МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

С.В. ВОСТОКИН

АРХИТЕКТУРА ОПЕРАЦИОННЫХ СИСТЕМ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии

САМАРА
Издательство Самарского университета
2023

УДК 004.451(075)
ББК 3973я7
В780

Рецензенты: д-р тех. наук, проф. С.П. О р л о в,
д-р физ.-мат. наук, проф. Д.Л. Г о л о в а ш к и н

Востокин, Сергей Владимирович

В780 Архитектура операционных систем: учебное пособие / *С.В. Востокин*. – Самара: Издательство Самарского университета, 2023. – 84 с.

ISBN 978-5-7883-1876-9

Рассмотрены базовые вопросы операционных систем: определения, классификация, проектирование и архитектуры, приведен исторический обзор операционных систем. Дается обзор языка программирования Си: указателей, адресной арифметики и типовых проблем при работе с указателями. Приведены варианты задач для самоконтроля, проведения лабораторных занятий и экзамена. Включен список экзаменационных вопросов.

Учебное пособие предназначено для изучающих дисциплину «Операционные системы» по образовательным программам бакалавриата в области информационных технологий.

УДК 004.451(075)
ББК 3973я7

ISBN 978-5-7883-1876-9 © Самарский университет, 2023

ОГЛАВЛЕНИЕ

Введение……………………………………………………………4

1. Основные понятия………………………………………………..7
1.1. Определение операционной системы.
Первые операционные системы………..……………………………7
1.2. Современные операционные системы…………………….....15
1.3. Вопросы……..……………………………………..………..28

2. Проектирование операционных систем………………………29
2.1. Требования к операционным системам
и обзор подходов их реализации………..……………………..…..29
2.2. Архитектуры операционных систем………………………....37
2.3. Вопросы…………………………………………………… 46

3. Введение в язык программирования Си……………………..48
3.1. Обзор основных свойств……………………..……………..48
3.2. Указатели и адресная арифметика…………………………...60
3.3. Изучение синтаксиса…………………………………………69
3.4. Изучение функций……………………………………………70
3.5. Работа с динамической памятью……………………………..73
3.6. Задачи ……………………………...……………………...75

Список литературы……………………………………………80

ВВЕДЕНИЕ

Предлагаемое вашему вниманию учебное пособие представляет собой сборник материалов для проведения лекционных, лабораторных и практических занятий по курсу «Операционные системы», читаемому автором студентам, обучающимся на бакалавриате ИТ-направлений института Информатики и кибернетики Самарского национального исследовательского университета имени академика С.П. Королева.

В данном учебном пособии представлена вводная часть курса, знакомящая с предметом, терминологией, основными понятиями в области операционных систем. В качестве материала для лабораторных и практических занятий во вводной части курса студентам предлагается освоить язык системного программирования Си, в частности, указатели и адресную арифметику, работу с динамической памятью с использованием стандартной библиотеки Си.

На базе освоенного по данному пособию вводного теоретического материала далее в основной теоретической части курса изучаются системы управления процессами, памятью и вводом-выводом. Основная практическая часть курса посвящена освоению работы с кучами процессов, реализации явной и неявной динамической компоновки, управлению процессами и потоками. Этот материал содержится в отдельном пособии.

Рекомендуется следующая методика освоения предлагаемого материала. Теоретическая и практическая части курса осваиваются на параллельно идущих (чередующихся) занятиях. Два раздела «Основные понятия» и «Проектирование операционных систем» содержат материалы для проведения лекционных занятий по четырем темам: «Определение операционной системы. Первые операционные системы»; «Современные операционные системы»; «Требования к операционным системам и обзор подходов их реализации»;

«Архитектуры операционных систем». Данный материал разбирается на интерактивных лекциях в очном либо онлайн формате на университетской платформе для проведения телеконференций. В качестве контроля освоения теоретического материала при проведении промежуточной аттестации обучающимся на экзаменационных занятиях предлагается самостоятельно ответить на два случайно выбранных вопроса из числа вопросов, приведенных в параграфах «Вопросы». Самостоятельная подготовка по теоретической части курса включает написание рукописного конспекта ответов на экзаменационные вопросы. При ответе на экзаменационные вопросы теоретической части курса, в том числе в случае онлайн формата промежуточной аттестации, допускается использование лично подготовленных рукописных конспектов.

Углубленное изложение предлагаемого теоретического материала можно найти, в частности, в учебниках Э. Таненбаума и других авторов, приведенных в списке литературы. При самостоятельной подготовке обучающимся рекомендуется ознакомиться с соответствующими главами дополнительной литературы, обзору которой обычно посвящается часть вводной лекции курса.

Практическая часть освоения курса «Операционные системы» по предлагаемому пособию включает изучение теории и практики программирования на языке Си, так как в настоящее время язык Си является основным языком разработки операционных систем и системного программного обеспечения. Предполагается, что студенты самостоятельно изучают теорию языка Си по литературе, в частности по монографии авторов языка Си Б. Кернигана и Д. Ритчи.

Для закрепления и обобщения теории языка Си в данном пособии приведен обзор языка Си в форме вопрос-ответ в части «Обзор основных свойств» и рассмотрены основные вопросы, возникаю-

щие при работе с указателями, в части «Указатели и адресная арифметика». На основе данных частей пособия обычно проводятся отдельные лекционные или практические закрепляющие занятия.

Результаты освоения языка Си проверяются с использованием тестовых задач на экзамене. Предлагается две тестовые задачи, требующие ответа без подготовки с пояснением выбранного варианта ответа. Примеры экзаменационных задач приведены в части «Задачи» данного учебного пособия.

Лабораторный практикум по основам языка Си включает три лабораторные работы, варианты заданий к которым приведены в третьем разделе данного пособия. Для выполнения заданий рекомендуется сразу начать работу в профессиональной интегрированной среде разработки на языках Си и C++, например, Microsoft Visual Studio или JetBrains CLion. Однако, в силу специфики заданий, обучающимся разрешается ограничиться упрощенными онлайн инструментами, например, https://www.onlinegdb.com/, https://cpp.sh/, https://coliru.stacked-crooked.com/. Правильность выполнения задач лабораторного практикума проверяется преподавателем путем аудита кода и проверки работоспособности программ. В качестве итогового отчета по результатам освоения курса обучающийся оформляет написанный на лабораторных занятиях и проверенный преподавателем код в виде электронного отчета. Отчет служит допуском для прохождения промежуточной аттестации по дисциплине «Операционные системы».

Задания к лабораторному практикуму рассчитаны на индивидуальное выполнение. Задания реализуются в виде консольных приложений. Приложение может быть выполнено, по желанию, как интерактивное, запрашивающее входные данные с консоли в диалоге с пользователем либо с набором заранее подготовленных в коде входных данных, результат обработки которых выводится в консоль.

1. ОСНОВНЫЕ ПОНЯТИЯ

1.1. Определение операционной системы. Первые операционные системы

Определение операционной системы, операционная система как виртуальная машина и как система управления ресурсами. Поколения компьютеров и операционных систем.

Операционная система — это комплекс программ, которые выступают как интерфейс между устройствами вычислительной системы и прикладными программами, предназначены для управления устройствами и вычислительными процессами, а также для эффективного распределения вычислительных ресурсов и организации надёжных вычислений.

Таким образом, операционная система - это программное обеспечение, выполняющее две функции: 1) предоставление пользователю-программисту более удобной в использовании «виртуальной машины», скрывающей реальное оборудование; 2) обеспечение эффективного использования компьютера путем рационального управления его ресурсами.

Рассмотрим для чего необходим программный интерфейс между оборудованием и программой пользователя на примере. Использование большинства компьютеров на уровне машинного языка затруднительно, особенно при организации ввода-вывода. Например, для чтения блока данных с гибкого диска программист может использовать 16 различных команд контроллера NEC PD765, каждая из которых требует 13 параметров: номера поверхности на диске, сектора на дорожке и других. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающих, в том числе, наличие и типы ошибок, которые необходимо анализировать. Кроме этого, работа с диском требует специальной

организации кода программы, связанной с необходимостью обработки прерываний, отслеживания состояния двигателя привода для уменьшения износа диска с сохранением высокой скорости обращения, организации хранения данных и учета других особенностей.

При работе с диском, используя функции операционной системы, программисту достаточно представлять данные в виде некоторого набора файлов, каждый из которых имеет имя. Работа с файлом заключается в открытии, выполнении чтения или записи, а затем в закрытии файла. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного чтения или записи файлов – это операционная система. Операционная система ограждает программистов от аппаратуры дискового накопителя и предоставляет ему простой файловый интерфейс, берет на себя обработку прерываний, управление таймерами и оперативной памятью, а также другие низкоуровневые проблемы. В каждом случае абстракция (например, файл), с которой, благодаря операционной системе, может иметь дело пользователь, гораздо проще и удобнее в обращении, чем реальная аппаратура, лежащая в основе этой абстракции. С этой точки зрения функцией операционной системы является предоставление пользователю некоторой расширенной или «виртуальной машины», которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.

Программист-пользователь считает, что операционная система прежде всего обеспечивает удобный интерфейс. Разработчик аппаратуры имеет представление об операционной системе как о некотором механизме, управляющим всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, накопителей на магнитных лентах, сетевой коммуникационной аппаратуры, принтеров и других устройств.

Функцией операционной системы является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. Операционная система должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность или реактивность системы. Управление ресурсами включает решение двух общих, не зависящих от типов ресурсов задач: планирование ресурсов и мониторинг ресурсов.

Планирование ресурса – это определение кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс.

Мониторинг (отслеживание состояния ресурса) – поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов, какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные операционные системы используют различные алгоритмы. Это, в конечном счете, определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс.

Рассмотренное функциональное определение операционной системы применимо к большинству современных операционных систем общего назначения.

Для понимания основных механизмов, используемых в современных операционных системах, кратко рассмотрим историю их развития. Применимость той или иной архитектурной идеи в программном обеспечении (software) во многом определяется наличием и доступностью аппаратного обеспечения (hardware).

Первое поколение компьютеров и операционных систем (1945-1955 гг.), электронные лампы и коммутационные панели.

Изобретателем цифрового компьютера считается английский математик Чарльз Беббидж. В 1833 году им был предложен проект механической универсальной цифровой вычислительной машины — прообраза современной ЭВМ. Первыми действующими компьютерами являлись: компьютер Z3, созданный немецким инженером Конрадом Цузе (1941); компьютер Марк I, созданный американским инженером Говардом Эйкеном (1941); компьютер ЭНИАК, разработанный Джоном Экертом и Джоном Мокли (1945). Первые компьютеры были электромеханическими (реле). В поздних моделях реле были заменены электронными лампами.

