

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к практическим занятиям
по дисциплине «Алгоритмы и структуры данных»
для студентов направления подготовки
09.03.01 Информатика и вычислительная техника,
Направленность (профиль) «Программное обеспечение средств
вычислительной техники и автоматизированных систем»

Ставрополь

2026

СОДЕРЖАНИЕ

ПРАКТИЧЕСКАЯ РАБОТА № 1.	6
Тема: Определение функции сложности алгоритмов	6
ПРАКТИЧЕСКАЯ РАБОТА № 2.	12
Тема: Имитационное моделирование системы обслуживания потока заданий	12
ПРАКТИЧЕСКАЯ РАБОТА № 3.	24
Тема: Построение графа среды для алгоритма поиска пути.	24
ПРАКТИЧЕСКАЯ РАБОТА № 4.	32
Тема: Симуляция работы мобильного робота на складе (используя Gazebo) ..	32
ПРАКТИЧЕСКАЯ РАБОТА № 5.	40
Тема: Оптимизация алгоритмов с помощью хеш-таблиц.	40
ПРАКТИЧЕСКАЯ РАБОТА № 6.	47
Реализация многопоточного алгоритма.	47
ПРАКТИЧЕСКАЯ РАБОТА № 7.	92
Тема: Реализация SLAM-алгоритма.	92
ПРАКТИЧЕСКАЯ РАБОТА № 8.	105
Тема: Знакомство с архитектурой YOLO для детекции объектов.	105
ПРАКТИЧЕСКАЯ РАБОТА № 9.	106
Тема: Оптимизация алгоритма с помощью профайлера.	106

ВВЕДЕНИЕ

Методические указания к выполнению лабораторных работ составлены в соответствии с рабочей программой дисциплины «Алгоритмы и структуры данных» и предназначены для студентов направления подготовки 09.03.01 Информатика и вычислительная техника.

Целью освоения дисциплины «Алгоритмы и структуры данных» является формирование у студентов теоретических знаний и практических навыков в области алгоритмов и структур данных, необходимых для эффективного решения задач робототехники, включая планирование движений, навигацию, обработку сенсорных данных и управление роботами в реальном времени.

1. Перечень планируемых результатов обучения по дисциплине (модулю), соотнесенных с планируемыми результатами освоения образовательной программы

Код, формулировка компетенции	Код, формулировка индикатора	Планируемые результаты обучения по дисциплине (модулю), характеризующие этапы формирования компетенций, индикаторов
ПК-1. Способен разрабатывать требования и проектировать программное обеспечение	ИД-1 _{ПК-1} понимает принципы построения архитектуры программного обеспечения и виды архитектуры программного обеспечения; понимает методы и средства проектирования программного обеспечения, методы и средства проектирования баз данных; понимает методы и средства проектирования	Способен самостоятельно проектировать и разрабатывать алгоритмическое обеспечение для робототехнических систем, учитывая их специфику и требования к производительности.

	программных интерфейсов; понимает методы и приемы формализации задач; понимает возможности современных и перспективных средств разработки программных продуктов, технических средств.	
	ИД-2_{ПК-1} проводит анализ исполнения требований; вырабатывает варианты реализации требований; проводит оценку и обоснование рекомендуемых решений; выбирает средства реализации требований к программному обеспечению; использует существующие типовые решения и шаблоны проектирования программного обеспечения; разрабатывает технические спецификации на программные компоненты и их взаимодействие.	
ПК-2. Способен осуществлять концептуальное, функциональное и логическое проектирование систем среднего и крупного масштаба и сложности	ИД-6_{ПК-2} Применяет методы и алгоритмы оптимизации исполняемого кода	Способен проектировать сложные алгоритмические системы в робототехнике, от концепции до логической реализации, с учётом масштабируемости, производительности и
	ИД-7_{ПК-2} Использует современные методологии разработки программного обеспечения и технологии программирования	

	ИД-14 _{ПК-2} Определяет и применяет необходимые алгоритмы и структуры данных для решения задач навигации, планирования, обработки сенсорных данных, управления и принятия решений в робототехнических системах	модульности, включая работу с ROS, распределёнными вычислениями и робототехническими платформами.
	ИД-9 _{ПК-2} Проектирует структуры данных	

ПРАКТИЧЕСКАЯ РАБОТА № 1.

Тема: «Определение функции сложности алгоритмов»

Цель работы – изучить принцип определения функции сложности алгоритма.

Теоретические сведения

1. Функция сложности алгоритма

Эффективность программы является очень важной ее характеристикой. Эффективность программы имеет две составляющие: память (или пространство) и время.

Пространственная эффективность измеряется количеством памяти, требуемой для выполнения программы. Компьютеры обладают ограниченным объемом памяти. Если две программы реализуют идентичные функции, то та, которая использует меньший объем памяти, характеризуется большей пространственной эффективностью.

Временная эффективность программы определяется временем, необходимым для ее выполнения. Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временной, так и пространственной сложности. Порядок сложности алгоритма выражает его эффективность обычно через количество обрабатываемых данных. Например, некоторый алгоритм может существенно зависеть от размера обрабатываемого массива. Если, скажем, время обработки удваивается с удвоением размера массива, то порядок временной сложности алгоритма определяется как размер массива. Порядок алгоритма — это функция, доминирующая над точным выражением временной сложности. Функция $f(n)$ имеет порядок $O(g(n))$, если имеется константа K и счетчик n_0 такие, что $f(n) \leq K \cdot g(n)$ для $n > n_0$.

Алгоритм имеет сложность $O(f(N))$, если с увеличением размерности исходных данных N время выполнения алгоритма возрастает с той же скоростью, что и функция $f(N)$.

Функция сложности O выражает относительную скорость алгоритма в зависимости от некоторой переменной (или переменных). Существуют три важных правила для определения функции сложности:

$$1) O(k \cdot f) = O(f)$$

$$2) O(f \cdot g) = O(f) \cdot O(g) \text{ или } O(f/g) = O(f)/O(g).$$

$$3) O(f+g) \text{ равна доминанте } O(f) \text{ и } O(g).$$

Здесь « k » обозначает константу, а f и g — функции.

Первое правило декларирует, что постоянные множители не имеют значения для определения порядка сложности

$$O(1,5 \cdot N) = O(N).$$

Из второго правила следует, что порядок сложности произведения двух функций равен произведению их сложностей

$$O((17 \cdot N) \cdot N) = O(17 \cdot N) \cdot O(N) = O(N) \cdot O(N) = O(N \cdot N) = O(N^2).$$

Из третьего правила следует, что порядок сложности суммы функций определяется как порядок доминанты первого и второго слагаемых, т. е. выбирается наибольший порядок

$$O(N^5 + N^2) = O(N^5).$$

2. Виды функции сложности алгоритмов $O(I)$

В алгоритмах константной сложности большинство операций в программе выполняются один или несколько раз. Любой алгоритм, всегда требующий независимо от размера данных одного и того же времени, имеет константную сложность.

1) $O(N)$. Время работы программы линейно, обычно, когда каждый элемент входных данных требуется обработать лишь линейное число раз.

2) $O(N^2)$, $O(N^3)$, $O(N^a)$. Полиномиальная сложность. $O(N^2)$ — квадратичная сложность, $O(N^3)$ — кубическая сложность

3) $O(\log(N))$. Когда время работы программы логарифмическое, программа начинает работать намного медленнее с увеличением N . Такое время работы встречается обычно в программах, которые делят большую проблему на маленькие и решают их по отдельности.

4) $O(N \cdot \log(N))$. Такое время работы имеют те алгоритмы, которые делят большую проблему на маленькие, а затем, решив их, соединяют их решения.

5) $O(2^n)$. Экспоненциальная сложность. Такие алгоритмы чаще всего возникают в результате подхода, именуемого методом грубой силы.

3. Анализ функции сложности по программе

Программист должен уметь проводить анализ алгоритмов и определять их сложность. Временная сложность алгоритма может быть подсчитана исходя из анализа его управляющих структур. Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность. Если нет рекурсии и циклов, все управляющие структуры могут быть сведены к структурам константной сложности. Следовательно, и весь алгоритм также характеризуется константной сложностью. Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов. Например, рассмотрим алгоритм обработки элементов массива (рис.11.1а).

Сложность этого алгоритма $O(N)$, так как тело цикла выполняется N раз, и сложность тела цикла равна $O(1)$. Если один цикл вложен в другой и оба цикла зависят от размера одной и той же переменной (рис.11.1б), то вся конструкция характеризуется квадратичной сложностью. Сложность такой программы — $O(N^2)$.

Наиболее сложными частями программы обычно является выполнение циклов и вызовов процедур. Если одна процедура вызывает другую, то необходимо более тщательно оценить сложность последней. Если в ней выполняется определенное число инструкций, например, вывод на печать, то на оценку порядка сложности она практически не влияет. С другой стороны, если в вызываемой процедуре выполняется $O(N)$ шагов, то функция может

значительно усложнять алгоритм. Если процедура вызывается внутри цикла, то влияние может быть намного больше.

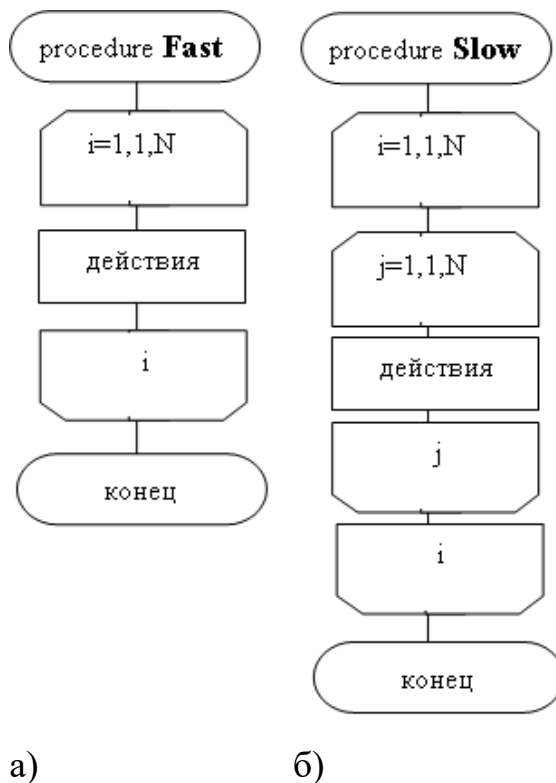


Рисунок 11.1 – Блок-схема алгоритма подпрограммы обработки одномерного (а) и двумерного массива (б)

В качестве примера возьмем программу, содержащую медленную процедуру Slow со сложностью порядка $O(N^2)$ и быструю процедуру Fast со сложностью порядка $O(N)$ – рис.11.1. Сложность всей программы зависит от соотношения между этими двумя процедурами.

Если при выполнении циклов процедуры Fast всякий раз вызывается процедура Slow, то сложности процедур перемножаются. Общая сложность равна произведению обеих сложностей. В данном случае сложность алгоритма составляет $O(N) * O(N^2)$ или $O(N * N^2) = O(N^3)$.

С другой стороны, если основная программа вызывает процедуры отдельно, их вычислительная сложность складывается. В этом случае итоговая сложность по порядку величины равна $O(N) + O(N^2) = O(N^2)$.

Рассмотрим алгоритм рис.11.2, который для матрицы $A[N \times N]$ находит максимальный элемент в каждой строке.

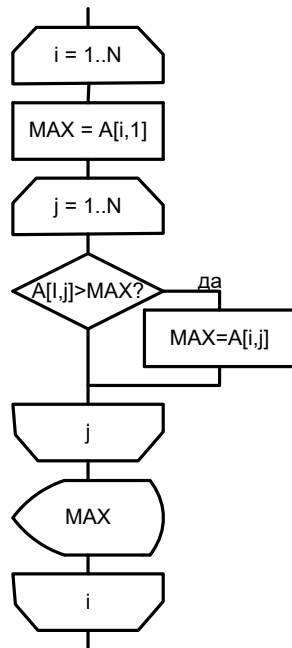


Рисунок 11.2 – Фрагмент алгоритма поиска максимального элемента в каждой строке массива

В этом алгоритме переменная i меняется от 1 до N . При каждом изменении i , переменная j тоже меняется от 1 до N . Во время каждой из N итераций внешнего цикла, внутренний цикл тоже выполняется N раз. Общее количество итераций внутреннего цикла равно $N \cdot N$. Это определяет сложность алгоритма $O(N^2)$.

$$O(f) = O(N) * (O(1) + (O(N) * (O(1) + O(1)))) + O(1) = O(N) * O(N) = O(N^2)$$

Как правило, около 90% времени работы программы требует выполнение повторений и только 10% составляют непосредственно вычисления. Анализ сложности программ показывает, на какие фрагменты выпадают эти 90% — это циклы наибольшей глубины вложенности. Повторения могут быть организованы в виде вложенных циклов или вложенной рекурсии.

Эта информация может использоваться программистом для построения более эффективной программы следующим образом. Прежде всего можно попытаться сократить глубину вложенности повторений. Затем следует рассмотреть возможность сокращения количества операторов в циклах с

наибольшей глубиной вложенности. Если 90% времени выполнения составляет выполнение внутренних циклов, то 30%-ное сокращение этих небольших секций приводит к 27%-ному снижению времени выполнения всей программы.

Порядок выполнения лабораторной работы

Необходимо реализовать три алгоритма поиска пути в графе. Сравнить и обосновать выбор наиболее оптимального из них для робототехнических систем.

Контрольные вопросы

- 1) Что определяет функция сложности алгоритма?
- 2) Укажите виды функции сложности алгоритмов.
- 3) Каким образом определяется временная функция сложности алгоритма?
- 4) Укажите способы анализа функции сложности по алгоритму.
- 5) Укажите уравнение, по которому определяется функция сложности.

ПРАКТИЧЕСКАЯ РАБОТА № 2.

Тема: Имитационное моделирование системы обслуживания потока заданий

Цель: практическое освоение динамической структуры данных Очередь.

Очередь характеризуется таким порядком обработки значений, при котором вставка новых элементов производится в конец очереди, а извлечение – из начала. Подобная организация данных широко встречается в различных приложениях. В качестве примера использования очереди предлагается задача разработки системы имитации однопроцессорной ЭВМ. Рассматриваемая в рамках лабораторной работы схема имитации является одной из наиболее простых моделей обслуживания заданий в вычислительной системе и обеспечивает тем самым лишь начальное ознакомление с проблемами моделирования и анализа эффективности функционирования реальных вычислительных систем.

Постановка учебной задачи

Основные понятия и определения

Имитационное моделирование реально существующих объектов и явлений – физических, химических, биологических, социальных процессов, живых и неживых систем, инженерных конструкций, конструируемых объектов – представляет собой построение математической модели, которая описывает изучаемое явление с достаточной точностью, и последующую реализацию разработанной модели на ЭВМ для проведения вычислительных экспериментов с целью изучения свойств моделируемых явлений. Использование имитационного моделирования позволяет проводить изучение исследуемых объектов и явлений без проведения реальных (натурных) экспериментов.

Очередь (англ. queue), – схема запоминания информации, при которой каждый вновь поступающий ее элемент занимает крайнее положение (*конец очереди*). При выдаче информации из очереди выдается элемент, расположенный в очереди первым (*начало очереди*), а оставшиеся элементы продвигаются к началу; следовательно, элемент, поступивший первым, выдается первым. Более строгое определение структуры дано в разделе 2 описания данной лабораторной работы.

Требования к лабораторной работе

Для вычислительной системы (ВС) с одним процессором и однопрограммным последовательным режимом выполнения поступающих заданий требуется разработать программную систему для имитации процесса обслуживания заданий в ВС. При построении модели функционирования ВС должны учитываться следующие основные моменты обслуживания заданий:

- генерация нового задания;
- постановка задания в очередь для ожидания момента освобождения процессора;
- выборка задания из очереди при освобождении процессора после обслуживания очередного задания.

По результатам проводимых вычислительных экспериментов система имитации должна выводить информацию об условиях проведения эксперимента (интенсивность потока заданий, размер очереди заданий, производительность процессора, число тактов имитации) и полученные в результате имитации показатели функционирования вычислительной системы, в т.ч.

- количество поступивших в ВС заданий;
- количество отказов в обслуживании заданий из-за переполнения очереди;
- среднее количество тактов выполнения заданий;
- количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания.

Показатели функционирования вычислительной системы, получаемые при помощи систем имитации, могут использоваться для оценки эффективности применения ВС; по результатам анализа показателей могут быть приняты рекомендации о целесообразной модернизации характеристик ВС (например, при длительных простоях процессора и при отсутствии отказов от обслуживания заданий желательно повышение интенсивности потока

обслуживаемых заданий и т.д.).

Условия и ограничения

Сделаем следующие основные допущения:

- При планировании очередности обслуживания заданий возможность задания приоритетов не учитывается.
- Моменты появления новых заданий и моменты освобождения процессора рассматриваются как случайные события.

Метод решения

Структуры данных

Напомним, что *динамическая структура* есть математическая структура, которой соответствует частично-упорядоченное (по включению) базовое множество M , операции вставки и удаления элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

Таким образом, *очередь* есть динамическая структура, операции вставки и удаления переводят очередь из одного состояния в другое, при этом добавление новых элементов осуществляется в конец очереди, а извлечение – из начала очереди (дисциплина обслуживания

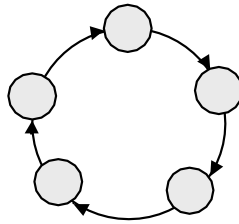
«первым пришел – первым обслужен.

Важной задачей при реализации системы обслуживания очереди является выбор структуры хранения, обеспечивающей решение проблемы эффективного использования памяти без перепакетов и без использования связанных списков (требующих дополнительных затрат памяти на указатели).

Как и в случае со стеком, в качестве *структуры хранения* очереди предлагается использовать одномерный (одноиндексный) массив, размещаемый в динамической области памяти. В связи с характером обработки значений, располагаемых в очереди, для указания хранимых в очереди данных необходимо иметь два указателя – на начало и конец очереди. Эти указатели увеличивают свое значение: один при вставке, другой при извлечении элемента. Таким образом, в ходе функционирования очереди может возникнуть ситуация, когда оба указателя достигнут своего наибольшего значения и дальнейшее

пополнение очереди станет невозможным, несмотря на наличие свободного пространства в очереди. Одним из решений проблемы «движения» очереди является организация на одномерном массиве кольцевого буфера. *Кольцевым буфером* называется структура хранения, получаемая из вектора расширением отношения следования парой $p(a_n, a_1)$.

Рис. 2. Общая схема структуры хранения вида кольцевой буфер



Структура хранения очереди в виде кольцевого буфера может быть определена как одномерный (одноиндексный) массив, размещаемый в динамической области памяти и расположение данных в котором определяется при помощи следующего набора параметров:

pMem – указатель на память, выделенную для кольцевого буфера,

MemSize – размер выделенной памяти,

MaxMemSize – размер памяти, выделяемый по умолчанию, если при создании кольцевого буфера явно не указано требуемое количество элементов памяти,

DataCount – количество запомненных в очереди значений,

Hi – индекс элемента массива, в котором хранится последний элемент очереди,

Li – индекс элемента массива, в котором хранится первый элемент очереди.

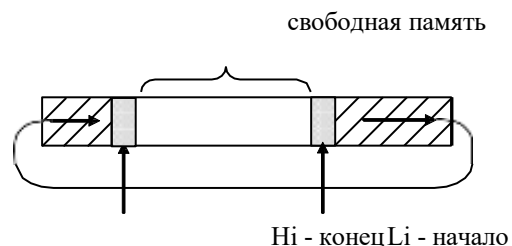


Рис. 3. Общая схема реализации кольцевого буфера на одноиндексном массиве

В связи с тем, что структура хранения очереди во многом аналогична структуре хранения стек, предлагается класс для реализации очереди построить наследованием от класса стек, описанного в лабораторной работе №3. При наследовании достаточно переопределить методы Get и GetNextIndex. В методе Get изменяется индекс для получения элемента (извлечение значений происходит из начала очереди), метод GetNextIndex реализует отношение следования на кольцевом буфере.

Алгоритмы

Очередь

Для работы с очередью предлагается реализовать следующие операции:

- Метод Put - добавить элемент;

Добавление элемента в очередь аналогично добавлению в стек, используется метод класса стек.

- Метод Get - удалить элемент;

При удалении элемента из очереди необходимо вернуть значение из динамического массива по индексу начала очереди, переместить указатель начала очереди и уменьшить количество элементов.

- Метод IsEmpty - проверить очередь на пустоту;
- Метод IsFull - проверить очередь на полноту. Данные методы наследуются из класса стек.

Моделирование потока задач

Для моделирования момента появления нового задания можно использовать значение датчика случайных чисел. Если значение датчика меньше некоторого порогового значения $q1$, $0 \leq q1 \leq 1$, то считается, что на данном такте имитации в вычислительную систему поступает новое задание (тем самым параметр $q1$ можно интерпретировать как величину, регулирующую интенсивность потока заданий – новое задание генерируется в среднем один раз за $(1/q1)$ тактов).

Моделирование работы процессора

Моделирование момента завершения обработки очередного задания также может быть выполнено по описанной выше схеме. При помощи датчика случайных чисел формируется еще одно случайное значение, и если это значение меньше порогового значения q_2 , $0 \leq q_2 \leq 1$, то принимается, что на данном такте имитации процессор завершил обслуживание очередного задания и готов приступить к обработке задания из очереди ожидания (тем самым параметр q_2 можно интерпретировать как величину, характеризующую производительность процессора вычислительной системы – каждое задание обслуживается в среднем за $(1/q_2)$ тактов).

Моделирование вычислительной системы

Возможная простая схема имитации процесса поступления и обслуживания заданий в вычислительной системе состоит в следующем.

1. Каждое задание в системе представляется некоторым однозначным идентификатором (например, порядковым номером задания).
2. Для проведения расчетов фиксируется (или указывается в диалоге) число тактов работы системы.
3. На каждом такте опрашивается состояние потока задач и процессор.
4. Регистрация нового задания в вычислительной системе может быть сведена к запоминанию идентификатора задания в очередь ожидания процессора. Ситуацию переполнения очереди заданий следует понимать как нехватку ресурсов вычислительной системы для ввода нового задания (отказ от обслуживания).
5. Для моделирования процесса обработки заданий следует учитывать, что процессор может быть занят обслуживанием очередного задания, либо же может находиться в состоянии ожидания (простоя).
6. В случае освобождения процессора предпринимается попытка его загрузки. Для этого извлекается задание из очереди ожидания.
7. Простой процессора возникает в случае, когда при завершении обработки очередного задания очередь ожидающих заданий оказывается пустой.
8. После проведения всех тактов имитации производится вывод характеристик вычислительной системы:
 - количество поступивших в вычислительную систему заданий в течение всего процесса имитации;
 - количество отказов в обслуживании заданий из-за переполнения очереди – целесообразно считать процент отказов как отношение количества не поставленных в очередь заданий к общему количеству сгенерированных заданий, умноженное на 100%;
 - среднее количество тактов выполнения задания;

- количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания – целесообразно считать процент простоя процессора как отношение количества тактов простоя процессора к общему количеству тактов имитации, умноженное на 100%.

Разработка программного комплекса

Структура

С учетом сформулированных выше предложений к реализации целесообразной представляется следующая модульная структура программы:

TStack.h, TStack.cpp – модуль с классом, реализующим операции над стеком;

TQueue.h, TQueue.cpp – модуль с классом, реализующим операции над очередью;

TJobStream.h, TJobStream.cpp – модуль с классом, реализующим поток задач;

TProc.h, TProc.cpp – модуль с классом, реализующим процессор;

QueueTestkit.cpp – модуль программы тестирования.

Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Класс *TStack* (файл TStack.h) Класс *TQueue* (файл TQueue.h):

```
class TQueue : public TStack {
protected:
    int Li; // индекс начала очереди
    int GetNextIndex (int index); // получить следующий индекс
public:
    TQueue (int Size = MaxMemSize):TStack(Size) {Li =
```

```

#include "Queue.h"

int Queue :: GetNextIndex(int index) { // Получение следующего индекса
    return ++ index % MemSize ;
}

TElem Queue :: Get ( void ) { // извлечь значение
    Head = GetNextIndex(
        Li ); DataCount--;
    return tmp;
}
}
if ( ~IsEmpty() ) { // очередь не пуста?
    TElem tmp = pMem[Li];

```

Тип `TElem` описывает тип значений элементов структуры данных (определяется в `TStack.h`). Реализация класса `TStack` подробно описана в лабораторной работе №3. Для определения методов класса `TQueue` ниже приведен фрагмент файла `TQueue.cpp` (модуль реализации).

В конструкторе вызывается конструктор базового класса, размер области памяти определяется как параметр конструктора (по умолчанию устанавливается `MaxMemSize`) и явно учитывает тип элемента очереди (`TElem`), кроме этого, устанавливается значение индекса начала очереди `Li = 0`.

Деструктор, методы `IsEmpty`, `IsFull`, `Put` наследуются из базового класса. Перед обращением к функции `Put` необходимо проверять очередь на полноту.

Метод `Get` переопределяется для очереди и обеспечивает получение значения из очереди по указателю начала очереди; выполнение операции выполняется с удалением возвращаемого значения из очереди. Перед обращением к этой функции необходимо проверять очередь на пустоту.

Метод `GetNextIndex` переопределяется в классе очереди и обеспечивает представление очереди в виде кольцевого буфера, позволяя устанавливать значения индексов начала и конца очереди при достижении ими граничного значения в 0.

Класс *`TJobStream`* (файл `TJobStream.h`)

```
class TJobStream {
private:
    // интенсивность потока
public:
    TJobStream (int Intens);
    int GetNewJob(void); // генерация нового задания
}
```

Класс *TProc* (файл TProc.h)

```
class TProc {
private:
    // производительность процессора
    double ProcRate;
    int
    int IsProcBusy(void) ; // процессор занят?
    int RunNewJob (int JobId); // приступить к выполнению нового
```

Этапы разработки

Как и ранее, важной частью лабораторной работы является организация поэтапной разработки программ. Предлагается следующая последовательность разработки (с учетом наличия работоспособного класса TStack):

Этап 1. Реализация программ поддержки очереди.

Этап 2. Реализация системы имитации обслуживания заданий.

Этап 3. Выполнение дополнительных заданий лабораторной работы.

При выполнении лабораторной работы следует продолжить практические освоение технологии итеративной разработки с самым тщательным тестированием правильности работы программ. Для проверки правильности должно разрабатываться достаточное количество тестов, которые гарантированно проверяют основные возможные ситуации при выполнении программ. Переход к следующему этапу разработки должен производиться только при успешном выполнении тестов; при проверке разработанных программ очередного этапа должны быть задействованы тесты всех предшествующих этапов.

Учитывая готовность класса TStack, при разработке программ очереди может быть предложена следующая последовательность реализации:

1. Реализация конструктора TQueue, перегрузка метода следования GetNextIndex. Тестирование разработанных программ.
2. Реализация метода извлечения из очереди. Тестирование.

Для этапа 2 может быть использована следующая последовательность разработки:

1. Разработка спецификации класса TJobStream, реализация конструктора и метода генерации нового задания.
2. Разработка спецификации класса TProc, реализация конструктора.
3. Реализация метода опроса состояния процессора и метода запуска нового задания на выполнение. Тестирование.
4. Разработка управляющей программы системы имитации, обеспечивающий циклическое выполнение тактов имитации: генерация нового задания, проверка состояния процессора, запуск нового задания. Тестирование.
5. Разработка программы вывода результатов имитации.
6. Выполнение вычислительных экспериментов. Подготовка отчета о выполнении лабораторной работы.

Входными параметрами вычислительного эксперимента являются: интенсивность потока задач, производительность процессора и количество тактов имитации. Максимальный размер очереди определен константным значением.

Рекомендации по разработке

Предусмотреть два режима работы программы – при небольшом числе тактов имитации следует выполнять потактовую распечатку состояния вычислительной системы и печать статистики, для длительных периодов имитации следует проводить только печать статистики.

Разработку программ лабораторной работы следует провести с обязательной проверкой возможных ошибочных ситуаций (например, выборка значения из пустой очереди). При обнаружении ошибочных ситуаций должно быть организовано завершение программы с выдачей аварийного кода завершения или, что более правильно, можно применить обработку исключений.

Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

1. Реализовать возможность задания приоритетов поступающих в систему задач – в этом случае задачи в очереди должны упорядочиваться в соответствии со своими приоритетами.
2. Дополнить постановку лабораторной работы возможностью

моделирования для поступающих заданий различной длительностью выполнения.

3. Расширить разработанную систему для возможности имитации многопроцессорных вычислительных систем, в которых может быть несколько процессоров. Возможное расширение постановки лабораторной работы может состоять в использовании нескольких процессоров для выполнения отдельных заданий. Важным моментом при анализе поведения таких систем является выбор стратегии выделения имеющихся процессоров для выполнения заданий (так, например, может возникать ситуации длительного откладывания выполнения заданий, для которых необходимо большое количество процессоров).

4. Расширить разработанную систему имитации возможностью использования нескольких очередей входных заданий. При расширении разработанных программ следует обратить внимание на трудоемкость необходимой модификации существующего программного кода: количество методов, в которые требуется внести изменения; количество новых внесенных ошибок разработки и т.п.

Критерии оценивания выполнения лабораторной работы

ПРАКТИЧЕСКАЯ работа считается сданной при выполнении минимального набора требований – реализованы все классы, при малых значениях количества тактов имитации итоговая статистика соответствует результатам потактового выполнения.

Набор тестов для проверки:

1. Количество тактов имитации 10

Размер очереди 3 Интенсивность потока задач 0.5

Производительность процессора 0.5

2. Количество тактов имитации 10

Размер очереди 3 Интенсивность потока задач 0.5

Производительность процессора 0.2

3. Количество тактов имитации 10

Размер очереди 3 Интенсивность потока задач 0.2

Производительность процессора 0.5

В первом тесте система является сбалансированной, во втором тесте малоинтенсивный поток задач при высокопроизводительном процессоре (в результате вычислительного эксперимента должен быть получен большой

процент проста процессора), в третьем тесте интенсивный поток задач при малопроизводительном процессоре (вычислительный эксперимент должен показать большое количество заданий, которым отказано в обслуживании из-за переполнения очереди заданий).

