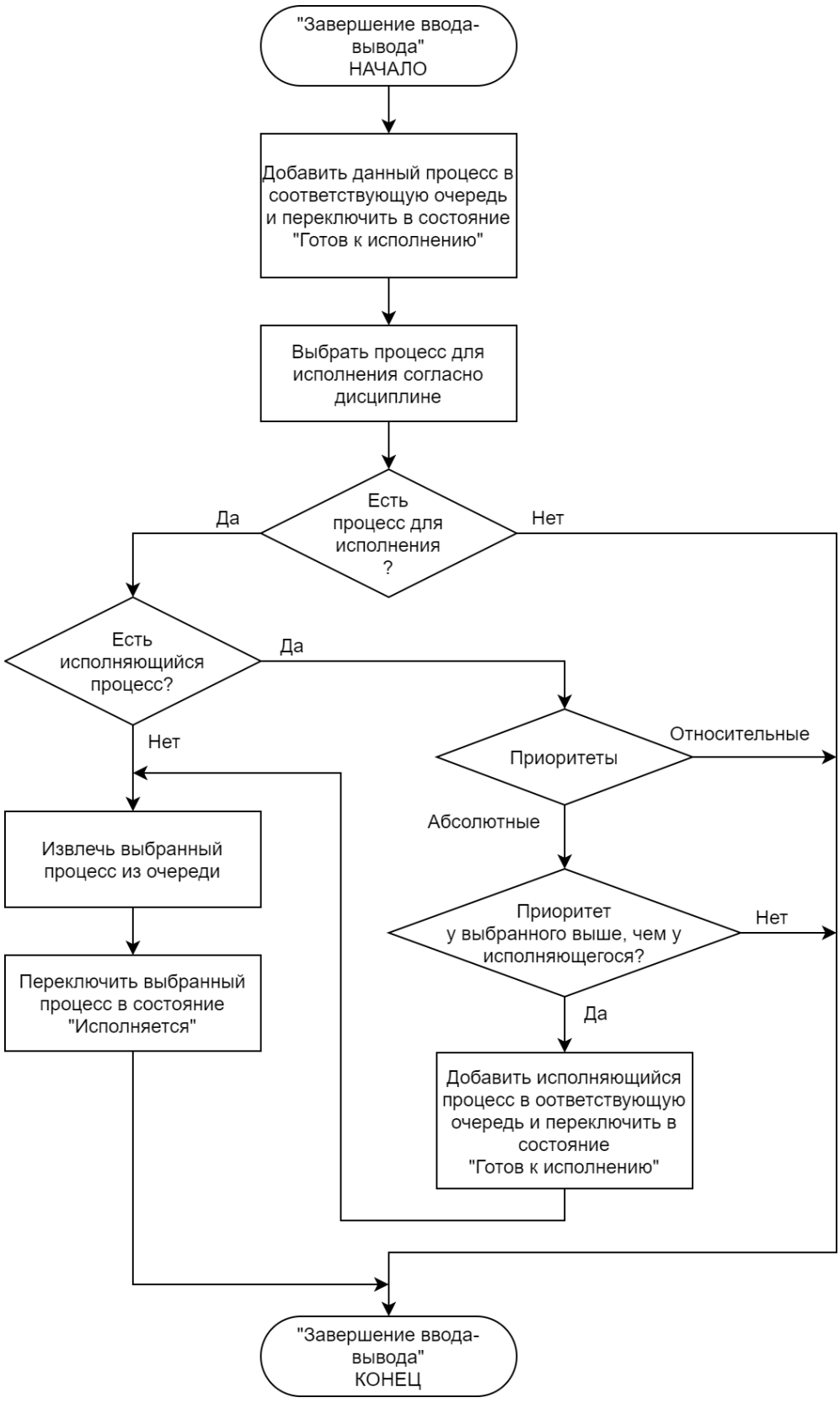


Рисунок 10 – Схема алгоритма обработки заявки «Завершение ввода-вывода»

3.4 Алгоритм генерации заданий

Основная идея алгоритма генерации заданий заключается в том, что каждая следующая заявка создается на основе текущего состояния системы. При этом тип дисциплины, тип заявки, корректность или некорректность заявки, а также параметры заявки определяются случайным образом.

Алгоритм генерации заданий состоит из следующих шагов:

1) выбрать случайным образом дисциплину;
2) создать пустой список заявок задания;
3) на основе текущего состояния системы сгенерировать корректные и некорректные заявки всех типов (по одной на каждый тип);
4) случайным образом определить должна ли текущая заявка быть корректной или нет;
5) если текущая заявка должна быть корректной, то из списка сгенерированных корректных заявок случайным образом выбрать одну и добавить ее в список заявок задания. В противном случае использовать список сгенерированных некорректных заявок;
6) если сгенерировано достаточное количество заявок, то завершить алгоритм, в противном случае перейти к пункту 7;
7) обновить текущее состояние системы в соответствии с выбранной заявкой и дисциплиной, перейти к пункту 3.

Схема алгоритма генерации задания представлена на рисунке 11.

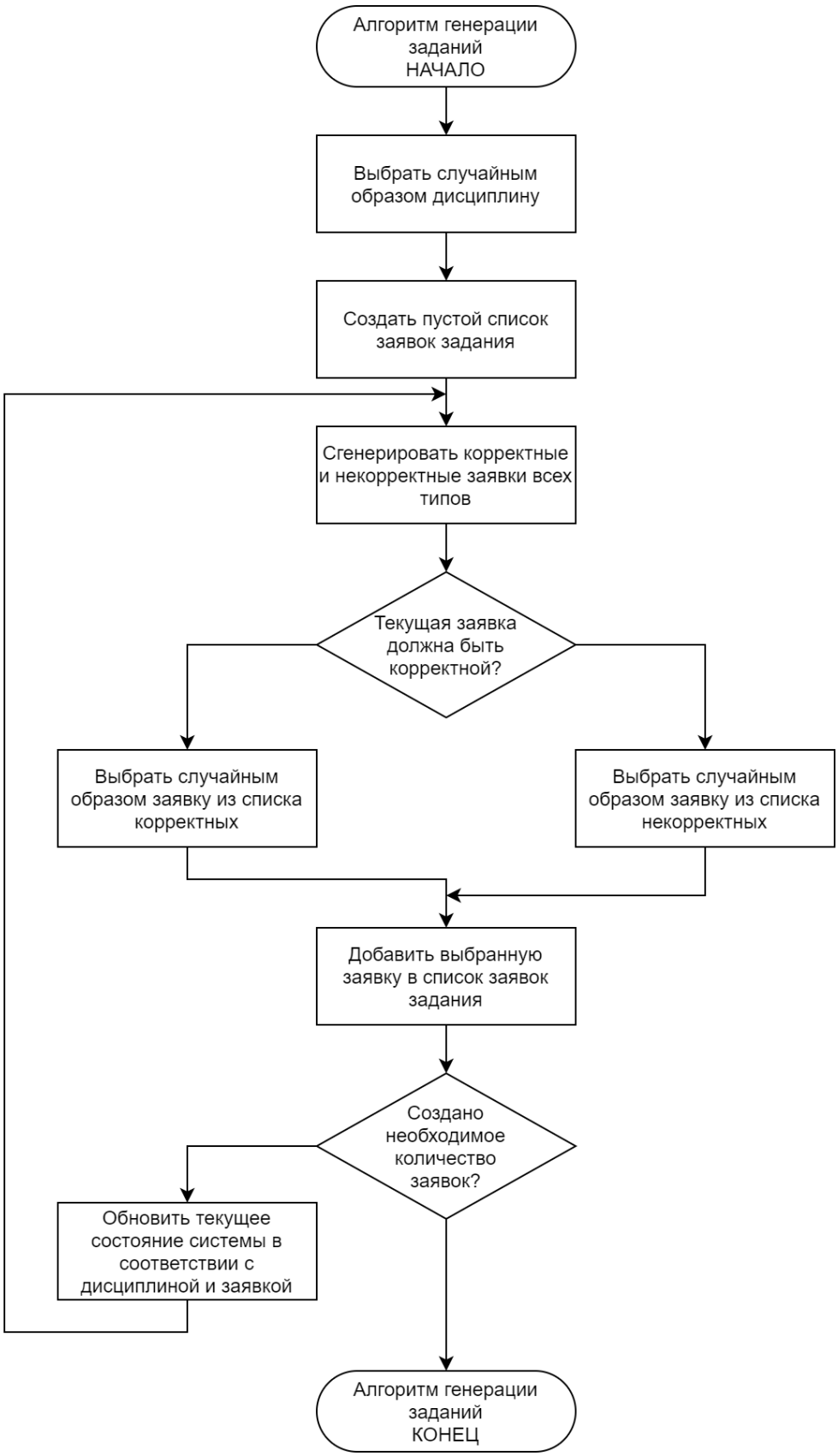


Рисунок 11 – Схема алгоритма генерации задания

3.5 Содержимое файла сессии пользователя

Сессия пользователя определяется списком заданий, состоящих из следующих компонентов:

- список всех заявок;
- количество уже выполненных заявок;
- количество допущенных пользователем ошибок;
- тип дисциплины;
- журнал действий пользователя;
- состояние модели после последней выполненной заявки.

Для каждого задания все перечисленные выше параметры должны быть сохранены в файл. Последний параметр позволяет защитить содержимое файла от изменений, направленных на подделку результатов выполнения задания, потому что для этого атакующему необходимо будет самостоятельно обработать заявки, чтобы получить искомое состояние модели.

Заявки для заданий по планировщику памяти определяются следующими параметрами (различными, в зависимости от типа заявки):

- идентификатор процесса;
- количество запрашиваемой памяти в байтах;
- адрес начала блока памяти.

Заявки для заданий по планировщику процессов определяются следующими параметрами (различными, в зависимости от типа заявки):

- идентификатор процесса;
- идентификатор родительского процесса;
- базовый приоритет;
- заявленное время выполнения.

Состояние модели для планировщика памяти определяется списком

всех блоков памяти, упорядоченных по начальному адресу, а также списком свободных блоков памяти. При этом каждый блок памяти определен следующими параметрами:

- адрес начала блока;
- размер блока в страницах;
- идентификатор процесса, которому принадлежит данный блок.

Состояние модели для планировщика процессов определяется списком процессов и состоянием очередей процессов. У процесса имеются следующие параметры:

- идентификатор процесса;
- идентификатор процесса родительского процесса;
- базовый приоритет;
- текущий приоритет;
- состояние (исполняется, в очереди, в ожидании);
- текущее время выполнения процесса;
- заявленное время выполнения процесса.

3.6 Обеспечение защиты файлов пользовательских сессий от несанкционированного доступа

В данном случае под несанкционированным доступом имеется в виду как открытие файла в программе пользователем, который его не сохранял, так и попытка расшифровки файла с последующей обратной шифровкой с другим ключом.

С целью защиты от подобных действие содержимое файлов шифруется с помощью симметричного алгоритма AES-256 в режиме CBC. Данный алгоритм хорошо проанализирован [2], широко используется при разработке программного обеспечения [2], а также имеет аппаратную

реализацию для большинства x86-совместимых процессоров [3].

Алгоритм AES представляет блок данных в виде двумерного байтового массива размером 4×4. Все операции производятся над отдельными байтами массива, а также над независимыми столбцами и строками [11].