Первые компьютеры программировались на абсолютном машинном языке, управление основными функциями машины осуществлялось путем соединения коммутационных панелей проводами. Такая панель являлась носителем программы и подключалась к компьютеру. Также для записи программ использовались перфорированные ленты. В начале 50-х панели и ленты были заменены перфокартами.

На компьютерах первого поколения занимались только прямыми числовыми вычислениями, например, расчетами таблиц синусов, косинусов, логарифмов. Компьютеры имели военные приложения: расчет стреловидных крыльев и управляемых ракет, расчет баллистических таблиц для стрельбы. Компьютеры не имели операционных систем. Программирование осуществлялось в интерактивном режиме: пользователь-программист получал полный контроль над машиной на время отладки программы и выполнения вычислений. В некотором смысле первые компьютеры напоминали современные персональные ЭВМ.

Второе поколение компьютеров и операционных систем (1955-1965 гг.), транзисторы и системы пакетной обработки заданий. С появлением в середине 50-х годов транзисторов компью-

теры стали достаточно надежными и могли без сбоев работать длительное время. Такие компьютеры назывались мейнфреймами. Они располагались в специальных помещениях с кондиционированным воздухом, ими управлял целый штат профессиональных операторов. Цена одного из известных мейнфреймов IBM 7090/94 рис. 1 составляла $2,900,000, а стоимость аренды $63,500 в месяц.

Рис. 1. Машинный зал с компьютером IBM 7094, НАСА.
На переднем плане внизу – считыватель перфокарт

Рассмотрим организацию работы на IBM 7090/94 и подобных машинах. Для программирования использовался язык высокого уровня Фортран или ассемблер. Программа для компьютера вначале писалась на бумаге, а затем переносилась на перфокарты при помощи перфораторов: электронно-механических устройств, похо-

жих на печатающие машинки (рис. 2). Каждая перфокарта представляла одну строчку кода программы. Перфокарты выполняли функции современных текстовых редакторов. Программа – это колода перфокарт, которая вставлялась в устройство для считывания. Результат работы очередной задачи отображался на принтере. Если в процессе расчетов был необходим компилятор языка Фортран, то оператору необходимо было брать его из карточного шкафа и загружать в машину отдельно.

Рис. 2. Перфоратор IBM 026

С учетом высокой стоимости оборудования необходимо было повысить эффективность использования машинного времени, сократить простои машины при загрузке программ в память. Данную функцию реализовывали первые операционные системы – системы пакетной обработки.

Рассмотрим автоматизацию расчетов на мейнфрейме IBM 7094 с использованием пакетной системы IBSYS (рис. 3).

Рис. 3. Схема работы пакетной системы

Для подготовки пакета заданий использовались относительно недорогие компьютеры IBM 1401 (рис. 4).

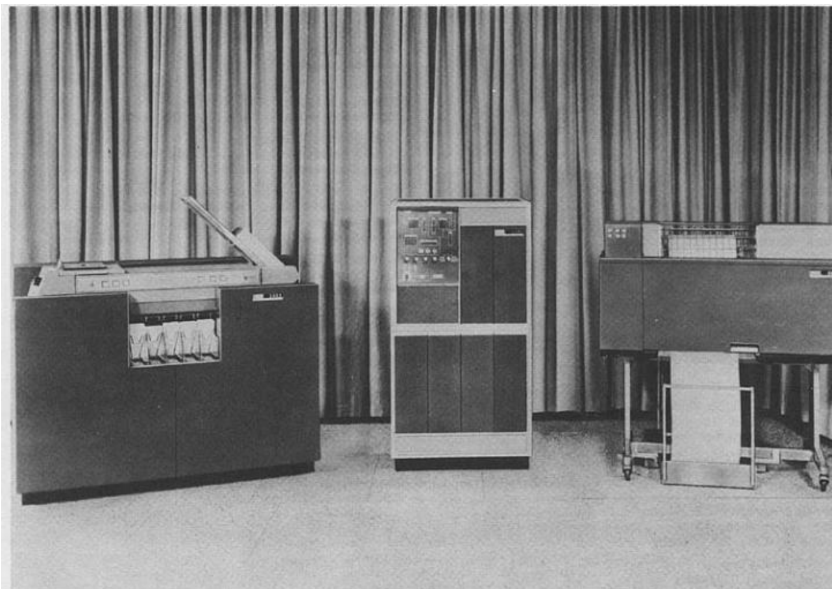

Рис. 4. Считыватель-перфоратор 1402, управляющий блок 1401 и принтер 1403 фирмы IBM

К такому компьютеру подключался считыватель перфокарт и устройство записывания магнитных лент. Примерно после часа сбора пакета заданий лента перематывалась, и ее относили в машинную комнату. Там ее устанавливали на лентопротяжное устройство, подключенное к мейнфрейму. Также к нему подключалась лента с программой операционной системы и выходная лента. По мере накопления выходные данные записывались на ленту вместо того, чтобы идти на печать. После обработки всего пакета заданий входные и выходные ленты менялись на новые для следующего цикла обработки. Печать результатов на принтере также осуществлялась под управлением маломощного и недорогого компьютера IBM 1401.

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике, и инженерных задачах. Поэтому режим работы с непродолжительными этапами ввода-вывода с использованием магнитных лент значительно оптимизировал загрузку мейнфрейма. Кроме этого, длительные операции, такие как считывание перфокарт и печать на медленных принтерах, происходили одновременно с расчетами. Такой способ взаимодействия с внешними устройствами получил название автономного (off-line), в отличие от интерактивного способа (on-line), используемого в системах третьего поколения.

Принципы пакетной обработки, несмотря на появление в 60-х годах, широко используются в современных системах для аналогичных целей: оптимизации загрузки дорогостоящей системы. Примером современных пакетных систем являются системы TORQUE (Terascale Open-Source Resource and QUEue Manager) и Slurm (Simple Linux Utility for Resource Management), служащие для управления суперкомпьютерами. Также пакетные режимы имеются во всех современных операционных системах разделения времени (Unix, macOS, Microsoft Windows).

1.2. Современные операционные системы

Поколения компьютеров и операционных систем: интегральные схемы и многозадачность, персональные компьютеры. Классификация и примеры операционных систем.

Третье поколение компьютеров и операционных систем (1965-1980 гг.), интегральные схемы и многозадачность. Дальнейшее развитие компьютерная индустрия получила при появлении технологии мелкомасштабных интегральных схем, которая давала преимущество в цене и качестве по сравнению с машинами второго поколения, созданными из отдельных транзисторов.

Основные разработки в области архитектуры современных операционных систем появились именно на компьютерах третьего поколения. Операционные системы, состоящие из миллионов строк, написанных тысячами программистов, на два или три порядка превышали по сложности первые системы типа IBSYS.

Основу к появлению сложного системного программного обеспечения заложила фирма IBM, которая выпустила серию программно-совместимых машин семейства IBM /360. Эти компьютеры различались только ценой и производительностью. Так как все машины семейства имели одинаковую структуру и набор команд, программы, написанные для одного компьютера, могли работать на других. Компьютеры семейства IBM /360 могли использоваться как для сложных научных расчетов, так и для статистической обработки, сортировки и печати. Предполагалось, что одну операционную систему можно будет использовать как с несколькими внешними устройствами, так и с большим их количеством. Одно семейство компьютеров было призвано удовлетворить потребности всех покупателей.

Первой операционной системой для компьютеров IBM /360 была OS /360 (IBM System /360 Operating System). Разработкой системы руководил Фред Брукс, описавший ее в знаменитой книге «Мифический человеко-месяц». В этом проекте впервые столкнулись с проблемой сложности программного обеспечения, с него начала развиваться дисциплина программной инженерии, впервые были применены многие технические приемы, используемые в современных операционных системах.

Самым важным достижением OS /360 являлось использование многозадачности. На компьютере IBM 7094, когда текущая работа приостанавливалась в ожидании данных с магнитной ленты или других устройств, центральный процессор бездействовал до окончания ввода-вывода. Данная ситуация была некритичной при обработке задач численного моделирования. Однако для универсальной системы, на которой обрабатывались задачи, связанные с сортировкой и другой обработкой информации на лентах, время простоя оказывалось существенным и составляло 80-90%. Необходимо было предложить решение, предотвращающее длительный простой дорогостоящих процессоров.

Решение проблемы было найдено. Оно заключалось в разбиении памяти на несколько разделов, каждому из которых давалось отдельное задание (рис. 5). Пока одно задание ожидало завершение операций ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти размещалось достаточное количество заданий, центральный процессор мог быть загружен почти 100% времени. Для реализации многозадачности потребовалось разработать аппаратуру защиты памяти и средства обработки прерываний. Применение многозадачности позволило отказаться от процедуры предварительной подготовки данных на магнитных лентах и выгрузки на ленту для печати. Данные процедуры являлись обычными фоновыми операциями ввода-вывода. Технический прием получил название спулинг (spooling – simultaneous peripheral operation on line).

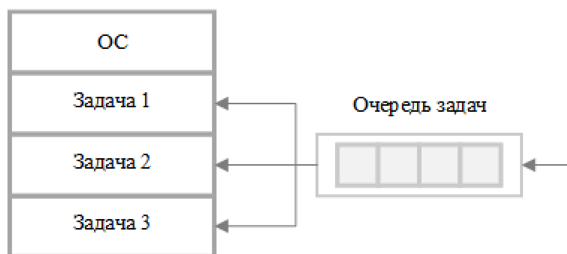

Рис. 5. Многозадачная система с тремя заданиями в памяти

Операционная система OS /360 и другие операционные системы третьего поколения являлись, по сути, сложными системами пакетной обработки. Главный недостаток таких систем заключался в том, что, несмотря на оптимизацию работы вычислительной системы, организация работы самих программистов являлась неэффективной. Промежуток времени между передачей задания и получением результата составлял несколько часов. Сбой при компиляции, связанный с незначительной синтаксической ошибкой, приводил к большой потере времени. Желание применить принципы интерактивной разработки, известные по компьютерам первого поколения, на новых системах привело к появлению режима разделения времени.

Режим разделения времени – вариант многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Так как пользователи чаще выдают короткие команды компиляции при отладке программ, компьютер может обеспечить быстрое интерактивное обслуживание нескольких пользователей.

Первая система разделения времени CTSS (Compatible Time-Sharing System) была разработана Фернандо Корбато и его командой в Массачусетском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094. Впоследствии в 1962 году решения, использовавшиеся в демонстрационной системе

CTSS, легли в основу проекта первой полноценной системы с разделением времени, названной Multiplexed Information and Computer System (Multics) на компьютере GE-645, позднее Honeywell 6180.