Построение подобных оценок результатов вычислительных экспериментов и рекомендации по изменению вычислительной системы могут быть включены в состав разрабатываемой системы имитации.

Вопросы и задания для самоконтроля

- Какие изменения могут потребоваться в структуре хранения при реализации очереди с приоритетами?
- Какие дополнительные понятия могут быть предложены при разработке системы имитации для более адекватного описания функционирования вычислительных систем?
- Насколько точно использованный способ генерации заданий описывает реально существующие потоки заданий в вычислительных системах? Какие дополнительные схемы генерации входного потока заданий могут быть предложены?
- Может ли разработанная система имитации быть использована в других областях приложений?
- Какие правила работы с очередью, реализация которой выполнена в библиотеке STL?
- Какие примеры задач можно привести, для решения которых может потребоваться использование очереди?

ПРАКТИЧЕСКАЯ РАБОТА № 3.

Тема: Построение графа среды для алгоритма поиска пути.

Цель работы – реализовать алгоритмы обработки графовых структур: поиск различных путей, проверку связности, построение остовых деревьев минимальной стоимости.

Краткие теоретические сведения

Граф – это конечное множество вершин и ребер, соединяющих их, т. е.:

$$G = \langle V, E \rangle,$$

где V – конечное непустое множество вершин; E – множество ребер (пар вершин).

Если пары E (ребра) имеют направление, то граф называется ориентированным (орграф), если иначе - неориентированный (неорграф). Если в пары E входят только различные вершины, то в графе нет петель. Если ребро графа имеет вес, то граф называется взвешенным. Степень вершины графа равна числу ребер, входящих и выходящих из нее (инцидентных ей). Неорграф называется связным, если существует путь из каждой вершины в любую другую.

Обозначим количество вершин как $n = |V|$, а количество ребер как $m = |E|$.

Графы в памяти могут представляться различным способом. Один из видов представления графов – это матрица смежности $B(n \times n)$; В этой матрице элемент $b[i,j]=1$, если ребро, связывающее вершины V_i и V_j существует и $b[i,j]=0$, если ребра нет. У неориентированных графов матрица смежности всегда симметрична.

Во многих случаях удобнее представлять граф в виде так называемого списка смежностей. Список смежностей содержит для каждой вершины из множества вершин V список тех вершин, которые непосредственно связаны с этой вершиной. Каждый элемент ($ZAP[u]$) списка смежностей является записью, содержащей данную вершину и указатель на следующую запись в

списке (для последней записи в списке этот указатель – пустой). Входы в списки смежностей для каждой вершины графа хранятся в таблице (массиве) (BEG [u])

Пример неориентированного графа приведен на рис. 1, матрица смежности для этого графа – на рис. 2, а список смежности – на рис. 3.

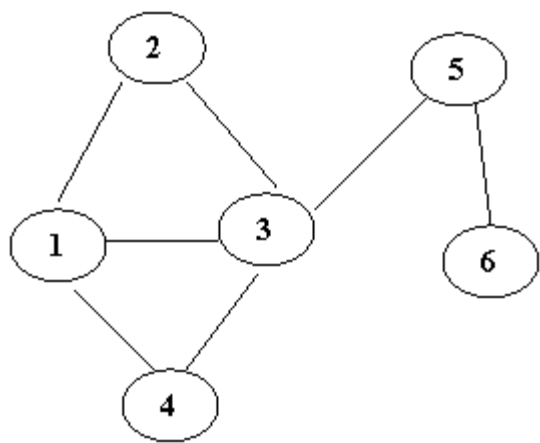


Рис. 1

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	0	0	0
3	1	1	0	1	1	0
4	1	0	1	0	0	0
5	0	0	1	0	0	1
6	0	0	0	0	1	0

Рис. 2

BEG[u]

ZAP[u]

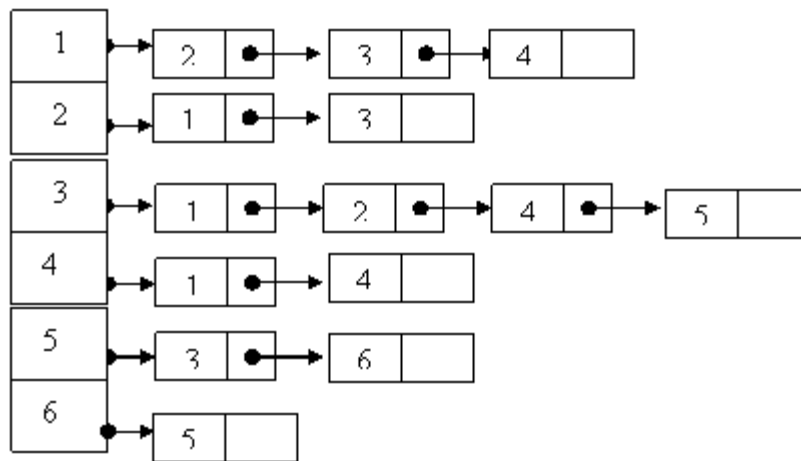


Рис. 3

Перечислим основные операции по работе с графовыми структурами:

- поиск кратчайшего пути от одной вершины к другой (если он есть);
- поиск кратчайшего пути от одной вершины ко всем другим;
- поиск кратчайших путей между всеми вершинами;
- поиск эйлера пути (если он есть);
- поиск гамильтонова пути (если он есть).

В основе построения большинства алгоритмов на графах лежит систематический перебор вершин графа, при котором каждая вершина просматривается в точности один раз, а количество просмотров ребер графа ограничено заданной константой (лучше – не более одного раза).

Один из основных методов проектирования графовых алгоритмов – это поиск (или обход графа) в глубину (depth first search, DFS), при котором начиная с произвольной вершины v_0 , ищется ближайшая смежная вершина v , для которой в свою очередь осуществляется поиск в глубину (т.е. снова ищется ближайшая, смежная с ней вершина) до тех пор, пока не встретится ранее просмотренная вершина, или не закончится список смежности вершины v (то есть вершина полностью обработана). Если нет новых вершин, смежных с v , то вершина v считается использованной, идет возврат в вершину, из которой попали в вершину v , и процесс продолжается до тех пор, пока не получим $v = v_0$. Иными словами, поиск в глубину из вершины v основан на поиске в глубину из всех новых вершин, смежных с вершиной v .

Путь, полученный методом поиска в глубину, в общем случае не является кратчайшим путем из вершины v в вершину u . Это общий недостаток поиска в глубину [1,2]. На рис. 4 показан путь, полученный обходом графа в глубину.

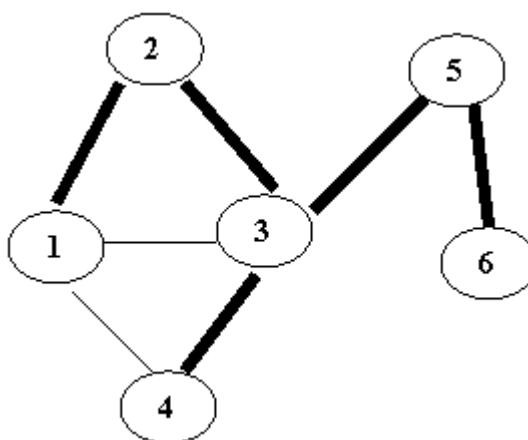


Рис. 4

Указанного недостатка лишен другой метод обхода графа – поиск в ширину (breadth first search, BFS). Обработка вершины v осуществляется путем просмотра сразу всех новых соседей этой вершины. При этом полученный путь является кратчайшим путем из одной вершины в другую [1].

На рис. 5 показан путь, найденный методом поиска в ширину

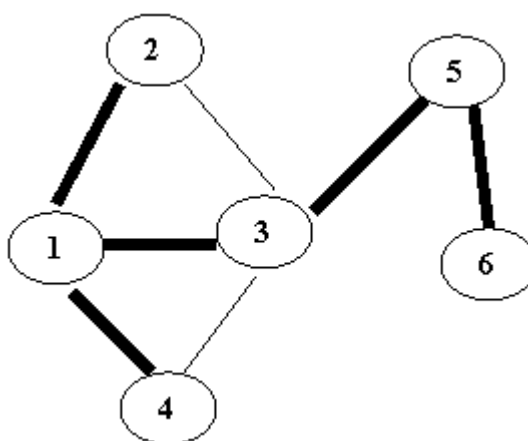


Рис. 5

При использовании алгоритмов DFS и BFS графы обходят разными способами, получая при этом некоторые подграфы, которые имеют специфические названия: каркасы, остовы или стягивающие деревья [1]. Все

вершины исходного графа входят в полученное стягивающее дерево, а все ветви дерева – это множество ребер из все множеств ребер графа (см. рис. 4, 5).

Произвольный путь в графе, проходящий через каждое ребро графа точно один раз, называется эйлеровым путем. При этом, если по некоторым вершинам путь проходит неоднократно, то он является непростым. Если путь замкнут, то имеем эйлеров цикл. Для существования эйлерова пути в связном графе необходимо и достаточно, чтобы граф содержал не более двух вершин нечетной степени [2,3]

Путь в графе, проходящий в точности один раз через каждую вершину графа (а не каждое ребро) и соответствующий цикл называются гамильтоновыми и существуют не для каждого графа, как и эйлеров путь. В отличие от эйлеровых путей неизвестно ни одного простого необходимого и достаточного условия для существования гамильтоновых путей. Неизвестен даже алгоритм полиномиальной сложности, проверяющий существование гамильтонова пути в произвольном графе. Проблема существования гамильтонова пути принадлежит к классу так называемых NP-полных задач [2, 3].

Поиск кратчайших путей до всех вершин из одной указанной вершины для взвешенного орграфа (имеющего значение, т.е. вес или стоимость, ребра) с неотрицательными ребрами осуществляется с использованием алгоритма Дейкстры. Алгоритм Дейкстры основан на выборе для включения в путь всякий раз той вершины, которая имеет наименьшую оценку кратчайшего пути (по весам ребер), то есть наименьший путь до этой вершины из всех возможных путей, которые были рассмотрены ранее [1].

Алгоритм Беллмана-Форда позволяет решить задачу о поиске кратчайших путях из одной выбранной вершины ко всем остальным вершинам при любых весах ребер, в том числе и отрицательных. Сначала ищется путь от выбранной вершины ко всем вершинам, связными с ней (аналогично поиску в ширину), а затем – кратчайший путь ко всем остальным вершинам, с попыткой последовательно пройти к ним, т.е. сначала через первую вершину, затем через

вторую, через третью и так далее до последней вершины. Кроме того алгоритм возвращает TRUE, если в графе нет цикла отрицательного веса, достижимого из данной вершины [2].

Для поиска кратчайших путей между всеми вершинами используется алгоритм Флойда-Уоршалла. По алгоритму Флойда-Уоршалла сначала ищется кратчайший путь от одной вершины ко всем вершинам, доступным из нее, затем проводятся те же действия, но пытаясь пройти от этой вершины ко всем доступным из нее, проходя каждый раз через новую вершину (сначала через первую, затем – через вторую и т.д.). Таким образом обрабатываются все вершины. [2].

Как уже было сказано, остовым деревом графа является дерево, содержащие все вершины графа. Стоимостью этого дерева является сумма стоимостей всех ребер [1].

Наиболее часто для построения остовых деревьев минимальной стоимости используют два алгоритма: алгоритм Крускала и алгоритм Прима.

Алгоритм Крускала, который при просмотре графа в лесу ребер, состоящему из нескольких связных компонент (деревьев), добавляет каждый раз минимальное ребро, концы которого лежат в разных компонентах леса.

Алгоритм Прима при просмотре графа строит одно дерево, в которое каждый раз добавляется ребро минимального веса [1].

Например, типичное применение остовых деревьев минимальной стоимости – это построение коммуникационных линий между городами, где стоимости ребер – это стоимость коммуникационных сетей.

Задание

Обработать графовую структуру в соответствии с заданным вариантом. Обосновать выбор необходимого алгоритма и выбор структуры для представления графов. Ввод данных осуществить на усмотрение программиста. Результат выдать в графической форме.

Примерные варианты заданий приведены в приложении.

Указания к выполнению работы

При отладке программы необходимо проверить правильность ввода данных. Программа должна адекватно реагировать на неверный ввод, в том числе на «пустой» ввод. Необходимо проверять поиск несуществующих путей в графе, использовать различные варианты графов для проверки правильности работы программы.

При разработке интерфейса программы следует предусмотреть:

- указание типа, формата и диапазона вводимых данных;
- указание действий, производимых программой;
- наличие пояснений при выводе результата;
- вывод графов осуществить в графическом виде (или предложить иную визуализацию в виде графа);
- вывод времени выполнения программы и объема требуемой памяти при использовании различных структур для представления графа.

При тестировании программы необходимо:

- о проверить правильность ввода и вывода данных (т.е. их соответствие требуемому типу и формату), обеспечить адекватную реакцию программы на неверный ввод данных;
- о обеспечить вывод сообщений при отсутствии входных данных («пустой ввод»);
- о проверить правильность выполнения операций;
- о предусмотреть вывод сообщения при поиске несуществующих путей в графе.

Содержание отчета

В отчете по лабораторной работе должен быть обоснован выбор формы представления графа и алгоритма, использованного для решения поставленной задачи. Следует указать сложность используемого алгоритма. Необходимо также предложить вариант реальной задачи, для которой можно использовать разработанную программу.

После выполнения работы студент должен уметь ответить на следующие вопросы:

1. Что такое граф?
2. Как представляются графы в памяти?
3. Какие операции возможны над графами?
4. Какие способы обхода графов существуют?
5. Где используются грифовые структуры?
6. Какие пути в графе Вы знаете?
7. Что такое каркасы графа?

Отчет представляется в электронном или печатном виде.

Примерные варианты заданий:

- 1 Найти все вершины графа, к которым от заданной вершины можно добраться по пути не длиннее A .
- 2 Задан граф не в виде дерева. Проверить, можно ли превратить его в дерево путем удаления одной вершины вместе с ее ребрами.
- 3 Задана система двусторонних дорог, где для любой пары городов есть соединяющий их путь. Найти город с минимальной суммой расстояний до остальных городов. Путь между двумя городами в две стороны может быть разным.
- 4 Задана система двусторонних дорог. Найти множество городов, расстояние от которых до выделенного города (столицы) больше, чем T . Путь между двумя городами в две стороны может быть разным.
- 5 Найти минимальное (по количеству ребер) подмножество ребер, удаление которых превращает заданный связный граф в несвязный

ПРАКТИЧЕСКАЯ РАБОТА № 4.

Тема: Симуляция работы мобильного робота на складе (используя Gazebo)

Цель: ознакомиться с RTS и системой симуляции Gazebo.

Для выполнения работы необходим MATLAB, примеры доступны по выполнению команды:

```
openExample('robotics/SimulateAMobileRobotInAWarehouseUsingGazeboExample')
```

Ход выполнения лабораторной работы

В этом примере показано, как симулировать складского робота в Gazebo. Gazebo позволил вам симулировать мобильного робота, который использует датчик области значений при выполнении определенных задач в симулированной среде. Этот пример детализирует, как использовать средство моделирования, чтобы применить Выполнить Задачи для Складского примера Робота, куда робот поставляет пакеты в складском сценарии. Робот использует симулированный датчик области значений в Gazebo, чтобы обнаружить возможные столкновения с динамической средой и избежать их.

Необходимые условия

- Загрузите [Виртуальную машину ROS и Gazebo](#), чтобы настроить симулированного робота.
- Рассмотрите [Выполнить Задачи для Складского примера Робота](#) для рабочего процесса планирования пути и перешедший в складском сценарии.
- Рассмотрите [Управление Робот с дифференциальным приводом в Gazebo c примером Simulink](#) для основных шагов сбора данных о датчике и управления роботом в Gazebo.

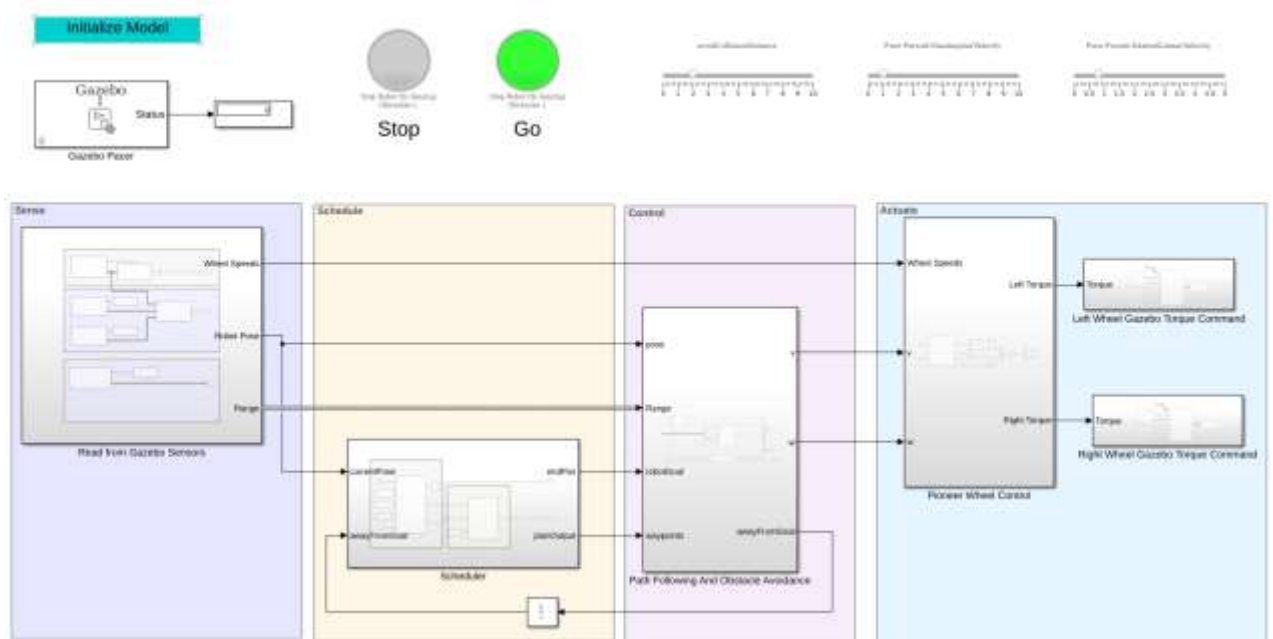
Обзор модели

Откройте модель.

```
open_system('simulateWarehouseRobotInGazebo.slx')
```

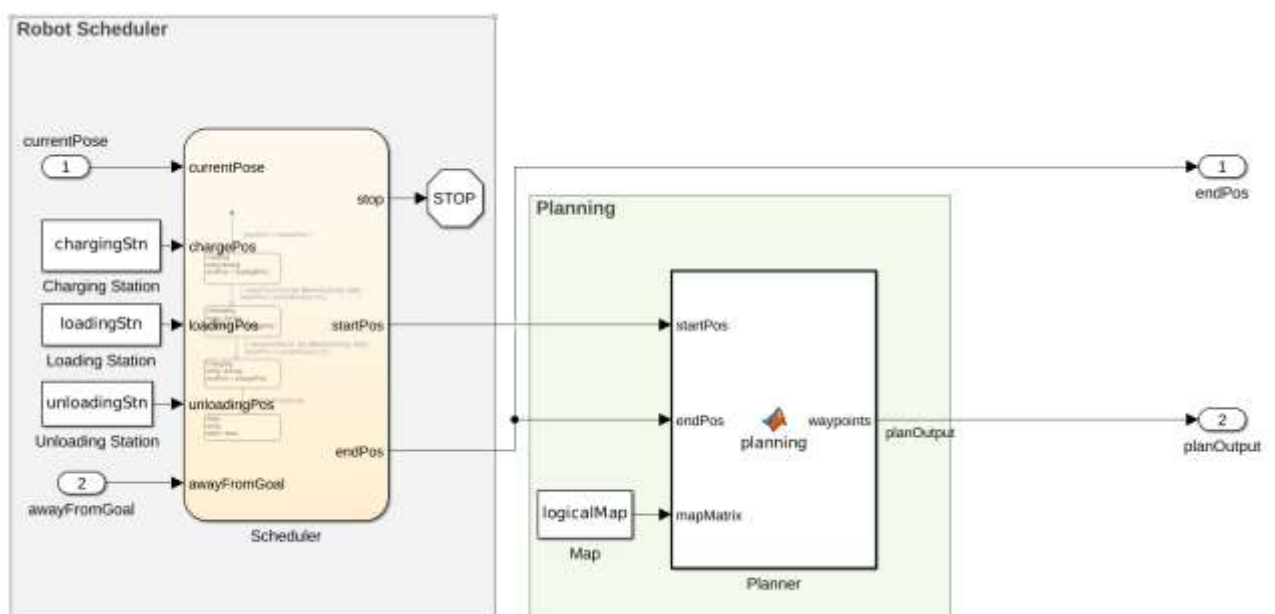
Модель может быть разделена на следующие элементы:

- Смысл: Считайте данные из датчиков в Gazebo.
- Расписание: Присвойте пакеты и пути к плану для роботов, чтобы поставить пакеты.
- Управление: Сгенерируйте команды, чтобы следовать за предопределенным путем и избежать препятствий.
- Приведите в движение: Отправьте команды в Gazebo, чтобы привести в движение робота в среде.



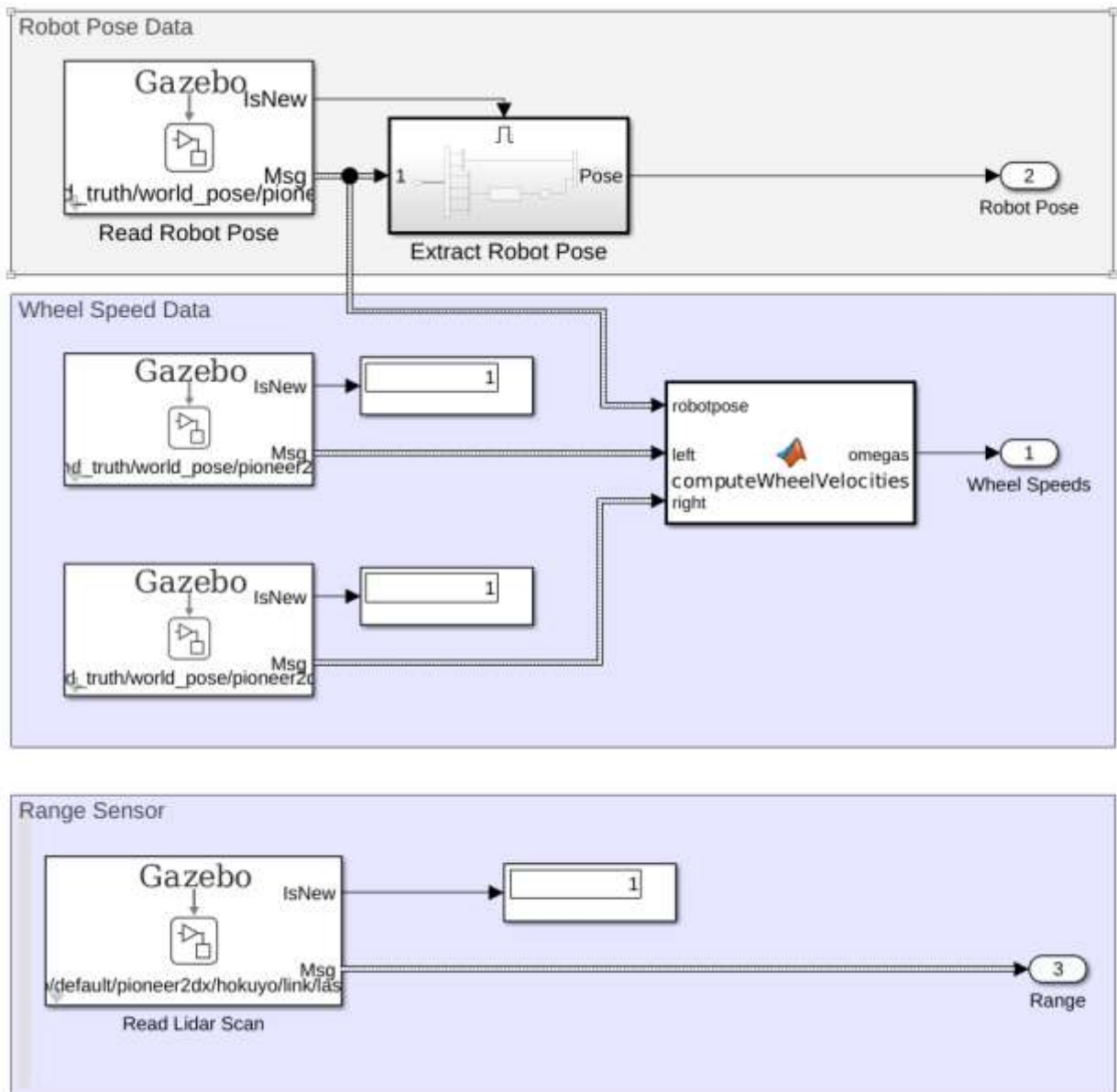
Расписание

Робот выполняет задачу движения между заряжающейся станцией, станцией загрузки и разгружающейся станцией, как ведется Планировщиком.



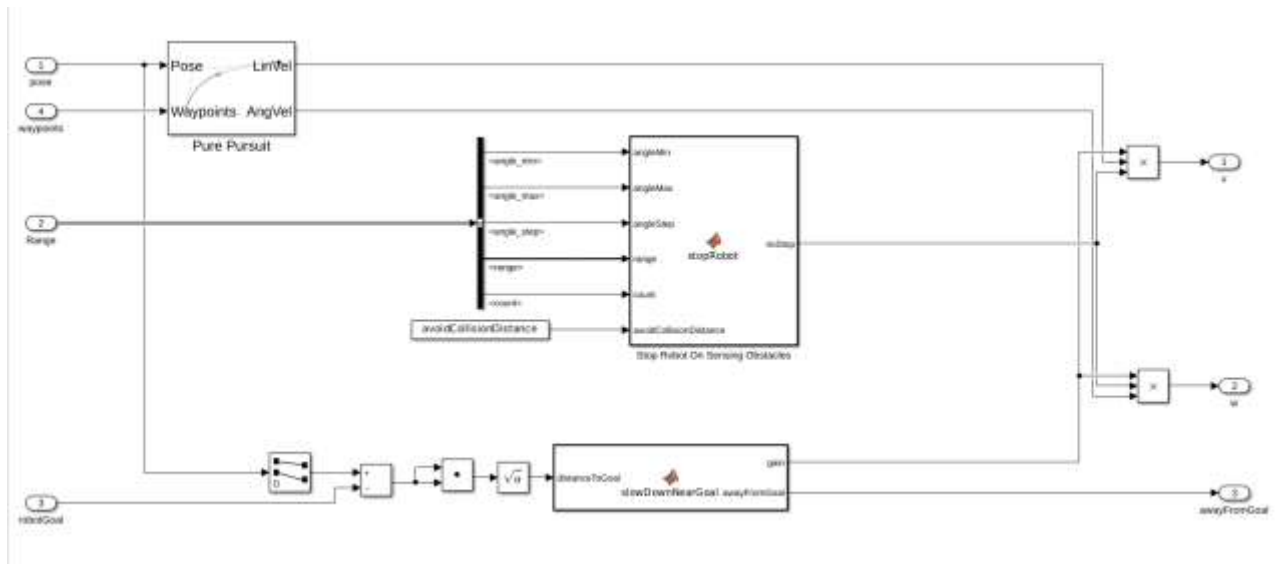
Смысл

Текущее положение робота, скорости колеса и показания датчика области значений читаются из симулированной среды в Gazebo. Рисунок ниже является расширенным представлением Чтения От подсистемы Датчиков Gazebo.



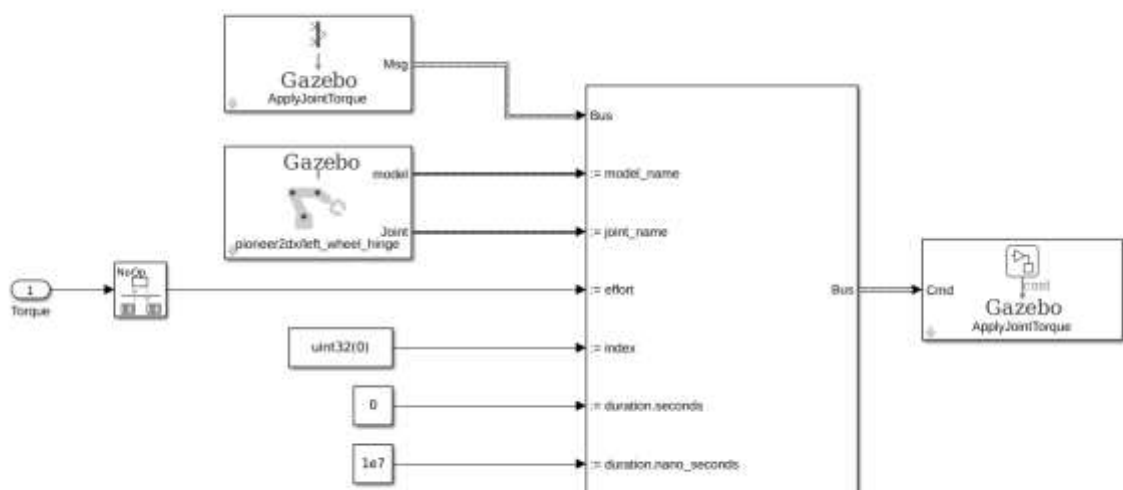
Управление

Контроллер генерирует команды управления для следующего waypoints использование блока Pure Pursuit. Если датчик области значений на роботе обнаруживает препятствие в avoidCollisionDistance порог, остановки робота. Кроме того, робот останавливается, когда добирается около достаточно до цели.



Привести в движение

На основе сгенерированных команд управления подсистема Управления Колесом Pioneer генерирует значение крутящего момента для каждого колеса. Крутящий момент применяется как ApplyJointTorque команда.



Настройка

Склад

Загрузите файл карты в качестве примера, map, который является матрицей логических значений, указывающих на занятое место на складе. Инвертируйте эту матрицу, чтобы указать на свободное пространство и сделать binaryOccupancyMap объект. Задайте разрешение 100 ячеек на метр.