В каждом раунде алгоритма выполняются следующие преобразования:

1) операция SubBytes, представляющая собой табличную замену каждого байта массива данных;
2) операция ShiftRows, которая выполняет циклический сдвиг влево всех строк массива данных, за исключением нулевой. Сдвиг i-ой строки массива (для $i = 1, 2, 3$) производится на i байт;
3) операция MixColumns. Выполняет умножение каждого столбца массива данных на полином $a(x) = 3x^3 + x^2 + x + 2$ по модулю $x^4 + 1$;
4) операция AddRoundKey выполняет наложение на массив данных материала ключа. А именно, на i-ый столбец массива данных ($i = 0 \ldots 3$) побитовой логической операцией «исключающее или» накладывается определенное слово расширенного ключа W_{4r+i}, где r – номер текущего раунда алгоритма, начиная с 1.

Для ключа размером 256 бит количество раундов алгоритма составляет 14. Перед первым раундом алгоритма выполняется предварительное наложение материала ключа с помощью операции AddRoundKey, которая выполняет наложение на открытый текст первых четырех слов расширенного ключа $W_0 \ldots W_3$. Последний же раунд отличается

от предыдущих тем, что в нем не выполняется операция MixColumns.

Расшифрование выполняется применением обратных операций в обратной последовательности [11]. Соответственно, перед первым раундом расшифрования выполняется операция AddRoundKey (которая является обратной самой себе), выполняющая наложение на шифротекст четырех последних слов расширенного ключа, т. е. $W_{4R} \dots W_{4R+3}$. Затем выполняется R раундов расшифрования, каждый из которых выполняет следующие преобразования:

1) операция InvShiftRows выполняет циклический сдвиг вправо трех последних строк массива данных на то же количество байт, на которое выполнялся сдвиг операцией ShiftRows при шифровании;
2) операция InvSubBytes выполняет побайтно обратную табличную замену;
3) операция AddRoundKey, как и при шифровании, выполняет наложение на обрабатываемые данные четырех слов расширенного ключа $W_4 r \dots W_{4r+3}$. Однако, нумерация раундов r при расшифровании производится в обратную сторону – от $R-1$ до 0;
4) операция InvMixColumns выполняет умножение каждого столбца массива данных аналогично операции MixColumns, но умножение производится на полином $a^{-1}(x) = Bx^3 + Dx^2 + 9x + E$.

Аналогично шифрованию, последний раунд расшифрования не содержит операцию InvMixColumns.

Схемы раундов шифрования и расшифрования представлены на рисунке 12.

В качестве ключа используется значение контрольной суммы SHA-256 от фамилии студента и секретной последовательности, хранящейся в

исполняемом файле приложения. За сбор пользовательских данных отвечает модуль графического интерфейса.

Таким образом, файл может открыть в программе только тот пользователь, который его сохранил.

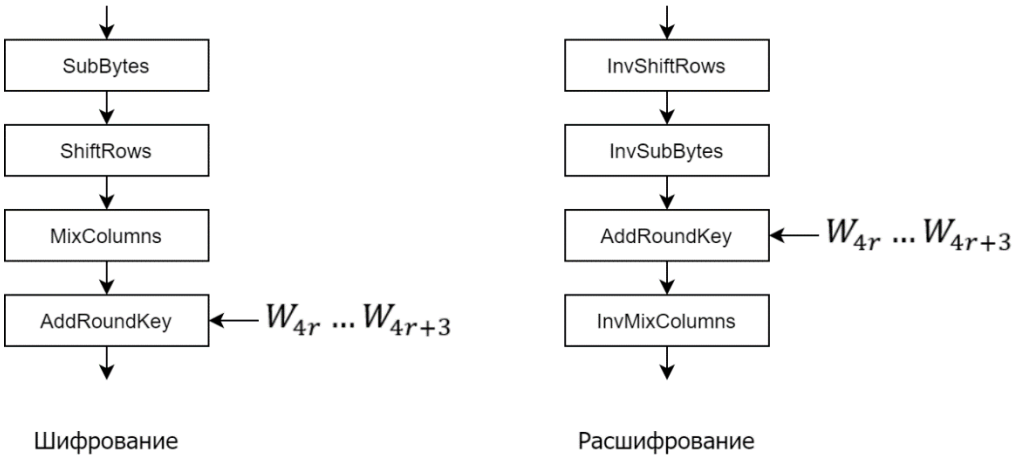


Рисунок 12 – Схема раундов шифрования и расшифрования AES

Выводы по разделу

На этапе разработки структуры приложения была составлена модульная структура, определены зависимости между модулями, разработаны алгоритмы функционирования планировщиков памяти и процессов; составлен общий алгоритм генерации заданий, определен состав содержимого файла сессии пользователя. Также обеспечена защита файлов от несанкционированного доступа.

Было замечено, что алгоритмы обработки заявок обоих планировщиков имеют общие части. Например, в планировщике процессов алгоритмы обработки заявок «Создан новый процесс» и «Завершение ввода-вывода» отличаются лишь первыми несколькими пунктами (1-4 для первого и 1-2 для второго). То же можно сказать и об алгоритмах «Запрос на ввод-вывод» и «Завершение процесса», которые также отличаются в первых

пунктах (1-2 для первого и 1-3 для второго). В планировщике памяти алгоритмы «Создан новый процесс» и «Выделение памяти существующему процессу» отличаются лишь в первых пунктах. Данные особенности можно учесть при программной реализации, вынеся общие части в отдельные функции.

4 Программная реализация

В данном разделе описаны выбранные инструменты для реализации приложения (язык программирования, библиотеки), разработан графический интерфейс пользователя, а также представлены детали реализации разработанных ранее модулей.

4.1 Выбор инструментов разработки

Так как приложение должно запускаться на различных ОС, а также быть простым в установке и запуске, было принято решение выбрать компилируемый язык программирования с возможностью создания родных для платформы исполняемых файлов. Помимо прочего, для реализации графического интерфейса необходимо выбрать кроссплатформенную библиотеку для его построения.

Среди доступных языков программирования имеются:

– Golang;

– C++;

– Rust;

– C# (.NET Core 3).

Среди кроссплатформенных библиотек для построения графического интерфейса имеются:

– GTK (от сокращения GIMP Toolkit) – кроссплатформенная библиотека элементов интерфейса. Написана на C, доступны интерфейсы для других языков программирования, в том числе для C++, C#, Python. Для проектирования графического интерфейса доступно приложение Glade [4];

– Qt. Кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++. Включает в

себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML. Комплектуется визуальной средой разработки графического интерфейса Qt Designer, позволяющей создавать диалоги и формы в режиме WYSIWYG [5, 6];

– wxWidgets. Кроссплатформенная библиотека инструментов с открытым исходным кодом для разработки кроссплатформенных на уровне исходного кода приложений. Основным применением wxWidgets является построение графического интерфейса пользователя, однако библиотека включает большое количество других функций и используется для создания весьма разнообразного ПО. Важная особенность wxWidgets: в отличие от некоторых других библиотек (GTK, Qt и др.), она максимально использует «родные» графические элементы интерфейса операционной системы всюду, где это возможно. Это существенное преимущество для многих пользователей, поскольку они привыкают работать в конкретной среде, и изменения интерфейса программ часто вызывают затруднения в их работе. Написана на C++. Для проектирования графического интерфейса доступно приложение wxGlade [7];

– AvaloniaUI. Кроссплатформенный XAML-фреймворк для построения графических интерфейсов пользователя для платформ .NET Framework, .NET Core и Mono [8, 9].

Следует отметить, что все перечисленные выше фреймворки и библиотеки имеют поддержку HiDPI-дисплеев.

При этом нужно учитывать, что не все комбинации языков программирования с библиотеками графического интерфейса возможны, удобны в разработке и достаточно стабильны. Для языков Golang и Rust не имеются стабильные версии перечисленных библиотек, для GTK сборка

C++-приложений под Windows требует нетривиальной настройки окружения вплоть до эмуляции Linux-окружения в Windows, а AvaloniaUI доступна только под C# и не имеет еще стабильной версии. Среди доступных вариантов остаются:

– C++ и wxWidgets;
– C# и GTK;
– C++ и Qt.

В ходе ознакомления с библиотекой wxWidgets были выявлены следующие недостатки: недостаточно подробная документация, нетривиальная настройка режима Drag’n’Drop (необходим для перемещения элементов ГПИ, представляющих блоки памяти), скудный функционал конструктора форм wxGlade.

У GTK графические элементы выглядят достаточно непривычно в сравнении с родными для Windows приложениями; имеет те же недостатки, что и wxWidgets.

По степени интеграции выигрывает Qt со средой разработки QtCreator.

В качестве системы сборки выбрана CMake – одна из наиболее популярных среди проектов, написанных на C++.

Для шифрования файлов сессий была выбрана библиотека Botan. Данная библиотека имеет хорошую документацию, написана на C++ и предоставляет удобные способы подключения к проектам, написанных на C++ [10].

В качестве формата файла задания был выбран JSON, потому что он имеет достаточно простой синтаксис, а также большой выбор библиотек для работы с ним. Для данного приложения была выбрана библиотека nlohmann::json.

4.2 Реализация модулей приложения

В данном разделе представлены детали реализации разработанных ранее модулей.

4.2.1Модуль обработки заявок планировщика памяти

В данном модуле реализованы алгоритмы обработки заявок планировщика памяти, а также определены следующие необходимые типы:

- структура MemoryBlock. Описывает отдельно взятый блок памяти. Основными атрибутами являются: PID (тип int), адрес начала блока (тип int) и размер (тип int). Для доступа к атрибутам реализованы методы pid(), address() и size() соответственно;
- структура MemoryState. Описывает состояние памяти. Содержит два атрибута: список блоков памяти, упорядоченных по начальному адресу (тип std::vector<MemoryBlock>) и список свободных блоков памяти (тип std::vector<MemoryBlock>), упорядоченных в соответствии с выбранной дисциплиной, однако их порядок может меняться при применении операций;
- структуры, описывающие заявки:
  - CreateProcessReq. Основные атрибуты: PID, количество запрашиваемой памяти в байтах, количество запрашиваемой памяти в страницах;
  - TerminateProcessReq. Атрибут: PID;
  - AllocateMemory. Основные атрибуты: PID, количество запрашиваемой памяти в байтах, количество запрашиваемой памяти в страницах;
  - FreeMemory. Основные атрибуты: PID, адрес начала блока памяти.

Для обеспечения возможности хранить заявки различного типа в таких контейнерах, как std::vector, был добавлен тип-сумма MemRequest,

являющийся типом std::variant.

В ходе анализа алгоритмов было выяснено, что дисциплины отличаются между собой только способом сортировки свободных блоков памяти. Поэтому было принято решение закодировать алгоритмы в виде методов отдельного класса – MemAbsStrategy, а особенность каждой дисциплины (способ сортировки блоков памяти) определить через виртуальный метод sortFreeBlocks(), реализованный в классах дисциплин, производных от базового класса MemAbsStrategy:

- FirstAppropriateStrategy – дисциплина «Первый подходящий»;
- MostAppropriateStrategy – дисциплина «Наименее подходящий»;
- LeastAppropriateStrategy – дисциплина «Наименее подходящий».

Диаграмма классов данного модуля представлена на рисунке 13.