Работа над системой Multics продолжалась вплоть до 1969 года, но коммерческого успеха она не имела. Однако этот проект имел определяющее значение для дальнейшего развития компьютерных технологий. В системе Multics впервые реализованы: сегментно-страничная виртуальная память; отображение файлов на адресное пространство процессов; динамическое связывание исполняемой программы с библиотеками. В ней применялись даже более сложные и гибкие механизмы взаимодействия процессов через разделяемые сегменты, чем в современных системах. Использовалось «горячее» реконфигурирование системы с заменой процессоров, блоков памяти, жестких дисков без остановки обслуживания пользователей. Multics – одна из первых мультипроцессорных систем. Multics была также одной из первых систем, в которой большое внимание уделялось безопасности взаимодействия между программами и пользователями. Более того, она была самой первой операционной системой, задуманной изначально и реализованной как безопасная. Дополнительно к этому, в Multics одной из первых была реализована иерархическая файловая система, имена файлов могли быть практически произвольной длины и содержать любые символы. Файл или директория могли иметь несколько имён (короткое и длинное), были доступны для использования символьные ссылки (symlink) между директориями. В Multics был впервые реализован (теперь уже стандартный) подход использования стеков для каждого вычислительного процесса в ядре системы: с отдельным стеком для каждого уровня безопасности вокруг ядра. Multics явилась одной из первых операционных систем, написанных на языке высокого уровня PL/I. Можно сказать, что идея «облачных» вычислений,

популярная в настоящее время, также берет начало от системы Multics – компьютерного предприятия общественного пользования 60-х и 70-х годов прошлого века.

Важным моментом развития компьютерной индустрии во времена третьего поколения был феноменальных рост миникомпьютеров, начиная с выпуска корпорацией DEC в 1961 году компьютера PDP-1 и наиболее коммерчески успешной модели PDP-11. Особенностью этого семейства являлась низкая цена ($120,000, что составляло 5% цены мейнфрейма IBM) и, сопоставимая с мейнфреймами, производительность на нечисловых задачах. Кроме этого, PDP-11 отличалась удачной «ортогональной» системой команд: можно было отдельно запоминать команды, и отдельно — методы доступа к операндам. Все регистры, в том числе служебные (указатель стека, счетчик команд), одинаково использовались со всеми командами.

Все эти факторы способствовали очень широкому распространению компьютеров этого семейства, а вместе с ними и операционной системы Unix. Операционная система UNIX была разработана сотрудниками Bell Labs, в первую очередь Кеном Томпсоном, Деннисом Ритчи и Дугласом Макилроем. В 1969 году Кен Томпсон, стремясь реализовать идеи, которые были положены в основу Multics, но на более скромном аппаратном обеспечении (DEC PDP-7), написал первую версию новой операционной системы. Брайан Керниган придумал для неё название – UNICS (UN-multiplexed Information and Computing System – примитивная информационная и вычислительная служба) – в противовес Multics. Позже это название сократилось до UNIX. В ноябре 1971 года вышла версия для PDP-11. Эта версия получила название «первая редакция» (Edition 1) и была первой официальной версией. Системное время все реализации UNIX отсчитывают с 1 января 1970.

UNIX-системы имеют большую историческую важность. UNIX популяризовала предложенную в Multics идею иерархической файловой системы с произвольной глубиной вложенности. Еще одной инновацией Multics, популяризированной UNIX, являлся интерпретатор команд. Так как интерпретатор и команды операционной системы - обычные программы, пользователь мог выбирать их в соответствии со своими предпочтениями или даже написать собственную оболочку. Наконец, новые команды можно добавлять к системе без перекомпиляции ядра. Предложенный в командной строке UNIX способ создания цепочек программ, последовательно обрабатывающих данные (конвейер, pipeline), способствовал использованию параллельной обработки данных.

Существенными особенностями UNIX были полная ориентация на текстовый ввод-вывод и предположение, что размер машинного слова кратен восьми битам. Первоначально в UNIX не было даже редакторов двоичных файлов – система полностью конфигурировалась с помощью текстовых команд. Наибольшей и наименьшей единицей ввода-вывода служил текстовый байт, что полностью отличало ввод-вывод UNIX от ввода-вывода других операционных систем. Ориентация на текстовый восьмибитный байт сделала UNIX более масштабируемой и переносимой, чем другие операционные системы. UNIX способствовала широкому распространению регулярных выражений, которые были впервые реализованы в текстовом редакторе ed для UNIX. Возможности, предоставляемые UNIX-программами, стали основой стандартных интерфейсов операционных систем (POSIX).

Примеры известных UNIX-подобных операционных систем: BSD, Solaris, Linux, Minix, Android, MeeGo, NeXTSTEP, macOS.

Отметим, что в нашей стране также были распространены компьютеры, аналогичные семействам IBM /360 и DECPDP. В конце

70-х начале 80-х годов в СССР и ряде соцстран разрабатывалось семейство управляющих ЭВМ (СМ ЭВМ). Наиболее развитой линией в семействе СМ ЭВМ был ряд машин, совместимых по системе команд и архитектуре с машинами PDP-11 фирмы DEC. На них, например, использовались операционные системы РАФОС (аналог RT-11, системы реального времени для компьютера PDP-11), ДЕМОС (советский клон Unix, созданный на основе BSD).

Аналогом серии мейнфреймов IBM /360 и /370 в СССР являлась «Единая система электронных вычислительных машин» (ЕС ЭВМ). Операционная система ОС ЕС разрабатывалась как клон OS /360 путем дизассемблирования и адаптации исходных кодов. Версия ОС ЕС 7 использовала концепцию гипервизора (виртуальной машины). Эта версия - аналог операционной системы VM, еще одного типа операционных систем компьютеров серии /360 /370.

В целом можно отметить, что третье поколение компьютеров являлось источником основных архитектурных инноваций, используемых в современных операционных системах. Роль четвертого поколения заключается в широком внедрении этих инноваций.

Четвертое поколение компьютеров и операционных систем (с 1980 года), персональные вычисления. Следующий этап эволюции операционных систем связан с появлением больших интегральных схем (БИС), по-английски large scale integration (LSI). С точки зрения аппаратной архитектуры персональные компьютеры (микрокомпьютеры) были похожи на PDP-11, но значительно уступали по цене. Теперь каждый желающий получил возможность купить собственный компьютер.

Первой операционной системой для микрокомпьютеров была CP/M (Control Program for Microcomputers), разработчик Гари Килдэлл, Digital Research. Она работала на первых 8-разрядных системах с процессором Intel 8080, Zilog Z80. С появлением в 1981 году новой серии IBM PC - совместимых компьютеров, на

базе 16-разрядного процессора Intel 8088, на рынке программного обеспечения микрокомпьютеров операционную систему CP/M сменила система MS- DOS компании Microsoft.

Операционные системы для первых микрокомпьютеров полностью основывались на вводе команд с клавиатуры. Однако еще в 60-е годы в научно-исследовательском институте Стэнфорда (Stanford Research Institute) Дугласом Энгельбартом был изобретен графический интерфейс пользователя (GUI, Graphical User Interface), в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений. Графический интерфейс был реализован в прототипе современных персональных компьютеров Xerox Alto, разработанном в исследовательском центре Xerox PARC в 1973 году.

На Xerox Alto использовались программы с графическими меню, пиктограммами и другими элементами, ставшие привычными лишь с появлением операционных систем Mac OS и Microsoft Windows. Для Xerox Alto были разработаны текстовые процессоры Bravo и Gypsy, работавшие по принципу WYSIWYG (предшественники Microsoft Word), редакторы графической информации (растровых изображений, печатных плат, интегральных схем и др.), среда Smalltalk (объектно-ориентированное программирование).

Первой успешной коммерческой реализацией GUI для персональных компьютеров была система Apple Macintosh, изготовленная под руководством Стива Джобса в 1984 году. Под влиянием успехов Apple, фирма Microsoft в 1985 году выпускает свою реализацию GUI – систему Windows. На протяжении 10 лет система Windows исполняла роль графической оболочки поверх MS-DOS. Были разработаны еще две линейки настольных операционных систем. Семейство Windows 9x включает в себя Windows 95, Windows 98 и

Windows ME. Оно являлось промежуточным семейством 32-разрядных систем. Семейство Windows NT – современные версии операционной системы Windows, берущие начало от системы Windows NT 3.1, выпущенной в 1993 году. Наиболее популярные версии операционных систем этого семейства: Windows NT 4.0; Windows 2000 (NT 5.0); Windows XP (NT 5.1); Windows Vista (NT 6.0); Windows 7 (NT 6.1); Windows 8 (NT 6.2); Windows 10 (NT 10.0.1) и Windows 11 (NT 10.0.2).

Главными конкурентами систем Windows в персональных вычислениях становятся различные производные системы Unix: семейство операционных систем Apple macOS (BSD), Google Android (Linux), Ubuntu (Linux). Для серверных приложений и высокопроизводительных вычислений операционная система Unix (в частности, Linux) является доминирующей операционной системой.

Современные компьютеры и операционные системы. Развитие аппаратных технологий приводит к уменьшению размеров, повышению быстродействия, увеличению коммуникационных возможностей, снижению стоимости и распространению вычислительной техники. В распоряжении крупнейших компаний, таких как Google, Amazon, Facebook, Microsoft и др. находится огромная вычислительная инфраструктура, включающая центры обработки и хранения данных, сети передачи данных, спутники связи и периферийное оборудование. Важнейшим видом периферийного оборудования являются мобильные устройства, включая смартфоны, планшетные компьютеры, автомобильные компьютеры, встраиваемые системы. По некоторым оценкам с данной инфраструктурой непосредственно взаимодействует примерно 90% всего населения Земли. Новые технологии в области операционных систем, по-видимому, будут связаны с развитием методов управления и внедрением вычислений во все сферы человеческой жизни. Для такой формы вычислений используется термин ubiquitous computing –

«вездесущие вычисления». «Вездесущие вычисления» охватывают широкий набор направлений, связанных с операционными системами, включая распределённые вычисления, мобильные вычисления, мобильные сети, сенсорные сети, человеко-машинное взаимодействие и искусственный интеллект.

Рассмотрим некоторые классификационные признаки и примеры операционных систем.

По признаку поддержки многозадачности различают однозадачные и многозадачные операционные системы. Однозадачные системы в основном выполняют функцию предоставления расширенной машины. Из рассмотренных операционных систем однозадачными являются MS-DOS, CP/M, RT-11 (версии SJ, SL), ранние версии Unix. Примеры многозадачных систем – Unix, Windows, операционная система персональных компьютеров с графическим интерфейсом IBM OS/2.

По числу пользователей операционные системы делятся на однопользовательские и многопользовательские. Главное отличие многопользовательских систем – это наличие средств защиты информации одного пользователя от несанкционированного доступа со стороны других пользователей. Многозадачные системы могут быть однопользовательскими, например, Windows 3.1. Теоретически возможны однозадачные многопользовательские системы. Одной из первых многопользовательских систем с развитыми механизмами защиты была система Multics. Пример другой однопользовательской системы Microsoft – MS-DOS. В системах на базе ядра NT реализованы полноценные механизмы защиты многопользовательского режима.