Карта базируется прочь warehouseExtensions.world файл, который был сделан с помощью Редактора Создания на том же масштабном коэффициенте, как упомянуто ниже. .png файл для карты может быть сделан с

помощью `collision_map_creator_plugin` плагин, чтобы сгенерировать матрицу карты. Детали о том, как установить плагин, могут быть найдены в Плагине Создателя Карты Столкновения.

```
mapScalingFactor = 100;
```

```
load gazeboWarehouseMap.mat map
```

```
logicalMap = ~map;
```

```
map = binaryOccupancyMap(logicalMap,mapScalingFactor);
```

```
show(map)
```

Присвойте *ху-местоположения* заряжающейся станции, сортировав станцию и разгружающееся местоположение около полок на складе. Выбранные значения основаны на `simulated` мире в Gazebo.

```
chargingStn = [12,5];
```

```
loadingStn = [24,5];
```

```
unloadingStn = [15,24];
```

Покажите различные местоположения на карте.

```
hold on;
```

```
text(chargingStn(1), chargingStn(2), 1, 'Charging');
```

```
plotTransforms([chargingStn, 0], [1 0 0 0])
```

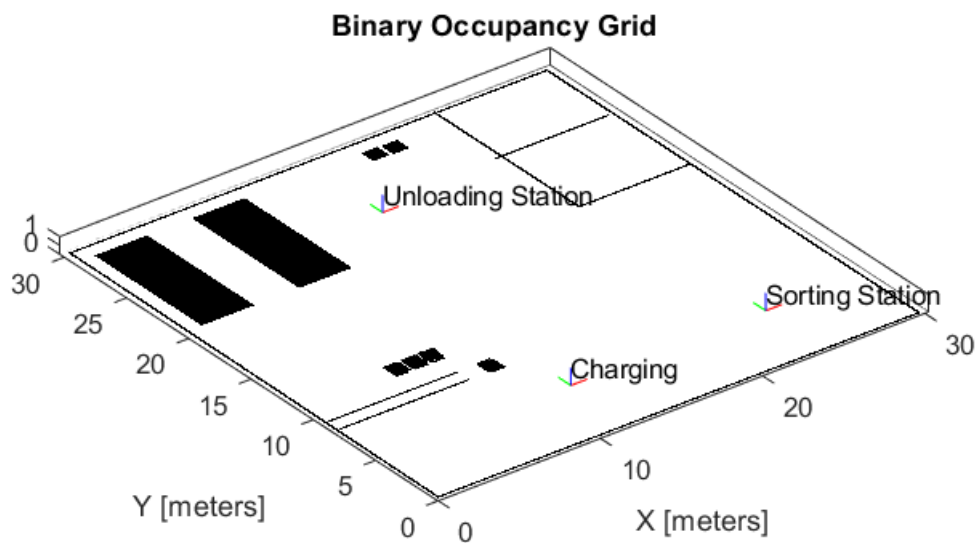
```
text(loadingStn(1), loadingStn(2), 1, 'Sorting Station');
```

```
plotTransforms([loadingStn, 0], [1 0 0 0])
```

```
text(unloadingStn(1), unloadingStn(2), 1, 'Unloading Station');
```

```
plotTransforms([unloadingStn, 0], [1 0 0 0])
```

```
hold off;
```



Датчик области значений

Блок Read Lidar Scan в разделе Sensing используется, чтобы считать значения диапазона из симулированного датчика области значений. warehouseExtensions.world файл содержит детали различных моделей и агентов (складские рабочие) в сцене. Поскольку `<actor>` теги являются статическими ссылками только с визуальными сетками, типом датчика датчика области значений является `gru_ray`.

Кроме того, датчик области значений использует 640 областей значений, но значение по умолчанию равняется 128. Это требует модификации шин, используемых в блоке Read Lidar Scan. Загрузите `exampleHelperWarehouseRobotWithGazeboBuses.mat` файл, чтобы получить модифицированную шину с `Gazebo_SL_Bus_gazebo_msgs_LaserScan.range` установите на 640. Модифицированные шины были сохранены в .mat файл с помощью Редактора Шины.

```
load exampleHelperWarehouseRobotWithGazeboBuses.mat
```

Предотвращение столкновения

Агенты в мире обходят predetermined траекторию. Робот использует датчик области значений, чтобы проверять на препятствия в диапазоне 2,0 м (`avoidCollisionDistance`) с углами диапазона от $[-\pi/10, \pi/10]$ После ненулевого чтения в той области значений и представления, остановок робота и только

возобновляется после того, как область значений будет ясна. "Робот Остановки На Распознающих Препятствиях" функциональный блок включает эту логику.

При выполнении симуляции лампа Остановки становится зеленой, когда смыслы робота, что хорошо продолжить. Если это остановилось, лампа покраснела.

`avoidCollisionDistance = 2;`

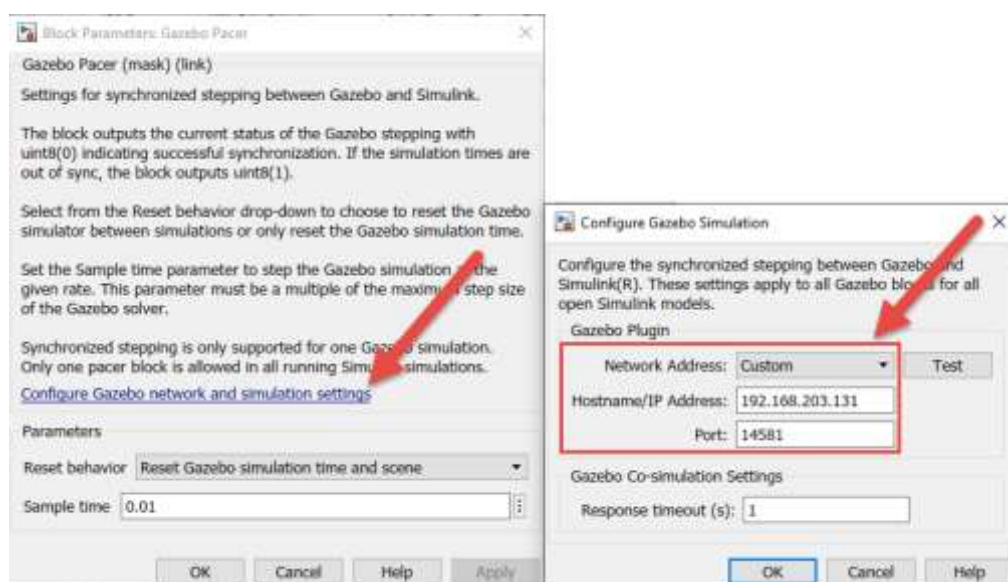
Симулировать

Чтобы симулировать сценарий, настройте связь с Gazebo.

Во-первых, запустите Средство моделирования Gazebo. В виртуальной машине кликните по Складскому значку Робота Gazebo. Если средству моделирования Gazebo не удастся открыться, вы, возможно, должны переустановить плагин. Смотрите Установку, Плагин Gazebo Вручную в Выполняет Со-симуляцию между Simulink и Gazebo.

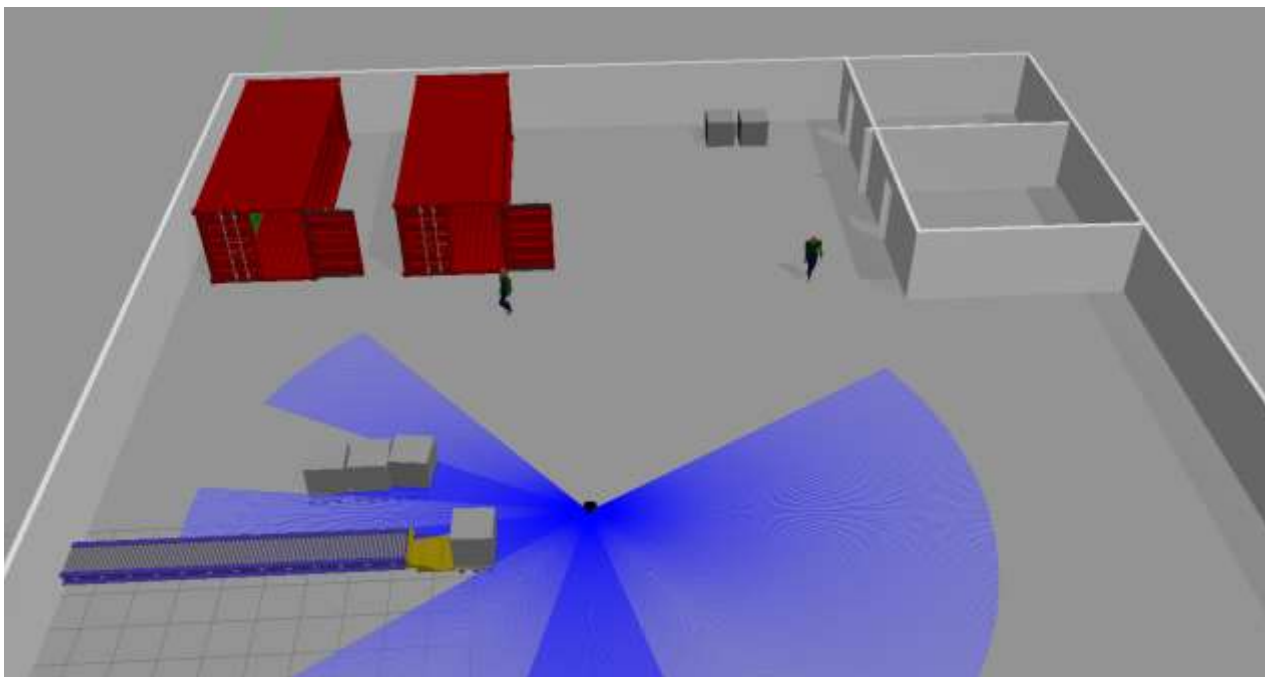
В Simulink, открытом блок Gazebo Pacer и сеть Configure Gazebo нажатия кнопки и настройки симуляции. Задайте Сетевой Адрес как Пользовательский, ИМЯ УЗЛА/IP-АДРЕС для вашей симуляции Gazebo и Порт 14581, который является портом по умолчанию для Gazebo. Рабочий стол VM отображает IP-адрес.

Для получения дополнительной информации о соединении с Gazebo, чтобы включить со-симуляцию, смотрите, Выполняют Со-симуляцию между Simulink и Gazebo.



Нажмите кнопку Initialize Model наверху модели, чтобы инициализировать все переменные, объявленные выше.

Запустите симуляцию. Робот управляет вокруг среды и остановок каждый раз, когда рабочий добирается в заданном пороге.



Дополнительные детали настройки представлены по ссылке <https://docs.exponenta.ru/robotics/ug/simulate-a-mobile-robot-using-gazebo.html>

Содержание отчета:

Пример выполнения представленной имитационной модели робота в Gazebo.

ПРАКТИЧЕСКАЯ РАБОТА № 5.

Тема: Оптимизация алгоритмов с помощью хеш-таблиц.

Цель: изучение принципов построения и работы хеш-таблиц.

Краткие теоретические сведения

Рассмотрим алгоритм построения двоичного дерева. Если при добавлении узлов в дерево мы будем их равномерно располагать слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. Такое дерево называется **идеально сбалансированным**.

Для построения идеально сбалансированного дерева используется рекурсия. Для дерева из n узлов, где nl - количество узлов в левом поддереве, nr - количество узлов в правом поддереве, алгоритм построения идеально сбалансированного дерева описывается следующим образом:

1. выбрать одну вершину в качестве корня;
2. рекурсивно построить левое поддерево с $nl = n \div 2$ узлами;
3. рекурсивно построить правое поддерево с $nr = n - nl - 1$ узлами.

Идеальная балансировка дает наименьшую высоту дерева, а так как высота дерева определяет длину пути поиска в нем, то, следовательно, и укорачивает поиск. Но поддержание идеальной сбалансированности дерева при включении или исключении элемента – это достаточно сложная процедура.

Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется *АВЛ-деревом*.

Использование этого критерия приводит к легко выполняемой балансировке. При этом средняя длина поиска остается практически такой же, как и у идеально сбалансированного дерева.

При включении узла в сбалансированное дерево возможны 3 случая: (рассматриваем включение в левое поддерево):

- 1) левое и правое поддеревья становятся неравной высоты, но критерий сбалансированности не нарушается;
- 2) левое и правое поддерево приобретают равную высоту и, таким образом, сбалансированность даже улучшается;

3) критерий сбалансированности нарушается, и дерево надо перестраивать.

Алгоритм включения и балансировки существенно зависит от способа хранения информации о сбалансированности дерева. Одно из решений – хранить в каждой вершине показатель ее сбалансированности. В этом случае сбалансированность будет определяться как разность между высотой правого и левого поддеревьев. Например, $Bal=1$, если правое поддерево выше левого, $Bal=0$, если правое и левое поддерева равной высоты и $Bal=-1$, если левое поддерево выше правого.

Процесс включения узла фактически состоит из трех последовательно выполняемых подзадач:

- 1) проход по пути поиска (пока не убедимся, что элемента с таким ключом в дереве нет);
- 2) включение нового узла и определение показателя сбалансированности (Bal);
- 3) «отступление» (возврат) по пути поиска, с проверкой показателей сбалансированности для каждой вершины, и если необходимо, то проведение балансировки.

Этот метод требует некоторой избыточной проверки, но зато его можно реализовать с помощью простого расширения алгоритма поиска с включением. На каждом шаге при этом необходима информация о высоте дерева.

Операция по балансировке состоит только из последовательных переприсваиваний ссылок. Фактически ссылки циклически меняются, что приводит к одно- или двукратному повороту двух или трех участвующих в процессе балансировки узлов. Кроме вращения, необходимо должным образом изменять и показатели сбалансированности этих узлов.[1]

Алгоритм удаления узла из сбалансированного дерева основан на алгоритме удаления из дерева (на замене удаляемого узла на самого левого потомка из правого поддерева или на самого правого потомка из левого поддерева) с учетом операции балансировки, то есть тех же поворотов узлов.[1]

Использование деревьев для поиска информации достаточно эффективно (трудоемкость – $O(\log_2 n)$). Можно ли создать еще более эффективную структуру или метод, позволяющий лучше осуществлять поиск информации? Для этого было бы хорошо по значению ключа сразу определять индекс элемента массива, в котором хранится информация. То есть необходимо создать такую функцию, по которой можно вычислить этот индекс. Такая функция называется *хеш-функцией* (от англ. to hash - крошить, рубить) и она ставит в соответствие каждому ключу k_i индекс ячейки j , где расположен элемент с этим ключом, таким образом:

$h(k_i) = j$, если $j=(1,m)$,

где j принадлежит множеству от 1 до m , а m – размерность массива.

Массив, заполненный в порядке, определенным хеш-функцией, называется *хеш-таблицей*. Минимальная трудоемкость поиска в хеш-таблице равна $O(1)$!

Принято считать, что хорошей является такая функция, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно.

Итак, первое свойство хорошей хеш-функции зависит от характеристик компьютера, а второе – от значений данных. Если бы все данные были случайными, то хеш-функции были бы очень простые (несколько битов ключа, например). Однако на практике случайные данные встречаются крайне редко, и приходится создавать функцию, которая зависела бы от всего ключа.

Может возникнуть ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда $h(K1)=h(K2)$, в то время как $K1 \neq K2$. Такая ситуация называется *коллизией*. В этом случае, очевидно, необходимо найти новое место для хранения ключей, претендующих на одну и ту же ячейку хеш-таблицы. Причем количество коллизий необходимо минимизировать. Таким образом, хорошая хеш-функция должна удовлетворять еще одному требованию, а именно: она должна минимизировать число коллизий.

Существует два основных типа хеширования, один из которых основан на делении, а другой на умножении. Впрочем, это не единственные методы, которые существуют, более того, они не всегда являются оптимальными.

Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно принять вычисление остатка от деления целочисленного ключа на размерность массива (m), то есть:

$$h(k_i) = (k_i \bmod m),$$

при $n < m$, где n – количество ключей.

Данная функция очень проста, хотя и не самая лучшая. Вообще, можно использовать любую размерность (константу m), но она должна быть такой, чтобы минимизировать число коллизий. Для этого лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен. Для символьной строки ключом может являться остаток от деления, например, суммы кодов символов строки на m .

На практике, метод деления – самый распространенный

Существует несколько возможных вариантов разрешения коллизий, которые имеют свои достоинства и недостатки.

Первый метод – внешнее (открытое) хеширование (метод цепочек)

В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким.

Другой путь решения проблемы, связанной с коллизиями – внутреннее (закрытое) хеширование (открытая адресация). Оно, состоит в том, чтобы полностью отказаться от ссылок. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ К или пустая позиция в таблице. При этом, если индекс следующего просматриваемого элемента определяется добавлением какого-то постоянного шага (от 1 до n), то данный способ разрешения коллизий называется линейной адресацией. Для вычисления шага можно также применить формулу:

$$h = h + a^2,$$

где а – это номер попытки поиска ключа. Этот вид адресации называется квадратичной или произвольной адресацией.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3–4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента

Задание

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице. Вывести на экран деревья и хеш-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи.

Примерные варианты заданий приведены в приложении.

Указания к выполнению работы

При разработке интерфейса программы следует предусмотреть:

- указание типа, формата и диапазона вводимых данных;
- указание действий, производимых программой;
- наличие пояснений при выводе результата;
- вывод деревьев осуществить в графическом виде (или предложить иную визуализацию в виде дерева);
- вывод на экран хеш-таблицы;
- возможность изменения хеш-функции при необходимости реструктуризации таблицы;
- вывод времени и количества сравнений при поиске одних и тех же данных в различных структурах данных.

При тестировании программы необходимо:

- о проверить правильность ввода и вывода данных (то есть их соответствие требуемому типу и формату), обеспечить адекватную реакцию программы на неверный ввод данных;
- о обеспечить вывод сообщений при отсутствии входных данных («пустой ввод»);
- о проверить правильность выполнения операций;
- о предусмотреть вывод сообщения при необходимости реструктуризации хеш-таблицы;
- о проверить поиск существующих и поиск несуществующих данных;

Содержание отчета

В отчете по лабораторной работе должны быть сделаны выводы о том, применение какой структуры (дерева двоичного поиска, сбалансированного дерева или хеш-таблицы) целесообразно для решения поставленной задачи, какие преимущества дает использование той или иной структуры. Выводы следует подтвердить результатом числовых сравнений расходования памяти и времени выполнения программы, а также количеством сравнений при поиске данных. В отчете также следует указать, в каком случае необходима реструктуризация хеш-таблицы и на что нужно обратить особое внимание при тестировании программы

В отчете по лабораторной работе должны быть даны ответы на следующие вопросы:

1. Чем отличается идеально сбалансированное дерево от AVL дерева?
2. Чем отличается поиск в AVL-дереве от поиска в дереве двоичного поиска?
3. Что такое хеш-таблица, каков принцип ее построения?
4. Что такое коллизии? Каковы методы их устранения.
5. В каком случае поиск в хеш-таблицах становится неэффективен?
6. Эффективность поиска в AVL деревьях, в дереве двоичного поиска и в хеш-таблицах

Отчет представляется в электронном или печатном виде.

Приложение

Примерные варианты заданий

1 Построить двоичное дерево поиска из букв вводимой строки. Вывести его на экран в виде дерева. Выделить цветом все буквы, встречающиеся более одного раза. Удалить из дерева эти буквы. Сбалансировать дерево после удаления повторяющихся букв. Вывести его на экран в виде дерева. Составить хеш-таблицу, содержащую буквы и количество их вхождений во введенной строке. Вывести таблицу на экран. Осуществить поиск введенной буквы в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

2 Построить хеш-таблицу из слов произвольного текстового файла, задав ее размерность с экрана. Вывести построенную таблицу слов на экран. Осуществить поиск введенного слова. Выполнить программу для различных размерностей таблицы и сравнить время поиска и количество сравнений. Удалить все слова, начинающиеся на указанную букву, вывести таблицу. Сравнить время поиска, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц.

3 Построить хеш-таблицу для зарезервированных слов, используемого языка программирования (не менее 20 слов), содержащую HELP для каждого слова. Выдать на экран подсказку по введенному слову. Сравнить время поиска, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц. Добавить подсказку по вновь введенному слову, используя при необходимости реструктуризацию таблицы. Сравнить эффективность добавления ключа в таблицу или ее реструктуризацию для различной степени заполненности таблицы.

4 В текстовом файле содержатся целые числа. Построить двоичное дерево поиска из чисел файла, сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

ПРАКТИЧЕСКАЯ РАБОТА № 6.

Тема: Реализация многопоточного алгоритма

Цель работы: ознакомиться с механизмом многозадачности в современных операционных системах, получить навыки по разработке многопоточных приложений в ОС Windows.

1.1 Теоретические сведения

1.1.1 Понятия мультипрограммирования и многозадачности

Важнейшей функцией операционной системы является организация рационального использования всех ее аппаратных и информационных ресурсов. К основным ресурсам могут быть отнесены процессоры, память, внешние устройства, данные и программы. Главные сложности возникают в мультипрограммных операционных системах (ОС), в которых за ресурсы конкурируют сразу несколько приложений. Именно поэтому большая часть всех проблем относится к мультипрограммным системам.

Мультипрограммирование, или *многозадачность (multitasking)* – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ [1]. Эти программы совместно используют не только процессор, но и другие ресурсы компьютера: оперативную и внешнюю память, устройства ввода-вывода, данные.

1.1.2 Понятия «процесс» и «поток»

Чтобы поддерживать мультипрограммирование, ОС должна определить и оформить для себя те внутренние единицы работы, между которыми будет разделяться процессор и другие ресурсы компьютера. В настоящее время в большинстве операционных систем определены два типа единиц работы. Более крупная единица работы, обычно носящая название *процесса*, или *задачи*, требует предварительного выполнения нескольких более мелких работ, для обозначения которых используют термины *«поток»*, или *«нить»* [1].

Таким образом, процесс – это объект, с которым сопоставляются все ресурсы, в том числе и адресное пространство, т. е. он является структурой в памяти. В адресном пространстве процесса, кроме кодов и данных, находятся также потоки.

Работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Чтобы этот программный код мог быть выполнен, его необходимо загрузить в оперативную память, возможно, выделить некоторое место на диске для хранения данных,

предоставить доступ к устройствам ввода-вывода, например, к последовательному порту для получения данных по подключенному к этому порту модему и т. д. Кроме того, необходимо предоставление программе процессорного времени, то есть времени, в течение которого процессор выполняет коды данной программы.

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является *изоляция* одного процесса от другого. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

Виртуальное адресное пространство процесса – это совокупность адресов, которыми может манипулировать программный модуль процесса [1]. Операционная система отображает виртуальное адресное пространство процесса в отведенной процессу физической памяти.

При необходимости взаимодействия процессы обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – конвейеры, почтовые ящики, разделяемые секции памяти и некоторые другие.

Потоки возникли в операционных системах как средство распараллеливания вычислений. Конечно, задача распараллеливания вычислений в рамках одного приложения может быть решена и традиционными способами.

Во-первых, программист может взять на себя сложную задачу организации параллелизма, выделив в приложении некоторую подпрограмму-диспетчер, которая периодически передает управление той или иной ветви вычислений. При этом программа получается логически весьма запутанной, с многочисленными передачами управления, что существенно затрудняет ее отладку и модификацию.

Во-вторых, решением является создание для одного приложения нескольких процессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что эти процессы решают единую задачу, а значит, имеют много общего между собой. А операционная система при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и с помощью универсальных механизмов обеспечивать их изоляцию друг от друга. Кроме того, на создание каждого процесса ОС тратит определенные системные ресурсы, которые в данном случае неоправданно дублируются – каждому процессу выделяются собственное виртуальное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т. п.

Таким образом, для эффективной обработки задач в любой операционной системе наряду с процессами необходим специальный механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС используют механизм *многопоточной обработки (multithreading)* [1]. При этом вводится новая единица работы – поток выполнения, а понятие «процесс» в значительной степени меняет смысл. Понятию «поток» соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков требует от ОС меньших накладных расходов, чем процессов. В отличие от процессов, которые принадлежат разным или конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Таким образом, мультипрограммирование более эффективно на уровне потоков, а не процессов [1].

Использование потоков связано не только со стремлением повысить производительность системы за счет параллельных вычислений, но и с целью создания более читабельных, логичных программ. Введение нескольких потоков выполнения упрощает программирование.

Наибольший эффект от введения многопоточной обработки достигается в мультипроцессорных системах, в которых потоки, в том числе и принадлежащие одному процессу, могут выполняться на разных процессорах действительно параллельно (а не псевдопараллельно).

1.1.3 Особенности реализации механизма мультипрограммирования для повышения эффективности использования вычислительной системы

Мультипрограммирование по своей сути призвано повысить эффективность использования вычислительной системы, однако эффективность может пониматься по-разному. Наиболее характерными критериями эффективности вычислительных систем являются [1]:

- 1) *пропускная способность* – количество задач, выполняемых вычислительной системой в единицу времени;
- 2) *удобство работы пользователей*;
- 3) *реактивность системы* – способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата.

В зависимости от выбранного критерия эффективности ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени [1]. Каждый тип ОС имеет специфические внутренние механизмы и особые области применения. Некоторые операционные системы могут поддерживать одновременно несколько режимов, например, часть задач может выполняться в режиме пакетной обработки, а часть – в режиме реального времени или в режиме разделения времени.

Системы пакетной обработки

При использовании мультипрограммирования для повышения пропускной способности компьютера главной целью является минимизация простоев всех устройств компьютера и прежде всего центрального процессора. Такие простои могут возникать из-за приостановки задачи по ее внутренним причинам, связанным, например, с ожиданием ввода данных для обработки. При возникновении такого рода блокировки выполняемой задачи выполняется переключение процессора на выполнение другой задачи, у которой есть данные для обработки. Такая концепция мультипрограммирования положена в основу пакетных систем.

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени.

Для достижения этой цели в системах пакетной обработки используется следующая схема функционирования: в начале работы формируется пакет заданий, из этого пакета заданий формируется мультипрограммная смесь, т. е. множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. Таким образом, выбирается «выгодное» задание. В вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени.

Совмещение во времени операций ввода-вывода и вычислений может достигаться разными способами. Один из них – использование компьютеров, имеющих специализированный процессор ввода-вывода (канал). Обычно канал имеет

систему команд, которые предназначены для управления внешними устройствами, например, «проверить состояние устройства», «установить магнитную головку», «установить начало листа», «напечатать строку». В системе команд центрального процессора предусматривается специальная инструкция, с помощью которой каналу передаются параметры и указания на то, какую программу ввода-вывода он должен выполнить. Начиная с этого момента центральный процессор и канал могут работать параллельно (рисунок 1, а).

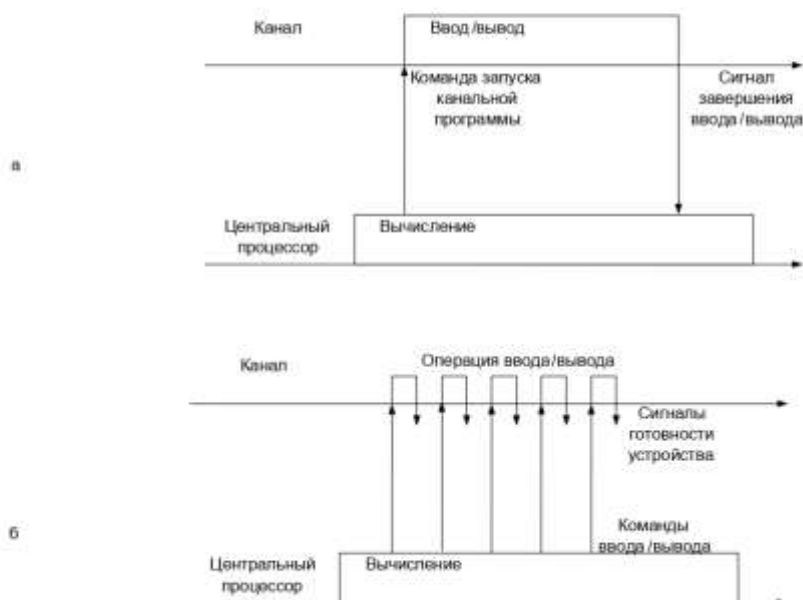


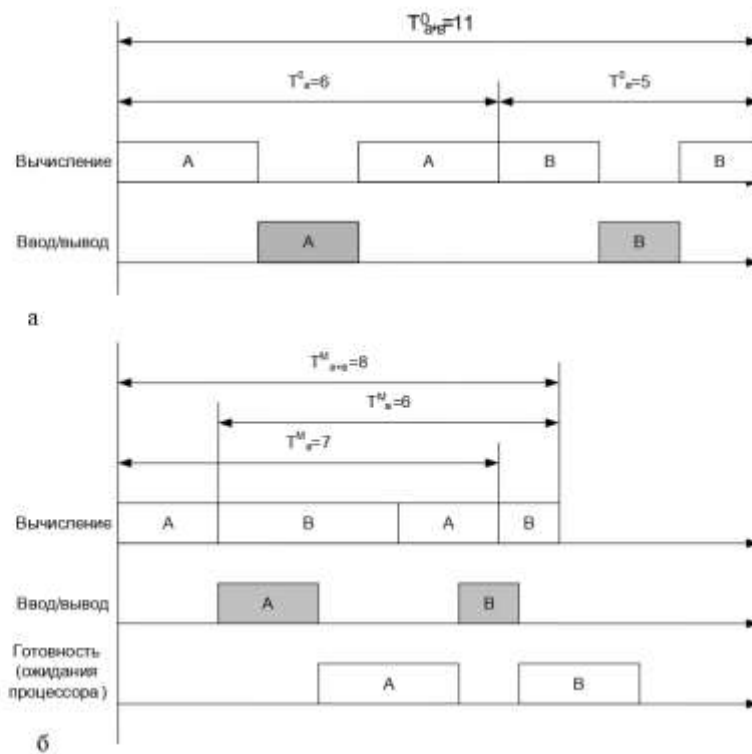
Рисунок 1 – Параллельное выполнение вычислений и операций ввода-вывода [1]

Другой способ совмещения вычислений с операциями ввода-вывода реализуется в компьютерах, в которых внешние устройства управляются не процессором ввода-вывода, а контроллерами. Каждое внешнее устройство (или группа внешних устройств одного типа) имеет свой собственный контроллер, который автономно обрабатывает команды, поступающие от центрального процессора. При этом контроллер и центральный процессор работают асинхронно. Контроллер выполняет свои команды управления устройствами существенно медленнее, чем центральный процессор – свои. Это обстоятельство используется для организации параллельного выполнения вычислений и операций ввода-вывода: в промежутке между передачей команд контроллеру центральный процессор может выполнять

вычисления (рисунок 1, б). Контроллер может сообщить центральному процессору о том, что он готов принять следующую команду, сигналом прерывания либо центральный процессор узнает об этом, периодически опрашивая состояние контроллера.