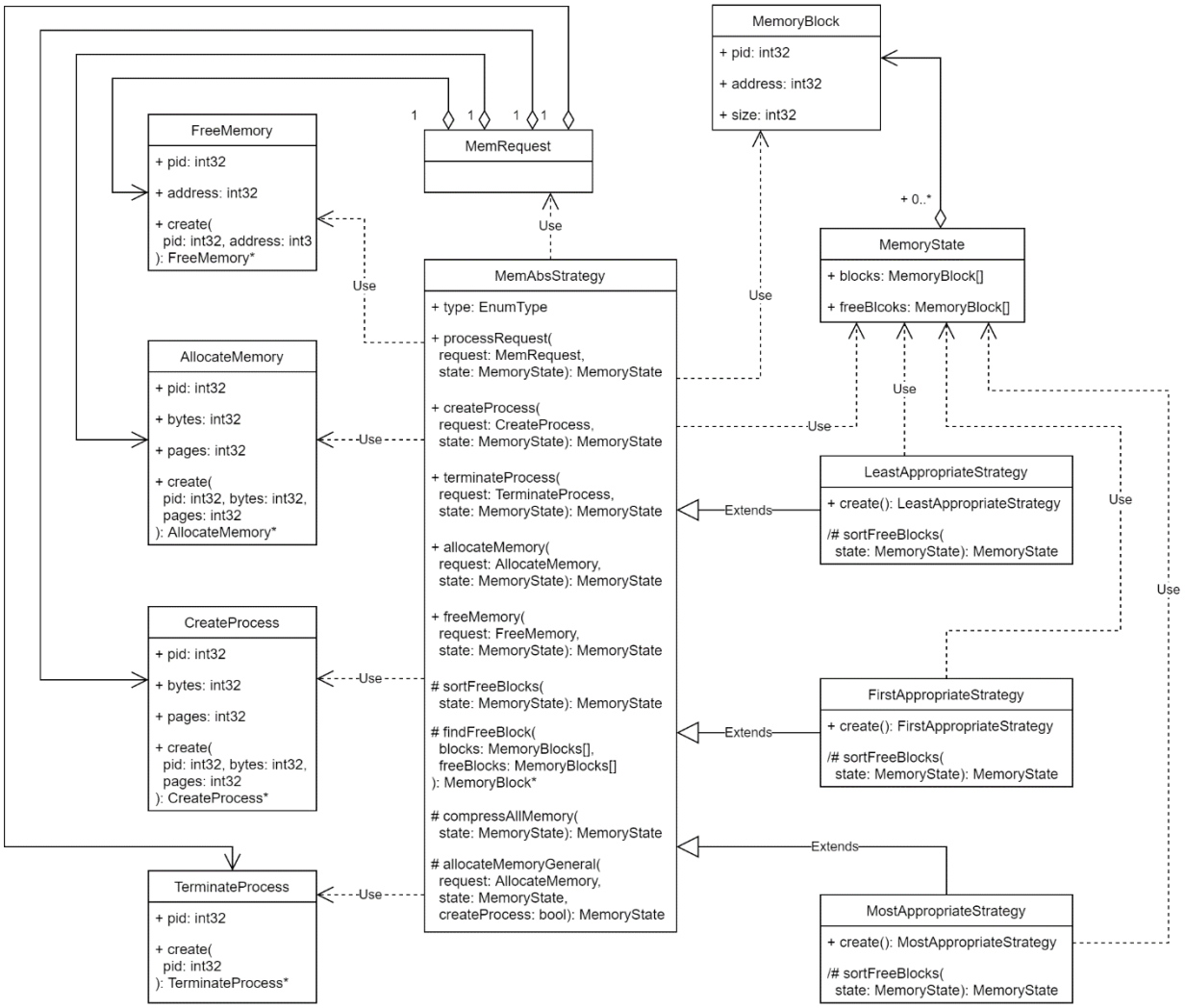


Рисунок 13 – Диаграмма классов модуля обработки заявок планировщика памяти

4.2.2Модуль обработки заявок планировщика процессов

В данном модуле реализованы алгоритмы обработки заявок планировщика процессов, а также определены следующие необходимые типы:

– структура Process. Описывает отдельно взятый процесс. Основными атрибутами являются: PID (тип int), PID родительского процесса, базовый и текущий приоритеты (тип int), состояние (тип int), текущее и заявленное времена выполнения процесса (тип int). Для доступа к атрибутам реализованы методы pid(), ppid() и basePriority(), priority(), timer() и

worktime() соответственно;

– структура ProcessesState. Описывает состояние системы. Содержит два атрибута: список процессов (тип std::vector< Process>) и список из 16 очередей процессов (тип std::array<std::deque<int>, 16>);

– структуры, описывающие заявки:

  - CreateProcessReq. Основные атрибуты: PID (тип int), PID родительского процесса (тип int), базовый приоритет (тип int), заявленное время выполнения (тип int);
  - TerminateProcessReq. Атрибуты: PID (тип int);
  - InitIO. Атрибуты: PID (тип int);
  - TerminateIO. Атрибуты: PID (тип int) и augment (тип int). augment используется только в дисциплине WinNT;
  - TransferControl. Атрибуты: PID (тип int);
  - TimeQuantumExpired.

Для обеспечения возможности хранить заявки различного типа в контейнерах, подобных std::vector, был добавлен тип-сумма ProcRequest, являющийся типом std::variant.

Дисциплины планирования выполнены в виде отдельных классов, реализующих интерфейс ProcAbsStrategy, который содержит следующие методы:

– toString() – возвращает строковое обозначение стратегии;

– type() – возвращает числовое обозначение стратегии;

– processRequest(ProcRequest, ProcessesState) – обрабатывает заявки всех типов;

– schedule(ProcessesState) – вызывает планировщик.

Диаграмма классов данного модуля представлена на рисунке 14.

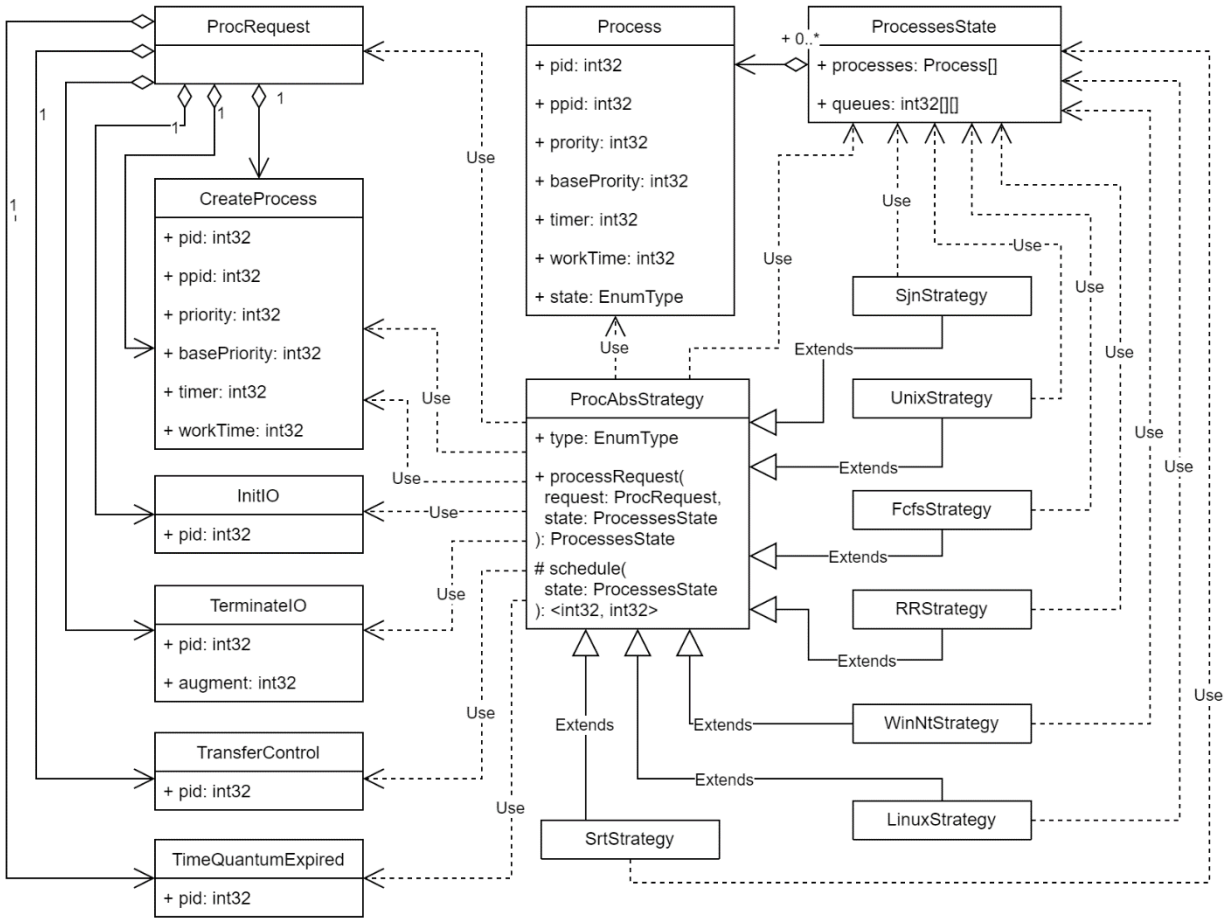


Рисунок 14 – Диаграмма классов модуля обработки заявок планировщика процессов

4.2.3Модуль загрузки и сохранения пользовательских сессий в файл

Данный модуль реализует функционал сохранения текущего прогресса выполнения заданий в файл, его загрузку из файла с защитой от непредвиденных изменений, а также шифрование содержимого файла.

Задания в файле сохранены в виде массива JSON-объектов. Структура объекта задания по планировщику памяти представлена в таблице 1.

Таблица 1 – Структура объекта задания по планировщику памяти

Поле	Тип	Описание
type	String	Тип задания. Значение: “MEMORY_TASK”
strategy	String	Название дисциплины. Допустимые значения: “FIRST_APPROPRIATE”, “MOST_APPROPRIATE”, “LEAST_APPROPRIATE”
completed	Number	Количество обработанных заявок
fails	Number	Количество допущенных пользователем ошибок
state	Object	Объект, описывающий состояние памяти
actions	Array	Массив строк, содержащих информацию о действиях пользователя для каждой заявки
requests	Array	Массив заявок, которые диспетчер должен обработать

В объекте, описывающем состояние памяти, (поле “state”) хранятся списки всех блоков памяти, упорядоченных по адресу. Структуры объектов состояния памяти, блока памяти и заявки представлены в таблицах 2, 3 и 4 соответственно.

Таблица 2 – Структура объекта состояния памяти

Поле	Тип	Описание
blocks	Array	Массив из дескрипторов всех доступных блоков памяти

Таблица 3 – Структура объекта блока памяти

Поле	Тип	Описание
pid	Number	PID процесса, которому выделен данный блок или -1, если блок свободный
address	Number	Адрес начала блока памяти в страницах
size	Number	Размер блока памяти в страницах

Таблица 4 – Структура объекта заявки

Заявка на создание процесса		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “CREATE_PROCESS”
pid	Number	PID процесса
bytes	Number	Запрашиваемый объем памяти в байтах
Заявка на завершение процесса		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “TERMINATE_PROCESS”
pid	Number	PID процесса
Заявка на выделение блока памяти процессу		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “ALLOCATE_MEMORY”
pid	Number	PID процесса
bytes	Number	Запрашиваемый объем памяти в байтах
Заявка на освобождение блока памяти		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “FREE_MEMORY”
pid	Number	PID процесса
address	Number	Адрес начала блока памяти в страницах

Структура объекта задания по планировщику процессов представлена в таблице 5.