По аппаратным средствам, на которых функционируют операционные системы, их можно разделить на системы для мейнфреймов, миникомпьютеров, микрокомпьютеров. Примеры этих опера-

ционных систем были рассмотрены в историческом обзоре. Существуют также встраиваемые системы. Они работают непосредственно в устройствах, которыми управляют: средствах управления техническими процессами; станках с ЧПУ; банкоматах, платежных терминалах; телекоммуникационном оборудовании (Windows IoT, QNX). В отдельную группу можно выделить операционные системы мобильных устройств: смартфонов, планшетных компьютеров, умных часов и др. (iOS, Android, Windows 10 Mobile). Самые производительные компьютеры нашего времени (суперкомпьютеры и кластерные системы) в основном работают под управлением Linux. В классификации, основанной на аппаратных средствах, разделение идет по возможности работы с устройствами ввода-вывода и периферийным оборудованием. Наименьшие возможности у встраиваемых систем интеллектуальных смарт-карт (Java Card, MULTOS). Наибольшие возможности – у суперкомпьютеров.

По особенностям работы в сети операционные системы делятся на сетевые и распределенные. Сетевая операционная система – это обычная система, расширенная поддержкой сетевого взаимодействия. По методам реализации сетевых функций системы различаются по способам реализации справочной информации о сетевых ресурсах и адресации; механизмам обеспечения прозрачности доступа. В линейке Windows первой сетевой системой была Windows for Workgroups 3.1. Распределённая операционная система, динамически и автоматически распределяя работы по различным машинам системы для обработки, заставляет набор сетевых машин обрабатывать информацию параллельно. Пользователь распределённой операционной системы не имеет сведений, на какой машине выполняется его работа. Примеры распределенных систем – Plan 9, Amoeba.

Существует несколько вариантов реализации многозадачных операционных систем. В зависимости от *способа переключения с за-*

дачи на задачу различают системы с не вытесняющей многозадачностью и вытесняющей многозадачностью. Вытесняющая многозадачность предполагает использование прерываний от таймера как основной способ переключения задач. Операционные системы семейств Windows 1.0-3.11 использовали не вытесняющую многозадачность, а Windows 9x, WindowsNT – вытесняющую.

Еще одним аспектом реализации многозадачности является *использование концепции нитей (потоков) исполнения* – многозадачности внутри одного процесса. Нити использовались в Windows, начиная с Windows 95.

Многозадачные системы можно разделить на три большие группы по критерию эффективности алгоритма планирования и, что эквивалентно, по областям применения многозадачных операционных систем.

Критерии эффективности *системы пакетной обработки* ориентированы на обеспечение максимальной пропускной способности, максимально полной загрузки вычислительной системы. Критерий характеризуется процентом загрузки центрального процессора; количеством заданий, выполняемых в единицу времени; оборотным временем (среднее время вычисления задачи). В таких системах выбор нового задания для выполнения определяется от текущей внутренней ситуации, складывающийся в системе. Время выполнения задачи зависит от того, какие задачи загружены одновременно с ней и как в них происходит ввод-вывод. Поэтому невозможно гарантировать выполнение некоторого задания в течение определенного интервала времени, что является недостатком.

Система разделения времени устраняет данный недостаток пакетных систем. В таких системах распределение времени процессора организовано на основе квантования: периодического переключения управления между всеми одновременно запущенными задачами. С точки зрения задачи это создает иллюзию монопольного

использования процессора (концепция терминала). Эффективность систем разделения времени определяется эргономическими показателями, связанными с субъективными ощущениями пользователя, как быстро система обрабатывает команды терминала. Системы разделения времени обладают меньшей пропускной способностью из-за того, что ресурсы процессора тратятся на постоянное переключение между задачами.

Реальное время в операционных системах — это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени. *Системы реального времени* применяют для управления техническими объектами. В них существует предельно допустимое время, в течение которого должна быть выполнена программа управления объектом. Это время называется временем реакции, а свойство операционных систем оперативно реагировать на события – реактивностью. Различают системы жёсткого реального времени — с режимом работы системы, при котором нарушение временных ограничений равнозначно отказу системы (RTLinux); системы мягкого реального времени — с режимом работы системы, при котором нарушения временных ограничений приводят к снижению качества работы (QNX, RT-11, Windows IoT).

Другими важными классификационными признаками являются: возможность исполнения операционной системы на разном типе компьютеров; особенности архитектур и типов ядра; используемые методы управления памятью; организация файловых систем. Они рассматриваются в следующих разделах лекционного курса.

———

1.3. Вопросы

1. Определение операционной системы и ее функции. Понятие виртуальной машины. Управление ресурсами.
2. История разработки операционных систем, поколения ЭВМ и операционных систем. Влияние аппаратуры на развитие операционных систем. Лампы - коммутационные панели. Транзисторы – пакетные системы. Интегральные схемы – многозадачность. СБИС – персональные компьютеры.
3. Классификация и примеры операционных систем. Многозадачность. Вид многозадачности. Многопоточная обработка. Критерии эффективности многозадачных операционных систем.

2. ПРОЕКТИРОВАНИЕ ОПЕРАЦИОННЫХ СИСТЕМ

2.1. Требования к операционным системам и обзор подходов их реализации

Обзор требований, предъявляемых к операционным системам: расширяемость, переносимость, надежность и отказоустойчивость, совместимость, безопасность, производительность.

Операционная система выполняет связующую роль между оборудованием и прикладными программами. Она является важнейшим элементом программной инфраструктуры, от свойств которой зависит качество работы прикладного программного обеспечения. Поэтому при проектировании операционных систем уделяют внимание специальным функциональным требованиям, более широким, чем при проектировании прикладных программ.

Расширяемость. Код операционной системы должен быть написан таким образом, чтобы при необходимости можно было легко внести дополнения и изменения, не нарушая целостности системы.

Во времена первых компьютеров считалось, что по мере обновления аппаратного обеспечения, код операционных систем будет создаваться заново, с нуля. Однако практика показала, что разработка программного обеспечения значительно более трудоемка, чем разработка аппаратуры. Возникла необходимость защиты инвестиций, как в разработку операционной системы, так и в прикладные программы, функционирующие под ее управлением.

В то время как аппаратная часть компьютера устаревает за несколько лет, полезная жизнь операционной системы может измеряться десятилетиями (Unix). Аппаратные изменения, к которым должна быть, в первую очередь, адаптирована архитектура опера-

ционных систем, связаны с развитием периферийного оборудования. Например, современные модификации в операционных системах связаны с новыми технологиями хранения данных (накопители SSD), сетевого взаимодействия (беспроводные сети, оптические каналы высокой пропускной способности), новыми пользовательскими интерфейсами (жестовый ввод, голосовой ввод, сенсорные панели). Сохранение целостности кода операционной системы, учитывая, что происходит постоянная модернизация аппаратуры, является одной из целей проектирования операционной системы.

Расширяемость может достигаться за счет модульной структуры операционной системы, при которой ее части состоят из сильно связанных внутри и слабо связанных между собой модулей. Взаимодействие между модулями организуется через стандартизированные интерфейсы. Новые компоненты, добавляемые в операционную систему, функционируют, используя интерфейсы, поддерживаемые существующими компонентами. Можно заменить код устаревших модулей, не затрагивая работоспособность системы в целом.

Использование объектов для представления системных ресурсов также улучшает расширяемость системы. Объекты позволяют единообразно управлять системными ресурсами. Добавление новых объектов не разрушает существующие объекты и не требует изменений существующего кода.

Вариантом модульной организации является применение архитектуры клиент-сервер и микроядра. В ней модули изолированы в отдельных процессах и могут замещаться даже без перекомпиляции и остановки вычислений.

Другой возможностью расширить функциональные возможности операционной системы являются средства вызова удаленных процедур (RPC), которые могут добавляться в любую машину сети

и немедленно поступать в распоряжение прикладных программ на других машинах сети.

Некоторые операционные системы для улучшения расширяемости поддерживают загружаемые драйверы, которые добавляются в систему во время ее работы. Новые файловые системы, устройства и сети могут поддерживаться путем написания драйвера устройства, драйвера файловой системы или транспортного драйвера и загрузки его в систему.

Переносимость. Код операционной системы должен легко переноситься между процессорами и аппаратными платформами различной архитектуры. Аппаратная платформа включает наряду с типом процессора и способ организации всей аппаратуры компьютера.

Расширяемость позволяет улучшать операционную систему по мере обновления периферийного оборудования. Переносимость дает возможность перемещать всю систему целиком на машину, базирующуюся на обновленном процессоре или аппаратной платформе, делая при этом по возможности небольшие изменения в коде. Операционные системы разрабатываются либо как переносимые, либо как непереносимые. Ключевым моментом в оценке переносимости является стоимость необходимых изменений.

При написании переносимой операционной системы нужно следовать перечисленным ниже правилам.

Большая часть кода должна быть написана на языке высокого уровня, например, как Unix на языке Си. Код, написанный на ассемблере, не является переносимым, если только он не переносится на машину, обладающую командной совместимостью с исходной машиной.

Необходимо учитывать аппаратную платформу, на которую операционная система должна быть перенесена. Например, система, построенная на 32-битовых адресах, не может быть перенесена на машину с 16-битовыми адресами.

Следует минимизировать или по возможности исключить части кода, которые непосредственно взаимодействуют с аппаратурой. Оставшийся после такой оптимизации аппаратно-зависимый код, который не может быть исключен, локализуется в отдельных модулях (HAL – hardware abstraction layer). Очевидные формы зависимости от аппаратуры включают прямое манипулирование регистрами процессора, аппаратно-зависимыми структурами данных (контекст процесса, дескрипторы страниц и сегментов), обращение к контроллерам периферийного оборудования.

Надежность и отказоустойчивость. Система должна быть защищена от отказов оборудования и внутренних ошибок. Действия операционной системы должны быть предсказуемыми. Прикладные процессы не должны иметь возможность наносить вред, как самой операционной системе, так и другим процессам. Надежность обеспечивается применением микроядерной архитектуры.

Совместимость. Операционная система должна иметь средства для выполнения прикладных программ, написанных для других операционных систем, или для более ранних версий данной операционной системы. Пользовательский интерфейс должен быть совместим с существующими системами и стандартами. Совместимость операционных систем рассматривается на двух уровнях как двоичная совместимость и совместимость по исходным кодам.

Двоичная совместимость достигается совместимостью на уровне команд процессора, системных вызовов и на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость по исходным кодам требует наличия соответствующего компилятора в составе программного обеспечения, а

также совместимости на уровне библиотек и системных вызовов, при этом необходима перекомпиляция.