Если же в системе выполняются одновременно несколько задач, появляется возможность совмещения вычислений одной задачи с вводом-выводом другой.

Общее время выполнения смеси задач часто оказывается меньше, чем их суммарное время последовательного выполнения (рисунок 2, а).



а – в однопрограммной системе; б – в мультипрограммной системе [1]
Рисунок 2 – Время выполнения двух задач

Однако при совместном использовании процессора в системе могут возникать ситуации, когда задача готова выполняться, но процессор занят выполнением

другой задачи. В таких случаях задача, завершившая ввод-вывод, вынуждена ждать освобождения процессора, и это удлиняет срок ее выполнения.

В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит по инициативе самой активной задачи. При этом одна задача может надолго занять процессор и выполнение интерактивных задач станет невозможным. Взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок повышает эффективность функционирования аппаратуры, но снижает эффективность работы пользователя.

Особенности реализации механизма мультипрограммирования в системах разделения времени

Повышение удобства и эффективности работы пользователя – цель исследования систем разделения времени. В них пользователям (или одному пользователю) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность «общения» с пользователем. При этом возможности диалога пользователя с приложением ограничены.

В системах разделения времени ОС всем приложениям попеременно выделяется квант процессорного времени, в течение которого программа обрабатывает поступившие в ее адрес сообщения, после чего вне зависимости от состояния программы система забирает управление от программы и передает ее другой программе.

Таким образом, пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки – изоляцию пользователя-программиста от процесса выполнения его задач.

Системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая «выгодна» системе. Это вполне соответствует тому, что критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя. Вместе с тем мультипрограммное выполнение интерактивных приложений повышает и пропускную способность компьютера (пусть и не в такой степени, как пакетные системы). Аппаратура загружается лучше, поскольку в то время, пока одно приложение ждет сообщения пользователя, другие приложения могут обрабатываться процессором.

Мультипрограммирование в системах реального времени

Еще одна разновидность мультипрограммирования используется в *системах реального времени*, предназначенных для управления от компьютера различными техническими объектами или технологическими процессами [1]. Критерием эффективности этих систем является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (т. е. время реакции системы). Это свойство системы называется реактивностью.

В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется по прерываниям (исходя из текущего состояния объекта) или в соответствии с расписанием плановых работ.

Способность аппаратуры компьютера и ОС к быстрому ответу зависит в основном от скорости переключения с одной задачи на другую и, в частности, от скорости обработки сигналов прерывания.

В системах реального времени не стремятся максимально загружать все устройства, наоборот, обычно закладывается некоторый «запас» вычислительной мощности на случай пиковой нагрузки.

Если система реального времени не спроектирована для поддержки пиковой нагрузки, то может случиться так, что система не справится с работой именно тогда, когда она нужна в наибольшей степени.

1.1.4 Планирование процессов и потоков

Одной из основных подсистем мультипрограммной ОС, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами и потоками, которая занимается их созданием и уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколькими одновременно существующими в системе процессами и потоками.

Подсистема управления процессами и потоками ответственна за *обеспечение процессов необходимыми ресурсами* [1]. ОС поддерживает в памяти специальные информационные структуры, в которые записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы в единоличное пользование или в совместное пользование с другими процессами. Подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление ресурсами, такими, как подсистема управления памятью, подсистема ввода-вывода, файловая система.

Когда в системе одновременно выполняется несколько независимых задач, то возникают дополнительные проблемы. Согласование скоростей потоков также очень важно для предотвращения эффекта «гонок» (когда несколько потоков пы-

таются изменить один и тот же файл), взаимных блокировок или других коллизий, которые возникают при совместном использовании ресурсов. *Синхронизация* потоков является одной из важных функций подсистемы управления процессами и потоками.

Каждый раз, когда процесс завершается, ОС предпринимает шаги, чтобы «зачистить следы» его пребывания в системе. Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса. Выполняется коррекция всевозможных очередей ОС и списков ресурсов, в которых имелись ссылки на завершаемый процесс.

1.1.5 Создание процессов и потоков

Создать процесс – это прежде всего означает создать *описатель процесса*, в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые операционной системе для управления им [1]. В число таких сведений могут входить, например, идентификатор процесса, данные о расположении в памяти исполняемого модуля, степень привилегированности процесса (приоритет и права доступа) и т. п.

Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого ОС должна обнаружить местоположение такой программы на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Затем необходимо считать программу в выделенные для нее участки памяти и, возможно, изменить параметры программы в зависимости от размещения в памяти. В системах с виртуальной памятью в начальный момент может загружаться только часть кодов и данных процесса, чтобы «подкачивать» остальные по мере необходимости.

В многопоточной системе при создании процесса ОС создает для каждого процесса как минимум один поток выполнения. При создании потока так же, как и при создании процесса, операционная система генерирует специальную информационную структуру – описатель потока, который содержит идентификатор потока, данные о правах доступа и приоритете, о состоянии потока и другую информацию. В исходном состоянии поток (или процесс, если речь идет о системе, в которой понятие «поток» не определяется) находится в приостановленном состоянии. Момент выборки потока на выполнение осуществляется в соответствии с принятым в данной системе правилом предоставления процессорного времени и с учетом всех существующих в данный момент потоков и процессов. Необходимым условием активизации потока процесса является также наличие места в оперативной памяти для загрузки его исполняемого модуля.

1.1.6 Состояния потока

ОС выполняет планирование потоков, принимая во внимание их состояние. В мультипрограммной системе поток может находиться в одном из трех основных состояний:

1) *выполнение* – активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

2) *ожидание* – пассивное состояние потока, находясь в котором, поток заблокирован по своим внутренним причинам (ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого потока или освобождения какого-либо необходимого ему ресурса);

3) *готовность* – также пассивное состояние потока, но в этом случае поток заблокирован в связи с внешним по отношению к нему обстоятельством (имеет все требуемые для него ресурсы, готов выполняться, однако процессор занят выполнением другого потока).

В течение своей жизни каждый поток переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятым в данной операционной системе.

Рассмотрим типичный граф состояния потока (рисунок 3). Только что созданный поток находится в состоянии готовности, он готов к выполнению и стоит в очереди к процессору. Когда в результате планирования подсистема управления потоками принимает решение об активизации данного потока, он переходит в состояние выполнения и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние ожидания какого-нибудь события, либо будет принудительно «вытеснен» из процессора, например, вследствие исчерпания отведенного данному потоку кванта процессорного времени. В последнем случае поток возвращается в состояние готовности. В это же состояние поток переходит из состояния ожидания, после того как ожидаемое событие произойдет.

В состоянии выполнения в однопроцессорной системе может находиться не более одного потока, а в каждом из состояний ожидания и готовности – несколько потоков. Эти потоки образуют очереди соответственно ожидающих и готовых потоков. Очереди потоков организуются путем объединения в списки описателей отдельных потоков. Таким образом, каждый описатель потока, кроме всего прочего, содержит по крайней мере один указатель на другой описатель, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать потоки, переводить потоки из одного состояния в другое. Если предположить, что на рисунке 4 показана очередь готовых потоков, то запланированный порядок выполнения выглядит так: А, В, Е, D, С.



Рисунок 3 – Граф состояний потока в многозадачной среде [1]

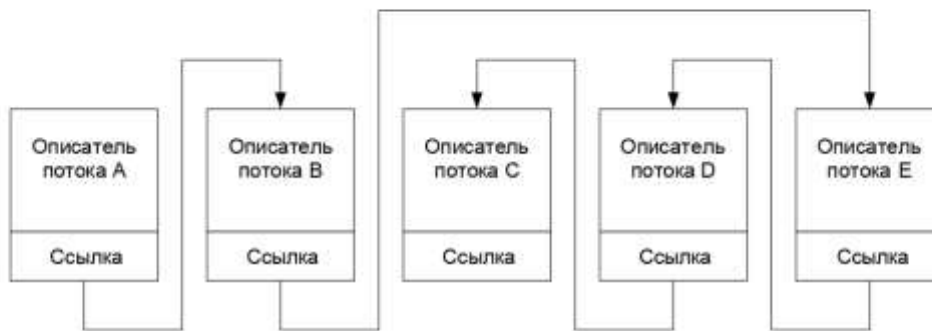


Рисунок 4 – Очередь потоков [1]

1.1.7 Вытесняющие и невытесняющие алгоритмы планирования

Все множество алгоритмов планирования можно разделить на два класса: вытесняющие и невытесняющие алгоритмы планирования.

1) *Невытесняющие (non-preemptive)* алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди очередной готовый к выполнению поток.

2) *Вытесняющие (preemptive)* алгоритмы – это такие способы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается операционной системой, а не активной задачей.

Основным различием между вытесняющими и невытесняющими алгоритмами является степень централизации механизма планирования потоков. При вытесняющем мультипрограммировании функции планирования потоков целиком сосредоточены в операционной системе. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активного потока, запоминает его контекст, выбирает из очереди готовых потоков следующий, запускает новый поток на выполнение, загружая его контекст.

При невытесняющем мультипрограммировании механизм планирования распределен между операционной системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения очередного цикла своего выполнения и только затем передает управление ОС с помощью какого-либо системного вызова. ОС формирует очереди потоков и выбирает в соответствии с некоторым правилом (например, с учетом приоритетов) следующий поток на выполнение. Такой механизм создает проблемы как для пользователей, так и для разработчиков приложений.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Поэтому разработчики приложений для операционной среды с невытесняющей многозадачностью вынуждены, возлагая на себя часть функций планировщика, создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Подобный метод разделения времени между задачами существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам. Для этого в программе должны быть предусмотрены частые передачи управления операционной системе. Крайним проявлением «не дружественности» приложения является его зависание, которое приводит к общему краху системы. В си-

стемах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм имеет возможность снять зависящую задачу с выполнения.

Однако распределение функций планирования потоков между системой и приложениями при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Существенным преимуществом невытесняющего планирования является более высокая скорость переключения с потока на поток.

Почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений, реализованы вытесняющие алгоритмы планирования потоков (процессов) [1, 2].

1.1.8 Планирование и диспетчеризация потоков

На протяжении существования процесса выполнение его потоков может быть многократно прервано и продолжено. Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации [1]. Работа по определению того, в какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться, называется *планированием* [1]. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании могут приниматься во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и другие факторы.

Планирование потоков, по существу, включает в себя решение двух задач:

- 1) определение момента времени для смены текущего активного потока;
- 2) выбор для выполнения потока из очереди готовых потоков.

Существует множество различных алгоритмов планирования потоков, по своему решающих каждую из приведенных выше задач. Алгоритмы планирования могут преследовать различные цели и обеспечивать разное качество мультипрограммирования.

В большинстве операционных систем универсального назначения планирование осуществляется *динамически* (on-line), т. е. решения принимаются во время работы системы на основе анализа текущей ситуации. ОС работает в условиях неопределенности – потоки и процессы появляются в случайные моменты времени и также непредсказуемо завершаются. Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации и не используют никаких предположений о мультипрограммной смеси. Для того чтобы оперативно найти в условиях такой неопределенности оптимальный в некотором смысле порядок выполнения задач, операционная система должна затрачивать значительные усилия.

Другой тип планирования – статический – может быть использован в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например, в системах реального времени. *Планировщик* называется *статическим* (или *предварительным*), если он принимает решения о планировании не во время работы системы, а заранее (off-line).

Результатом работы статического планировщика является таблица, называемая расписанием, в которой указывается, какому потоку/процессу, когда и на какое время должен быть предоставлен процессор. Для построения расписания планировщику нужны как можно более полные предварительные знания о характеристиках набора задач.

Готовое расписание может использоваться операционной системой для переключения потоков и процессов. При этом накладные расходы ОС на исполнение расписания сводятся лишь к диспетчеризации потоков/процессов.

Диспетчеризация заключается в реализации найденного в результате планирования (динамического или статического) решения, то есть в переключении процессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст, с тем чтобы впоследствии использовать эту информацию для последующего возобновления выполнения данного потока. Контекст отражает, во-первых, состояние аппаратуры компьютера в момент прерывания потока: значение счетчика команд, содержимое регистров общего назначения, режим работы процессора, флаги, маски прерываний и другие параметры. Во-вторых, контекст включает параметры операционной среды, а именно ссылки на открытые файлы, данные о незавершенных операциях ввода-вывода, коды ошибок выполняемых данным потоком системных вызовов и т. д.

Диспетчеризация сводится к следующему [1]:

- 1) сохранение контекста текущего потока, который требуется сменить;
- 2) загрузка контекста нового потока, выбранного в результате планирования;
- 3) запуск нового потока на выполнение.

Поскольку операция переключения контекстов существенно влияет на производительность вычислительной системы, программные модули ОС выполняют диспетчеризацию потоков совместно с аппаратными средствами процессора.

Для реализации алгоритма планирования ОС должна получать управление всякий раз, когда в системе происходит событие, требующее перераспределения процессорного времени. К таким событиям могут быть отнесены следующие [1]:

- 1) прерывание от таймера, сигнализирующее, что время, отведенное активной задаче на выполнение, закончилось – планировщик переводит задачу в состояние готовности и выполняет перепланирование;

2) активная задача выполнила системный вызов, связанный с запросом на ввод-вывод или на доступ к ресурсу, который в настоящий момент занят, – планировщик переводит задачу в состояние ожидания и выполняет перепланирование;

3) активная задача выполнила системный вызов, связанный с освобождением ресурса, – планировщик проверяет, не ожидает ли этот ресурс какая-либо задача;

4) внешнее (аппаратное) прерывание, которое сигнализирует о завершении периферийным устройством операции ввода-вывода, переводит соответствующую задачу в очередь готовых, и выполняется планирование;

5) внутреннее прерывание сигнализирует об ошибке, которая произошла в результате выполнения активной задачи – планировщик снимает задачу и выполняет перепланирование.

При возникновении каждого из этих событий планировщик выполняет просмотр очередей и решает вопрос о том, какая задача будет выполняться следующей. Помимо указанных, существует и ряд других событий (часто связанных с системными вызовами), требующих перепланировки. Например, запросы приложений и пользователей на создание новой задачи или повышение приоритета уже существующей задачи создают новую ситуацию, которая требует пересмотра очередей и, возможно, переключения процессора.

В системах реального времени для отработки статического расписания планировщик активизируется по прерываниям от таймера. Эти прерывания пронизывают всю временную ось, возникая через короткие постоянные интервалы времени. После каждого прерывания планировщик просматривает расписание и проверяет, не пора ли переключить задачи. Кроме прерываний от таймера в системах реального времени перепланирование задач может происходить по прерываниям от внешних устройств – различного вида датчиков и исполнительных механизмов.

1.1.9 Алгоритмы планирования, основанные на квантовании

В основе многих вытесняющих алгоритмов планирования лежит концепция *квантования* [1]. В соответствии с этой концепцией каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени – *квант*. Смена активного потока происходит, если:

- 1) поток завершился и покинул систему;
- 2) произошла ошибка;
- 3) поток перешел в состояние ожидания;
- 4) исчерпан квант процессорного времени, отведенный данному потоку.

Поток, который исчерпал свой квант, переводится в состояние готовности и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый поток из

очереди готовых. Граф состояний потока, изображенный на рисунке 5, соответствует алгоритму планирования, основанному на квантовании.



Рисунок 5 – Граф состояний потока в системе с квантованием [1]

Кванты, выделяемые потокам, могут быть одинаковыми для всех потоков или различными. Рассмотрим, например, случай, когда всем потокам предоставляются кванты одинаковой длины q (рисунок 6). Если в системе имеется n потоков, то время, которое поток проводит в ожидании следующего кванта, можно грубо оценить как $q(n - 1)$. Чем больше потоков в системе, тем больше время ожидания и тем меньше возможности вести одновременную интерактивную работу нескольким пользователям. Но если величина кванта выбрана очень небольшой, то значение произведения $q(n - 1)$ все равно будет достаточно мало для того, чтобы пользователь не ощущал дискомфорта от присутствия в системе других пользователей. Типичное значение кванта в системах разделения времени составляет десятки миллисекунд.

Если квант короткий, то суммарное время, которое проводит поток в ожидании процессора, прямо пропорционально времени, требуемому для его выполнения (т. е. времени, которое потребовалось бы для выполнения этого потока при монопольном использовании вычислительной системы). Чем больше квант, тем выше вероятность того, что потоки завершатся в результате первого же цикла выполнения, и тем менее явной становится зависимость времени ожидания потоков от их времени выполнения. При достаточно большом кванте алгоритм квантования вырождается в алгоритм последовательной обработки, присущий однопользовательным системам, при котором время ожидания задачи в очереди вообще никак не зависит от ее длительности.

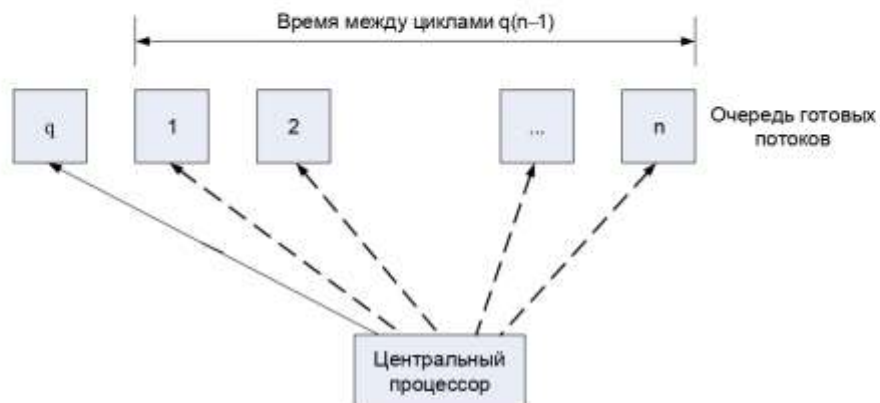


Рисунок 6 – Иллюстрация расчета времени ожидания в очереди

Кванты, выделяемые одному потоку, могут быть фиксированной величины, а могут и изменяться в разные периоды жизни потока. Такой подход позволяет уменьшить накладные расходы на переключение задач в том случае, когда сразу несколько задач выполняют длительные вычисления.

Потоки получают для выполнения квант времени, но некоторые из них используют его не полностью, например, из-за необходимости выполнить ввод или вывод данных. Алгоритм планирования может исправить эту «несправедливость». В качестве компенсации за неиспользованные полностью кванты потоки получают привилегии при последующем обслуживании. Для этого планировщик создает две очереди готовых потоков (рисунок 7).



Рисунок 7 – Квантование с предпочтением потоков, интенсивно обращающихся к вводу-выводу [1]

Очередь 1 образована потоками, которые пришли в состояние готовности в результате исчерпания кванта времени, а очередь 2 – потоками, у которых завершилась операция ввода-вывода. При выборе потока для выполнения на процессоре прежде всего просматривается вторая очередь, и только если она пуста, квант выделяется потоку из первой очереди.

Многозадачные ОС теряют некоторое количество процессорного времени для выполнения вспомогательных работ во время переключения контекстов задач. При этом запоминаются и восстанавливаются регистры, флаги и указатели стека, а также проверяется статус задач для передачи управления. Затраты на эти вспомогательные действия не зависят от величины кванта времени, поэтому чем больше квант, тем меньше суммарные накладные расходы, связанные с переключением потоков [1].

1.1.10 Алгоритмы планирования, основанные на приоритетах

Другой важной концепцией, лежащей в основе многих вытесняющих алгоритмов планирования, является *приоритетное обслуживание* [1]. Приоритетное

обслуживание предполагает наличие у потоков приоритета, на основании которого определяется порядок их выполнения. *Приоритет* – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в частности процессорного времени; чем выше приоритет, тем выше привилегии и тем меньше времени будет проводить поток в очередях [1].

Приоритет может выражаться целым или дробным, положительным или отрицательным значением. В некоторых ОС принято, что приоритет потока тем выше, чем больше (в арифметическом смысле) число, обозначающее приоритет. В других системах, наоборот, чем меньше число, тем выше приоритет.

В большинстве операционных систем, поддерживающих потоки, приоритет потока непосредственно связан с приоритетом процесса, в рамках которого выполняется данный поток. Приоритет процесса назначается операционной системой при его создании. Значение приоритета включается в описатель процесса и используется при назначении приоритета потокам этого процесса. При назначении приоритета вновь созданному процессу ОС учитывает, является этот процесс системным или прикладным, каков статус пользователя, запустившего процесс, было ли явное указание пользователя на присвоение процессу определенного уровня приоритета.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменение приоритета могут происходить по инициативе самого потока или по инициативе пользователя. Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются *динамическими* в отличие от неизменяемых, *фиксированных*, приоритетов [1].

От того, какие приоритеты назначены потокам, существенно зависит эффективность работы всей вычислительной системы. В большинстве же случаев ОС присваивает приоритеты потокам по умолчанию.

В качестве примера рассмотрим схему назначения приоритетов потокам, принятую в операционной системе Windows NT (рисунок 8). В системе определено 32 уровня приоритетов и два класса потоков – потоки реального времени и потоки с переменными приоритетами. Диапазон от 1 до 15 включительно отведен для потоков с переменными приоритетами, а от 16 до 31 – для более критичных ко времени потоков реального времени (приоритет 0 зарезервирован для системных целей).

При создании процесса поток в зависимости от класса получает по умолчанию базовый приоритет в верхней или нижней части диапазона. Базовый приоритет процесса в дальнейшем может быть повышен или понижен операционной системой.

В Windows NT с течением времени приоритет потока, относящегося к классу потоков с переменными приоритетами, может отклоняться от базового приори-

тета потока. ОС может повышать приоритет потока (который в этом случае называется динамическим) в тех случаях, когда поток не полностью использовал отведенный ему квант, или понижать приоритет, если квант был использован полностью. ОС наращивает приоритет дифференцированно в зависимости от того, какого типа событие не дало потоку полностью использовать квант. В частности, ОС повышает приоритет в большей степени q потоков, которые ожидают ввода с клавиатуры (интерактивным приложениям) и в меньшей степени – потокам, выполняющим дисковые операции. Именно на основе динамических приоритетов осуществляется планирование потоков.

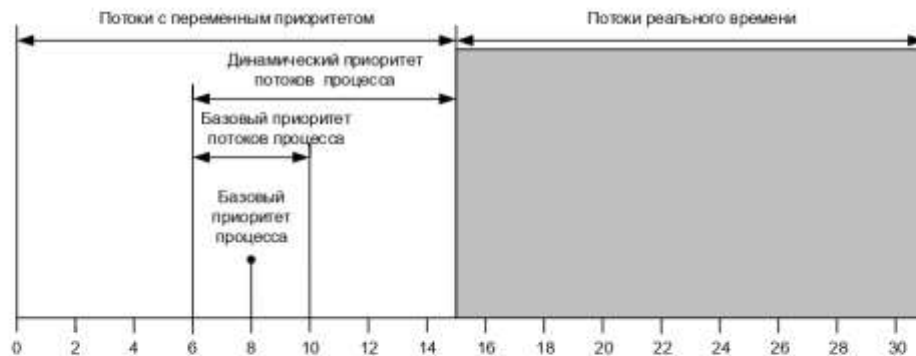


Рисунок 8 – Схема назначения приоритетов в Windows NT [1]

Существуют две разновидности приоритетного планирования: обслуживание с относительными приоритетами и обслуживание с абсолютными приоритетами.

В обоих случаях выбор потока на выполнение из очереди готовых осуществляется одинаково: выбирается поток, имеющий наивысший приоритет. Однако проблема определения момента смены активного потока решается по-разному. В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (или же произойдет ошибка, или поток завершится). На рисунке 9 показан граф состояний потока в системе с относительными приоритетами.

В системах с абсолютными приоритетами выполнение активного потока прерывается, кроме указанных выше причин, еще при одном условии: если в очереди готовых потоков появился поток, приоритет которого выше приоритета активного потока. В этом случае прерванный поток переходит в состояние готовности (рисунок 10).

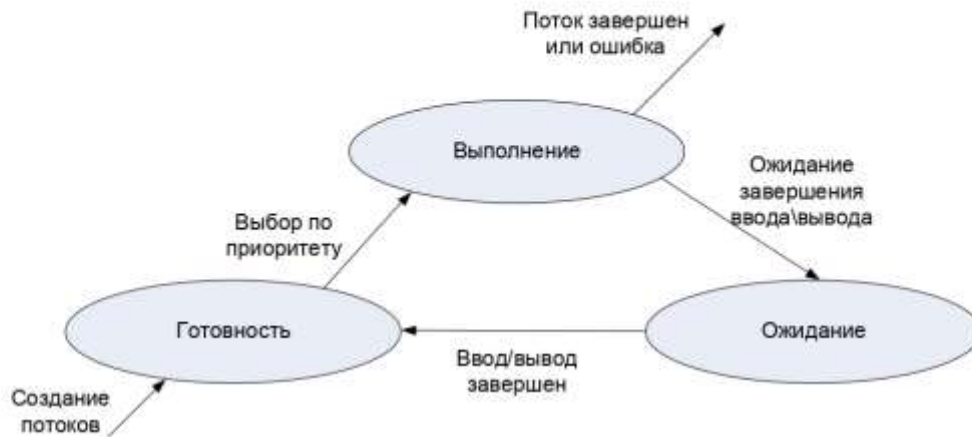


Рисунок 9 – Графы состояний потоков в системах с относительными приоритетами [1]

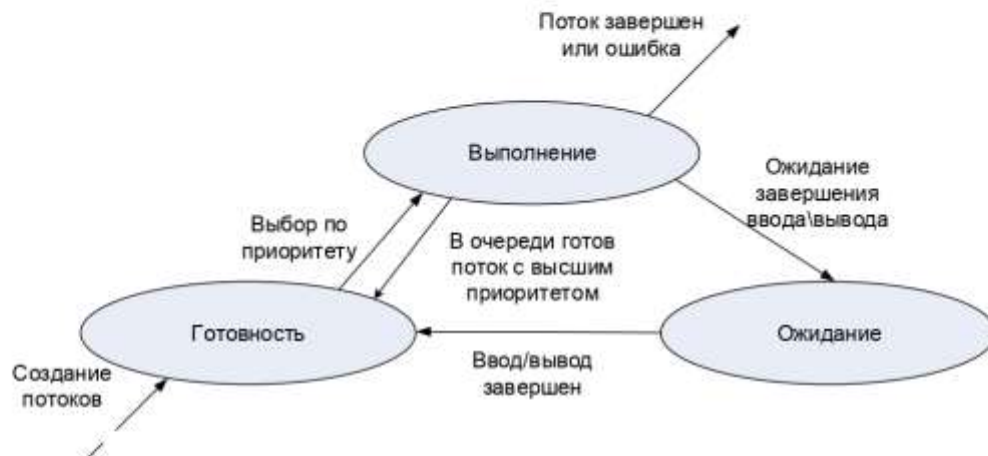


Рисунок 10 – Графы состояний потоков в системах с абсолютными приоритетами [1]

В системах, в которых планирование осуществляется на основе относительных приоритетов, минимизируются затраты на переключения процессора с одной работы на другую. С другой стороны, здесь могут возникать ситуации, когда одна задача занимает процессор долгое время. Ясно, что для систем разделения времени и реального времени такая дисциплина обслуживания не подходит: интер-

активное приложение может ждать своей очереди часами, пока вычислительной задаче не потребуются ввод-вывод.

В системах с абсолютными приоритетами время ожидания потока в очередях может быть сведено к минимуму, если ему назначить самый высокий приоритет. Такой поток будет вытеснять из процессора все остальные потоки (кроме потоков, имеющих такой же наивысший приоритет). Это делает планирование на основе абсолютных приоритетов подходящим для систем управления объектами, в которых важна быстрая реакция на событие.

1.1.11 Смешанные алгоритмы планирования

Во многих операционных системах алгоритмы планирования построены с использованием как концепции квантования, так и приоритетов [1]. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора потока из очереди готовых определяется приоритетами потоков. Именно так реализовано планирование в системе Windows NT, в которой квантование сочетается с динамическими абсолютными приоритетами. На выполнение выбирается готовый поток с наивысшим приоритетом. Ему выделяется квант времени. Если во время выполнения в очереди готовых появляется поток с более высоким приоритетом, то он вытесняет выполняемый поток. Вытесненный поток возвращается в очередь готовых, причем он становится впереди всех остальных потоков имеющих такой же приоритет.

1.1.12 Задача синхронизации потоков

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия – Inter Process Communications (IPC) [1]. Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов или потоков связано с их *синхронизацией*, которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и ОС должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые операционной системой в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества операционной системы они не могут приостановить друг друга или оповестить о произошедшем событии. Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально специализированными. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильному решению задачи или даже к краху системы. Рассмотрим, например, задачу ведения базы данных клиентов некоторого предприятия (рисунки 11). Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля Заказ и Оплата. Программа, ведущая базу данных, оформлена как единый процесс, имеющий несколько потоков, в том числе поток А, который заносит в базу данных информацию о заказах, поступивших от клиентов, и поток В, который фиксирует в базе данных сведения об оплате клиентами выставленных счетов.