Таблица 5 – Структура объекта задания по планировщику процессов

Поле	Тип	Описание
type	String	Тип задания. Значение: “PROCESSES_TASK”
strategy	String	Название дисциплины. Допустимые значения: “FCFS”, “SJN”, “SRT”, “ROUNDROBIN”, “UNIX”, “WINNT”, “LINUXO1”
completed	Number	Количество обработанных заявок
fails	Number	Количество допущенных пользователем ошибок

Продолжение таблицы 5

state	Object	Объект, описывающий состояние процессов
actions	Array	Массив строк, содержащих информацию о действиях пользователя для каждой заявки
requests	Array	Массив заявок, которые диспетчер должен обработать

В объекте, описывающем состояние процессов (поле “state”), хранится список всех процессов, а также содержимое всех очередей. Структуры объектов состояния процессов, заявки и дескриптора процесса представлены в таблицах 6, 7 и 8 соответственно.

Таблица 6 – Структура объекта состояния процессов

Поле	Тип	Описание
processes	Array	Массив из дескрипторов процессов
queues	Array	Массив из 16 очередей, каждая - массив из идентификаторов процессов

Таблица 7 – Структура объекта заявки

Заявка на создание процесса		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “CREATE_PROCESS”
pid	Number	PID процесса
ppid	Number	PID родительского процесса или -1
priority	Number	Приоритет
basePriority	Number	Базовый приоритет
timer	Number	Время работы
workTime	Number	Заявленное время работы
Заявка на завершение процесса		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “TERMINATE_PROCESS”
pid	Number	PID процесса

Продолжение таблицы 7

Заявка на инициализацию ввода/вывода		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “INIT_IO”
pid	Number	PID процесса
Заявка на завершение ввода/вывода		
Поле	Тип	Описание
type	String	Тип заявки. Значение: “TERMINATE_IO”
pid	Number	PID процесса
augment	Number	Прибавка к текущему приоритету
Заявка на передачу управления операционной системе		
type	String	Тип заявки. Значение: “TRANSFER_CONTROL”
pid	Number	PID процесса
Заявка на обработку события "Истек квант времени"		
type	String	Тип заявки. Значение: “QUANTUM_EXPIRED”

Таблица 8 – Структура объекта дескриптора процесса

Поле	Тип	Описание
pid	Number	PID процесса
ppid	Number	PID родительского процесса или -1
priority	Number	Приоритет
basePriority	Number	Базовый приоритет
timer	Number	Время работы
workTime	Number	Заявленное время работы
state	String	Состояние процесса. Допустимые значения: “ACTIVE”, “EXECUTING”, “WAITING”

Загрузка сессий из файла реализована в функции loadTasks(), сохранение – saveTasks().

Диаграмма классов данного модуля представлена на рисунке 15.

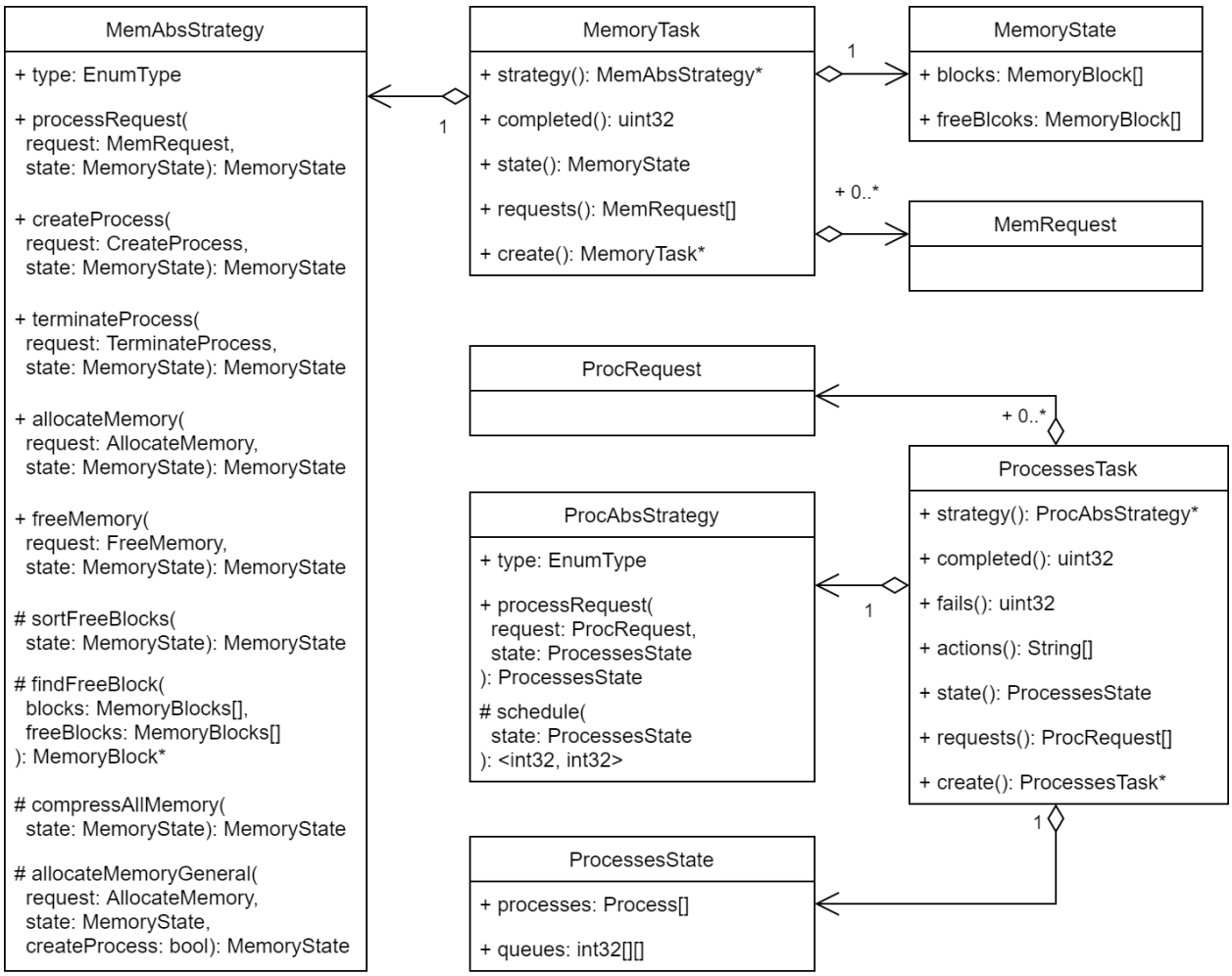


Рисунок 15 – Диаграмма классов модуля загрузки и сохранения пользовательских сессий в файл

4.2.4Модуль-генератор заданий

Генератор заданий для планировщика памяти реализован следующими функциями:

– функции генерации заявок на основе данного состояния памяти: genCreateProcess(), genTerminateProcess(), genAllocateMemory(), genFreeMemory();

– функция выбора случайным образом дисциплины;

– функция generate(), использующая описанные выше функции для построения задания. Количество заявок по умолчанию – 40.

Опытным путем было выбрано соотношение между количеством

некорректных и корректных заявок – 1 к 7.

Также в генераторе установлено ограничение на максимальное количество процессов – 16.

Генератор заданий для планировщика процессов исполнен в виде набора функций и классов, реализующих алгоритм генерации заданий для каждой дисциплины в отдельности с вынесением общих частей в базовый класс AbstractTaskGenerator, содержащий следующие методы:

– basis(ProcessesState, bool) – создает экземпляры заявок (корректные или некорректные) в определенном соотношении по типам. Соотношение между типами заявок зависит количества процессов, находящихся в различных состояниях;

– applyFilters(std::vector<Requests>, Request, bool) – исключает вырожденные случаи следующих типов:

  – две подряд идущие заявки TimeQuantumExpired или TransferControl;
  – после заявки CreateProcessReq идет заявка TransferControl или TerminateProcessReq с таким же PID;
  – после InitIO идет заявка TerminateIO с таким же PID;
  – первая в списке заявка не является CreateProcessReq;

– generate(ProcessesState, Request, bool) – возвращает список из сгенерированных заявок для выбора одной из них.

Были разработаны следующие функции:

– randStrategy() – выбирает случайным образом дисциплину планирования;

– collectStats(ProcessesState) – собирает статистику по количеству процессов, находящихся в состояниях ожидания, исполнения и готовности;

– generate() – использует описанные выше функции и классы генераторов для построения задания. Количество заявок по умолчанию – 40.

Опытным путем были подобраны следующие соотношения:

– соотношение между количеством некорректных и корректных заявок – 1 : 7;

– соотношения между количеством заявок различных типов, генерируемых в функции basis:

– CreateProcessReq : TerminateProcessReq : InitIO : TerminateIO : TransferControl = 2 : 3 : 1 : 1 : 2;

– если процессов, находящихся в состоянии готовности меньше 5, то добавляются 2 заявки CreateProcessReq;

– если процессов, находящихся в состоянии готовности больше 7, то добавляются 3 заявки TerminateProcessReq;

– если процессов, находящихся в состоянии ожидания меньше 1, то добавляются 2 заявки InitIO;

– если процессов, находящихся в состоянии ожидания больше 3, то добавляются 2 заявки TerminateIO;

– для вытесняющих дисциплин добавляются 2 заявки TimeQuantumExpired;

– в 3 из 5 случаев заявки CreateProcessReq соответствуют созданию дочерних процессов.

Также в генераторе установлено ограничение на максимальное количество процессов – 24.

Диаграмма классов данного модуля представлена на рисунке 16.

4.2.5Модуль графического интерфейса

Было решено организовать графический интерфейс программы следующим образом. Основное окно программы разделено на две области:

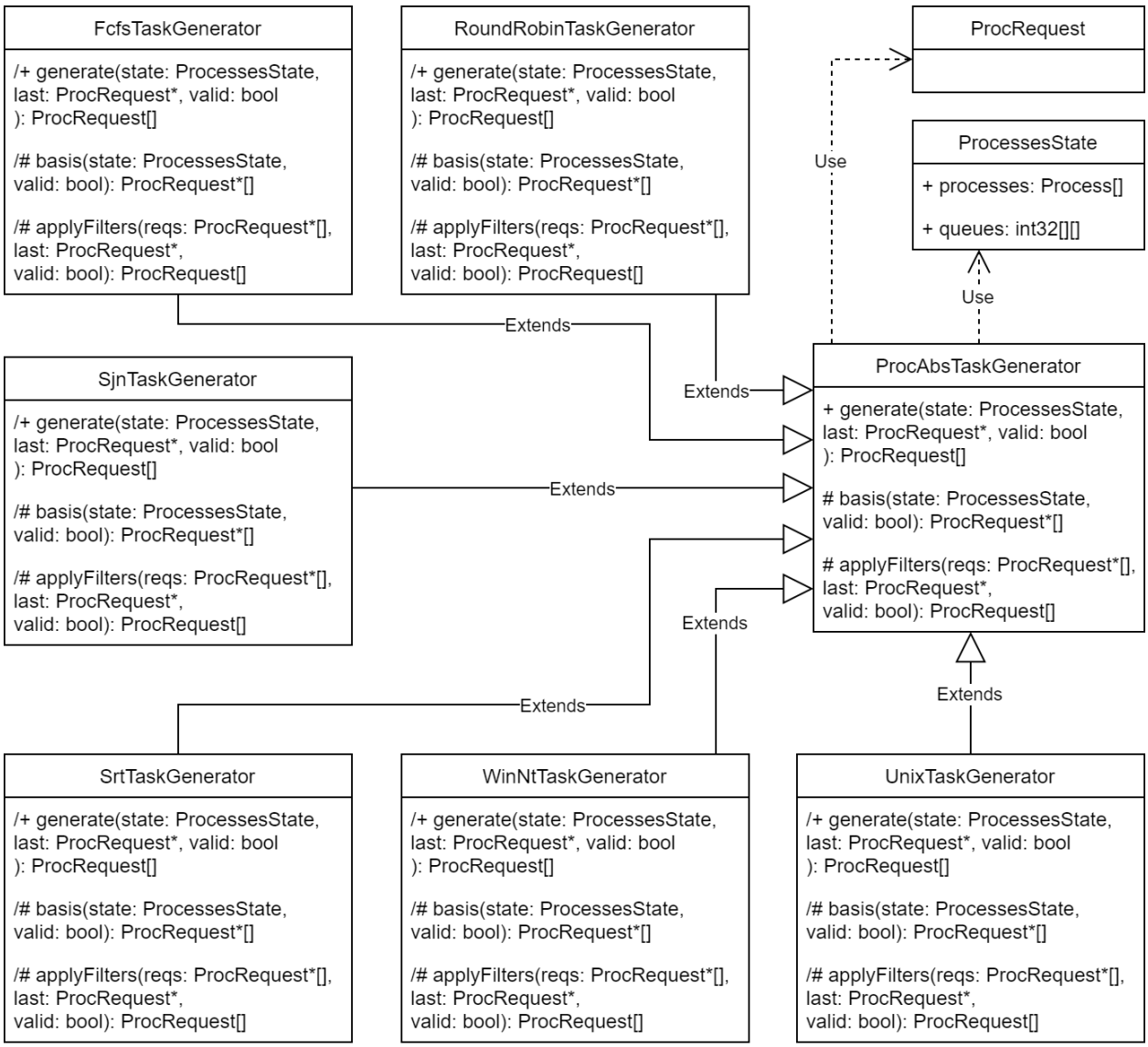


Рисунок 16 – Диаграмма классов модуля-генератора заданий

строка меню и область вкладок, в которых размещены задания. При запуске пользователю выводится диалоговое окно с предложением ввести свое имя. Ввод имени обязателен, так как с помощью него будет осуществлено шифрование сохраняемых файлов. Имя пользователя будет выведено в заголовке окна. При попытке закрыть основное окно программы будет выводиться диалог подтверждения, чтобы исключить потерю результатов из-за случайного закрытия программы.

В строке меню имеются следующие пункты:

– «Задание». Через этот пункт можно открыть файл сессии,

сохранить сессию в файл, сгенерировать новое задание и завершить работу программы

– «Справка». В этот пункте меню можно получить информацию о программе: версия, автор, краткое описание.

Проанализировав интерфейс предыдущей установки, было принято решение внести в интерфейс вкладок заданий следующие изменения:

– добавить кнопку «Сбросить». При нажатии на нее все изменения, сделанные в ходе обработки текущей заявки, будут сброшены;

– вместо пиктограмм на кнопках будут содержаться соответствующие их смыслу надписи;

– для удобства пользования для кнопок «Подтвердить» и «Сбросить» добавлены клавиатурные сочетания «Alt+Enter» и «Ctrl+Z» (⌘+Z для macOS) соответственно;

– для состояний процессов сделаны более заметные пиктограммы в виде кружков разного цвета;

– в диалоговом окне создания нового процесса в зависимости от текущей дисциплины планирования некоторые поля становятся неактивными;

– для свободных и занятых блоков памяти сделаны более заметные пиктограммы в виде кружков разного цвета.

Диаграмма классов данного модуля представлена на рисунке 17.

Экранные формы основного окна программы представлены на рисунках 18-19. Экранные формы диалогового окна создания нового процесса и окна ввода имени пользователя представлены на рисунках 20-21.

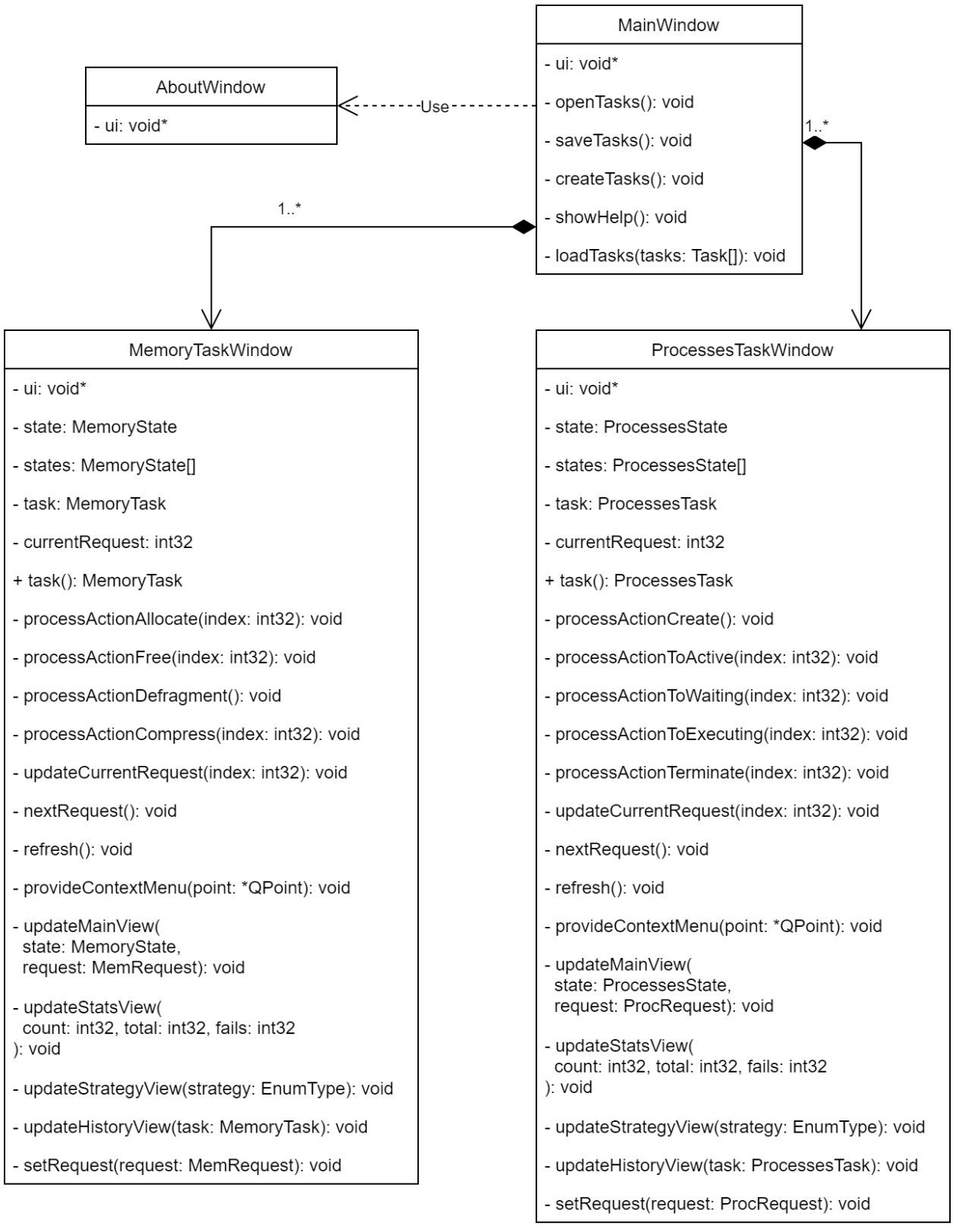


Рисунок 17 – Диаграмма классов модуля графического интерфейса

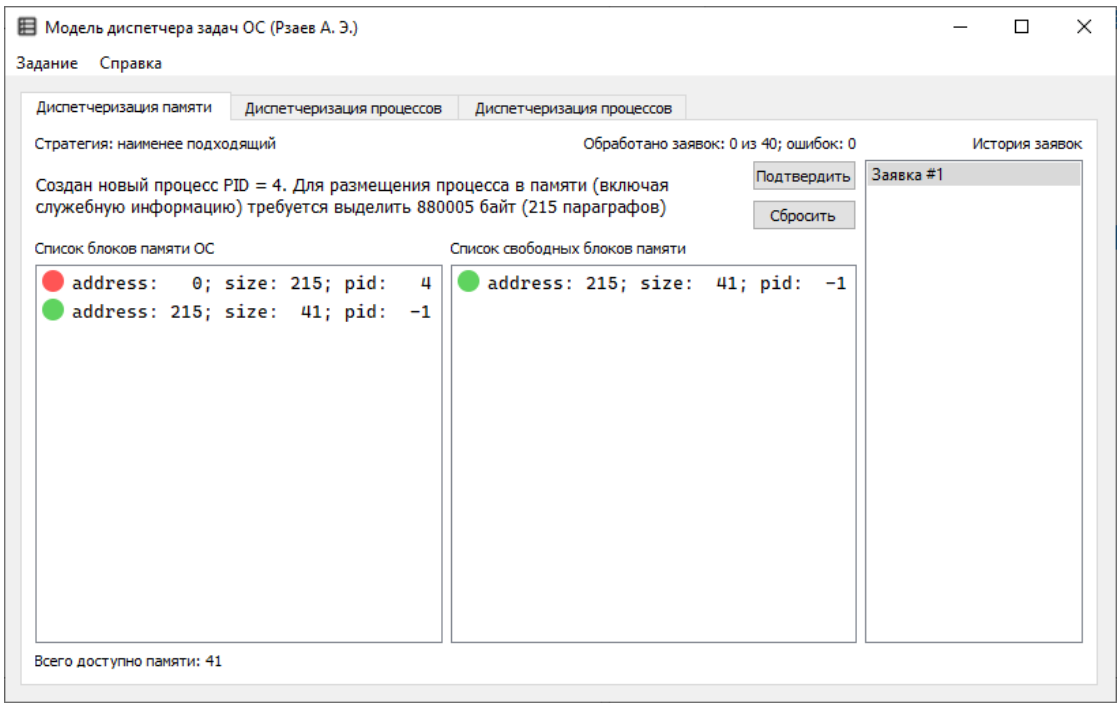


Рисунок 18 – Экранная форма основного окна программы – задание по планировщику памяти