Совместимость на уровне исходных текстов важна для разработчиков приложений. Для конечных пользователей практическое значение имеет только двоичная совместимость, благодаря которой один и тот же коммерческий продукт можно использовать в различных операционных средах и на различных машинах. Совместимость зависит от многих факторов, самый важный фактор – архитектура процессора. Если процессор, на который переносится операционная система, использует аналогичный набор команд и тот же диапазон адресов, двоичная совместимость может быть достигнута достаточно просто.

Двоичная совместимость между процессорами, основанными на разных архитектурах, требует написания специальных эмуляторов и использования прикладных сред. Для исполнения кода в гостевой операционной системе требуется: процедура загрузки, адаптированная под формат исполняемого файла; интерпретация команд целевого процессора на гостевом процессоре (если процессорные архитектуры различаются); имитация системных вызовов (вызовов библиотечных функций операционной системы) с использованием заранее написанной библиотеки функций аналогичного назначения.

Примером прикладных сред в Windows NT являются подсистемы для исполнения Win32 приложений, консольных приложений OS/2 и Unix, 16-разрядных DOS и Win16 приложений. Среда исполнения языка Java, как основной механизм переносимости программ, использует программную эмуляцию архитектуры Java, которая может быть реализована полностью на аппаратном уровне. Для запуска Win32(64) приложений на Linux используется программная среда Wine. Среда Cygwin, наоборот, представляет

собой инструмент для переноса программ UNIX в Windows и является библиотекой, которая реализует интерфейс к системным вызовам Win32.

Средствами обеспечения совместимости на уровне исходных кодов являются стандартизированный язык высокого уровня и стандартизированные интерфейсы системных вызовов. Примером стандартизированного процедурного языка программирования является Си (действующий стандарт для языка ISO/IEC 9899:2018 вышел в июне 2018 года, выпуск новой версии запланирован на 2023 год). Наиболее известным набором стандартов, описывающих интерфейсы между операционной системой и прикладной программой, является POSIX (Portable Operating System Interface for Unix — переносимый интерфейс операционных систем Unix). Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода. Использование стандарта POSIX (ISO/IEC/IEEE 9945:2009) позволяет создавать программы в стиле UNIX, которые могут легко переноситься из одной вычислительной системы в другую.

Для семейства операционных систем, основанных на Linux, имеется стандарт совместимости LSB (Linux Standard Base). LSB специфицирует: стандартные библиотеки, несколько команд и утилит в дополнение к стандарту POSIX, структуру иерархии файловой системы, уровни запуска и различные расширения системы X Window System.

Безопасность. Операционная система должна обладать средствами защиты. Правила безопасности определяют способы защиты ресурсов пользователя и устанавливают квоты по ресурсам для предотвращения захвата пользователем всех системных ресурсов (например, памяти).

Основы безопасности были заложены стандартом «Критерии оценки надежных компьютерных систем» (Department of DefenseTrusted Computer System Evaluation Criteria, TCSEC, DoD 5200.28-STD, 26 декабря 1985 года). Этот документ часто называют «Оранжевой книгой». Аналогом «Оранжевой книги» является международный стандарт ISO/IEC 15408-5:2022, опубликованный в 2022 году.

В соответствии с требованиями «Оранжевой книги» безопасной считается такая система, которая «посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации».

Иерархия уровней безопасности, приведенная в «Оранжевой книге», выделяет четыре уровня (D, C, B, A) и 6 классов безопасности внутри уровней (C1, C2, B1, B2, B3, A1). В уровень D попадают системы, оценка которых выявила их несоответствие требованиям безопасности. Уровень C обеспечивает произвольное управление доступом; уровень B – принудительное управление доступом; уровень A – верифицируемую безопасность. В каждом классе от C1 к A1 требования по безопасности расширяются.

Коммерческие системы обычно соответствуют уровню C. Вот некоторые требования безопасности этого уровня: 1) наличие защищенных средств секретного входа, обеспечивающих идентификацию пользователя путем ввода логина и пароля; 2) избирательный (дискреционный) контроль доступа, позволяющий владельцу ресурса определить, кто имеет доступ к ресурсу, и что он может с ним делать, путем предоставляемых прав доступа пользователю или группе пользователей; 3) средства учета и наблюдения, обеспечивающие возможность обнаружить и зафиксировать важные события,

связанные с безопасностью: любые попытки создать, получить доступ и удалить системные ресурсы; 4) защита памяти, обеспечивающая инициализацию перед повторным использованием.

Системы уровня В реализуют мандатный (принудительный) контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень, в отличие от уровня С, обеспечивает централизованное управление потоками данных в системе и защищает систему от ошибочного поведения пользователя.

Однако существует проблема, впервые описанная Батлером Лэмпсоном в 1973 году, известная как «скрытый канал» или проблема ограждения. Лэмпсон показал, что в защищенной системе возможны неконтролируемые принудительной системой безопасности потоки информации. Определяют два вида скрытых каналов.

Скрытый канал памяти. Процессы взаимодействуют благодаря тому, что один может прямо или косвенно записывать информацию в некоторую область памяти, а второй считывать. Обычно имеется в виду, что у процессов с разными уровнями безопасности имеется доступ к некоторому ресурсу (например, возможность проверить наличие файла).

Скрытый канал времени. Один процесс посылает информацию другому, модулируя своё собственное использование системных ресурсов (например, процессорное время) таким образом, что эта операция воздействует на реальное время отклика, наблюдаемое вторым процессом.

Критерии «Оранжевой книги» требуют, чтобы анализ скрытых каналов памяти был классифицирован как требование для системы класса В2, а анализ скрытых каналов времени как требование для класса В3.

Уровень А является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня В выполнения

формального (математически обоснованного) доказательства соответствия системы требованиям безопасности.

Производительность. Система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа. На практике удовлетворение рассмотренных выше требований к операционным системам, особенно по надежности и безопасности, приводит к большому потреблению системных ресурсов на функционирование самой операционной системы. Снижение ресурсных требований и повышение производительности является нетривиальной исследовательской задачей при проектировании новых операционных систем.

2.2. Архитектуры операционных систем

Монолитные системы: модульные и многоуровневые. Клиент-серверные архитектуры: микроядерные и гибридные. Объектно-ориентированные архитектуры. Виртуальные машины: гипервизоры, экзоядра, наноядра.

Монолитная система – классическая, наиболее распространённая архитектура ядер операционных систем. Все части монолитного ядра работают в одном адресном пространстве (рис. 6).

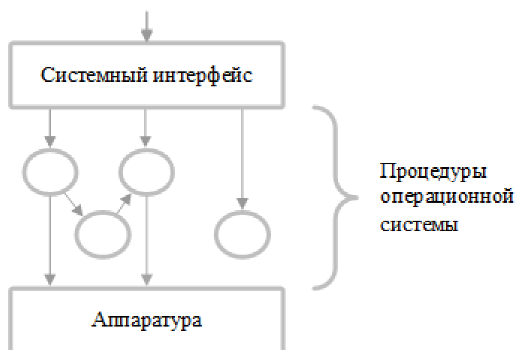

Рис. 6. Монолитная операционная система

При использовании этой технологии код ядра операционной системы состоит из процедур. Каждая процедура системы имеет определённый интерфейс и может вызывать любую другую процедуру, а также обращаться непосредственно к аппаратуре. Код ядра выполняется в режиме супервизора.

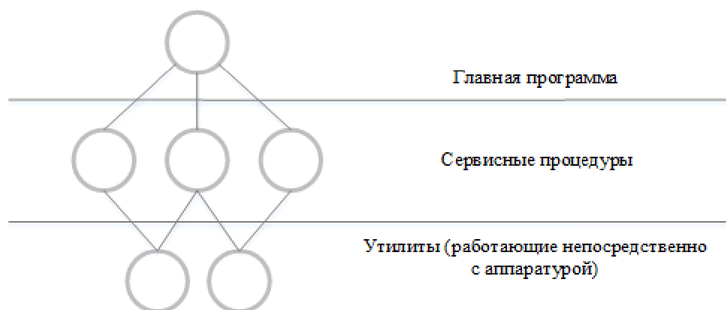

Рис. 7. Структура ядра монолитной системы

Обычно процедуры операционной системы разделяют на главную программу, сервисные процедуры и утилиты (рис. 7). Взаимодействие кода пользователя с операционной системой происходит посредством системных вызовов. В отличие от обычных вызовов подпрограмм, в системном вызове происходит передача управления и данных между кодом режима пользователя и кодом режима супервизора (ядра). При обращении к системным вызовам главная программа помещает параметры вызова в строго определённое место (регистры, стек) и переключает машину из режима пользователя в режим ядра (например, вызывая специальное программное прерывание).

В режиме ядра по сформированному коду системного вызова вычисляется адрес сервисной процедуры в пространстве ядра и выполняется вызов. У каждого системного вызова имеется своя сервисная процедура, а утилиты выполняют функции, нужные нескольким сервисным процедурам.

В классической монолитной архитектуре все процедуры операционной системы собраны в один объектный файл. *Модульное ядро* – современная, усовершенствованная модификация архитектуры монолитных ядер, в которой код ядра разделяется на отдельно компилируемые и загружаемые модули.

Модульные ядра предоставляют механизм загрузки модулей ядра, поддерживающих аппаратное обеспечение (например, драйверов). Загрузка модулей может быть как динамической (без перезагрузки операционной системы), так и статической (выполняемой при перезагрузке). Большинство современных ядер, такие как OpenVMS, Linux, FreeBSD, NetBSD и Solaris, позволяют во время работы динамически (по необходимости) подгружать и выгружать модули, выполняющие часть функций ядра.

Модули ядра работают в адресном пространстве ядра и могут пользоваться всеми функциями, предоставляемыми ядром. Поэтому модульные ядра продолжают оставаться монолитными. Модульность ядра осуществляется на уровне бинарного образа, а не на архитектурном уровне ядра, так как динамически подгружаемые модули загружаются в адресное пространство ядра и в дальнейшем работают как его часть.

Имеется вариант монолитной архитектуры, когда код ядра операционной системы строится как иерархия уровней. Такая архитектура ядра называется *многоуровневой*. Уровни образуются группами функций, каждый уровень взаимодействует только со своими непосредственными соседями через строго определенные интерфейсы. Многоуровневая архитектура была предложена Эдсгером Дейкстра в проекте пакетной системы THE в 1968 году.

В системе THE функции ядра распределялись по уровням следующим образом. Уровень 0 занимался распределением времени процессора, реализуя базовую многозадачность. Уровень 1 осу-

ществлял управление памятью через механизм виртуальной памяти. Уровень 2 отвечал за связь между консолью оператора и процессами. Уровень 3 управлял буферизацией и взаимодействием с устройствами ввода-вывода. На уровне 4 работали программы пользователя. Уровень 5 – это пользователь системы.