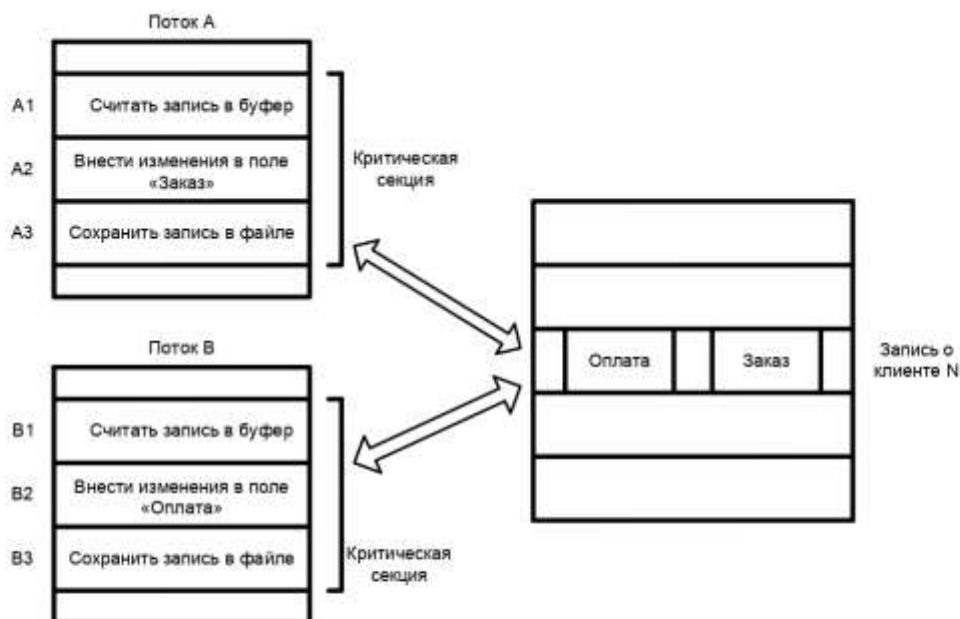


Рисунок 11 – Возникновение гонок при доступе к разделяемым данным

Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага [1]:

- 1) считать из файла базы данных в буфер запись о клиенте с заданным идентификатором;
- 2) внести новое значение в поле Заказ (для потока А) или Оплата (для потока В);
- 3) вернуть модифицированную запись в файл базы данных.

Обозначим соответствующие шаги для потока А как А1, А2 и А3, а для потока В – как В1, В2 и В3. Предположим, что в некоторый момент поток А обновляет поле Заказ записи о клиенте N. Для этого он считывает эту запись в свой буфер (шаг А1), модифицирует значение поля Заказ (шаг А2), но внести запись в базу данных (шаг А3) не успевает, так как его выполнение прерывается, например, вследствие завершения кванта времени.

Предположим также, что потоку В также потребовалось внести сведения об оплате относительно того же клиента N. Когда подходит очередь потока В, он успевает считать запись в свой буфер (шаг В1) и выполнить обновление поля Оплата (шаг В2), а затем прерывается.

Когда в очередной раз управление будет передано потоку А, то он, продолжая свою работу, запишет запись о клиенте N с модифицированным полем Заказ в базу данных (шаг А3). После прерывания потока А и активизации потока В последний запишет в базу данных поверх только что обновленной записи о клиенте N свой вариант записи, в которой обновлено значение поля Оплата. Таким образом, в базе данных будут зафиксированы сведения о том, что клиент N произвел оплату, но информация о его заказе окажется потерянной (рисунок 12, а).

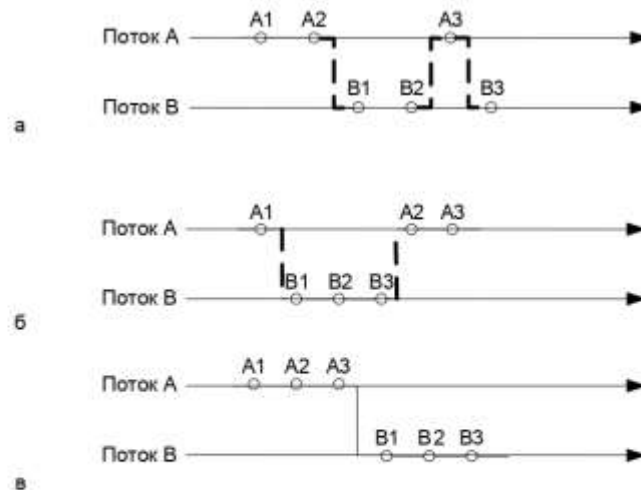


Рисунок 12 – Влияние относительных скоростей потоков на результат решения задачи

Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций. Так, в предыдущем примере можно представить и другое развитие событий: могла быть потеряна информация не о заказе, а об оплате (рисунок 12, б) или, напротив, все исправления были успешно внесены (рисунок 12, в). Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей. Ситуации, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются *гонками*.

1.1.13 Понятие критической секции

Важным понятием синхронизации потоков являются понятия «*критическая секция*» и «*блокирующие переменные*». *Критическая секция* – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным *критическим данным*, при несогласованном изменении которых могут возникнуть нежелательные эффекты. В разных потоках критическая секция состоит в общем случае из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. Этот прием называют *взаимным исключением*. Операционная система использует разные способы реализации взаимного исключения.

Самый простой и в то же время самый неэффективный способ обеспечения взаимного исключения состоит в том, что операционная система позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управление системой пользовательскому потоку – он может надолго занять процессор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Блокирующие переменные – переменные, к которым все потоки процесса имеют прямой доступ. С ними программист работает, не обращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рисунке 13 показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную $F(D)$.

Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D . Если переменная $F(D)$ установлена в 0, то данные заняты и проверка циклически повторяется. Если же данные свободны ($F(D) = 1$), то значение переменной $F(D)$ устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными D , значение переменной $F(D)$ снова устанавливается равным 1.

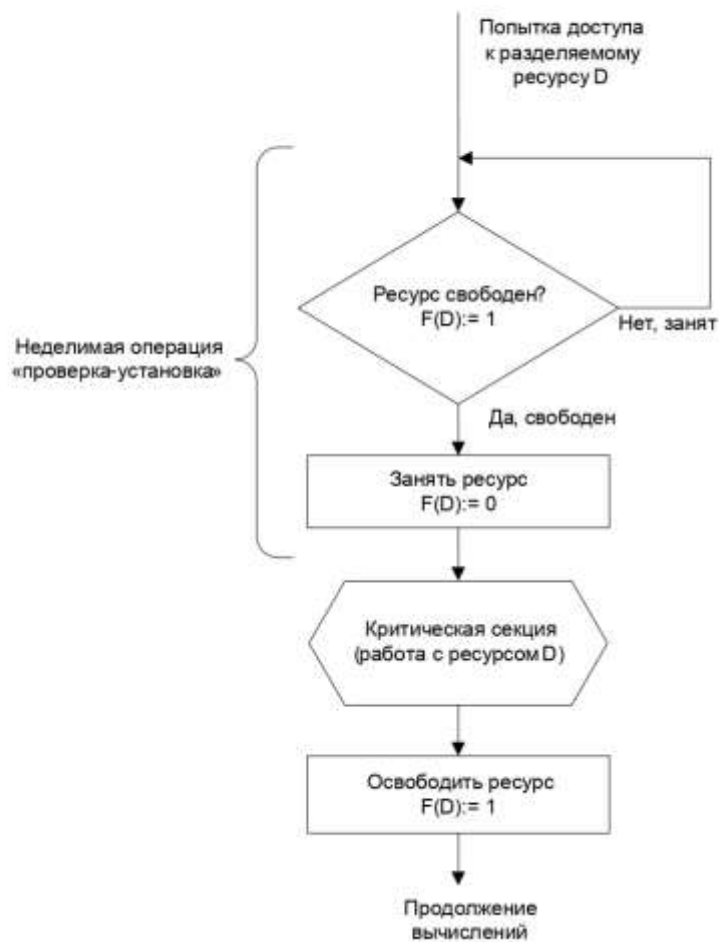


Рисунок 13 – Реализация критических секций с использованием блокирующих переменных [1]

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида.

Потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Однако нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Этот способ ограничения прерываний имеет существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ

к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока.

Итак, на рисунке 14 показано, как с помощью функций для работы с критической секцией реализовано взаимное исключение в операционной системе Windows NT.

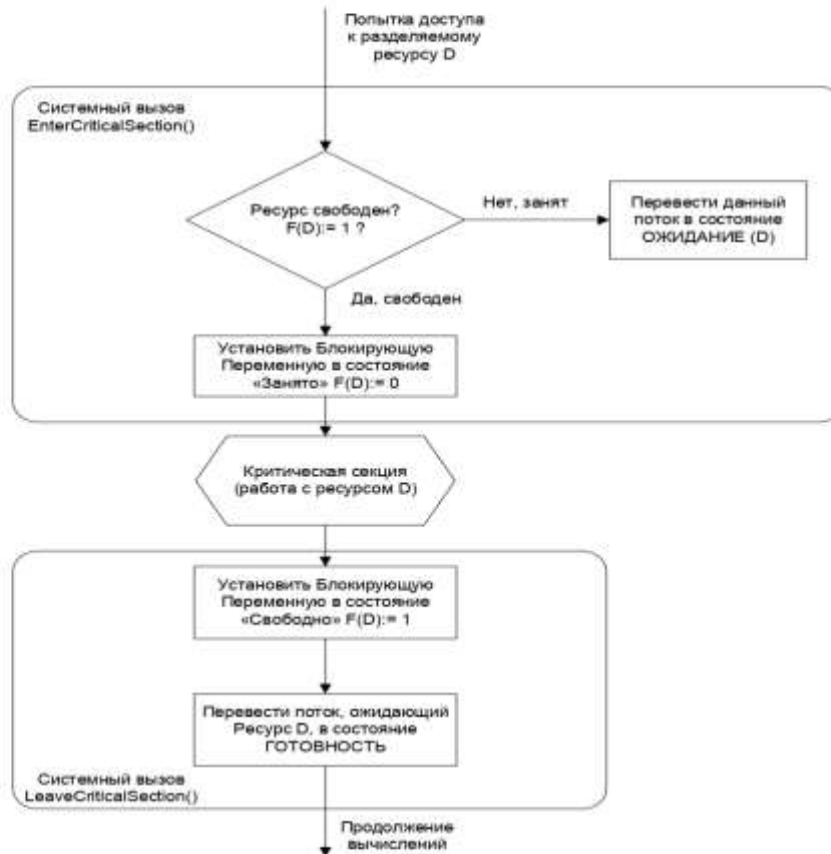


Рисунок 14 – Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее [1]

Перед тем как начать изменение критических данных, поток выполняет системный вызов EnterCriticalSection(). В рамках этого вызова сначала выполняется проверка блокирующей переменной, отражающей состояние критического ресурса. Если системный вызов определил, что ресурс занят ($F(D) = 0$), он переводит поток в состояние ожидания (D) и делает отметку о том,

что данный поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции должен выполнить системную функцию `LeaveCriticalSection()`, в результате чего блокирующая переменная принимает значение, соответствующее свободному состоянию ресурса ($F(D) = 1$), а операционная система просматривает очередь ожидающих этот ресурс потоков и переводит первый поток из очереди в состояние готовности.

Таким образом, исключается непроизводительная потеря процессорного времени на циклическую проверку освобождения занятого ресурса.

1.1.14 Синхронизирующие объекты ОС

Рассмотренные выше механизмы синхронизации, основанные на использовании глобальных переменных процесса, обладают существенным недостатком – они не подходят для синхронизации потоков разных процессов. В таких случаях операционная система должна предоставлять потокам системные объекты синхронизации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов ОС являются системные семафоры, мьютексы, события, таймеры и др.

Кроме того, для синхронизации могут быть использованы такие «обычные» объекты ОС, как файлы, процессы и потоки. Все эти объекты могут находиться в двух состояниях: сигнальном и несигнальном – свободном. Для каждого объекта смысл, вкладываемый в понятие «сигнальное состояние», зависит от типа объекта. Так, например, поток переходит в сигнальное состояние тогда, когда он завершается. Процесс переходит в сигнальное состояние тогда, когда завершаются все его потоки. Файл переходит в сигнальное состояние в том случае, когда завершается операция ввода-вывода для этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вызовов. Приостановка и активизация потоков осуществляются в зависимости от состояния синхронизирующих объектов ОС.

Потоки с помощью специального системного вызова сообщают операционной системе о том, что они хотят синхронизировать свое выполнение с состоянием некоторого объекта. Будем далее называть этот системный вызов `Wait(X)`, где `X` – указатель на объект синхронизации. Системный вызов, с помощью которого поток может перевести объект синхронизации в сигнальное состояние, назовем `Set(X)`.

Поток, выполнивший системный вызов `Wait(X)`, переводится операционной системой в состояние ожидания до тех пор, пока объект `X` не перейдет в сигнальное состояние. Примерами системных вызовов типа `Wait()` и `Set()` являются вызовы `WaitForSingleObject()` и `SetEvent()` в Windows NT.

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких. Поток может в качестве аргумента системного вызова Wait() указать также максимальное время, которое он будет ожидать перехода объекта в сигнальное состояние, после чего ОС должна его активизировать в любом случае.

Синхронизация тесно связана с планированием потоков. Во-первых, любое обращение потока с системным вызовом Wait(X) влечет за собой действия в подсистеме планирования – этот поток снимается с выполнения и помещается в очередь ожидающих потоков, а из очереди готовых потоков выбирается и активизируется новый поток. Во-вторых, при переходе объекта в сигнальное состояние (в результате выполнения некоторого потока – либо системного, либо прикладного) ожидающий этот объект поток (или потоки) переводится в очередь готовых к выполнению потоков. В обоих случаях осуществляется перепланирование потоков.

Однако круг событий, с которыми потоку может потребоваться синхронизировать свое выполнение, отнюдь не исчерпывается завершением потока, процесса или операции ввода-вывода. Поэтому в ОС, как правило, имеются и другие, более универсальные объекты синхронизации, такие как *событие (event)*, *мьютекс (mutex)*, *системный семафор* и др.

Семафоры – переменные, которые могут принимать целые неотрицательные значения.

Для работы с семафорами вводятся два примитива, традиционно обозначаемые, как P и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом:

1) V(S): переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции;

2) P(S): уменьшение S на 1, если это возможно. Если S=0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и P недопустимы.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую называют двоичным семафором.

Рассмотрим использование семафоров на примере взаимодействия двух выполняющихся в режиме мультипрограммирования потоков, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула. Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. В общем случае поток-писатель и поток-читатель могут иметь различные

скорости и обращаться к буферному пулу с переменной интенсивностью. Для правильной совместной работы поток-писатель должен приостанавливаться, когда все буферы оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, поток-читатель должен приостанавливаться, когда все буферы пусты, и активизироваться при появлении хотя бы одной записи.

Введем два семафора: e – число пустых буферов и f – число заполненных буферов, причем в исходном состоянии $e = N$, а $f = 0$. Тогда работа потоков с общим буферным пулом может быть описана следующим образом (рисунок 15).

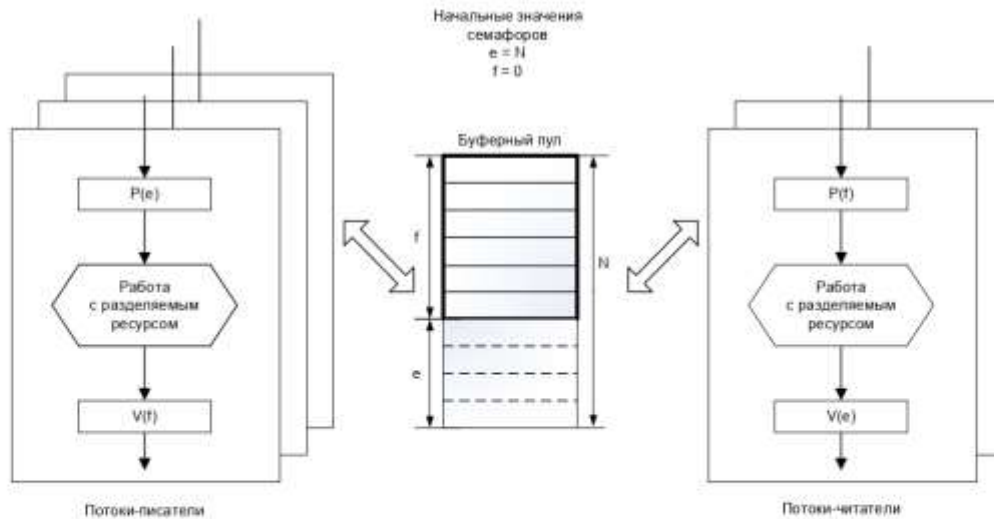


Рисунок 15 – Использование семафоров для синхронизации потоков [1]

Поток-писатель прежде всего выполняет операцию $P(e)$, с помощью которой он проверяет, имеются ли в буферном пуле незаполненные буферы. В соответствии с семантикой операции P , если семафор e равен 0 (т. е. свободных буферов в данный момент нет), то поток-писатель переходит в состояние ожидания. Если же значением e является положительное число, то он уменьшает число свободных буферов, записывает данные в очередной свободный буфер и после этого наращивает число занятых буферов операцией $V(f)$. Поток-читатель действует аналогичным образом с той разницей, что он начинает работу с проверки наличия заполненных буферов, а после чтения данных наращивает количество свободных буферов.

В данном случае предпочтительнее использовать семафоры вместо блокирующих переменных. Действительно, критическим ресурсом здесь является бу-

ферный пул, который может быть представлен как набор идентичных ресурсов – отдельных буферов. Семафор допускает к разделяемому пулу ресурсов заданное количество потоков. Так, в примере с буферным пулом могут работать максимум N потоков, часть из которых может быть «писателями», а часть – «читателями».

Таким образом, семафоры позволяют эффективно решать задачу синхронизации доступа к ресурсным пулам.

Семафор может использоваться и в качестве блокирующей переменной. В рассмотренном выше примере, для того чтобы исключить коллизии при работе с разделяемой областью памяти, будем считать, что запись в буфер и считывание из буфера являются критическими секциями. Взаимное исключение будем обеспечивать с помощью двоичного семафора (рисунок 16).

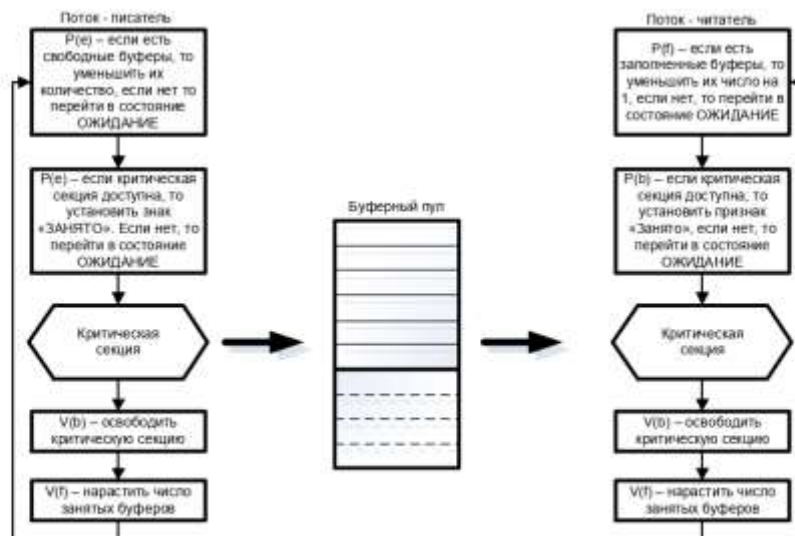


Рисунок 16 – Использование двоичного семафора [1]

Оба потока после проверки доступности буферов должны выполнить проверку доступности критической секции.

Мьютекс, как и семафор, обычно используется для управления доступом к данным.

В отличие от объектов-потоков, объектов-процессов и объектов-файлов, которые при переходе в сигнальное состояние переводят в состояние готовности все потоки, ожидающие этого события, объект-мьютекс «освобождает» из очереди ожидающих только один поток.

Работа мьютекса хорошо поясняется в терминах «владения». Пусть поток, пытаясь получить доступ к критическим данным, выполнил системный вызов `Wait(X)`, где `X` – указатель на мьютекс. Если мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работу с критическими данными, он «отдает» мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс свободен и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одновременно мьютекс переходит в несигнальное состояние.

Объект-событие (в данном случае слово «событие» используется в узком смысле, как обозначение конкретного вида объектов синхронизации) обычно используется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. Пусть, например, в некотором приложении работа организована таким образом, что один поток читает данные из файла в буфер памяти, а другие потоки обрабатывают эти данные, затем первый поток считывает новую порцию данных, а другие потоки снова ее обрабатывают и так далее. В начале работы первый поток устанавливает объект-событие в несигнальное состояние. Все остальные потоки выполнили вызов `Wait(X)`, где `X` – указатель события, и находятся в приостановленном состоянии, ожидая наступления этого события. Как только буфер заполняется, первый поток сообщает об этом операционной системе, выполняя вызов `Set(X)`. Операционная система просматривает очередь ожидающих потоков и активизирует все потоки, которые ждут этого события.

1.1.15 Потоки в Windows. Функция потока. Создание потока

Функция потока [2]

В первичном потоке функцией потока является `main`, `wmain`, `WinMain` или `wWinMain`. Если создать вторичный поток, в нем тоже будет входная функция примерно следующего вида:

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
    DWORD rtwResult = 0;
    ....
    return(dwResult);
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент поток остановится и память, отведенная под его стек, будет освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра «процесс», он может жить гораздо дольше, чем сопоставленный с ним поток.

Создание потока [2]

При вызове функции **CreateProcess** появляется первичный поток процесса. Для создания дополнительных потоков нужно вызывать из первичного потока функцию **CreateThread**:

```
HANDLE CreateThread(  
LPSECURITY_ATTRIBUTES psa,  
DWORD cbStack,  
LPTHREAD_START_ROUTINE pfnStartAddr,  
LPVOID pvParam,  
DWORD tdwCreate,  
LPDWORD pdwThreadId);
```

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке.

Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

Параметр *psa* является указателем на структуру **SECURITY_ATTRIBUTES**. Если необходимо, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию, в этом параметре передается **NULL**. А чтобы дочерние процессы смогли наследовать описатель этого объекта, необходимо определить структуру **SECURITY_ATTRIBUTES** и инициализировать ее элемент *hInheritHandle* значением **TRUE**.

Параметр *cbStack*. Этот параметр определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Функция **CreateProcess**, запуская приложение, вызывает **CreateThread**, и эта функция инициализирует первичный поток процесса. При этом **CreateProcess** заносит в параметр *cbStack* значение, хранящееся в самом исполняемом файле.

Параметры `pfnStartAddr` и `pvParam`. Параметр `pfnStartAddr` определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр `pvParam` идентичен параметру `pvParam` функции потока. `CreateThread` лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией.

Параметр `fdwCreate`. Этот параметр определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений. 0 (исполнение потока начинается немедленно) или `CREATE_SUSPENDED`. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг `CREATE_SUSPENDED` позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код.

Параметр `pdwThreadId`. Последний параметр функции – это адрес переменной типа `DWORD`, в которой функция возвращает идентификатор, присвоенный системой новому потоку.

Завершение потока [2]

Поток можно завершить четырьмя способами:

- 1) функция потока возвращает управление (рекомендуемый способ);
- 2) поток самоуничтожается вызовом функции `ExitThread` (нежелательный способ);
- 3) один из потоков данного или стороннего процесса вызывает функцию `TerminateThread` (нежелательный способ);
- 4) завершается процесс, содержащий данный поток (нежелательно).

Возврат управления функцией потока. Функцию рекомендуется проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших потоку.

Функция `ExitThread`. Поток можно завершить принудительно, вызвав:

```
VOID ExitThread(DWORD dwExitCode);
```

При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но `C/C++` – ресурсы (например, объекты, созданные из `C++`-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию `ExitThread`.

В параметр *dwExitCode* помещается значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, потому что после ее вызова поток перестает существовать.

Функция *TerminateThread*. Вызов этой функции также завершает поток:

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode);
```

В отличие от *ExitThread*, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread*. В параметр *dwExitCode* помещается значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей уменьшится на 1.

Приостановка и возобновление потоков [2]

В объекте ядра «поток» имеется переменная – счетчик числа простоев данного потока. При вызове **CreateProcess** или **CreateThread** он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, **CreateProcess** или **CreateThread** проверяет, не передан ли ей флаг **CREATE_SUSPENDED**, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется.

Создав поток в приостановленном состоянии, можно настроить некоторые его свойства. Закончив настройку, необходимо разрешить выполнение потока. Для этого используется функция **ResumeThread**:

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов **ResumeThread** прошел успешно, эта функция возвращает предыдущее значение счетчика простоев данного потока; в ином случае – 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен три раза, то и возобновлен он должен быть тоже три раза. Выполнение потока можно приостановить не только при его создании с флагом **CREATE_SUSPENDED**, но и вызовом **SuspendThread**:

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока.

Функция Sleep [2]

Поток может приказать системе не выделять ему процессорное время на определенный период, вызвав:

```
VOID Sleep (DWORD dwMilliseconds);
```

Эта функция приостанавливает поток на *dwMilliseconds*.

Особенности:

1) вызывая Sleep, поток добровольно отказывается от остатка выделенного ему кванта времени;

2) система прекращает выделять потоку процессорное время на период, примерно равный заданному;

3) можно вызвать Sleep и передать в *dwMilliseconds* значение INFINITE, вообще запретив планировать поток;

4) можно вызвать Sleep и передать в *dwMilliseconds* нулевое значение – т. е. отказаться от остатка своего кванта времени и заставить систему подключить к процессору другой поток.

При загрузке системы создается особый поток – поток обнуления страниц (zero page thread), которому присваивается нулевой уровень приоритета. Ни один поток, кроме этого, не может иметь нулевой уровень приоритета. Он обнуляет свободные страницы в оперативной памяти при отсутствии других потоков, требующих внимания со стороны системы.

1.1.16 Синхронизация потоков в WINDOWS

Синхронизация с помощью критических секций [2–4]

В Windows для реализации критических секций используется структура CRITICAL_SECTION, работа с которой осуществляется исключительно через функции Windows, передавая им адрес соответствующего экземпляра этой структуры. Структура инициализируется с помощью следующей функции:

```
VOID InitializeCriticalSection(  
                                LPCRITICAL_SECTION pcs);
```

Эта функция инициализирует элементы структуры CRITICAL_SECTION, на которую указывает параметр *pcs*. Поскольку вся работа данной функции заключается в инициализации нескольких переменных-членов, она не дает сбоев и поэтому ничего не возвращает (void). InitializeCriticalSection должна быть вызвана до того, как один из потоков обратится к EnterCriticalSection. Участок кода, работающий с разделяемым ресурсом, начинается вызовом:

```
VOID EnterCriticalSection(LPCRITICAL_SECTION pcs);
```

EnterCriticalSection — исследует значения элементов структуры **CRITICAL_SECTION**. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. **EnterCriticalSection** выполняет следующие действия.

Если ресурс свободен, **EnterCriticalSection** модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу.

Если ресурс уже захвачен вызывающим потоком, **EnterCriticalSection** обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление. Такая ситуация бывает нечасто — лишь тогда, когда поток два раза подряд вызывает **EnterCriticalSection** без промежуточного вызова **LeaveCriticalSection**.

Если значения элементов структуры указывают на то, что ресурс занят другим потоком, **EnterCriticalSection** переводит вызывающий поток в режим ожидания. Система запоминает, что данный поток хочет получить доступ к ресурсу, и — как только поток, занимавший этот ресурс, вызывает **LeaveCriticalSection** — вновь начинает выделять данному потоку процессорное время.

В конце участка кода, использующего разделяемый ресурс, должен присутствовать вызов:

```
VOID LeaveCriticalSection(LPCRITICAL_SECTION pcs);
```

Эта функция просматривает элементы структуры **CRITICAL_SECTION** и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, **LeaveCriticalSection** ничего не делает и просто возвращает управление.

Если значение счетчика достигло 0, **LeaveCriticalSection** сначала выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове **EnterCriticalSection**. Если есть хотя бы один такой поток, функция настраивает значения элементов структуры, чтобы они сигнализировали о занятости ресурса, и отдает его одному из ждущих потоков (поток выбирается «по справедливости»). Если же ресурс никому не нужен, **LeaveCriticalSection** сбрасывает элементы структуры.

Пример реализации критической секции:

```
HANDLE htl;  
CRITICAL_SECTION cs;  
int x;  
PVOID pvParam = (PVOID) &x;
```

```

//--Инициализация критической секции-----
InitializeCriticalSection ( (LPCRITICAL_SECTION) & cs );

//--Функция потока ThreadFunc1-----
DWORD WINAPI ThreadFunc1 (PVOID pvParam) {
while (true) {
EnterCriticalSection ( (LPCRITICAL_SECTION) & cs );
...
// --Код, который работает с общим ресурсом
...
//--Выход из критической секции
LeaveCriticalSection ( (LPCRITICAL_SECTION) & cs );
Sleep(100);
}
}
...
//---удаление критической секции-----
DeleteCriticalSection (&cs);

```

Синхронизация потоков с помощью объектов ядра. Функции ожидания объектов ядра [5,6]

В Windows есть специальные функции ожидания синхронизирующих объектов ядра (wait-функции), которые позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется `WaitForSingleObject`:

```

DWORD WaitForSingleObject ( HANDLE hHandle, DWORD dwMil-
liseconds );

```

Когда поток вызывает эту функцию, параметр `hHandle` идентифицирует объект ядра, поддерживающий состояние «свободен-занят». Второй параметр, `dwMilliseconds`, указывает сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем `hProcess`.

```

WaitForSingleObject(hProcess, INFINITE);

```

В данном случае константа `INFINITE` подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность. Именно эта константа

обычно и передается функции `WaitForSingleObject`, но можно указать любое значение в миллисекундах.

Функция `WaitForMultipleObjects` позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects( DWORD dwCount,  
CONST HANDLE* phHandles, BOOL bWaitAll, DWORD dwMilli-  
seconds);
```

Параметр *dwCount* определяет количество интересующих объектов ядра. Его значение должно быть в пределах от 1 до `MAXIMUM_WAIT_OBJECTS` (в заголовочных файлах Windows оно определено как 64). Параметр *phHandles* – это указатель на массив описателей объектов ядра.