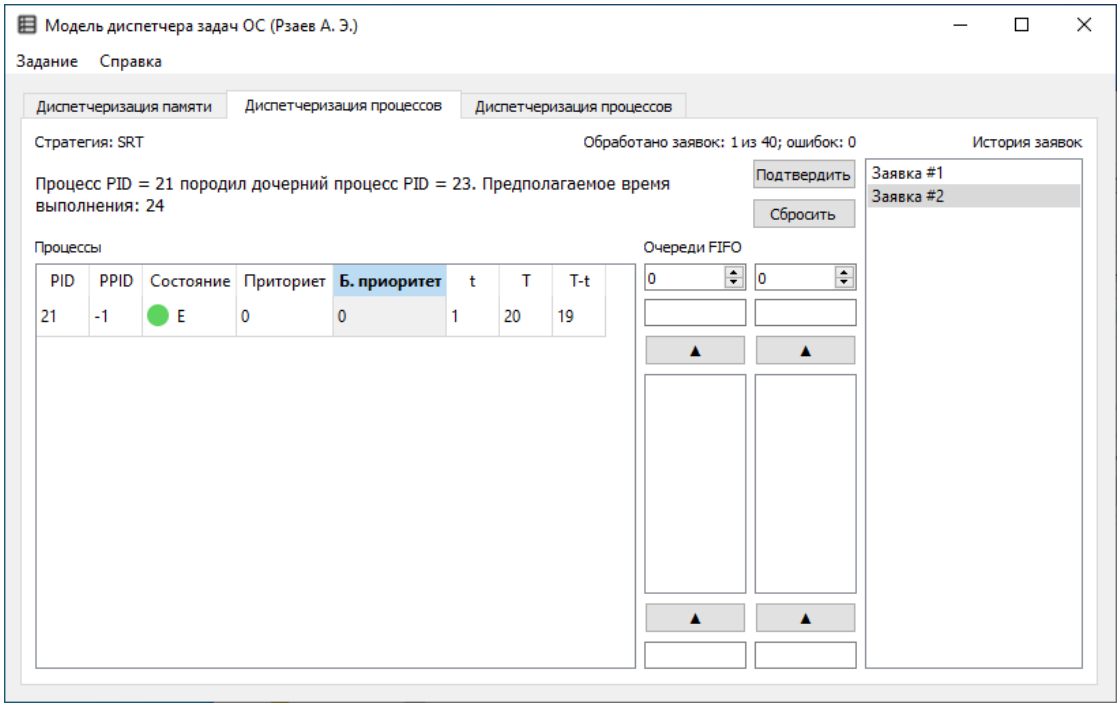


Рисунок 19 – Экранная форма основного окна программы – задание по планировщику процессов

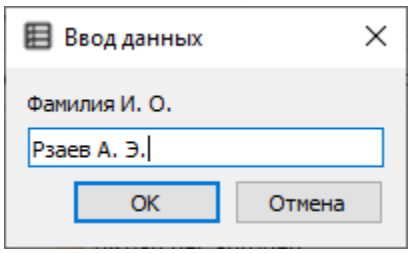


Рисунок 20 – Экранная форма окна ввода имени пользователя

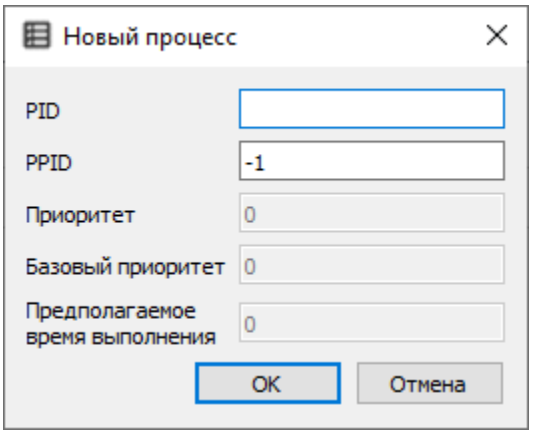


Рисунок 21 – Экранная форма окна создания нового процесса

Выводы по разделу

На основе разработанной структуры приложения и алгоритмов функционирования была выполнена программная реализация приложения. Модули приложения были реализованы в виде классов и функций; в графический интерфейс пользователя было внесено несколько улучшений, а также был разработан формат файлов заданий. Диаграмма разработанных классов представлена в приложении А, листинг кода представлен в приложении В.

Заключение

В ходе выполнения выпускной квалификационной работы было разработано программное обеспечение – программа моделирования работы планировщиков памяти и процессов в операционных системах. Изначально предполагалось исправление и доработка существующей установки, однако такая задача оказалось крайне сложной в сравнении с разработкой нового приложения.

Были устранены следующие проблемы:

– доступность приложения только для ОС Windows. Запуск новой программной модели обеспечен на Windows, macOS, а также на наиболее популярных дистрибутивах Linux;

– нечеткость, «размытость» интерфейса на дисплеях со сверхвысоким разрешением устранена за счет использования фреймворка Qt;

– недостаточное количество вариантов заданий. В новой программной модели имеется возможность генерации задания;

– стабильность новой программной модели достигнута за счет модульных тестов;

– исходный код программной модели выложен в открытый доступ на сервисе GitHub, чтобы свести к минимуму вероятность его утери, а также упростить дальнейшую разработку.

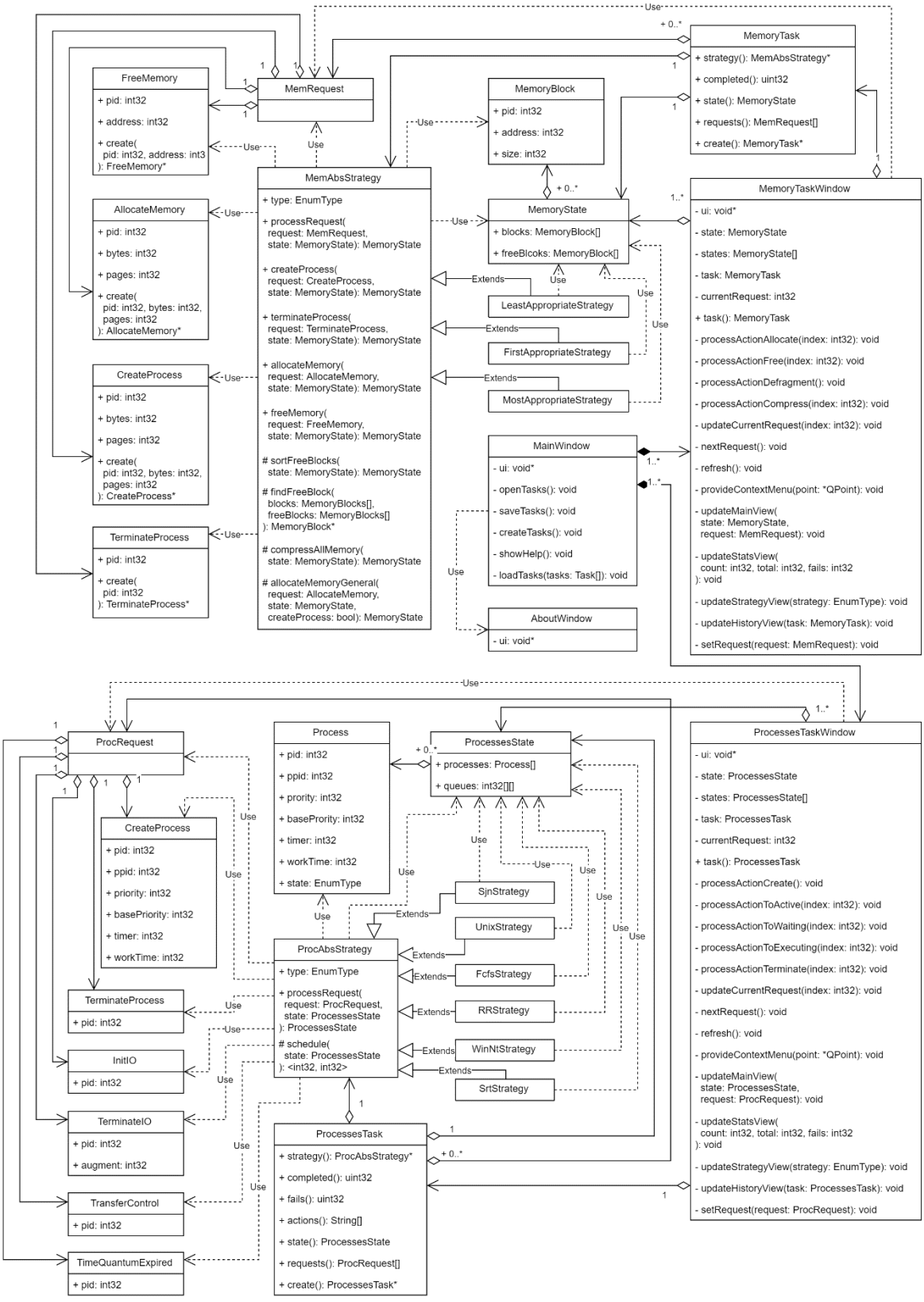
В качестве направления дальнейшего развития можно выбрать разработку отдельного конструктора заданий для преподавателя.

В процессе реализации данное приложение было протестировано, отлажено и готово к эксплуатации.

Приложение А

(обязательное)

Диаграмма классов



Приложение Б

(обязательное)

Авторская справка

Я, Рзаев Али Эльдар оглы, автор выпускной квалификационной работы «Разработка программы моделирования работы планировщиков памяти и процессов в операционных системах» сообщаю, что мне известно о персональной ответственности автора за разглашение сведений, подлежащих защите законами РФ о защите объектов интеллектуальной собственности.

Одновременно сообщаю, что:

1. При подготовке к защите выпускной квалификационной работы не использованы источники (документы, отчеты, диссертации, литература и т.п.), имеющие гриф секретности или «Для служебного пользования» ФГБОУ ВО «Вятский государственный университет» или другой организации.

2. Данная работа не связана с незавершенными исследованиями или уже с завершенными, но еще официально не разрешенными к опубликованию ФГБОУ ВО «Вятский государственный университет» или другими организациями.

3. Данная работа не содержит коммерческую информацию, способную нанести ущерб интеллектуальной собственности ФГБОУ ВО «Вятский государственный университет» или другой организации.

4. Данная работа не является результатом НИР или ОКР, выполняемой по договору с организацией.

5. В предлагаемом к опубликованию тексте нет данных по незащищенным объектам интеллектуальной собственности других авторов.

6. Использование моей дипломной работы в научных исследованиях оформляется в соответствии с законодательством РФ о защите интеллектуальной собственности отдельным договором.

Автор: Рзаев А. Э. «___» _____2020 г. _____

подпись

Сведения по авторской справке подтверждаю: «___» _____2020 г.

Заведующий кафедрой ЭВМ: Д. А. Страбыкин _____

подпись

Приложение В

(обязательное)