Многоуровневая архитектура операционной системы THE, в основном, служила средством разработки. Однако каждый уровень может быть наделён привилегиями и выполнять системный вызов с контролем параметров для обращения к другому уровню. В данной архитектуре уровни называются *кольцами защиты*. Многоуровневая архитектура с кольцами защиты была реализована в операционной системе Multics.

Проблемой многоуровневой архитектуры является множественность и размытость интерфейсов между слоями, так как сложно выполнить однозначную группировку функций по уровням. Основной проблемой монолитной архитектуры является низкая надежность. Она вызвана большим объемом критического к сбою кода, который сложно сопровождать. Также из-за того, что весь код работает в общем адресном пространстве, ошибка в одной из процедур может привести к повреждению разделяемых всеми процедурами данных и выходу системы из строя. Подобная ситуация может быть и при злонамеренном повреждении кода через модифицированные злоумышленниками подгружаемые модули ядра.

Монолитная архитектура имеет преимущества с точки зрения удобства организации взаимодействия между частями операционной системы. Оно организуется проще, так как может использовать глобальные структуры данных системы. По этой же причине и по причине отсутствия переключения режима процессора взаимодействие происходит быстрее. Это свойство существенно, например, для эффективной реализации файловых систем.

Клиент – серверная архитектура. Эта архитектурная модель предполагает наличие программного компонента потребителя сервиса, который называется *клиентом*, и программного компонента поставщика сервиса (*сервера*). Взаимодействие клиента и сервера стандартизировано. Инициатором обмена является клиент, который посылает запрос серверу, находящемуся в состоянии ожидания запроса.

Применительно к структурированию операционной системы подход состоит в разделении ее на несколько процессов-серверов, каждый из которых выполняет отдельный набор сервисных функций, например, управление памятью, планирование процессов. Серверные процессы работают в пользовательском режиме. Микроядро работает в привилегированном режиме и выполняет функции доставки сообщений нужному серверу и передачи результатов клиенту (рис. 8).

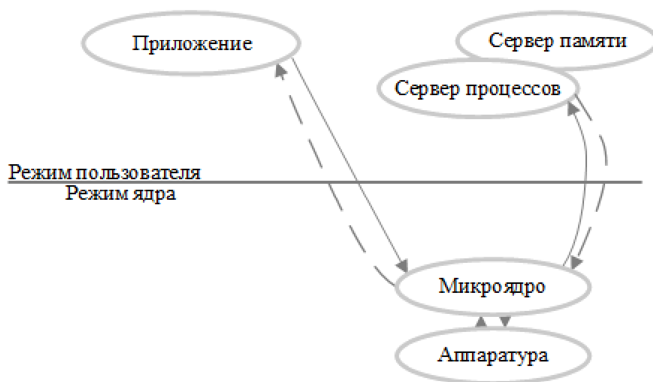

Рис. 8. Клиент-серверная архитектура

В представленной теоретической модели существует проблема: для реализации своих функций серверы должны иметь до-

ступ к аппаратуре. В зависимости от того как разрешается эта архитектурная проблема различают микроядерную и гибридную реализацию клиент-серверного подхода к построению ядра операционной системы.

Архитектурный подход при проектировании *микроядра* заключается в помещении в ядро только тех функций, которые можно исполнить в режиме супервизора (привилегированном режиме). Все остальные функции распределяются между серверами пользовательского режима. При этом используется парадигма разделения механизма и политики. Ядро отвечает за механизм реализации некоторого решения по управлению ресурсами, а сервер пользовательского режима отвечает за политику, то есть отвечает за принятие решений.

Пример 1. Очевидно, что запуск процесса требует привилегированного режима, так как связан с манипулированием аппаратно зависимыми структурами данных и передачей управления между процессами. Сам запуск процесса требует доступа к аппаратуре и выполняется ядром. Решение о приоритетах процессов, дисциплине постановки их в очередь может принимать работающий вне ядра планировщик. Пользователь может использовать тот планировщик, который подходит для его прикладной задачи.

Пример 2. Для управления памятью в системе имеется планировщик, который определяет стратегию замещения страниц и работает вне ядра как сервер пользовательского режима (pager). Все остальные аппаратно зависимые функции по управлению памятью реализуются в ядре.

Пример 3. У драйверов устройств можно выделить общий интерфейс, работающий в режимах ядра. В его функции входит работа с аппаратными прерываниями и доступ к управляющим регистрам

устройств ввода-вывода. Драйвер, работающий в режиме пользователя, например, драйвер СУБД, может включать оптимизацию под конкретный способ доступа к диску, работая на уровне сегментов диска, а не с файлами.

Преимуществом микроядерной архитектуры является высокая надежность и гибкость. Надежность обеспечивается тем, что возможные сбои локализуются в сервере режима пользователя и не затрагивают другие сервисы и ядро. Восстановление осуществляется перезапуском отказавшего сервиса без необходимости остановки всех процессов и перезапуска операционной системы. Само ядро, в силу малого объема кода, легко проанализировать на наличие ошибок. Например, размер кода ядра L4 составляет 14 килобайт и содержит 7 системных вызовов.

Микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с обменом сообщениями, что отрицательно влияет на производительность. Для того, чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется обеспечить правильное разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем – необходимость очень аккуратного проектирования.

Примерами микроядер являются Mach, L4, Minix, QNX. Наиболее известная операционная система, основанная на микроядре – Symbian OS.

Гибридное ядро. Этот архитектурный подход заключается в размещении аппаратно зависимых сервисов в режиме ядра (безопасность и управление процессами, памятью, вводом-выводом). При этом сохраняется ориентированный на сообщения механизм взаимодействия клиент-серверной архитектуры.

Основным преимуществом подхода является значительное повышение быстродействия системы. Это происходит потому, что не требуется частого переключения из режима ядра в режим пользователя при передаче сообщений между серверами, реализующими сложный системный вызов.

Недостатком является некоторая потеря гибкости и надежности. Такие известные семейства операционных систем, как Windows NT и Mac OS X построены на основе гибридных ядер, соответственно, NT kernel и XNU.

Оригинальное решение проблемы производительности с сохранением высокой надежности и гибкости микроядерного подхода предложено Microsoft и развивается в рамках проекта операционной системы Singularity. В данной операционной системе серверные процессы и ядро исполняются на одном уровне привилегий. Защита процессов производится не путем организации аппаратно-защищенных адресных пространств, а путем использования промежуточного языка и его верификации перед компиляцией в код процессора.

Объектно-ориентированные архитектуры операционных систем. Преимущества объектной модели (инкапсуляция, наследование, полиморфизм) используются также при написании операционных систем.

Можно выделить три способа применения объектов. Код ядра пишется на процедурном языке, а объекты моделируются средствами языка, например, Си. Такой подход используется в ядре Windows NT. Так пользователю доступны описатели (типа HANDLE) для работы с объектами ядра. Имеются полиморфные функции (например, CloseHandle(), выполняющая очистку произвольного объекта ядра). Сокрытие реализации объектов ядра выполняется аппаратными механизмами защиты.

Код операционной системы может быть реализован непосредственно на объектно-ориентированном языке программирования. Например, язык Objective C используется в операционных системах Mac OS X и iOS.

Шире всего объектно-ориентированные технологии применяются в программном обеспечении промежуточного уровня - программной прослойке между операционными системами и прикладным программным обеспечением. Примером данного вида программного обеспечения Microsoft является архитектура COM (component object model). На основе COM были реализованы технологии Microsoft: OLE Automation, ActiveX, DCOM, COM+, DirectX. В COM используется IDL для описания интерфейсов компонента и язык реализации C++. Кроссплатформенным аналогом технологии COM является архитектура CORBA, позволяющая организовать взаимодействие систем, написанных на разных языках программирования и работающих на разных операционных системах.

Виртуальная машина. Концепция виртуальной машины была разработана в конце 60-х годов в исследовательском центре IBM (Кембридж, Массачусетс) в процессе работы над операционной системой CP/CMS. Последующая коммерческая реализация VM/CMS используется в компьютерах IBM до настоящего времени.

Проект изначально разрабатывался как альтернатива OS/360. Эта система разделения времени обеспечивает с одной стороны многозадачность, а с другой - расширенную машину. Эти функции можно полностью разделить. Монитор VM работает с оборудованием и обеспечивает многозадачность. Но представленная монитором машина не является расширенной: она полностью идентична оборудованию. На ней может работать любая операционная система, которая работает на аппаратуре, в том числе сама VM. Эта

операционная система, в свою очередь, реализует расширенную машину. Такая архитектура оказывается проще и надежнее обычной многозадачной операционной системы OS/360.

На использовании виртуализации основаны интенсивно развивающиеся в настоящее время «облачные» вычисления. Например, одним из основных решений для сглаживания неравномерности нагрузки «облачного» поставщика услуг является размещение слоя серверной виртуализации между слоем программных услуг и аппаратным обеспечением. Виртуализация позволяет реализовать балансировку нагрузки посредством программного распределения виртуальных серверов по реальным компьютерам без остановки вычислений.

Экзоядро развивает идею виртуальной машины. Основная идея операционной системы на основе экзоядра состоит в том, что ядро должно выполнять лишь функции координатора для небольших процессов, связанных только одним ограничением: экзоядро должно иметь возможность гарантировать безопасное выделение и освобождение ресурсов оборудования. В отличие от операционных систем на основе микроядра, экзоядра обеспечивают большую эффективность за счет отсутствия необходимости в переключении между процессами при каждом обращении к оборудованию.

Наноядро — архитектура ядра операционной системы компьютеров, в рамках которой крайне упрощённое ядро выполняет лишь одну задачу — обработку аппаратных прерываний, генерируемых устройствами компьютера. Наиболее часто в современных компьютерах наноядра используются для виртуализации аппаратного обеспечения реальных компьютеров или для обеспечения возможности запуска «старой» операционной системы на новом, несовместимом аппаратном обеспечении без её полного переписывания.

2.3. Вопросы

1. Функциональные требования, предъявляемые к операционным системам, и способы их реализации. Расширяемость. Переносимость. Надежность. Совместимость. Безопасность. Производительность.
2. Основные архитектуры операционных систем: монолитные, многоуровневые, микроядерные, объектно-ориентированные, виртуальные машины.

3. ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

3.1. Обзор основных свойств

Обзор основных понятий языка Си в форме вопрос-ответ: классы памяти, область видимости, типы литералов (констант), escape-символы, размер типа, тип void, возвращение результата функции, операции с указателями, доступ к элементам массива, typedef, инструкции выбора, инструкции циклов, goto, массив в качестве аргумента функции, функция в качестве аргумента функции, порядок сборки программы, препроцессор.

Какие классы памяти используются в языке Си?