`WaitForMultipleObjects` приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр *bWaitAll* определяет, что именно требуется от функции. Если он равен `TRUE`, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Параметр *dwMilliseconds* идентичен одноименному параметру функции `WaitForSingleObject`.

Возвращаемое значение функции `WaitForMultipleObjects` сообщает, почему возобновилось выполнение вызвавшего ее потока.

Синхронизация потоков с помощью мьютексов [4]

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом `CreateMutex`:

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES psa, BOOL  
bInitialOwner, LPCTSTR pszName);
```

Любой процесс может получить свой («процессозависимый») описатель существующего объекта «мьютекс», вызвав `OpenMutex`:

```
HANDLE OpenMutex( DWORD fdwAccess, BOOL bInheritHandle,  
PCTSTR pszName);
```

Параметр *fInitialOwner* определяет начальное состояние мьютекса. Если в нем передается `FALSE` (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается `TRUE`, идентификатор потока, принадлежащий мьютексу, приравнивается иден-

тификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции `ReleaseMutex`:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать `ReleaseMutex` столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-нибудь другие потоки. Если да, система «по-честному» выберет один из ждущих потоков и передаст ему во владение объект-мьютекс.

Остальные объекты ядра отличаются от объекта-мьютекса тем, что способны запоминать, какому потоку они принадлежат. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции `ReleaseMutex`, то она, проверив идентификаторы потоков и обнаружив их несовпадение, ничего делать не станет, а просто вернет `FALSE`.

Пример использования мьютекса приведен ниже.

```
HANDLE htl, hMutex;  
int x;  
BOOL TermFlag1=FALSE;
```

```
//--Создание мьютекса с помощью функции CreateMutex ----
```

```
hMutex=CreateMutex(  
    NULL,  
    FALSE,  
    NULL);
```

```
//--Функция потока ThreadFunc1-----
```

```
DWORD WINAPI ThreadFunc1() {  
    while (!TermFlag1) {  
        WaitForSingleObject(  
            hMutex,  
            INFINITE);
```

```
    ...
```

```
// --Код, который работает с общим ресурсом
```

```
    ...
```

```

ReleaseMutex(hMutex);
Sleep(1500);
}
return(0);
}
}

//--- Создание потока -----
DWORD dwThread;
TermFlag1=FALSE;
    ht1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
ThreadFunc1, (PVOID)&x, 0, &dwThread);
...

//---Закрытие мьютекса-----
CloseHandle(hMutex);

```

Синхронизация потоков с помощью семафоров [4]

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Для семафоров определены следующие правила:

- 1) когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- 2) если этот счетчик равен 0, семафор занят;
- 3) система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- 4) счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Объект ядра «семафор» создается вызовом **CreateSemaphore**:

```

HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTE psa,
LONG lInitialCount, LONG lMAXimumCount, LPCTSTR
pszName);

```

Любой процесс может получить свой («процессозависимый») описатель существующего объекта «семафор», вызвав **OpenSemaphore**.

```
HANDLE OpenSemaphore( DWORD fdwAccess,
  BOOL bInheritHandle, LPCTSTR pszName);
```

Параметр *IMAXimumCount* сообщает системе максимальное число ресурсов, обрабатываемое приложением. Поскольку это 32-битное значение со знаком, предельное число ресурсов может достигать 2 147 483 647. Параметр *InitialCount* указывает, сколько из этих ресурсов доступны изначально (на данный момент).

Поток получает доступ к ресурсу, вызывая одну из Wait-функций и передавая ей описатель семафора, который охраняет этот ресурс. Wait-функция проверяет у семафора счетчик текущего числа ресурсов, если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток.

Если Wait-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время.

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию *ReleaseSemaphore*:

```
BOOL ReleaseSemaphore( HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount);
```

ReleaseSemaphore просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но это вовсе не обязательно. Функция возвращает исходное значение счетчика ресурсов в **lpPreviousCount*.

Пример реализации семафоров:

```
HANDLE h1, hSemaphore;
BOOL TermFlag1=FALSE;
```

```
//--Создание семафора с помощью функции CreateSemaphore
hSemaphore=CreateSemaphore(
```

```
    NULL,
    1,
    1,
    NULL);
```

```
...
```

```
//--Функция потока ThreadFunc1-----
```

```
DWORD WINAPI ThreadFunc1() {
```

```

while (!TermFlag1) {
WaitForSingleObject(
    hSemaphore,
    INFINITE);

...
//--Код, который работает с общим ресурсом
...
ReleaseSemaphore (hSemaphore, 1, NULL);
Sleep(1500);
}
return(0);
}

...

//--- Создание потока -----
    DWORD dwThread;
int x;
    TermFlag1=FALSE;
    ht1=CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
ThreadFunc1, (PVOID)&x, 0, &dwThread);

//---Закрытие семафора-----
CloseHandle (hSemaphore);

```

1.2 Порядок выполнения работы

- 1 Изучить теоретические основы реализации механизма многопоточности в операционных системах и набор системных функций для работы с потоками в Windows.
- 2 Получить задание у преподавателя.
- 3 В среде Builder C++ или MS Visual Studio создать приложение и продемонстрировать правильность его работы преподавателю.
- 4 Оформить отчет по лабораторной работе.

1.3 Содержание отчета

- 1 Название работы и цель работы.
- 2 Исходное задание.
- 3 Листинг разработанной программы.
- 4 Выводы.

1.4 Контрольные вопросы

- 1 Как приостановить/возобновить выполнение потока системными средствами Windows?
- 2 Какие системные ресурсы сопоставляются с объектом «поток»?
- 3 В чем отличие объекта «мьютекс» от объекта «семафор» применительно к задаче синхронизации потоков?
- 4 Можно ли, используя критическую секцию, синхронизировать потоки для разных процессов ?
- 5 Как принудительно завершить поток системными средствами Windows?
- 6 Какие существуют способы синхронизации потоков в современных ОС?
- 7 В чем функциональное отличие объектов «поток» и «процесс»?
- 8 В каких трех основных состояниях может находиться поток?
- 9 Можно ли влиять на время, которое отдельно взятый поток находится в очереди за ресурсами?

ПРАКТИЧЕСКАЯ РАБОТА № 7.

Тема: Реализация SLAM-алгоритма

Цель: Ознакомление и реализация SLAM-алгоритма

Разработайте визуальный алгоритм SLAM Используя нереальную симуляцию Engine

В этом примере показано, как разработать визуальный алгоритм Одновременной локализации и картографии (SLAM) с помощью данных изображения, полученных из Нереальной среды симуляции Engine®.

Все примеры и подробное описание представлено по ссылке:

<https://docs.exponenta.ru/R2021b/driving/ug/develop-visual-slam-algorithm-using-unreal-engine-simulation.html>

Визуальный SLAM является процессом вычисления положения и ориентации камеры относительно ее среды, одновременно сопоставляя среду. Разработка визуального алгоритма SLAM и оценка его эффективности в различных условиях являются сложной задачей. Одна из самых сложных задач генерирует основную истину датчика камеры, особенно в наружных средах. Использование симуляции позволяет тестировать под множеством сценариев и настроек камеры при обеспечении точной основной истины.

Этот пример демонстрирует использование Нереальной симуляции Engine, чтобы разработать визуальный алгоритм SLAM или для монокуляра или для стереофотоаппарата в сценарии парковки. Для получения дополнительной информации о реализации визуальных трубопроводов SLAM, смотрите [Монокулярный Визуальный Одновременный пример Локализации и картографии](#) и [Визуальный Одновременный пример Локализации и картографии Стерео](#).

Настройте сценарий в среде симуляции

Используйте Симуляцию 3D Блок Configuration Сцены, чтобы настроить среду симуляции. Выберите встроенную Большую сцену Парковки, которая содержит несколько припаркованных транспортных средств. Визуальный

алгоритм SLAM совпадает с функциями через последовательные изображения. Чтобы увеличить число потенциальных соответствий функции, можно использовать Припаркованную подсистему Транспортных средств, чтобы добавить более припаркованные транспортные средства в сцену. Чтобы задать положения парковки транспортных средств, используйте `helperAddParkedVehicle` функция. Если вы выбираете более естественную сцену, присутствие дополнительных транспортных средств не необходимо. Естественные сцены обычно имеют достаточно структуры и показывают разнообразие, подходящее для соответствия функции.

Можно последовать [Выбрать](#) примеру [Waypoints for Unreal Engine Simulation](#), чтобы в интерактивном режиме выбрать последовательность парковки местоположений. Можно использовать тот же подход, чтобы выбрать последовательность waypoints и сгенерировать ссылочную траекторию для автомобиля, оборудованного датчиком. Этот пример использует записанную ссылочную траекторию и припаркованные местоположения транспортного средства.

```
% Load reference path
data = load('parkingLotReferenceData.mat');

% Set reference trajectory of the ego vehicle
refPosesX = data.refPosesX;
refPosesY = data.refPosesY;
refPosesT = data.refPosesT;

% Set poses of the parked vehicles
parkedPoses = data.parkedPoses;

% Display the reference path and the parked vehicle locations
sceneName = 'LargeParkingLot';
hScene = figure;
helperShowSceneImage(sceneName);
hold on
plot(refPosesX(:,2), refPosesY(:,2), 'LineWidth', 2, 'DisplayName', 'Reference Path');
scatter(parkedPoses(:,1), parkedPoses(:,2), [], 'filled', 'DisplayName', 'Parked
Vehicles');
xlim([-60 40])
ylim([10 60])
hScene.Position = [100, 100, 1000, 500]; % Resize figure
legend
hold off
```

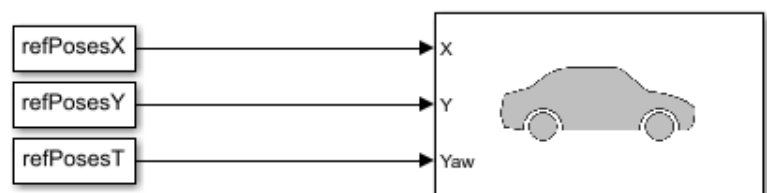
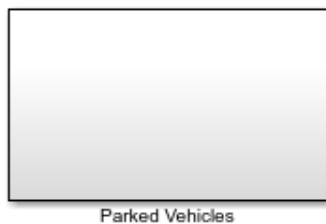
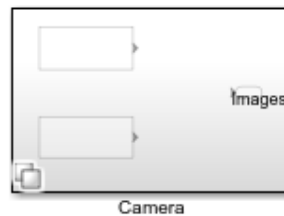
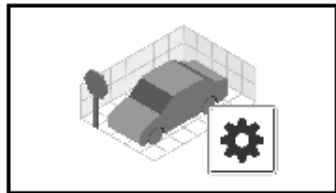


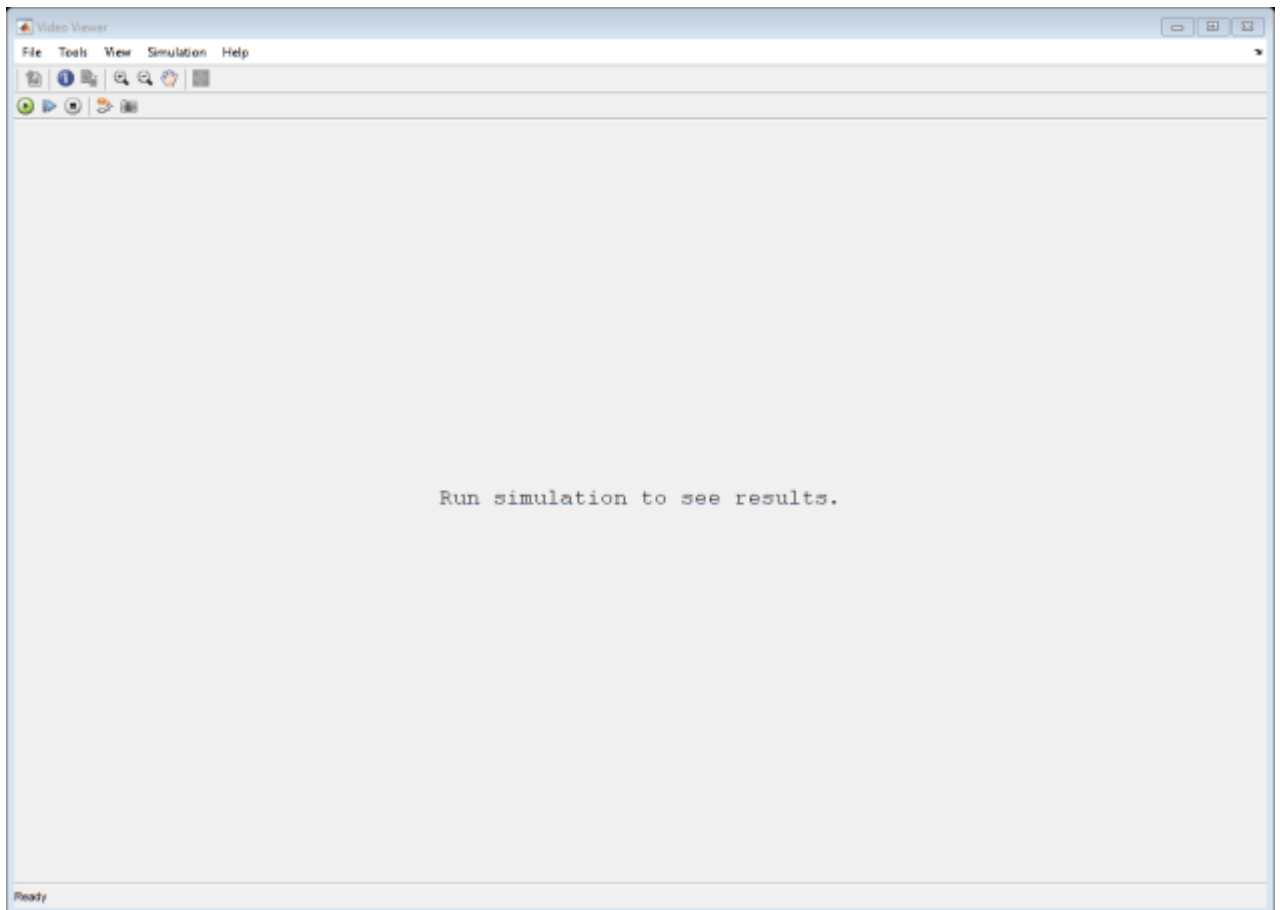
Откройте модель и добавьте припаркованные транспортные средства

```
modelName = 'GenerateImageDataOfParkingLot';
open_system(modelName);
snapnow;

helperAddParkedVehicles(modelName, parkedPoses);
```

Generate Image Data of Parking Lot





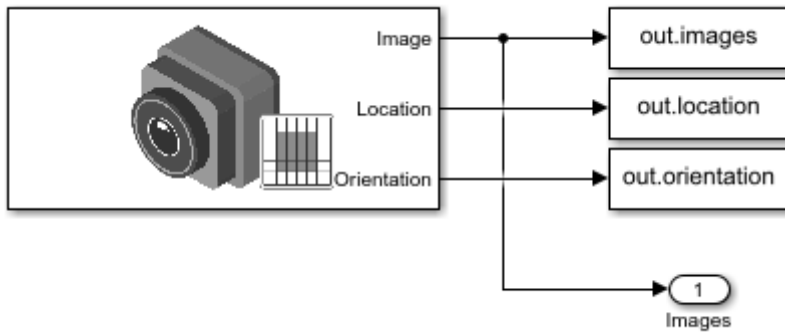
Настройте датчик автомобиля, оборудованного датчиком и камеры

Настройте автомобиль, оборудованный датчиком, проходящий заданный ссылочный путь при помощи Симуляции 3D Транспортное средство с блоком Ground Following. Подсистема Варианта Камеры содержит две настройки датчиков камеры: монокуляр и стерео. В обеих настройках камера смонтирована на центре крыши транспортного средства. Можно использовать [Camera Calibrator](#) или приложение [Stereo Camera Calibrator](#), чтобы оценить внутренние параметры фактической камеры, которую вы хотите симулировать. Этот пример показывает монокулярный рабочий процесс камеры, сначала сопровождаемый рабочим процессом стереофотоаппарата.

```
% Select monocular camera
useMonoCamera = 1;

% Inspect the monocular camera
open_system([modelName, '/Camera/Monocular']);
snapnow;

% Camera intrinsics
focalLength    = [700, 700]; % specified in units of pixels
principalPoint = [600, 180]; % in pixels [x, y]
imageSize      = [370, 1230]; % in pixels [mrows, ncols]
intrinsics      = cameraIntrinsics(focalLength, principalPoint, imageSize);
```



Визуализируйте и запишите данные о датчике

Запустите симуляцию, чтобы визуализировать и записать данные о датчике. Используйте блок Video Viewer, чтобы визуализировать изображение выход из датчика камеры. Используйте блок To Workspace, чтобы записать местоположение основной истины и ориентацию датчика камеры.

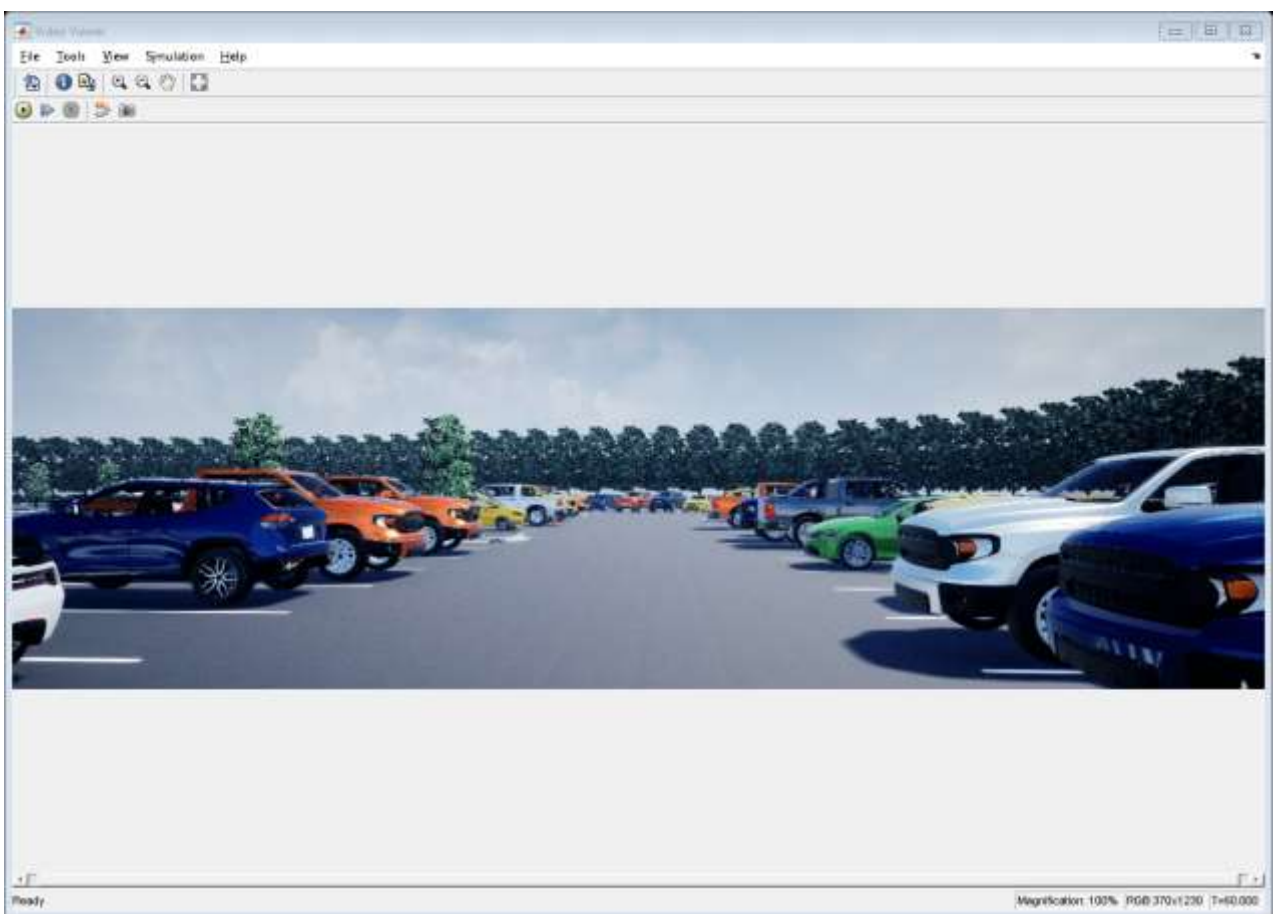
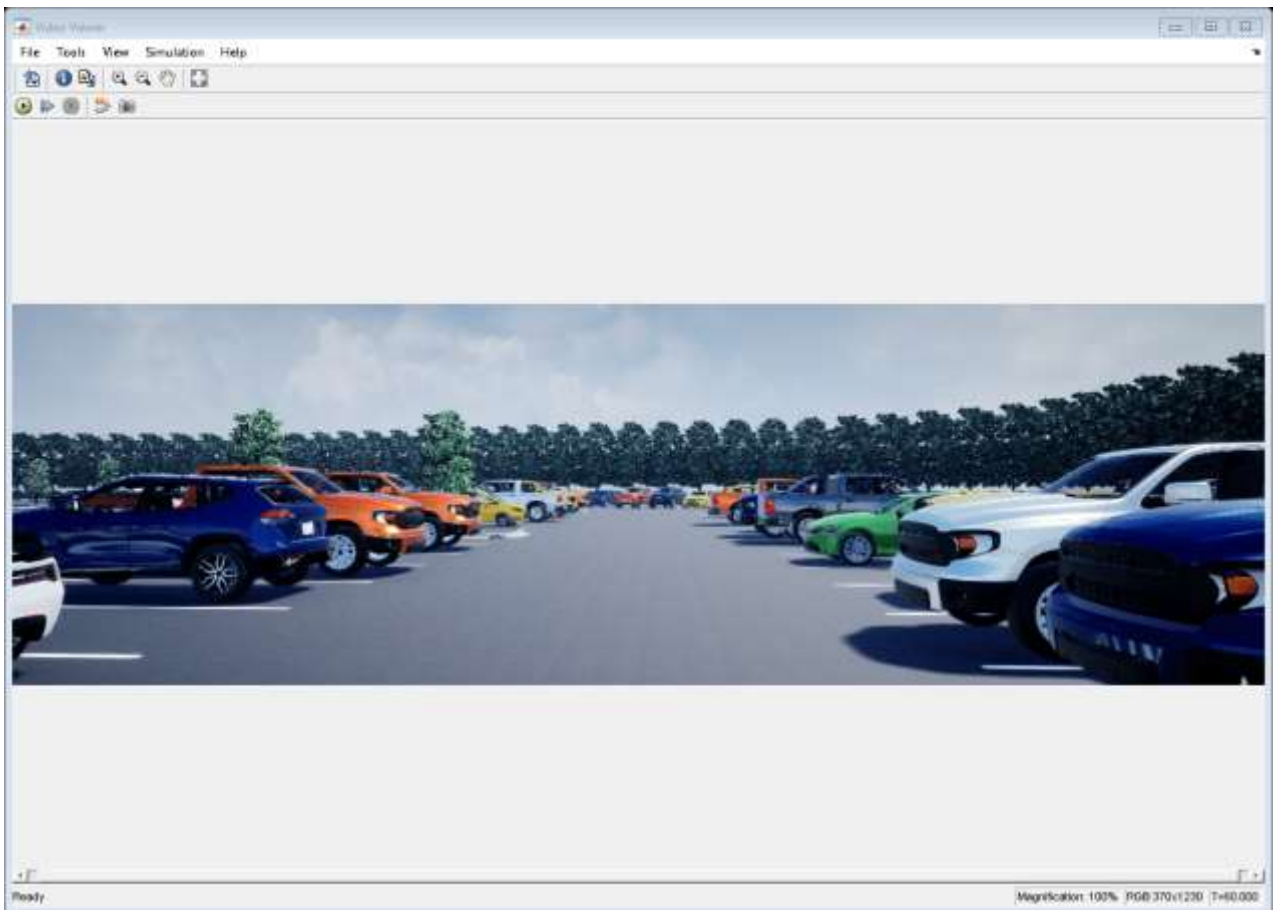
```
close(hScene)
```

```
if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + "
Windows" + char(174) + ".");
end
```

```
% Run simulation
simOut = sim(modelName);
snapnow;
```

```
% Extract camera images as an imageDatastore
imds = helperGetCameraImages(simOut);
```

```
% Extract ground truth as an array of rigid3d objects
gTruth = helperGetSensorGroundTruth(simOut);
```



Разработайте монокулярный визуальный алгоритм SLAM

Используя записанные данные

Используйте изображения, чтобы оценить монокулярный визуальный алгоритм SLAM. Функциональный `helperVisualSLAM` реализует монокулярный трубопровод ORB-SLAM:

Инициализация карты: ORB-SLAM запускается путем инициализации карты 3-D точек от двух изображений. Используйте `relativeCameraPose` чтобы вычислить относительное положение на основе 2D ORB показывающих соответствия и `triangulate` вычислить 3-D точки карты. Две системы координат хранятся в `imageviewset` в качестве ключевых кадров. 3-D точки карты и их соответствия к ключевым кадрам хранятся в `worldpointset` объект.

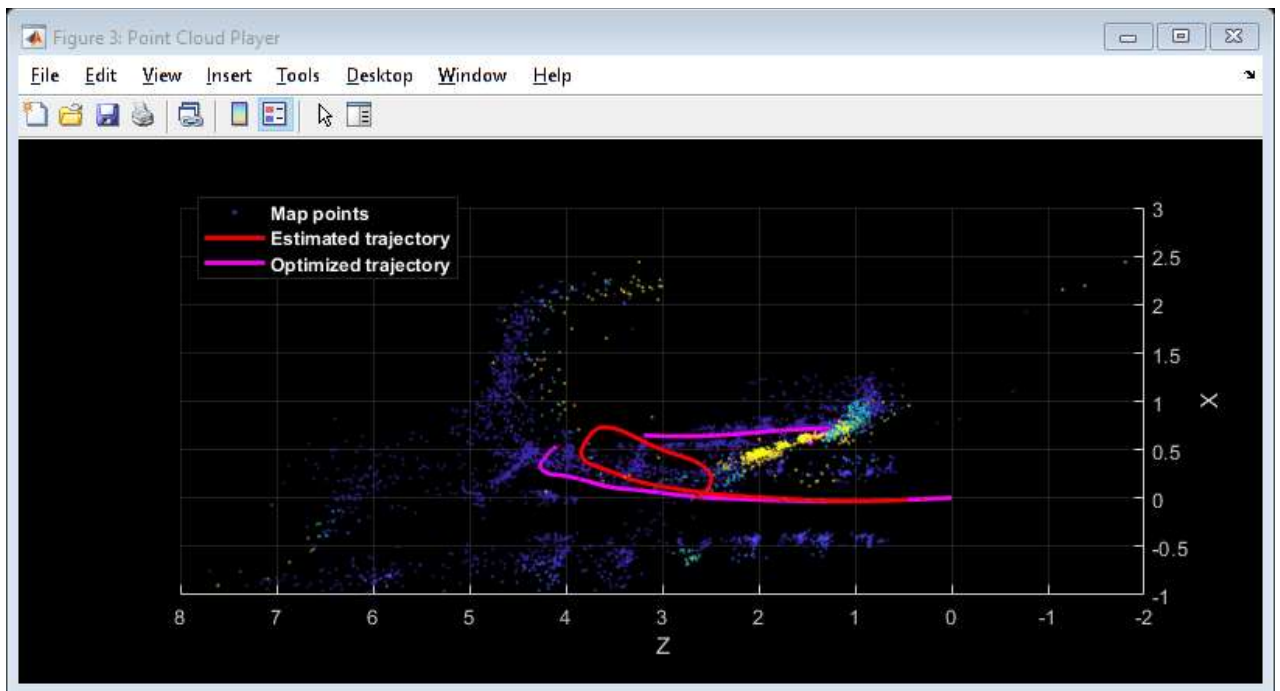
Отслеживание: Если карта инициализируется, для каждого нового изображения, функционального `helperTrackLastKeyFrame` оценивает положение камеры путем соответствия с функциями в текущей системе координат к функциям в последнем ключевом кадре. Функциональный `helperTrackLocalMap` совершенствовал предполагаемое положение камеры путем отслеживания локальной карты.

Локальное Отображение: текущая система координат используется, чтобы создать новые 3-D точки карты, если она идентифицирована как ключевой кадр. На данном этапе, `bundleAdjustment` используется, чтобы минимизировать ошибки перепроекции путем корректировки положения камеры и 3-D точек.

Заккрытие цикла: Циклы обнаруживаются для каждого ключевого кадра путем сравнения его против всех предыдущих ключевых кадров с помощью подхода набора признаков. Если закрытие цикла обнаруживается, график положения оптимизирован, чтобы совершенствовать положения камеры всех ключевых кадров с помощью `optimizePoseGraph` (Navigation Toolbox) функция.

Для получения дополнительной информации реализации алгоритма смотрите [Монокулярный Визуальный Одновременный пример Локализации и картографии](#).