Листинг кода

```
#pragma once

#include <stdexcept>
#include <string>

namespace Utils {
class BaseException : public std::logic_error {
public:
  BaseException(const std::string &what_arg) : std::logic_error(what_arg) {}
};

class TaskException : public BaseException {
public:
  TaskException(const std::string &what_arg) : BaseException(what_arg) {}
};
} // namespace Utils
#pragma once

#include <cstdint>
#include <array>
#include <deque>
#include <iomanip>
#include <istream>
#include <map>
#include <ostream>
#include <string>
#include <vector>
#include <utility>

#include <mapbox/variant.hpp>

#include <config.h>

#include "../algo/processes/types.h"
#include "tasks.h"

namespace Utils::details {
using Utils::TaskException;

/**
 *  @brief Создает объект задания "Диспетчеризация памяти" из JSON-объекта.
 *
 *  @param obj JSON-объект.
 *
 *  @return Объект задания.
```

```
 *
 *  @throws Utils::TaskException Исключение возникает в следующих случаях:
 *
 *  "UNKNOWN_STRATEGY" - неизвестный тип стратегии выбора блока памяти;
 *  "UNKNOWN_REQUEST" - неизвестный тип заявки.
 */
inline MemoryTask loadMemoryTask(const nlohmann::json &obj) {
  using namespace MemoryManagement;

  auto toPair = [](auto strategy) -> std::pair<std::string, StrategyPtr> {
    return {strategy->toString(), strategy};
  };

  std::map<std::string, StrategyPtr> strategies = {
      toPair(FirstAppropriateStrategy::create()),
      toPair(MostAppropriateStrategy::create()),
      toPair(LeastAppropriateStrategy::create())};

  auto strategyType = obj["strategy"];
  if (strategies.find(strategyType) == strategies.end()) {
    throw TaskException("UNKNOWN_STRATEGY");
  }

  StrategyPtr strategy = strategies[strategyType];

  uint32_t completed = obj["completed"];

  uint32_t fails = 0;
  if (obj.contains("fails")) {
    fails = obj["fails"];
  }

  std::vector<Request> requests;
  for (auto req : obj["requests"]) {
    if (req["type"] == "CREATE_PROCESS") {
      requests.push_back(CreateProcessReq(req["pid"], req["bytes"]));
    } else if (req["type"] == "TERMINATE_PROCESS") {
      requests.push_back(TerminateProcessReq(req["pid"]));
    } else if (req["type"] == "ALLOCATE_MEMORY") {
      requests.push_back(AllocateMemory(req["pid"], req["bytes"]));
    } else if (req["type"] == "FREE_MEMORY") {
      requests.push_back(FreeMemory(req["pid"], req["address"]));
    } else {
      throw TaskException("UNKNOWN_REQUEST");
    }
  }

  std::vector<MemoryBlock> blocks, freeBlocks;
  for (auto block : obj["state"]["blocks"]) {
    blocks.emplace_back(block["pid"], block["address"], block["size"]);
```

```
  }
  for (auto blockObj : obj["state"]["free_blocks"]) {
    freeBlocks.emplace_back(
        blockObj["pid"], blockObj["address"], blockObj["size"]);
  }

  std::vector<std::string> actions(0);

  /*
   * Если размер массива actions не совпадает с количеством
   * выполненных заданий (completed) или ни одно задание не выполнено,
   * то информация из массива отбрасывается.
   */
  if (obj.contains("actions") && obj["actions"].is_array() &&
      !(completed == 0 || completed != obj["actions"].size())) {
    for (const std::string action : obj["actions"]) {
      actions.push_back(action);
    }
  }

  return MemoryTask::create(
      strategy, completed, fails, {blocks, freeBlocks}, requests, actions);
}

/**
 *  @brief Создает объект задания "Диспетчеризация процессов" из JSON-объекта.
 *
 *  @param obj JSON-объект.
 *
 *  @return Объект задания.
 *
 *  @throws Utils::TaskException Исключение возникает в следующих случаях:
 *
 *  "UNKNOWN_STRATEGY" - неизвестный тип планировщика;
 *  "UNKNOWN_REQUEST" - неизвестный тип заявки;
 *  "UNKNOWN_PROCSTATE" - неизвестное состояние процесса.
 */
inline ProcessesTask loadProcessesTask(const nlohmann::json &obj) {
  using namespace ProcessesManagement;

  auto toPair = [](auto strategy) -> std::pair<std::string, StrategyPtr> {
    return {strategy->toString(), strategy};
  };

  std::map<std::string, StrategyPtr> strategies = {
      toPair(RoundRobinStrategy::create()),
      toPair(FcfsStrategy::create()),
      toPair(SjnStrategy::create()),
      toPair(SrtStrategy::create()),
      toPair(WinNtStrategy::create()),
```

```
    toPair(UnixStrategy::create()),
    toPair(LinuxO1Strategy::create())};

auto strategyType = obj["strategy"];
if (strategies.find(strategyType) == strategies.end()) {
  throw TaskException("UNKNOWN_STRATEGY");
}

StrategyPtr strategy = strategies[strategyType];

uint32_t completed = obj["completed"];

uint32_t fails = 0;
if (obj.contains("fails")) {
  fails = obj["fails"];
}

std::vector<Request> requests;
for (auto req : obj["requests"]) {
  if (req["type"] == "CREATE_PROCESS") {
    requests.push_back(CreateProcessReq(req["pid"],
                                        req["ppid"],
                                        req["priority"],
                                        req["basePriority"],
                                        req["timer"],
                                        req["workTime"]));
  } else if (req["type"] == "TERMINATE_PROCESS") {
    requests.push_back(TerminateProcessReq(req["pid"]));
  } else if (req["type"] == "INIT_IO") {
    requests.push_back(InitIO(req["pid"]));
  } else if (req["type"] == "TERMINATE_IO") {
    requests.push_back(TerminateIO(req["pid"], req["augment"]));
  } else if (req["type"] == "TRANSFER_CONTROL") {
    requests.push_back(TransferControl(req["pid"]));
  } else if (req["type"] == "TIME_QUANTUM_EXPIRED") {
    requests.push_back(TimeQuantumExpired());
  } else {
    throw TaskException("UNKNOWN_REQUEST");
  }
}

std::map<std::string, ProcState> stateMap = {
    {"ACTIVE", ProcState::ACTIVE},
    {"EXECUTING", ProcState::EXECUTING},
    {"WAITING", ProcState::WAITING}};

std::vector<Process> processes;
for (auto process : obj["state"]["processes"]) {
  if (stateMap.find(process["state"]) == stateMap.end()) {
    throw TaskException("UNKNOWN_PROCSTATE");
```

```
    }
    processes.emplace_back(Process{}
                           .pid(process["pid"])
                           .ppid(process["ppid"])
                           .priority(process["priority"])
                           .basePriority(process["basePriority"])
                           .timer(process["timer"])
                           .workTime(process["workTime"])
                           .state(stateMap[process["state"]]));
  }
  std::array<std::deque<int32_t>, 16> queues;
  for (size_t i = 0; i < queues.size(); ++i) {
    for (int32_t pid : obj["state"]["queues"][i]) {
      queues[i].push_back(pid);
    }
  }

  std::vector<std::string> actions(0);

  /*
   * Если размер массива actions не совпадает с количеством
   * выполненных заданий (completed) или ни одно задание не выполнено,
   * то информация из массива отбрасывается.
   */
  if (obj.contains("actions") && obj["actions"].is_array() &&
      !(completed == 0 || completed != obj["actions"].size())) {
    for (const std::string action : obj["actions"]) {
      actions.push_back(action);
    }
  }

  return ProcessesTask::create(
      strategy, completed, fails, {processes, queues}, requests, actions);
}
} // namespace Utils::details

namespace Utils {
/**
 *  @brief Загружает задания из файла.
 *
 *  @param is Дескриптор файла.
 *
 *  @return Массив из объектов заданий.
 *
 *  @throws Utils::TaskException Исключение возникает в
 *  следующих случаях:
 *
 *  "UNKNOWN_TASK" - неизвестный тип задания.
 */
inline std::vector<Task> loadTasks(std::istream &is) {
```

```
  nlohmann::json obj;
  is >> obj;
  std::vector<Task> tasks;

  for (auto task : obj) {
    if (task["type"] == "MEMORY_TASK") {
      tasks.emplace_back(details::loadMemoryTask(task));
    } else if (task["type"] == "PROCESSES_TASK") {
      tasks.emplace_back(details::loadProcessesTask(task));
    } else {
      throw TaskException("UNKNOWN_TASK");
    }
  }

  return tasks;
}

/**
 * @brief Сохраняет задания в файл.
 *
 * @param tasks Массив из объектов заданий.
 *
 * @param os Дескриптор файла.
 */
inline void saveTasks(const std::vector<Task> &tasks, std::ostream &os) {
  auto obj = nlohmann::json::array();

  for (Task task : tasks) {
    auto task_json = task.match([](const auto &task) { return task.dump(); });
    obj.push_back(task_json);
  }

  os
#ifdef DISPATCHER_DEBUG
      << std::setw(2)
#endif
      << obj;
}
} // namespace Utils
#pragma once

#include <cstdint>
#include <string>
#include <utility>
#include <vector>

#include <mapbox/variant.hpp>
#include <nlohmann/json.hpp>
#include <tl/optional.hpp>
```

```
#include "../algo/memory/exceptions.h"
#include "../algo/memory/requests.h"
#include "../algo/memory/strategies.h"
#include "../algo/processes/exceptions.h"
#include "../algo/processes/requests.h"
#include "../algo/processes/strategies.h"
#include "exceptions.h"

namespace Utils {
namespace Memory = MemoryManagement;
namespace Processes = ProcessesManagement;

/**
 *  @brief Задание "Диспетчеризация памяти".
 */
class MemoryTask {
private:
  Memory::StrategyPtr _strategy;

  uint32_t _completed;

  Memory::MemoryState _state;

  std::vector<Memory::Request> _requests;

  uint32_t _fails;

  std::vector<std::string> _actions;

  /**
   *  @brief Создает объект задания "Диспетчеризация памяти".
   *
   *  @param strategy Стратегия выбора блока памяти.
   *  @param completed Количество обработанных заявок.
   *  @param fails Количество допущенных пользователем ошибок.
   *  @param state Дескриптор состояния памяти.
   *  @param requests Список заявок для обработки.
   *  @param actions Массив строк с информацией о действиях пользователя для
   *  каждой заявки.
   */
  MemoryTask(Memory::StrategyPtr strategy,
             uint32_t completed,
             uint32_t fails,
             const Memory::MemoryState &state,
             const std::vector<Memory::Request> requests,
             const std::vector<std::string> actions)
      : _strategy(strategy), _completed(completed), _state(state),
        _requests(requests), _fails(fails), _actions(actions) {}

public:
```

```
/**
 *  @brief Создает объект задания "Диспетчеризация памяти".
 *
 *  @see Utils::MemoryTask::MemoryTask().
 */
static MemoryTask create(Memory::StrategyPtr strategy,
                         uint32_t completed,
                         uint32_t fails,
                         const Memory::MemoryState &state,
                         const std::vector<Memory::Request> requests,
                         const std::vector<std::string> actions) {
  validate(strategy, completed, state, requests);
  return {strategy, completed, fails, state, requests, actions};
}

/**
 *  @brief Создает объект задания "Диспетчеризация памяти".
 *
 *  Количество допущенных пользователем ошибок по умолчанию - 0.
 *  Массив действий actions пользователя по умолчанию пуст.
 *
 *  @see Utils::MemoryTask::MemoryTask().
 */
static MemoryTask create(Memory::StrategyPtr strategy,
                         uint32_t completed,
                         const Memory::MemoryState &state,
                         const std::vector<Memory::Request> requests) {
  validate(strategy, completed, state, requests);
  return {strategy, completed, 0, state, requests, {}};
}

/**
 *  @brief Проверяет параметры конструктора.
 *
 *  @param strategy Стратегия выбора блока памяти.
 *  @param completed Количество обработанных заявок.
 *  @param state Дескриптор состояния памяти.
 *  @param requests Список заявок для обработки.
 *
 *  @throws Utils::TaskException Исключение возникает, если
 *  переданные параметры не соответствуют заданным ограничениям.
 */
static void validate(Memory::StrategyPtr strategy,
                     uint32_t completed,
                     const Memory::MemoryState &state,
                     const std::vector<Memory::Request> requests) {
  try {
    Memory::MemoryState::validate(state.blocks, state.freeBlocks);
  } catch (Memory::BaseException &ex) {
    throw TaskException(ex.what());
```

```
  }

  if (requests.size() < completed) {
    throw TaskException("INVALID_TASK");
  }
  auto currentState = Memory::MemoryState::initial();
  try {
    for (auto req = requests.begin(); req != requests.begin() + completed;
         ++req) {
      currentState = strategy->processRequest(*req, currentState);
    }
    if (currentState != state) {
      throw TaskException("STATE_MISMATCH");
    }
  } catch (Memory::BaseException &ex) {
    throw TaskException(ex.what());
  }
}

Memory::StrategyPtr strategy() const { return _strategy; }

uint32_t completed() const { return _completed; }

const Memory::MemoryState &state() const { return _state; }

const std::vector<Memory::Request> &requests() const { return _requests; }

uint32_t fails() const { return _fails; }

const std::vector<std::string> actions() const { return _actions; }

/**
 *  Возвращает задание в виде JSON-объекта.
 */
nlohmann::json dump() const {
  nlohmann::json obj;

  obj["type"] = "MEMORY_TASK";

  obj["strategy"] = strategy()->toString();

  obj["completed"] = completed();

  obj["state"] = state().dump();

  obj["fails"] = fails();

  obj["requests"] = nlohmann::json::array();

  for (auto request : requests()) {
```

```
    auto req_json = request.match([](const auto &req) { return req.dump(); });
    obj["requests"].push_back(req_json);
  }

  obj["actions"] = nlohmann::json(_actions);

  return obj;
}

/**
 *  Проверяет, выполнено ли задание полностью.
 */
bool done() const { return _completed == _requests.size(); }

/**
 *  @brief Проверяет, правильно ли обработана текущая заявка.
 *
 *  @param state Дескриптор состояния памяти после обработки заявки.
 *
 *  @return Пара <bool ok, MemoryTask task>.
 *
 *  Если заявка была обработана правильно, то @a ok == true, а в @a task
 *  хранится новый объект задания с обновленными состоянием памяти и
 *  счетчиком завершенных заданий (увеличивается на 1). Если задание выполнено
 *  полностью, то в @a task будет хранится текущий объект задания.
 *
 *  Если заявка была обработана неправильно, то @a ok == false, а в @a task
 *  хранится текущий объект задания с увеличенным на единицу счетчиком ошибок.
 */
std::pair<bool, MemoryTask> next(const Memory::MemoryState &state) const {
  if (done()) {
    return {true, *this};
  }
  try {
    auto request = _requests[_completed];
    auto expected = _strategy->processRequest(request, _state);
    if (expected == state) {
      return {true,
              MemoryTask{_strategy,
                         _completed + 1,
                         _fails,
                         expected,
                         _requests,
                         _actions}};
    } else {
      return {false,
              MemoryTask{_strategy,
                         _completed,
                         _fails + 1,
                         _state,
```

```
                    _requests,
                    _actions}};
      }
     } catch (...) {
      return {
         false,
         MemoryTask{
            _strategy, _completed, _fails + 1, _state, _requests, _actions}};
    }
   }
};

/**
 *  @brief Задание "Диспетчеризация процессов".
 */
class ProcessesTask {
private:
  Processes::StrategyPtr _strategy;

  uint32_t _completed;

  Processes::ProcessesState _state;

  std::vector<Processes::Request> _requests;

  uint32_t _fails;

  std::vector<std::string> _actions;

  /**
   *  @brief Создает объект задания "Диспетчеризация процессов".
   *
   *  @param strategy Планировщик.
   *  @param completed Количество обработанных заявок.
   *  @param fails Количество допущенных пользователем ошибок.
   *  @param state Дескриптор состояния процессов.
   *  @param requests Список заявок для обработки.
   *  @param actions Массив строк с информацией о действиях пользователя для
   *  каждой заявки.
   */
  ProcessesTask(Processes::StrategyPtr strategy,
                uint32_t completed,
                uint32_t fails,
                const Processes::ProcessesState &state,
                const std::vector<Processes::Request> requests,
                const std::vector<std::string> actions)
      : _strategy(strategy), _completed(completed), _state(state),
        _requests(requests), _fails(fails), _actions(actions) {}

public:
```

```
/**
 *  @brief Создает объект задания "Диспетчеризация процессов".
 *
 *  @see Utils::ProcessesTask::ProcessesTask().
 */
static ProcessesTask create(Processes::StrategyPtr strategy,
                            uint32_t completed,
                            uint32_t fails,
                            const Processes::ProcessesState &state,
                            const std::vector<Processes::Request> requests,
                            const std::vector<std::string> actions) {
  validate(strategy, completed, state, requests);
  return {strategy, completed, fails, state, requests, actions};
}

/**
 *  @brief Создает объект задания "Диспетчеризация процессов".
 *
 *  Количество допущенных пользователем ошибок по умолчанию - 0.
 *  Массив действий actions пользователя по умолчанию пуст.
 *
 *  @see Utils::ProcessesTask::ProcessesTask().
 */
static ProcessesTask create(Processes::StrategyPtr strategy,
                            uint32_t completed,
                            const Processes::ProcessesState &state,
                            const std::vector<Processes::Request> requests) {
  validate(strategy, completed, state, requests);
  return {strategy, completed, 0, state, requests, {}};
}

/**
 *  @brief Проверяет параметры конструктора.
 *
 *  @param strategy Стратегия выбора блока памяти.
 *  @param completed Количество обработанных заявок.
 *  @param state Дескриптор состояния памяти.
 *  @param requests Список заявок для обработки.
 *
 *  @throws Utils::TaskException Исключение возникает, если
 *  переданные параметры не соответствуют заданным ограничениям.
 */
static void validate(Processes::StrategyPtr strategy,
                     uint32_t completed,
                     const Processes::ProcessesState &state,
                     const std::vector<Processes::Request> requests) {
  try {
    Processes::ProcessesState::validate(state.processes, state.queues);
  } catch (Processes::BaseException &ex) {
    throw TaskException(ex.what());
```

```
  }

  if (requests.size() < completed) {
    throw TaskException("INVALID_TASK");
  }
  auto currentState = Processes::ProcessesState::initial();
  try {
    for (auto req = requests.begin(); req != requests.begin() + completed;
         ++req) {
      currentState = strategy->processRequest(*req, currentState);
    }
    if (currentState != state) {
      throw TaskException("STATE_MISMATCH");
    }
  } catch (Memory::BaseException &ex) {
    throw TaskException(ex.what());
  }
}

/**
 *  Возвращает задание в виде JSON-объекта.
 */
nlohmann::json dump() const {
  nlohmann::json obj;

  obj["type"] = "PROCESSES_TASK";

  obj["strategy"] = strategy()->toString();

  obj["completed"] = completed();

  obj["state"] = state().dump();

  obj["fails"] = fails();

  obj["requests"] = nlohmann::json::array();

  for (auto request : requests()) {
    auto req_json = request.match([](const auto &req) { return req.dump(); });
    obj["requests"].push_back(req_json);
  }

  obj["actions"] = nlohmann::json(_actions);

  return obj;
}

Processes::StrategyPtr strategy() const { return _strategy; }

uint32_t completed() const { return _completed; }
```

```
const Processes::ProcessesState &state() const { return _state; }

const std::vector<Processes::Request> &requests() const { return _requests; }

uint32_t fails() const { return _fails; }

const std::vector<std::string> actions() const { return _actions; }

/**
 *  Проверяет, выполнено ли задание полностью.
 */
bool done() const { return _completed == _requests.size(); }

/**
 *  @brief Проверяет, правильно ли обработана текущая заявка.
 *
 *  @param state Дескриптор состояния процессов после обработки заявки.
 *
 *  @return Пара <bool ok, ProcessesTask task>.
 *
 *  Если заявка была обработана правильно, то @a ok == true, а в @a task
 *  хранится новый объект задания с обновленными состоянием процессов и
 *  счетчиком завершенных заданий (увеличивается на 1). Если задание выполнено
 *  полностью, то в @a task будет хранится текущий объект задания.
 *
 *  Если заявка была обработана неправильно, то @a ok == false, а в @a task
 *  хранится текущий объект задания с увеличенным на единицу счетчиком ошибок.
 */
std::pair<bool, ProcessesTask>
next(const Processes::ProcessesState &state) const {
 if (done()) {
  return {true, *this};
 }
 try {
  auto request = _requests[_completed];
  auto expected = _strategy->processRequest(request, _state);
  if (expected == state) {
   return {true,
        ProcessesTask{_strategy,
                _completed + 1,
                _fails,
                expected,
                _requests,
                _actions}};
  } else {
   return {false,
        ProcessesTask{_strategy,
                _completed,
                _fails + 1,
```

```
                    _state,
                    _requests,
                    _actions}};
    }
   } catch (...) {
    return {
       false,
       ProcessesTask{
          _strategy, _completed, _fails + 1, _state, _requests, _actions}};
   }
  }
};

using Task = mapbox::util::variant<MemoryTask, ProcessesTask>;
} // namespace Utils

#pragma once

#include <QDialog>
#include <QWidget>

namespace Ui {
class AboutWindow;
}

class AboutWindow : public QDialog {
  Q_OBJECT

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

private:
  Ui::AboutWindow *ui;
};
#pragma once

#include <vector>

#include <QCloseEvent>
#include <QMainWindow>
#include <QString>
#include <QWidget>

#include <utils/tasks.h>

#include "models.h"

namespace Ui {
class MainWindow;
```

```
}

class MainWindow : public QMainWindow {
  Q_OBJECT

public:
  explicit MainWindow(const QString &student, QWidget *parent = nullptr);

  ~MainWindow() override;

private:
  Ui::MainWindow *ui;

  QString student;

  void openTasks();

  void saveTasks();

  void loadTasks(const std::vector<Utils::Task> &tasks);

  void createTasks();

  void dumpTasks(const std::vector<Utils::Task> &tasks);

  void showHelp();

  void openTmpDir();

  void closeEvent(QCloseEvent *event) override;
};
#pragma once

#include <cstddef>
#include <cstdint>
#include <string>
#include <vector>

#include <QPoint>
#include <QString>
#include <QWidget>

#include <algo/memory/requests.h>
#include <algo/memory/strategies.h>
#include <algo/memory/types.h>
#include <utils/tasks.h>

#include "models.h"
#include "taskgetter.h"
```