В языке Си используются автоматический и статический классы памяти. Автоматические объекты локальны в блоке, при выходе из него значения таких объектов теряются, так как они размещаются во фрейме вызова в стеке. Статические объекты могут быть локальными в блоке или располагаться вне блоков, но в обоих случаях их значения сохраняются после выхода из блока (или функции) до повторного входа в него. Они размещаются в памяти, инициализируемой при загрузке программы по фиксированному адресу. Объект (переменная) - часть памяти с двумя главными характеристиками: класс памяти (время жизни памяти, связанной с объектом), тип (род значений, хранящихся в объекте).

Какие правила определяют область видимости имен?

Существуют два вида областей видимости имен: первая область – это лексическая область идентификатора, область в тексте программы, где имеют смысл все его характеристики; вторая область – это область, ассоциируемая с объектами и функциями, имеющими внешние связи, устанавливаемые между идентификаторами из раздельно компилируемых единиц трансляции.

Лексическая область видимости определяется по следующим принципам. Идентификатор объекта (функции) во внешнем объявлении виден от места объявления и до конца единицы трансляции. Параметр функции – в пределах всего тела функции. Переменная внутри функции – от места объявления до конца тела функции. Идентификатор в блоке – от места объявления и до конца блока. Метка - во всем теле функции, где встречается эта метка.

Если идентификатор явно объявлен в начале блока (в том числе тела функции), то любое объявление того же идентификатора, находящееся снаружи этого блока, временно перестает действовать вплоть до конца блока.

Как синтаксически выражается принадлежность переменной к классу памяти?

Возможны следующие варианты выражения принадлежности переменной к классу памяти. Если переменная объявлена внутри блока, а спецификатор не задан - это автоматический объект. Внутри блока со спецификатором **`auto`** - автоматический объект. Внутри блока со спецификатором **`register`** - автоматический объект, который по возможности располагается в регистре машины. Внутри блока со спецификатором **`static`** - статический объект. Вне всех блоков при незаданном спецификаторе - статический глобальный объект. Вне всех блоков со спецификатором **`extern`** - статический глобальный объект (атрибут внешней связи). Вне всех блоков со спецификатором **`static`** - статический локальный (в пределах модуля) объект (атрибут внутренней связи).

Как можно определить константу (литерал) целого типа?

Константа считается восьмеричной, если начинается с **`0`** (цифры ноль); шестнадцатеричной, если начинается с символов **`0x`** или **`0X`**, и десятичной во всех остальных случаях. Если добавить

суффикс **u** или **U**, то константа будет беззнаковой; суффикс **l** или **L** – типа long. Тип константы определяется по следующим правилам. Берется первый из перечисленных типов, который годится для ее представления:

- десятичная без суффикса - **int**, **long int**, **unsigned long int**;
- восьмеричная или шестнадцатеричная без суффикса - **int**, **unsigned int**, **long int**, **unsigned long int**;
- с суффиксом **U** - **unsigned int**, **unsigned long int**;
- с суффиксом **L** - **long int**, **unsigned long int**;
- с суффиксом **UL** - **unsigned long int**.

Что такое escape-символы?

Escape - символы, определенные в языке Си

описание символа	обозначение ASCII	запись в Си
новая строка	**NL или LF**	**\n**
горизонтальная табуляция	**HT**	**\t**
вертикальная табуляция	**VT**	**\v**
возврат на шаг	**BS**	**\b**
возврат каретки	**CR**	**\r**
перевод страницы	**FF**	**\f**
сигнал звонка	**BEL**	**\a**
обратная наклонная черта	****	****
знак вопроса	**?**	**\?**
одиночная кавычка	**'**	**\'**
двойная кавычка	**"**	**\"**
восьмеричный код	**Ooo**	**\ooo**
шестнадцатеричный код	**Hh**	**\xhh**

Escape - символы (escape - последовательности) - это коды текстовых знаков, которые в данной программе имеют специальное управляющее назначение и потому не могут быть записаны напрямую. Они используются, например, при необходимости добавить в строковую константу символа двойных кавычек, который является управляющим и обозначает конец объявления строки.

Как определяется константа (литерал) символьного типа, как такая константа будет храниться в памяти?

Константа символьного типа заключается в одиночные кавычки (`'x'`). Один символ будет храниться как численное значение этого символа в кодировке, принятой на данной машине (например, ASCII). Хранение нескольких символов зависит от реализации.

Символьная константа принимает тип **`char`** или **`wchar_t`**. Последний тип определяет расширенную константу и используется для расширенного набора символов, который не может быть охвачен типом **`char`**. Чтобы ввести расширенную константу, нужно записать перед ней букву **`L`**: **`L'x'`**.

Как определяется строковый литерал. Как поступают, если он не умещается на одной строке?

Строковый литерал определяется как последовательность символов, заключающихся в двойные кавычки (**`" "`**). Строка имеет тип «массив символов» и память класса **`static`**. Размер строки определяется нулевым символом **`'\0'`**, который символизирует ее конец.

Объединить несколько строк в один строковый литерал можно двумя способами: либо написав в конце строки обратную наклонную черту (поскольку символ \ и следующий за ним символ новой строки выбрасываются препроцессором), либо закрыв двойные кавычки в конце одной строки и вновь открыв их на другой.

Как определить размер типа?

Размер типа определяется с помощью оператора **`sizeof «тип»`**, который возвращает размер памяти, выделяемый под объект данного типа в байтах.

Что такое тип `void`, как он используется?

Тип **`void`** обозначает отсутствие значения, а поэтому применять его можно там, где значение не требуется. Например, как тип возвращаемого значения функции он явным образом подчеркивает факт того, что результат функции отбрасывается.

Указатель на тип **`void`** можно присваивать, передавать в качестве параметра и результата функции, но операции косвенного обращения и адресной арифметики с ним недопустимы. Кроме того, он играет роль обобщенного указателя: указатель на любой тип данных можно привести к типу **`void*`**, а затем обратно без потери данных.

Это используется в функциях, работающих с любым типом данных. Например, WINAPI - функция вторичного потока

```
DWORD WINAPI ThreadFunc(LPVOID);
```

получает в качестве параметра обобщенный указатель, который затем можно явно привести к нужному типу в теле функции.

Как можно вернуть результат, вычисленный функцией?

Результат, вычисленный функцией, можно вернуть либо через инструкцию **`return`** (предварительно указав в определении функции тип возвращаемого значения), либо передав в функцию указатель на объект (и, тем самым, получить возможность работать с ним, а не с его локальной копией).

Какие операции применимы к указателям?

В языке Си можно производить следующие операции с указателями:

- присваивание значения указателя другому указателю того же типа;
- сложение и вычитание указателя и целого;

- вычитание и сравнение двух указателей, указывающих на элементы одного и того же массива;
- присваивание указателю нуля и сравнение указателя с нулем.

В язык С++, помимо этого, добавлена возможность присваивания значения одного указателя другому, даже если эти указатели имеют различный тип. При этом производится неявное преобразование типа.

Напишите функцию, обменивающую значения двух переменных типа `int`.

Функция может выглядеть следующим образом.

```
void swap(int *a, int *b){
  int v=0; v=*a; *a=*b; *b=v;
}
```

Переменные передаются в функцию по ссылке. Обмен производится через временную вспомогательную переменную. Вызов имеет вид: `swap(&x,&y)`.

Как осуществляется доступ к элементам массива?

К i-тому элементу массива можно обратиться либо явно по номеру элемента: `a[i]`, либо с использованием адресной арифметики: `*(a+i)`. Последний вариант возможен, поскольку переменная массива есть указатель на его нулевой элемент.

Для чего используется ключевое слово `typedef`?

Ключевое слово `typedef` используется для задания произвольных имен (typedef - имен) существующим типам взамен или в дополнение к стандартным.

Например, после

```
typedef long Blockno, *Blockptr;
typedef struct {double r, theta;} Complex;
```

допустимы следующие объявления:

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

Переменная **b** принадлежит типу **long**, **bp** – типу «указатель на **long**»; **z** – это структура заданного вида, а **zp** принадлежит типу «указатель на такую структуру».

Перечислите инструкции выбора языка Си, как они работают.

1) Инструкция **if-else** - принятие решения:

```
if (выражение) инструкция1 else инструкция2;
```

2) Инструкция **else-if** - многоступенчатое принятие решения:

```
if (выражение)
   инструкция
else if (выражение)
   инструкция
else if (выражение)
   инструкция
else
   инструкция;
```

3) Переключатель **switch**:

```
switch (выражение) {
   case константное выражение:
        инструкции
          break;/*если требуется выйти из
          switch*/
   case константное выражение:
        инструкции
```

```
        break;/*если требуется выйти из
        switch*/
    default: инструкции
}
```

Особенностью **switch** является то, что для выполнения только одной ветви case необходимо вставлять в нее **break**, иначе инструкции продолжают выполняться последовательно, и возможен проход по нескольким вариантам выбора.

Так как в местах, где по синтаксису полагается одна инструкция, иногда возникает необходимость выполнить несколько, предусматривается возможность задания составной инструкции (которую также называют блоком). Тело определения функции есть составная инструкция.

Перечислите инструкции циклов языка Си, как они работают.

1) Цикл с предусловием **while**:

```
while(выражение)
    инструкция;
```

2) Цикл с постусловием **do-while**:

```
do
    инструкция
while(выражение);
```

Является аналогом **repeat-until** в Паскале (инструкция будет выполнена хотя бы один раз). Единственное отличие: постусловие является условием продолжения, а не завершения цикла.

3) Цикл **for**:

```
for(выражение1;выражение2;выражение3)инструкция;
```

относится к циклам с предусловием, поскольку аналогичен следующему коду:

```
выражение1;
while (выражение2) {инструкция; выражение3;}
```

Отличие от цикла **for** из Паскаля в том, что в Си индекс и его предельное значение могут изменяться внутри цикла, и значение индекса после выхода из цикла всегда определено. Кроме того, все три компоненты цикла могут быть произвольными выражениями, поэтому применение **for**-циклов не ограничивается только случаем арифметической прогрессии.

Где может понадобиться инструкция goto?

Наиболее типичный случай применения инструкции **goto** – прерывание обработки в некоторой глубоко вложенной структуре и выход сразу из двух или большего числа вложенных циклов. Инструкция **break** здесь не поможет, так как она обеспечит выход только из самого внутреннего цикла. В качестве примера можно взять следующую конструкцию:

```
for(...)
for(...){
   /* если бедствие, уйти на ошибку */
   if (disaster) goto error;
}
error: /* обработка ошибки */
```

Другой пример: необходимо определить, есть ли в двух массивах совпадающие элементы. При использовании цикла **for** инструкция **goto** позволяет избежать дополнительных проверок условий:

```
for(i=0;i<n;i++)
for(j=0;j<m;j++)if(a[i]==b[i]) goto found;
   /* нет одинаковых элементов */
found:
   /* обнаружено совпадение:a[i]==b[i] */
```

В других случаях следует избегать использования инструкции безусловного перехода по метке, так как это ухудшает читаемость программы.