```
[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAM(imds, intrinsics);  
Map initialized with frame 1 and frame 3  
Loop edge added between keyframe: 7 and 156
```



Оцените против основной истины

Можно оценить оптимизированную траекторию камеры против основной истины, полученной из симуляции. Поскольку изображения сгенерированы от монокулярной камеры, траектория камеры может только быть восстановлена до

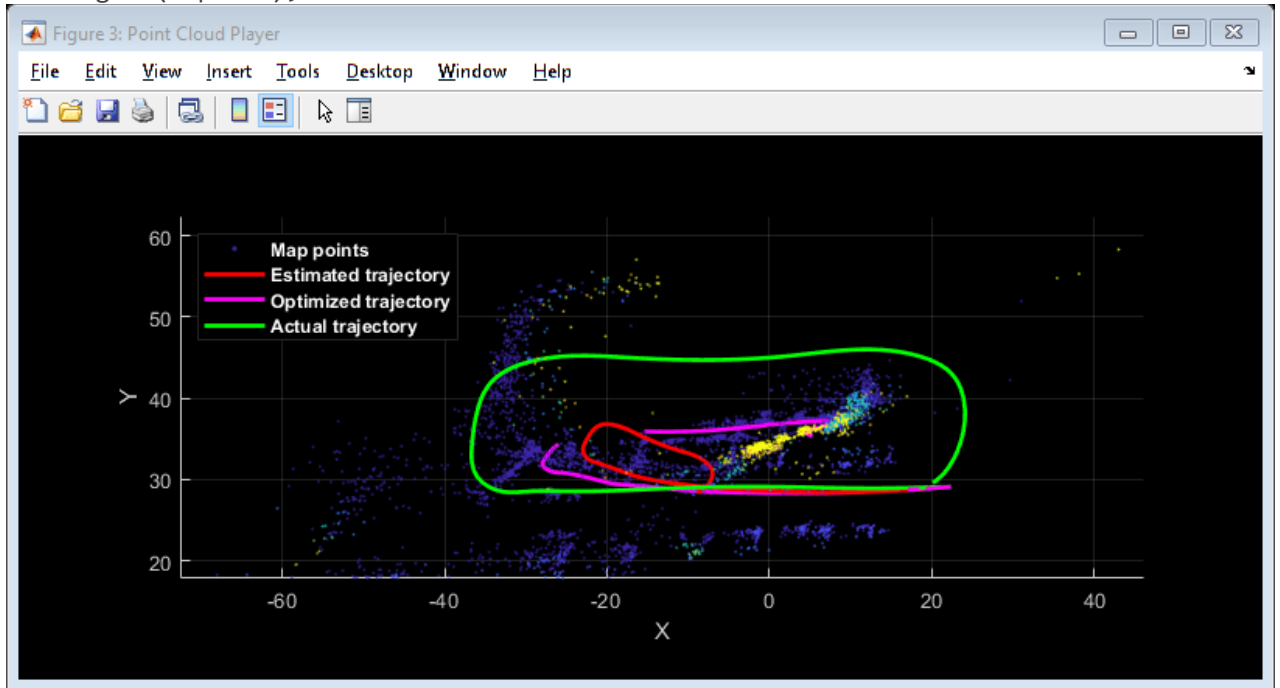
неизвестного масштабного коэффициента. Можно приблизительно вычислить масштабный коэффициент из основной истины, таким образом симулировав то, что вы обычно получали бы из внешнего датчика.

```
% Plot the camera ground truth trajectory
```

```
scaledTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx),  
optimizedPoses);
```

```
% Show legend
```

```
showLegend(mapPlot);
```



Можно также вычислить среднеквадратичную ошибку (RMSE) оценок траектории.

```
helperEstimateTrajectoryError(gTruth(addedFramesIdx), scaledTrajectory);
```

Absolute RMSE for key frame trajectory (m): 11.8389

Визуальный алгоритм SLAM стерео

В монокулярном визуальном алгоритме SLAM глубина не может быть точно определена с помощью одной камеры. Шкала карты и предполагаемой траектории неизвестна и дрейфует в зависимости от времени. Кроме того, потому что точки карты часто не могут триангулироваться от первой системы координат, загрузка системы требует, чтобы несколько представлений произвели первоначальную карту. Используя стереофотоаппарат решает эти задачи и предоставляет более надежное визуальное решение SLAM. Функциональный `helperVisualSLAMStereo` реализует визуальный трубопровод SLAM стерео. Основное отличие от монокулярного трубопровода на этапе инициализации карты, трубопровод стерео создает 3-D точки карты из

пары стереоизображений той же системы координат, вместо того, чтобы создать их из двух изображений различных систем координат. Для получения дополнительной информации реализации алгоритма смотрите [Визуальный Одновременный пример Локализации и картографии Стерео](#).

```
% Select stereo camera
```

```
useMonoCamera = 0;
```

```
% Inspect the stereo camera
```

```
open_system([modelName, '/Camera/Stereo']);
```

```
snapnow;
```

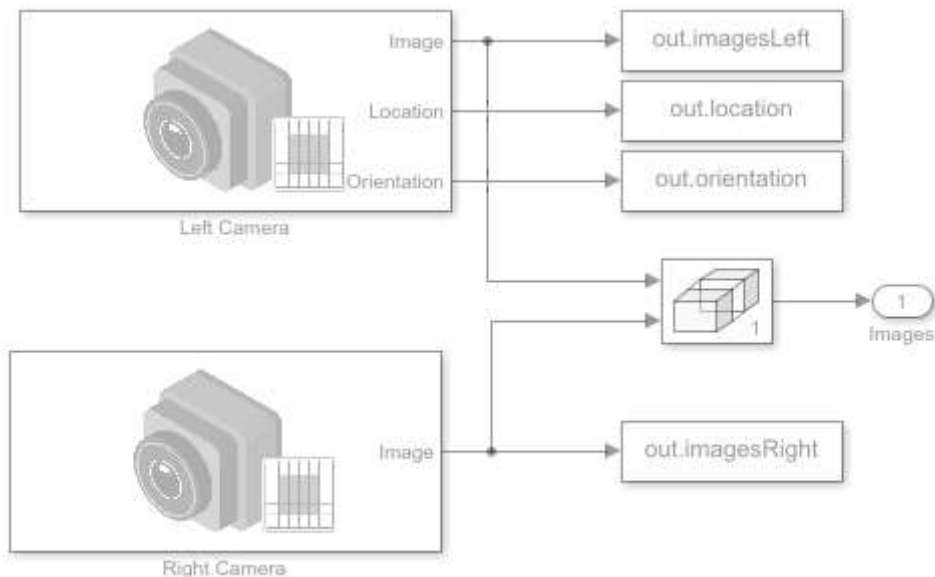
```
% Set stereo camera baseline
```

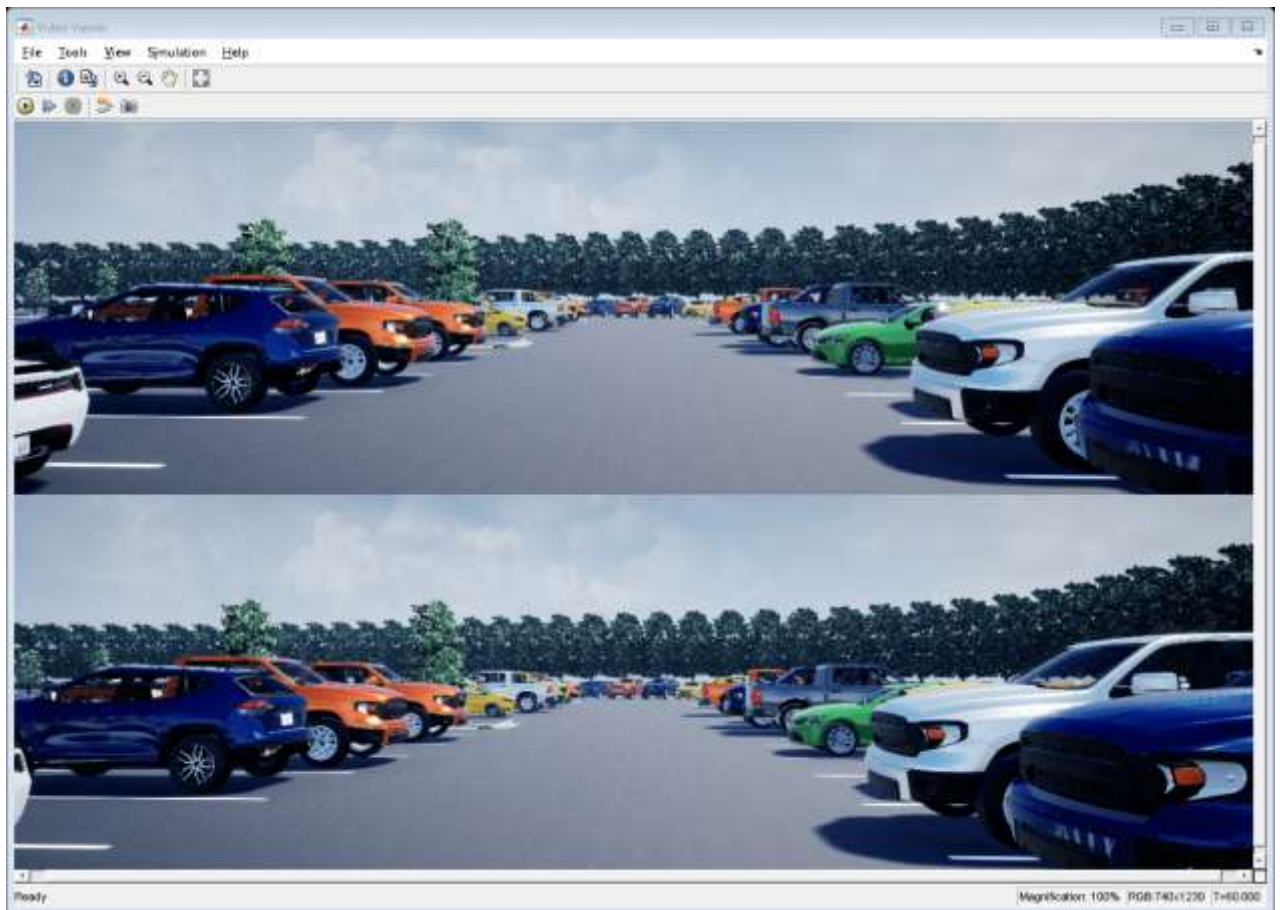
```
baseline = 0.5; % In meters
```

```
% Run simulation
```

```
simOut = sim(modelName);
```

```
snapnow;
```





Извлеките стереоизображения

```
[imdsLeft, imdsRight] = helperGetCameraImagesStereo(simOut);

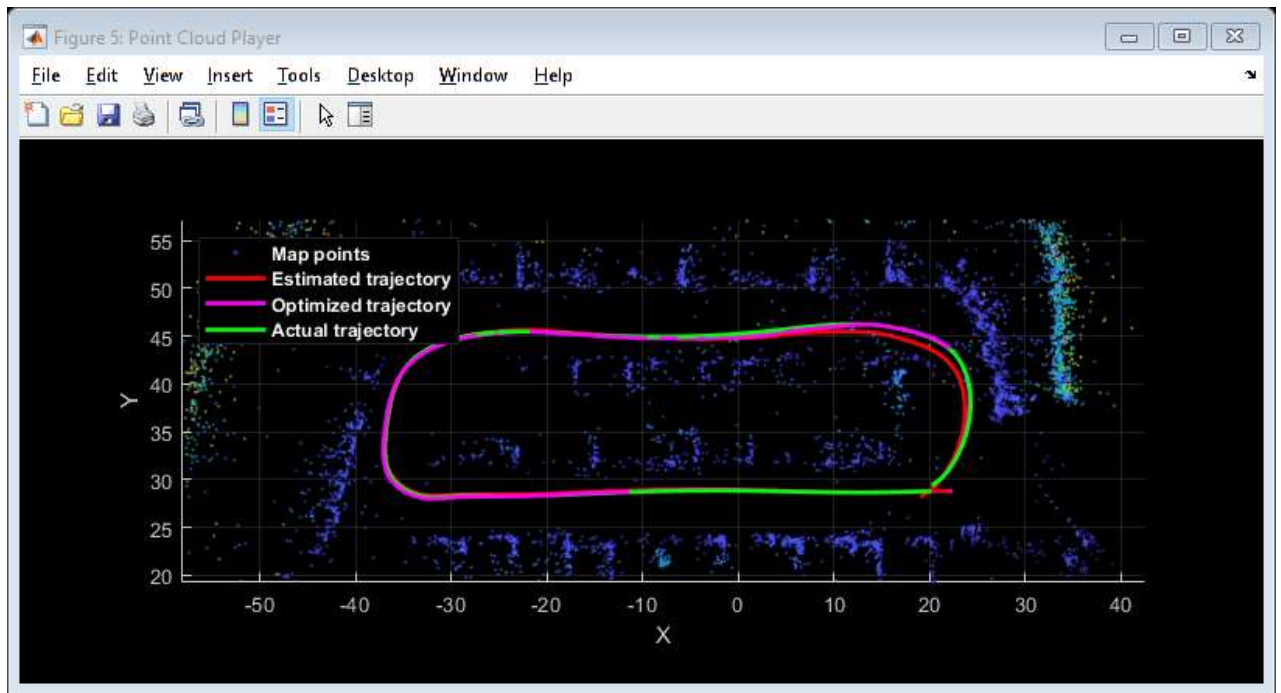
% Extract ground truth as an array of rigid3d objects
gTruth = helperGetSensorGroundTruth(simOut);
Запустите визуальный алгоритм SLAM стерео

[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAMStereo(imdsLeft,
imdsRight, intrinsics, baseline);

% Plot the camera ground truth trajectory
optimizedTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx));

% Show legend
showLegend(mapPlot);

% Calculate the root mean square error (RMSE) of trajectory estimates
helperEstimateTrajectoryError(gTruth(addedFramesIdx), optimizedTrajectory);
Loop edge added between keyframe: 5 and 145
Absolute RMSE for key frame trajectory (m): 0.25164
```



По сравнению с монокулярным визуальным алгоритмом SLAM визуальный алгоритм SLAM стерео производит более точную оценку траектории камеры.

Закройте модель и фигуры.

```
close_system(modelName, 0);
close all
```

Вспомогательные Функции

helperGetCameraImages Выводили камеру

```
function imds = helperGetCameraImages(simOut)
% Save image data to a temporary folder
dataFolder = fullfile(tempdir, 'parkingLotImages', filesep);
folderExists = exist(dataFolder, 'dir');
if ~folderExists
    mkdir(dataFolder);
end

files = dir(dataFolder);
if numel(files) < 3
    numFrames = numel(simOut.images.Time);
    for i = 3:numFrames % Ignore the first two frames
        img = squeeze(simOut.images.Data(:,:,i));
```

```

        imwrite(img, [dataFolder, sprintf('%04d', i-2), '.png'])
    end
end

% Create an imageDatastore object to store all the images
imds = imageDatastore(dataFolder);
end

helperGetCameraImagesStereo Выводили камеру

function [imdsLeft, imdsRight] = helperGetCameraImagesStereo(simOut)
% Save image data to a temporary folder
dataFolderLeft = fullfile(tempdir, 'parkingLotStereoImages', filesep, 'left',
filesep);
dataFolderRight = fullfile(tempdir, 'parkingLotStereoImages', filesep, 'right',
filesep);
folderExists = exist(dataFolderLeft, 'dir');
if ~folderExists
    mkdir(dataFolderLeft);
    mkdir(dataFolderRight);
end

files = dir(dataFolderLeft);
if numel(files) < 3
    numFrames = numel(simOut.imagesLeft.Time);
    for i = 3:numFrames % Ignore the first two frames
        imgLeft = squeeze(simOut.imagesLeft.Data(:,:,i));
        imwrite(imgLeft, [dataFolderLeft, sprintf('%04d', i-2), '.png'])

        imgRight = squeeze(simOut.imagesRight.Data(:,:,i));
        imwrite(imgRight, [dataFolderRight, sprintf('%04d', i-2), '.png'])
    end
end

% Use imageDatastore objects to store the stereo images
imdsLeft = imageDatastore(dataFolderLeft);
imdsRight = imageDatastore(dataFolderRight);
end

helperGetSensorGroundTruth Сохраняют основную истину датчика

function gTruth = helperGetSensorGroundTruth(simOut)
numFrames = numel(simOut.location.Time);
gTruth = repmat(rigid3d, numFrames-2, 1);
for i = 1:numFrames-2 % Ignore the first two frames
    gTruth(i).Translation = squeeze(simOut.location.Data(:, :, i+2));
    % Ignore the roll and the pitch rotations since the ground is flat
    yaw = double(simOut.orientation.Data(:, 3, i+2));
    gTruth(i).Rotation = [cos(yaw), sin(yaw), 0; ...
        -sin(yaw), cos(yaw), 0; ...
        0, 0, 1];
end
end

helperEstimateTrajectoryError Вычисляют ошибку отслеживания

function rmse = helperEstimateTrajectoryError(gTruth, scaledLocations)
gLocations = vertcat(gTruth.Translation);

rmse = sqrt(mean( sum((scaledLocations - gLocations).^2, 2) ));
disp(['Absolute RMSE for key frame trajectory (m): ', num2str(rmse)]);
end

```

ПРАКТИЧЕСКАЯ РАБОТА № 8.

Тема: Знакомство с архитектурой YOLO для детекции объектов.

Цель: ознакомление с методами, применяемыми для компьютерного зрения роботов.

Выполнение работы

Получите у преподавателя файл NNMA_YOLO_Lab. Запустите его в Jupiter notebook и продемонстрируйте результаты обработки видеофайла.

Работоспособный файл является отчетом по лабораторной работе.

ПРАКТИЧЕСКАЯ РАБОТА № 9.

Тема: Оптимизация алгоритма с помощью профайлера

Цель: Ознакомиться с инструментами профилирования программного кода.

При разработке приложений периодически у разработчиков возникает вопрос - по какой причине реализованный код работает медленно, и где теряются драгоценные время и ресурсы? Здесь на помощь приходят инструменты профилирования. В текущем посте рассмотрим возможности профилирования приложений реализованных на Python с помощью различных инструментов профилирования, а также рассмотрим средства позволяющие строить графическое представление результатов профилирования и создавать отчеты.

Профилирование как понятие

Для выявления мест где приложение растрчивает ресурсы нам потребуется следующая информация:

- порядок выполнения кода (какие функции в каком порядке вызываются);
- количество вызовов отдельных фрагментов кода (функций/процедур/методов и т.п.);
- время затрачиваемое на выполнение отдельного фрагмента кода (функции/метода/процедуры и т.п.);

Имея вышеуказанную информацию мы можем выявить узкие места в нашем проекте, и сконцентрировавшись на отдельном фрагменте кода выполнить его рефакторинг для повышения эффективности выполнения приложения в целом.

Профиль - набор статистических данных который описывает как часто и долго выполнялись различные части программы.

Инструменты профилирования Python

cProfile и profile

Python имеет встроенные инструменты для выполнения профилирования, которые находятся в модулях cProfile и profile. Данные модули позволяют

получить статистику, которая в дальнейшем может быть преобразована в отчет с помощью модуля `pstats` или других инструментов. Модуль `cProfile` реализован как Си расширение с невысоким оверхедом и рекомендован для использования как основной. Официальная документация [здесь](#): [The Python Profilers \(opens new window\)](#).

Создадим небольшое приложение Python в котором осуществляется вызов нескольких функций и вывод сообщений в консоль разместим код в модуле `test.py`

```
def print_message(message):
    print(message)

def range_output(external_step, internal_steps_number):
    for internal_step in range(internal_steps_number):
        print_message(f'[{external_step}: {internal_step}]')

def main():
    for i in range(3):
        range_output(external_step=i, internal_steps_number=2)

if __name__ == '__main__':
    main()
```

Выполним профилирование данного кода с помощью `cProfile`:

```
python -m cProfile test.py
[0: 0]
[0: 1]
[1: 0]
[1: 1]
[2: 0]
[2: 1]
      19 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1    0.000    0.000    0.000    0.000 test.py:10(main)
   1    0.000    0.000    0.000    0.000 test.py:3(<module>)
   6    0.000    0.000    0.000    0.000 test.py:3(print_message)
   3    0.000    0.000    0.000    0.000 test.py:6(range_output)
   1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
   6    0.000    0.000    0.000    0.000 {built-in method builtins.print}
   1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
objects}
```

В результате мы получили интересующую нас информацию:

- `ncalls` - количество вызовов функции;
- `tottime` - общее время затраченное в данной функции (без учета времени затраченного на вызовы подфункций);

- `percall` - время затраченное на 1 вызов функции (`tottime / ncalls`);
- `cumtime` - общее время затраченное в данной функции, включая время затраченное на вызов всех подфункций (от вызова до выхода). Данная величина верна и для рекурсивных функций;
- `percall` - время затраченное на 1 вызов функции (`cumtime / ncalls`);
- `filename:lineno(function)` - соответственно <имя файла>:<№ строки>(<имя функции>)

Также мы можем создать файл в формате `pstats` для использования в качестве данных инструментов построения графических отчетов (рассмотрены далее), что реализуется через флаг `-o` с указанием имени файла с расширением `.pstats` например *`cprofile_output.pstats`*:

```
python -m cProfile -o cprofile_output.pstats test.py
```

Yappi

Библиотека Yappi (`opens new window`)("Yet Another Python Profiler") - обладает более широким функционалом, чем встроенные средства Python. Yappi реализованная как Си библиотека - является быстрой, позволяет выполнить профилирование многопоточных приложений, приложений использующих `asyncio` (`opens new window`)и `gevent` (`opens new window`), позволяет вывести результаты в формате `callind` (`opens new window`)и `pstat` (`opens new window`), в результататы работы профилировщика могут отображать `Wall time` (`opens new window`)или `CPU Time` (`opens new window`)и могут быть агрегированы из разных сеансов. Кроме того при использовании PyCharm профилировщик Yappi используется в данной IDE по умолчанию.

Установка Yappi осуществляется следующим образом:

```
pip install yappi
```

Теперь немного изменим наш код, чтобы Yappi мог выполнить его анализ и собрать статистику в форматах `pstat` и `callgrind`:

```
import yappi
```

```

def print_message(message):
    print(message)

def range_output(external_step, internal_steps_number):
    for internal_step in range(internal_steps_number):
        print_message(f'[{external_step}: {internal_step}]')

def main():
    for i in range(3):
        range_output(external_step=i, internal_steps_number=2)

if __name__ == '__main__':
    yappi.start() # Здесь начинается наблюдение за выполняемым кодом
    main()
    func_stats = yappi.get_func_stats() # Собираем статистику выполненных функций
    func_stats.print_all() # Выведем в консоль собранную статистику
    func_stats.save('yappi_output.pstats', 'PSTAT') # Сохраняем в формате pstat
    func_stats.save('yappi_output.callgrind', 'CALLGRIND') # Сохраняем в формате
callgrind
    yappi.stop() # Останавливаем наблюдение
    yappi.clear_stats() # Очищаем статистику

```

Выполнение данного кода создаст файл *output.pstats* который может быть обработан с помощью ранее рассмотренного *gprof2dot*, и файл *output.callgrind* в формате *callgrind* (как использовать рассмотрим ниже) и выведет результаты профилирования в консоль:

```
python test.py
```

```
Clock type: CPU
```

```
Ordered by: totaltime, desc
```

name	ncall	tsub	ttot	tavg
../ver/MarketAnalyser/test.py:10 main	1	0.000017	0.000141	0.000141
../ketAnalyser/test.py:6 range_output	3	0.000029	0.000125	0.000042
../etAnalyser/test.py:3 print_message	6	0.000028	0.000095	0.000016

Здесь отображается:

- **Clock type: CPU** - указывает, что показанная статистика времени профилирования извлекается с использованием тактовой частоты ЦП. Это означает, что отображается фактическое время процессора, затраченное на функцию.
- **Ordered by: totaltime, desc** - показывает параметр по которому выполнена сортировка и ее порядок, в нашем случае сортировка выполнена по итоговому времени (*totaltime*) в порядке убывания
- **name** - полное уникальное имя вызываемой функции
- **ncall** - количество вызовов функции
- **tsub** - общее время, затраченной функцией, без дополнительных вызовов
- **ttot** - общее время, затраченное функцией, включая вспомогательные вызовы

- `tavg` - среднее время, затраченной функцией, включая вспомогательные вызовы.

Главное преимущество `yappi` - это реализация профилирования в многопоточных приложениях и корутинах. Более подробную информацию и примеры реализации можно посмотреть на GitHub странице проекта ([opens new window](#)).

Построение отчетов по результатам профилирования

Информация полученная с помощью вышеуказанных средств профилирования весьма полезна, но вывод ее не очень удобен для анализа, особенно в случае с реальным приложением где количество фрагментов кода и их вызовов будет огромным. По этой причине зачастую требуется помощь дополнительных инструментов позволяющих в графическом виде проанализировать данную информацию.

gprof2dot

Библиотека `gprof2dot` ([opens new window](#)), позволяет преобразовать вывод результатов профилирования в формате `pstat` преобразовать в удобный график вызовов функций который для итогового приложения.

`gprof2dot` для построения графики использует библиотеку `graphviz`.

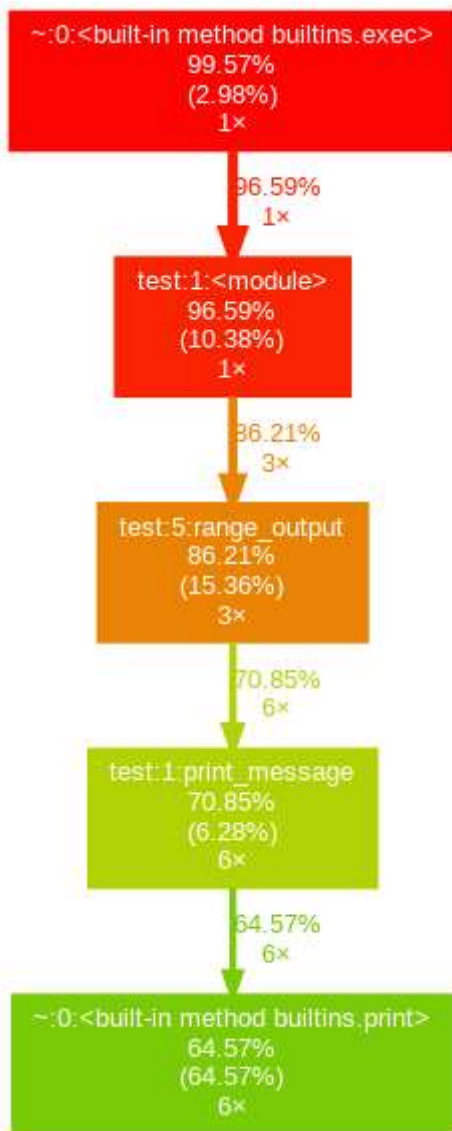
Для установки в ОС Linux необходимо выполнить следующие команды:

```
sudo apt install graphviz
pip install gprof2dot
```

Теперь отобразим графики вызовов, созданные нами ранее с помощью `gprof2dot` которые будут сохранены в формате `png`:

```
gprof2dot -f pstats cprofile_output.pstats | dot -Tpng -o cprofile_output.png
gprof2dot -f pstats yappi_output.pstats | dot -Tpng -o yappi_output.png
```

Откроем изображение *cprofile_output.png*:



Здесь мы наглядно видим что *86.21%* времени выполнения заняло выполнение функции `range_output` и данный метод был вызван 3 раза и за это время он 6 раз вызвал метод `print_message`. Таким образом с помощью `gprof2dot` можно строить удобные графические представления для профилирования.

Граф полученные в результате работы `yappi` выглядит несколько короче по причине того что запись ведется с момента указанного в коде:



Здесь аналогично с примером cProfile мы видим граф вызовов функций с начала записи `yappi.start()` до завершения `yappi.stop()`.

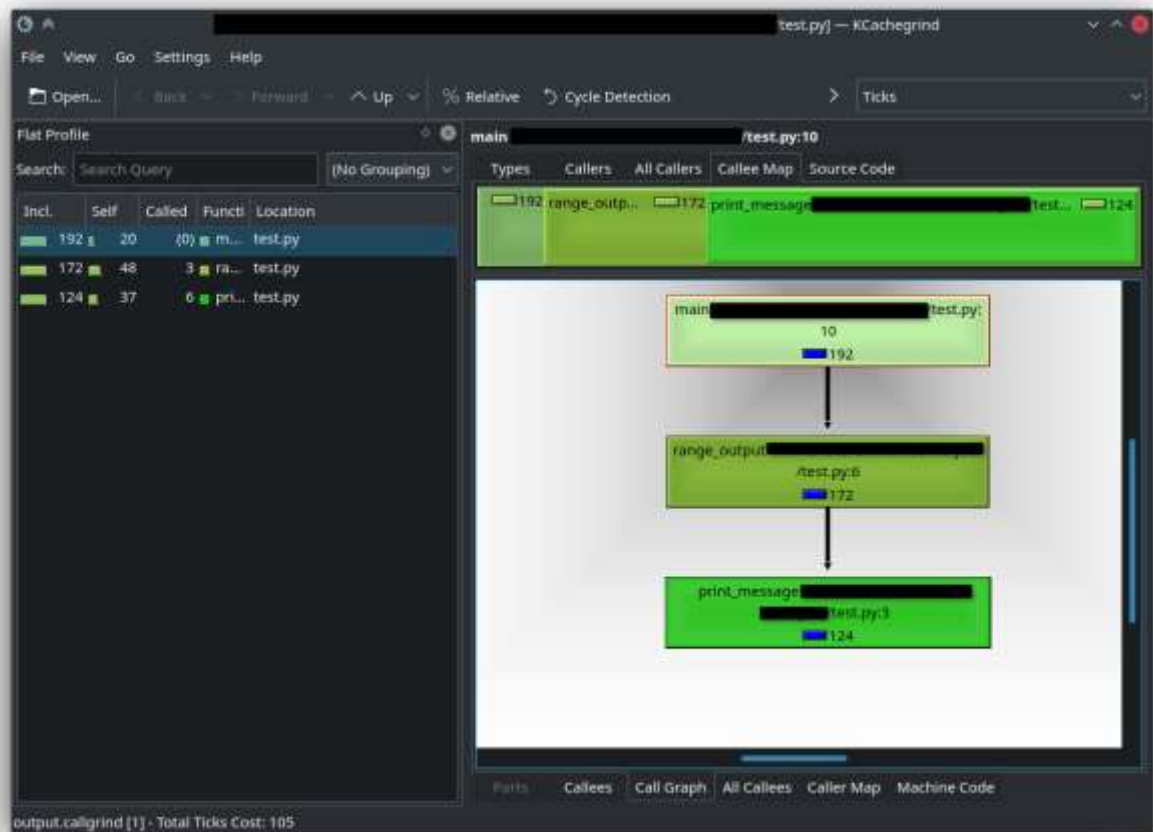
Как мы видим инструмент `gprof2dot` позволяет нам строить графические представления последовательностей вызовов функций нашего приложения с указанием количества вызовов и затрачиваемых ресурсов. Но `yappi` позволяет кроме того выполнить вывод результатов профилирования в формате Valgrind который называется `callgrind` (opens new window). Соответственно данный формат может быть открыт сторонними приложениями умеющими работать с ними одним из таких является `KCachegrind`.

KCachegrind

Создав файл в формате `callgrind` мы можем использовать продвинутые средства для просмотра результатов профилирования `KCachegrind` (opens new window). В среде Debian она может быть установлена:

```
sudo apt install kcache
```

После установки, запуском приложения `kcache` и открываем созданный нами файл `output.callgrind` и получаем подробнейший интерактивный отчет:



PyCharm Profile

Также если вы используете в работе IDE PyCharm Pro, данная IDE по умолчанию использует Yarpri как профайлер и имеет встроенные средства для графического отображения результатов профилирования и запустив профилирование в IDE мы получаем отчет в виде таблицы и графического представления:

Statistics			
Call Graph			
Name	Call Count	Time (ms)	Own Time (ms) ▾
test.py	1	0 0,0 %	0 0,0 %
main	1	0 0,0 %	0 0,0 %
range_output	3	0 0,0 %	0 0,0 %
print_message	6	0 0,0 %	0 0,0 %

print_message times;6Total: **0ms** 0,0 %Own: **0ms** 0,0 %**range_output times;3**Total: **0ms** 0,0 %Own: **0ms** 0,0 %**main times;1**Total: **0ms** 0,0 %Own: **0ms** 0,0 %**test.py times;1**Total: **0ms** 0,0 %Own: **0ms** 0,0 %

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕ-
ЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

**по организации и проведению самостоятельной
работы студентов по дисциплине «Алгоритмы и
структуры данных»**

направления подготовки 09.03.01 Информатика и вычислительная техника,
Направленность (профиль) «Программное обеспечение средств
вычислительной техники и автоматизированных систем»

Ставрополь, 2026

Методические указания по организации самостоятельной работы студентов составлено в соответствии с требованиями программы дисциплины «Алгоритмы и структуры данных» для бакалавров направления подготовки 09.03.01 Информатика и вычислительная техника. Учебно-методическое пособие посвящено планируемой учебной работе студентов, выполняемой во внеаудиторное (аудиторное) время по заданию и при методическом руководстве преподавателя, но без его непосредственного участия, при частичном, непосредственном участии преподавателя.

СОДЕРЖАНИЕ

1. Самостоятельная работа как важнейшая форма учебного процесса
2. Цели и основные задачи СРС
3. Виды самостоятельной работы
4. Организация СРС
5. Общие рекомендации по организации самостоятельной работы
6. Самостоятельная работа студента - необходимое звено становления исследователя
7. Методические рекомендации для студентов по отдельным формам самостоятельной работы
8. Учебно-методическое и информационное обеспечение дисциплины

1. Самостоятельная работа как важнейшая форма учебного процесса

Самостоятельная работа - планируемая учебная, учебно-исследовательская, научно-исследовательская работа студентов, выполняемая во внеаудиторное (аудиторное) время по заданию и при методическом руководстве преподавателя, но без его непосредственного участия (при частичном непосредственном участии преподавателя, оставляющем ведущую роль за работой студентов).

Необходимо ознакомиться с технологической картой самостоятельной работы обучающегося.

Самостоятельная работа студентов в ВУЗе является важным видом учебной и научной деятельности студента. Самостоятельная работа студентов играет значительную роль в современной технологии обучения. Обучение в ВУЗе включает в себя две, практически одинаковые по объему и взаимовлиянию части – процесса обучения и процесса самообучения. Поэтому СРС должна стать эффективной и целенаправленной работой студента.

Концепцией модернизации российского образования определены основные задачи профессионального образования - "подготовка квалифицированного работника соответствующего уровня и профиля, конкурентоспособного на рынке труда, компетентного, ответственного, свободно владеющего своей профессией и ориентированного в смежных областях деятельности, способного к эффективной работе по специальности на уровне мировых стандартов, готового к постоянному профессиональному росту, социальной и профессиональной мобильности".

Решение этих задач невозможно без повышения роли самостоятельной работы студентов над учебным материалом, усиления ответственности преподавателей за развитие навыков самостоятельной работы, за стимулирование профессионального роста студентов, воспитание творческой активности и инициативы.

К современному специалисту общество предъявляет достаточно широкий перечень требований, среди которых немаловажное значение имеет наличие у выпускников определенных способностей и умения самостоятельно добывать знания из различных источников, систематизировать полученную информацию, давать оценку конкретной финансовой ситуации. Формирование такого умения происходит в течение всего периода обучения через участие студентов в лабораторных занятиях, выполнение контрольных заданий и тестов, написание курсовых и выпускных квалификационных работ. При этом самостоятельная работа студентов играет решающую роль в ходе всего учебного процесса.

Формы самостоятельной работы студентов разнообразны. По данной дисциплине «Алгоритмы и структуры данных в робототехнике» предусмотрены следующие:

Использование материала учебно-методического комплекса дисциплины

На первом этапе необходимо ознакомиться с обязательным минимумом содержания основной образовательной программы по изучаемой дисциплине. Далее необходимо изучить рабочую программу дисциплины, в которой рассмотрено содержание тем дисциплины лекционного курса, взаимосвязь тем лекций с лабораторных занятий, темы и виды самостоятельной работы. По каждому виду самостоятельной работы предусмотрены определённые формы отчетности, представленные в учебной программе по дисциплине «Алгоритмы и структуры данных в робототехнике».

Работа с литературой

Для успешного освоения дисциплины, необходимо самостоятельно детально изучить представленные темы по рекомендуемым источникам информации, которые представлены в учебной программе.

Самостоятельная работа приобщает студентов к научному творчеству, поиску и решению актуальных современных проблем.

2. Цели и основные задачи СРС

Ведущая цель организации и осуществления СРС должна совпадать с целью обучения студента – подготовкой бакалавра с высшим образованием. При организации СРС важным и необходимым условием становятся формирование умения самостоятельной работы для приобретения знаний, навыков и возможности организации учебной и научной деятельности.

Целью самостоятельной работы студентов является овладение фундаментальными знаниями, профессиональными умениями и навыками деятельности по профилю, опытом творческой, исследовательской деятельности. Самостоятельная работа студентов способствует развитию самостоятельности, ответственности и организованности, творческого подхода к решению проблем учебного и профессионального уровня.

Задачами СРС являются:

- систематизация и закрепление полученных теоретических знаний и практических умений студентов;
- углубление и расширение теоретических знаний;
- формирование умений использовать нормативную, правовую, спра-

вочную документацию и специальную литературу;

- развитие познавательных способностей и активности студентов: творческой инициативы, самостоятельности, ответственности и организованности;
- формирование самостоятельности мышления, способностей к саморазвитию, самосовершенствованию и самореализации;
- развитие исследовательских умений;
- использование материала, собранного и полученного в ходе самостоятельных занятий на семинарах, на лабораторных занятиях, при написании курсовых и выпускной квалификационной работ, для эффективной подготовки к итоговым зачетам и экзаменам.

3. Виды самостоятельной работы

В образовательном процессе высшего профессионального образовательного учреждения выделяется два вида самостоятельной работы – аудиторная, под руководством преподавателя, и внеаудиторная. Тесная взаимосвязь этих видов работ предусматривает дифференциацию и эффективность результатов ее выполнения и зависит от организации, содержания, логики учебного процесса (межпредметных связей, перспективных знаний и др.):

Аудиторная самостоятельная работа по дисциплине выполняется на учебных занятиях под непосредственным руководством преподавателя и по его заданию.

Внеаудиторная самостоятельная работа выполняется студентом по заданию преподавателя, но без его непосредственного участия.

Основными видами самостоятельной работы студентов без участия преподавателя по данной дисциплине являются:

- формирование и усвоение содержания конспекта лекций на базе рекомендованной лектором учебной литературы, включая информационные образовательные ресурсы (электронные учебники, электронные библиотеки и др.);
- подготовка к лабораторным работам, их оформление.

Основными видами самостоятельной работы студентов с участием преподавателей являются:

- текущие консультации;
- прием и защита лабораторных работ (во время проведения л/р).
- выполнение курсовой работы.

4. Организация СРС

Методика организации самостоятельной работы студентов зависит от структуры, характера и особенностей дисциплины «Алгоритмы и структуры данных в робототехнике», объема часов на ее изучение, вида заданий для самостоятельной работы студентов, индивидуальных качеств студентов и условий учебной деятельности.

Процесс организации самостоятельной работы студентов включает в себя следующие этапы:

- подготовительный (определение целей, составление программы, подготовка методического обеспечения, подготовка оборудования);
- основной (реализация программы, использование приемов поиска информации, усвоения, переработки, применения, передачи знаний, фиксирование результатов, самоорганизация процесса работы);
- заключительный (оценка значимости и анализ результатов, их систематизация, оценка эффективности программы и приемов работы, выводы о направлениях оптимизации труда).

Деятельность студентов по формированию и развитию навыков учебной самостоятельной работы

В процессе самостоятельной работы студент приобретает навыки самоорганизации, самоконтроля, самоуправления, саморефлексии и становится активным самостоятельным субъектом учебной деятельности.

Выполняя самостоятельную работу под контролем преподавателя студент должен:

- освоить минимум содержания, выносимый на самостоятельную работу студентов и предложенный преподавателем в соответствии с образовательными стандартами высшего образования;
- планировать самостоятельную работу в соответствии с графиком самостоятельной работы, предложенным преподавателем;
- самостоятельную работу студент должен осуществлять в организационных формах, предусмотренных учебным планом и рабочей программой преподавателя;
- выполнять самостоятельную работу и отчитываться по ее результатам в соответствии с графиком представления результатов, видами и сроками отчетности по самостоятельной работе студентов.

студент может:

сверх предложенного преподавателем (при обосновании и согласовании с ним) и минимума обязательного содержания, определяемого программой по данной дисциплине:

- самостоятельно определять уровень (глубину) проработки содержания материала;
- предлагать темы и вопросы для самостоятельной проработки;
- в рамках общего графика выполнения самостоятельной работы предлагать обоснованный индивидуальный график выполнения и отчетности по результатам самостоятельной работы;
- предлагать свои варианты организационных форм самостоятельной работы;
- использовать для самостоятельной работы методические пособия, учебные пособия, разработки сверх предложенного преподавателем перечня;
- использовать не только контроль, но и самоконтроль результатов самостоятельной работы в соответствии с методами самоконтроля, предложенными преподавателем или выбранными самостоятельно.

Самостоятельная работа студентов должна оказывать важное влияние на формирование личности будущего бакалавра, она планируется студентом самостоятельно. Каждый студент самостоятельно определяет режим своей работы и меру труда, затрачиваемого на овладение учебным содержанием по каждой дисциплине. Он выполняет внеаудиторную работу по личному индивидуальному плану, в зависимости от его подготовки, времени и других условий.

5. Общие рекомендации по организации самостоятельной работы

Основной формой самостоятельной работы студента является изучение конспекта лекций, их дополнение, рекомендованной литературы, активное участие на лабораторных занятиях. Но для успешной учебной деятельности, ее интенсификации, необходимо учитывать следующие субъективные факторы:

1. Знание школьного программного материала, наличие прочной системы знаний, необходимой для усвоения основных вузовских курсов. Это особенно важно для математических дисциплин. Необходимо отличать пробелы в знаниях, затрудняющие усвоение нового материала, от малых способностей.
2. Затратив силы на преодоление этих пробелов, студент обеспечит себе нормальную успеваемость и поверит в свои способности.
3. Наличие умений, навыков умственного труда:
 - а) умение конспектировать на лекции и при работе с книгой;
 - б) владение логическими операциями: сравнение, анализ, синтез, обобщение, определение понятий, правила систематизации и классификации.

4. Специфика познавательных психических процессов: внимание, память, речь, наблюдательность, интеллект и мышление. Слабое развитие каждого из них становится серьезным препятствием в учебе.

5. Хорошая работоспособность, которая обеспечивается нормальным физическим состоянием. Ведь серьезное учение - это большой многосторонний и разнообразный труд. Результат обучения оценивается не количеством сообщаемой информации, а качеством ее усвоения, умением ее использовать и развитием у себя способности к дальнейшему самостоятельному образованию.

6. Соответствие избранной деятельности, профессии индивидуальным способностям. Необходимо выработать у себя умение саморегулировать свое эмоциональное состояние и устранять обстоятельства, нарушающие деловой настрой, мешающие намеченной работе.

7. Овладение оптимальным стилем работы, обеспечивающим успех в деятельности. Чередование труда и пауз в работе, периоды отдыха, индивидуально обоснованная норма продолжительности сна, предпочтение вечерних или утренних занятий, стрессоустойчивость на экзаменах и особенности подготовки к ним,

8. Уровень требований к себе, определяемый сложившейся самооценкой.

Адекватная оценка знаний, достоинств, недостатков - важная составляющая самоорганизации человека, без нее невозможна успешная работа по управлению своим поведением, деятельностью.

Одна из основных особенностей обучения в высшей школе заключается в том, что постоянный внешний контроль заменяется самоконтролем, активная роль в обучении принадлежит уже не столько преподавателю, сколько студенту. Зная основные методы научной организации умственного труда, можно при наименьших затратах времени, средств и трудовых усилий достичь наилучших результатов. Эффективность усвоения поступающей информации зависит от работоспособности человека в тот или иной момент его деятельности.

Работоспособность - способность человека к труду с высокой степенью напряженности в течение определенного времени. Различают внутренние и внешние факторы работоспособности.

К внутренним факторам работоспособности относятся интеллектуальные особенности, воля, состояние здоровья.

К внешним:

- организация рабочего места, режим труда и отдыха;
- уровень организации труда - умение получить справку и пользоваться информацией;

- величина умственной нагрузки.

Выдающийся русский физиолог Н. Е. Введенский выделил следующие условия продуктивности умственной деятельности:

- во всякий труд нужно входить постепенно;
- мерность и ритм работы. Разным людям присущ более или менее разный темп работы;
- привычная последовательность и систематичность деятельности;
- правильное чередование труда и отдыха.

Отдых не предполагает обязательного полного бездействия со стороны человека, он может быть достигнут простой переменой дела. В течение дня работоспособность изменяется. Наиболее плодотворным является *утреннее время* (с 8 до 14 часов), причем максимальная работоспособность приходится на период с 10 до 13 часов, затем *послеобеденное* - (с 16 до 19 часов) и *вечернее* (с 20 до 24 часов). Очень трудный для понимания материал лучше изучать в начале каждого отрезка времени (лучше всего утреннего) после хорошего отдыха. Через 1-1,5 часа нужны перерывы по 10 - 15 мин, через 3 - 4 часа работы отдых должен быть продолжительным - около часа.

Составной частью научной организации умственного труда является овладение техникой умственного труда.

Время, которым располагает студент для выполнения учебного плана, складывается из двух составляющих: одна из них - это аудиторная работа в вузе по расписанию занятий, другая - внеаудиторная самостоятельная работа. Задания и материалы для самостоятельной работы выдаются во время учебных занятий по расписанию, на этих же занятиях преподаватель осуществляет контроль за самостоятельной работой, а также оказывает помощь студентам по правильной организации работы.

Самостоятельные занятия потребуют интенсивного умственного труда, который необходимо не только правильно организовать, но и стимулировать. При этом очень важно уметь поддерживать устойчивое внимание к изучаемому материалу. Выработка внимания требует значительных волевых усилий. Именно поэтому, если студент замечает, что он часто отвлекается во время самостоятельных занятий, ему надо заставить себя сосредоточиться. Подобную процедуру необходимо проделывать постоянно, так как это является тренировкой внимания. Устойчивое внимание появляется тогда, когда человек относится к делу с интересом.

Следует правильно организовать свои занятия по времени: 50 минут - работа, 5-10 минут - перерыв; после 3 часов работы перерыв - 20-25 минут. Иначе нарастающее утомление повлечет неустойчивость внимания. Очень существен-

ным фактором, влияющим на повышение умственной работоспособности, являются систематические занятия физической культурой. Организация активного отдыха предусматривает чередование умственной и физической деятельности, что полностью восстанавливает работоспособность человека.

6. Самостоятельная работа студента - необходимое звено становления исследователя

Прогресс науки и техники, информационных технологий приводит к значительному увеличению научной информации, что предъявляет более высокие требования не только к моральным, нравственным свойствам человека, но и в особенности, постоянно возрастающие требования в области образования – обновление, модернизация общих и профессиональных знаний, умений специалиста.

Всякое образование должно выступать как динамический процесс, присущий человеку и продолжающийся всю его жизнь. Овладение научной мыслью и языком науки является необходимой составляющей в самоорганизации будущего специалиста исследователя. Под этим понимается не столько накопление знаний, сколько овладение научно обоснованными способами их приобретения. В этом, вообще говоря, состоит основная задача вуза.

Специфика вузовского учебного процесса, в организации которого самостоятельной работе студента отводятся все больше места, состоит в том, что он является как будто бы последним и самым адекватным звеном для реализации этой задачи. Ибо во время учебы в вузе происходит выработка стиля, навыков учебной (познавательной) деятельности, рациональный характер которых будет способствовать постоянному обновлению знаний высококвалифицированного выпускника вуза.

Однако до этого пути существуют определенные трудности, в частности, переход студента от синтетического процесса обучения в средней школе, к аналитическому в высшей. Это связано как с новым содержанием обучения (расширение общего образования и углубление профессиональной подготовки), так и с новыми, неизвестными до сих пор формами: обучения (лекции, семинары, лабораторные занятия и т.д.). Студент получает не только знания, предусмотренные программой и учебными пособиями, но он также должен познакомиться со способами приобретения знаний так, чтобы суметь оценить, что мы знаем, откуда мы это знаем и как этого знания мы достигли. Ко всему этому приходят через собственную самостоятельную работу.

Это и потому, что самостоятельно приобретенные знания являются более оперативными, они становятся личной собственностью, а также мотивом поведения, развивают интеллектуальные черты, внимание, наблюдательность, критичность, умение оценивать. Роль преподавателя в основном заключается в руководстве накопления знаний (по отношению к первокурсникам), а в последующие годы учебы, на старших курсах, в совместном установлении проблем и заботе о самостоятельных поисках студента, а также контролирования за их деятельностью. Отметим, что нельзя ограничиваться только приобретением знаний предусмотренных программой изучаемой дисциплины, надо постоянно углублять полученные знания, сосредотачивая их на какой-нибудь узкой определенной области, соответствующей интересам студента. Углубленное изучение всех предметов, предусмотренных программой, на практике является возможным, и хорошая организация работы позволяет экономить время, что создает условия для глубокого, систематического, заинтересованного изучения самостоятельно выбранной студентом темы.

Конечно, все советы, примеры, рекомендации в этой области, даваемые преподавателем, или определенными публикациями, или другими источниками, не гарантируют никакого успеха без проявления собственной активности в этом деле, т. е. они не дают готовых рецептов, а должны способствовать анализу собственной работы, ее целей, организации в соответствии с индивидуальными особенностями. Учитывая личные возможности, существующие условия жизни и работы, навыки, на основе этих рекомендаций, возможно, выработать индивидуально обоснованную совокупность методов, способов, найти свой стиль или усовершенствовать его, чтобы изучив определенный материал, иметь время оценить его значимость, пригодность и возможности его применения, чтобы, в конечном счете, обеспечить успешность своей учебы с будущей профессиональной деятельности

7. Методические рекомендации для студентов по отдельным формам самостоятельной работы

Система вузовского обучения подразумевает значительно большую самостоятельность студентов в планировании и организации своей деятельности.

Работа с литературой

При работе с литературой необходимо подобрать литературу, научиться правильно ее читать, вести записи. Для подбора литературы в библиотеке используются алфавитный и систематический каталоги.

Важно помнить, что рациональные навыки работы с книгой - это всегда большая экономия времени и сил.

Правильный подбор учебников рекомендуется преподавателем, читающим лекционный курс. Необходимая литература может быть также указана в методических разработках по данному курсу.

Изучая материал по учебнику, следует переходить к следующему вопросу только после правильного уяснения предыдущего, описывая на бумаге все выкладки и вычисления (в том числе те, которые в учебнике опущены или на лекции даны для самостоятельного вывода).

При изучении любой дисциплины большую и важную роль играет самостоятельная индивидуальная работа.

Особое внимание следует обратить на определение основных понятий курса. Студент должен подробно разбирать примеры, которые поясняют такие определения, и уметь строить аналогичные примеры самостоятельно. Нужно добиваться точного представления о том, что изучаешь. Полезно составлять опорные конспекты. При изучении материала по учебнику полезно в тетради (на специально отведенных полях) дополнять конспект лекций. Там же следует отмечать вопросы, выделенные студентом для консультации с преподавателем.

Выводы, полученные в результате изучения, рекомендуется в конспекте выделять, чтобы они при перечитывании записей лучше запоминались.

Опыт показывает, что многим студентам помогает составление листа опорных сигналов, содержащего важнейшие и наиболее часто употребляемые формулы и понятия. Такой лист помогает запомнить формулы, основные положения лекции, а также может служить постоянным справочником для студента.

Различают два вида чтения; первичное и вторичное. *Первичное* - это внимательное, неторопливое чтение, при котором можно остановиться на трудных местах. После него не должно остаться ни одного непонятого слова. Содержание не всегда может быть понятно после первичного чтения.

Задача *вторичного* чтения полное усвоение смысла целого (по счету это чтение может быть и не вторым, а третьим или четвертым).

Чтение научного текста является частью познавательной деятельности. Ее цель – извлечение из текста необходимой информации. От того насколько осознанна читающим собственная внутренняя установка при обращении к печатному слову (найти нужные сведения, усвоить информацию полностью или частично, критически проанализировать материал и т.п.) во многом зависит эффективность осуществляемого действия.

Выделяют четыре основные установки в чтении научного текста:

1. информационно-поисковый (задача – найти, выделить искомую информацию)

2. усваивающая (усилия читателя направлены на то, чтобы как можно полнее осознать и запомнить как сами сведения излагаемые автором, так и всю логику его рассуждений)

3. аналитико-критическая (читатель стремится критически осмыслить материал, проанализировав его, определив свое отношение к нему)

4. творческая (создает у читателя готовность в том или ином виде – как отправной пункт для своих рассуждений, как образ для действия по аналогии и т.п. – использовать суждения автора, ход его мыслей, результат наблюдения, разработанную методику, дополнить их, подвергнуть новой проверке).

С наличием различных установок обращения к научному тексту связано существование и нескольких **видов чтения**:

1. библиографическое – просматривание карточек каталога, рекомендательных списков, сводных списков журналов и статей за год и т.п.;

2. просмотрное – используется для поиска материалов, содержащих нужную информацию, обычно к нему прибегают сразу после работы со списками литературы и каталогами, в результате такого просмотра читатель устанавливает, какие из источников будут использованы в дальнейшей работе;

3. ознакомительное – подразумевает сплошное, достаточно подробное прочтение отобранных статей, глав, отдельных страниц, цель – познакомиться с характером информации, узнать, какие вопросы вынесены автором на рассмотрение, провести сортировку материала;

4. изучающее – предполагает доскональное освоение материала; в ходе такого чтения проявляется доверие читателя к автору, готовность принять изложенную информацию, реализуется установка на предельно полное понимание материала;

5. аналитико-критическое и творческое чтение – два вида чтения близкие между собой тем, что участвуют в решении исследовательских задач. Первый из них предполагает направленный критический анализ, как самой информации, так и способов ее получения и подачи автором; второе – поиск тех суждений, фактов, по которым или в связи с которыми, читатель считает нужным высказать собственные мысли.

Из всех рассмотренных видов чтения основным для студентов является изучающее – именно оно позволяет в работе с учебной литературой накапливать знания в различных областях. Вот почему именно этот вид чтения в рамках учебной деятельности должен быть освоен в первую очередь. Кроме того, при овладении данным видом чтения формируются основные приемы, повышающие эффективность работы с научным текстом.

Основные виды систематизированной записи прочитанного:

1. Аннотирование – предельно краткое связное описание просмотренной или прочитанной книги (статьи), ее содержания, источников, характера и назначения;
2. Планирование – краткая логическая организация текста, раскрывающая содержание и структуру изучаемого материала;
3. Тезирование – лаконичное воспроизведение основных утверждений автора без привлечения фактического материала;
4. Цитирование – дословное выписывание из текста выдержек, извлечений, наиболее существенно отражающих ту или иную мысль автора;
5. Конспектирование – краткое и последовательное изложение содержания прочитанного.

Конспект – сложный способ изложения содержания книги или статьи в логической последовательности. Конспект аккумулирует в себе предыдущие виды записи, позволяет всесторонне охватить содержание книги, статьи. Поэтому умение составлять план, тезисы, делать выписки и другие записи определяет и технологию составления конспекта.

Методические рекомендации по составлению конспекта:

1. Внимательно прочитайте текст. Уточните в справочной литературе непонятные слова. При записи не забудьте вынести справочные данные на поля конспекта.
2. Выделите главное, составьте план.
3. Кратко сформулируйте основные положения текста, отметьте аргументацию автора.
4. Законспектируйте материал, четко следуя пунктам плана. При конспектировании старайтесь выразить мысль своими словами. Записи следует вести четко, ясно.
5. Грамотно записывайте цитаты. Цитируя, учитывайте лаконичность, значимость мысли.

В тексте конспекта желательно приводить не только тезисные положения, но и их доказательства. При оформлении конспекта необходимо стремиться к емкости каждого предложения. Мысли автора книги следует излагать кратко, заботясь о стиле и выразительности написанного. Число дополнительных элементов конспекта должно быть логически обоснованным, записи должны распределяться в определенной последовательности, отвечающей логической структуре произведения. Для уточнения и дополнения необходимо оставлять поля.

Овладение навыками конспектирования требует от студента целеустремленности, повседневной самостоятельной работы.

Лабораторные занятия

Следует подчеркнуть, что только после усвоения лекционного материала с определенной точки зрения (а именно с той, с которой он излагается на лекциях) он будет закрепляться на лабораторных занятиях как в результате обсуждения и анализа лекционного материала, так и с помощью выполнения заданий лабораторной работы, поставленных задач. При этих условиях студент не только хорошо усвоит материал, но и научится применять его на практике, а также получит дополнительный стимул (и это очень важно) для активной проработки лекции.

При самостоятельном выполнении заданий нужно обосновывать каждый этап решения, исходя из теоретических положений курса. Если студент видит несколько путей решения проблемы (задачи), то нужно сравнить их и выбрать самый рациональный. Полезно до начала вычислений составить краткий план решения проблемы (задачи). Решение задач или примеров следует излагать подробно, вычисления располагать в строгом порядке, отделяя вспомогательные вычисления от основных. Решения при необходимости нужно сопровождать комментариями, схемами, чертежами и рисунками.

Следует помнить, что выполнение каждого учебного задания должно доводиться до окончательного логического ответа, которого требует условие, и по возможности с выводом. Выполнение заданий данного типа нужно продолжать до приобретения твердых навыков в их решении.

Самопроверка

После изучения определенной темы по записям в конспекте и учебнику, а также решения достаточного количества соответствующих заданий на лабораторных занятиях и самостоятельно студенту рекомендуется, используя лист опорных материалов, воспроизвести по памяти определения, выводы формул, формулировки основных положений и доказательств.

В случае необходимости нужно еще раз внимательно разобраться в материале.

Иногда недостаточность усвоения того или иного вопроса выясняется только при изучении дальнейшего материала. В этом случае надо вернуться назад и повторить плохо усвоенный материал. Важный критерий усвоения теоретического материала - умение решать задачи или пройти тестирование по пройденному материалу. Однако следует помнить, что правильное решение за-

дачи может получиться в результате применения механически заученных формул без понимания сущности теоретических положений.

Консультации

Если в процессе самостоятельной работы над изучением теоретического материала или при выполнении заданий у студента возникают вопросы, разрешить которые самостоятельно не удастся, необходимо обратиться к преподавателю для получения у него разъяснений или указаний. В своих вопросах студент должен четко выразить, в чем он испытывает затруднения, характер этого затруднения. За консультацией следует обращаться и в случае, если возникнут сомнения в правильности ответов на вопросы самопроверки.

Подготовка к экзамену по дисциплине «Алгоритмы и структуры данных»

Изучение многих общепрофессиональных и специальных дисциплин завершается зачетом. Подготовка к экзамену способствует закреплению, углублению и обобщению знаний, получаемых, в процессе обучения, а также применению их к решению практических задач. Готовясь к нему, студент ликвидирует имеющиеся пробелы в знаниях, углубляет, систематизирует и упорядочивает свои знания.

Систематическая подготовка к занятиям в течение семестра позволит использовать время экзаменационной сессии для систематизации знаний.

Текущая аттестация студентов проводится преподавателями, ведущими лабораторные занятия по дисциплине в следующих формах: собеседование при защите лабораторной работы с предоставлением письменного отчета. Допуск к лабораторным работам происходит при наличии у студентов печатного варианта отчета. Защита отчета проходит в форме собеседования. Экзамен студент сдает в указанное время, по билету отвечает на теоретические вопросы и реализует задачу.

8. Учебно-методическое и информационное обеспечение дисциплины

Основная литература	1. Павлов, Л. А. Структуры и алгоритмы обработки данных : учебник для вузов / Л. А. Павлов, Н. В. Первова. — 3-е изд., стер. — Санкт-Петербург : Лань, 2021. — 256 с. — ISBN 978-5-8114-7259-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/156929 (дата обращения: 11.04.2025). — Режим доступа: для авториз. пользователей. 2. Макаров, Л. М. Проектирование беспилотных транспортных средств : учебное пособие / Л. М. Макаров. — Санкт-Петербург : СПбГУТ им. М.А. Бонч-Бруевича, 2023. — 107 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/381488 (дата обращения: 11.04.2025). — Режим доступа: для авториз. пользователей.
Дополнительная литература	3. Корк, П. Машинное зрение. Основы и алгоритмы с примерами на Matlab : руководство / П. Корк ; перевод с английского В. С. Яценкова. — Москва : ДМК Пресс, 2023. — 584 с. — ISBN 978-5-93700-222-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/456581 (дата обращения: 11.04.2025). — Режим доступа: для авториз. пользователей.
Программное обеспечение	Альт Рабочая станция 10 Альт Рабочая станция К Альт «Сервер» Пакет офисных программ - Р7-Офис