Приложение Г

(справочное)

Список сокращений и обозначений

ГПИ – графический пользовательский интерфейс

ОС – операционная система

ПК – персональный компьютер

CBC – cipher block chaining, режим сцепления блоков шифротекста

HiDPI – high dots per inch, высокая плотность пикселей

PID – process identifier, идентификатор процесса

Приложение Д

(справочное)

Библиографический список

1. Караваева О. В. Операционные системы. Управление памятью и процессами [Текст]: Учебно-методическое пособие / О. В. Караваева. – Киров: ФГБОУ ВО «ВятГУ», 2016. – 39 с.

2. Block Ciphers – Botan [Электронный ресурс]. – Режим доступа: https://botan.randombit.net/handbook/api_ref/block_cipher.html.

3. AES instruction set – Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/AES_instruction_set.

4. GTK – Wikipedia [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/GTK.

5. Qt – Wikipedia [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Qt.

6. Qt Reference Pages | Qt 5.14 [Электронный ресурс]. – Режим доступа: https://doc.qt.io/qt-5/reference-overview.html.

7. wxWidgets – Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/WxWidgets.

8. AvaloniaUI/Avalonia: A multi-platform .NET UI framework [Электронный ресурс]. – Режим доступа: https://github.com/AvaloniaUI/Avalonia.

9. Avalonia мои за и против / Хабр [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/471728/.

10. Botan: Crypto and TLS for Modern C++ – Botan [Электронный ресурс]. – Режим доступа: https://botan.randombit.net/.

11. AES (Advanced Encryption Standard) – Национальная библиотека им. Н. Э. Баумана [Электронный ресурс]. – Режим доступа: https://ru.bmstu.wiki/AES_(Advanced_Encryption_Standard).