Как передать массив в качестве аргумента функции?

Массив в качестве аргумента функции можно передавать только по ссылке, например:

```
void sorting(int *a,int n);
```

или

```
void sorting(int a[],int n);
```

с последующим вызовом

```
sorting(a,n); .
```

Если требуется передать массив по значению, то необходимо заключить его в какую-либо структуру и передать через нее, поскольку структуры, в отличие от массивов, можно передавать по значению.

Как передать функцию в функцию, как выглядят объявления и вызов переданной функции?

В языке Си функции нельзя определять внутри других функций, а сама функция является не переменной, а константой (адресом первой машинной инструкции в коде функции). Однако можно определить указатель на функцию и работать с ним, как с обычной переменной, в том числе и передавать в качестве параметра функции.

Примером может служить функция из обобщенного алгоритма сортировки:

```
void qsort(void*lineptr[],
   int left, int right,
   int (*comp)(void*,void*)
);
```

В качестве своих аргументов функция **qsort** ожидает массив указателей, два целых значения и функцию с двумя аргументами типа указатель на **void**. Внутри тела функции **qsort** мы сможем работать с функцией **comp** как с обычной переменной, использовав ее, например, в проверке условия **if**:

```
if((*comp)(v[i],v[left])<0)/*перестановка*/ .
```

Как производится сборка программы из файлов. Перечислите основные стадии этого процесса?

Сборка программы из файлов осуществляется следующим образом. В первую очередь исходный код обрабатывается препроцессором, который включает нужные файлы и осуществляет макроподстановки. Полученный код компилируется в объектный файл. Заключительный этап проводится линковщиком, который устанавливает в объектном файле все внешние связи. Конечный результат зависит от типа создаваемой программы (библиотека, исполняемый файл и другие возможные варианты).

Также следует отменить, что проводить полную сборку из-за одного незначительного изменения было бы неэффективно, поэтому язык Си разбивает исходный код на единицы компиляции и при повторной сборке перекомпилирует только те единицы, которые изменились.

Как работает препроцессор языка Си, как исполняются директивы #include, #define, #ifdef (#ifndef)?

Препроцессор языка Си работает на первом шаге компиляции и осуществляет:

1) Включение файла. Любая строка вида

```
#include "имя-файла"
```

или

```
#include <имя-файла>
```

заменяется содержимым файла с именем «имя-файла».

2) Макроподстановку.

```
#define имя замещающий текст
```

Во всех местах, где встречается лексема «имя», вместо нее будет помещен замещающий текст.

3) Условную компиляцию.

```
#if !defined(HDR)
#define HDR
   /* здесь содержимое hdr.h */
#endif
```

Условная компиляция – это выборочное включение того или иного текста программы в зависимости от значения условия, вычисляемого во время компиляции (схоже с инструкциями if-else). Она позволяет управлять ходом препроцессирования.

Само препроцессирование проистекает в нескольких логически последовательных фазах (в отдельных реализациях некоторые фазы объединены):

1) Трехзнаковые последовательности заменяются их эквивалентами. Между строками вставляются символы новой строки, если того требует операционная система.
2) Выбрасываются пары символов, состоящие из обратной наклонной черты с последующим символом новой строки, тем самым осуществляется «склеивание» строк.
3) Комментарии заменяются единичными пробелами. Затем выполняются директивы препроцессора и макроподстановки.
4) Escape-последовательности в символьных константах и строковых литералах заменяются на символы, которые они обозначают. Соседние строковые литералы конкатенируются.

Результат препроцессирования транслируется в объектный

код. Затем устанавливаются связи с другими программами и библиотеками посредством сборки необходимых программ и данных и соединения ссылок на внешние функции и объекты с их определениями.

3.2. Указатели и адресная арифметика

Графическое представление состояния памяти и семантика операторов * и &, семантика выражений, многократная разадресация, typedef для адресных типов, указатель на функцию, выделение памяти в куче, указатели и массивы, висячие указатели, утечка памяти.

Язык программирования Си был разработан в начале 70-х годов прошлого века и является основой таких языков, как C++, C#, Java, JavaScript, Go и других. Поэтому сейчас, как правило, студенты, приступающие к изучению операционных систем, уже знакомы с производными от Си языками. Однако язык Си обладает особенностями, значительно отличающими его от перечисленных современных языков. Это связано с тем, что Си используется, прежде всего, как замена ассемблера при написании кода операционных систем, драйверов, системного программного обеспечения и должен напрямую эффективно работать с памятью. Для этого в языке Си используется особая семантика выражений, операторы для получения адресов переменных в памяти и обращения к содержимому памяти по адресам. При этом язык Си – это язык высокого уровня со статической типизацией. Ниже кратко рассмотрены основные особенности языка Си, связанные с указателями и адресной арифметикой. Для подробного изучения этих вопросов рекомендуется использовать книгу «Язык программирования Си» Б. Кернигана и Д. Ритчи и другие источники.

Базовыми операторами при работе с памятью являются оператор получения (взятия) адреса переменной **&** и оператор обращения к переменной по её адресу ***** (разадресации). Кроме использования в выражениях, оператор ***** также применяется в объявлениях типов переменных-указателей. Указатели – это специальные переменные, содержащие адрес других переменных.

Семантику (смысловое содержание) переменных и операторов можно пояснять с использованием графического представления ленты памяти, как показано на рис. 9. Переменные Си – это одна или несколько ячеек памяти, расположенных по смежным адресам на ленте памяти. Переменные имеют адрес – младший из адресов ячеек памяти, входящих в состав переменной. Переменные имеют тип, который определяет: (а) количество ячеек памяти, занимаемых переменной; (б) способ интерпретации содержимого этих ячеек (какую информацию содержат 8-ми битные ячейки памяти, составляющие переменную); (в) допустимые операции с переменной в выражениях, определяемые механизмом статического контроля типов языка Си.

```
int a; int*y; a=10; y=&a;
```

Рис. 9. Графическое представление семантики операций с указателями

Для обозначения адреса переменной, но без привязки к конкретному положению переменной на ленте памяти, используются стрелки, указывающие на прямоугольник, обозначающий переменную.

Некоторые переменные имеют имена, по которым к переменным обращаются в выражениях. Заметим, что не все переменные имеют имена, а только переменные, размещаемые в статической и автоматической памяти программы. Переменные, размещаемые в куче (динамической памяти), имен не имеют. К таким переменным обращаются косвенно по их адресам.

Некоторые переменные являются указателями, то есть содержат адреса других переменных. Если переменным присвоены обычные значения, например, числа, то их вписывают в прямоугольники-символы переменных. Когда же переменные, в соответствии со своим типом, содержат адреса других переменных, то внутри прямоугольников-символов переменных рисуются основания стрелок, обозначающих адреса переменных как показано на рис. 9. Представленный рисунок поясняет смысл, то есть описывает семантику следующего кода на языке Си:

```
int a; int*y; a=10; y=&a;
```

или в сокращенном виде

```
int a = 10, *y = &a; .
```

Заметим, что оператор **`*`** в объявлении переменной при обработке компилятором ассоциируется с именем переменной **`y`**, а не именем типа **`int`**. Кроме того, оператор **`*`** является разделителем, то есть его не нужно обособлять от соседних лексем пробелами или другими пробельными символами и можно писать слитно с именем типа, именем переменной или даже сплошным текстом, как показано выше.

Объявление переменной в языке Си можно представлять как особое уравнение, которое задает тип переменной следующим способом: переменная имеет такой тип, что применение к этому типу операторов из объявления даст в итоге базовый (примитивный) тип

Си, указанный в начале объявления. Поясним это на примере переменной **y**. Тип этой переменной – **int***. Если к выражению типа **int*** применить оператор *****, то тип результирующего выражения будет **int**, что и записано в объявлении переменной **y**.

Типы переменных нужно уметь правильно записывать, так как они являются аргументами операторов приведения типов **(тип)**. Общий способ получения типа переменной из объявления: взять объявление, исключить из него имя переменной и разделитель ‘;’. Оставшиеся символы – это тип переменной.

Вернемся к основным операторам работы с адресами **&** и *****. Следует понимать, что это обычные унарные операторы, аргументом которых являются выражения, а не только идентификаторы переменных. С другой стороны, любые выражения языка Си обозначают некоторое значение или переменную. Если выражение обозначает переменную, то в зависимости от оператора, применяемого к выражению, оно может пониматься как место помещения значения или как само значение, содержащееся в переменной. Про такие выражения говорят, что они являются «левосторонними», или **lvalue**. Название связано с тем, что такие выражения могут использоваться в левой части оператора присваивания. Термин **lvalue** можно встретить в сообщениях компилятора об ошибках, например, при попытке присвоить значение другому значению.

Оператор ***** ожидает в качестве аргумента выражение, обозначающее адрес переменной. Адрес – это разновидность значений. В частном случае адрес содержится в переменной-указателе. Результатом выполнения оператора ***** является переменная «целиком», то есть тип выражения ***(выражение)** всегда **lvalue**.

В некоторых случаях, например при разработке драйверов устройств, адрес может быть передан в оператор ***** непосредственно своим числовым значением. Так в случае рис. 9 выражение ***(int*)8** обозначает переменную **a**.

Оператор **&** ожидает в качестве аргумента выражение типа **lvalue**. В частном случае это выражение – имя переменной. Результатом выполнения оператора **&** является адрес переменной.

Описанные свойства операторов языка Си можно легко проверить экспериментально, например с использованием онлайн компилятора. Если определить переменную **int z;** с учетом записанного выше кода для переменных **a** и **y** будут показаны следующие значения для выражений:

```
(z=*y)        →    10
&(z=*y)       →    0x7f4e6e697294
&z            →    0x7f4e6e697294,
```

где 0x7f4e6e697294 некоторый адрес переменной **z** в шестнадцатеричном представлении, зависящий от конкретного размещения этой переменной в памяти.

Из примера также видно, что значением выражения с оператором присваивания является переменная **z** в левой части присваивания. Благодаря этому свойству работают цепочечные присваивания вида **xx = yy = zz = 0;** .

Действия операторов **&** и ***** противоположно, но следует учитывать, что результатом применения оператора **&** является значение, а не **lvalue**. Например, ***(&a) = 10;** и **a = 10;** – одинаковые выражения, но **&(*y)** не тоже самое, что **y**, хотя оба выражения при распечатке покажут одинаковое значение адреса. Дело в том, что для выражения **&(*y)** не допустимо присваивание: компиляция выражения вида **&(*y) = <выражение>;** приведет к ошибке.

Адресные выражения можно разадресовывать несколько раз. Соответствующие объявления типов выглядят следующим образом и читаются как «двойной указатель» или «указатель на указатель» на переменную типа **int**.

```
int **x; x=&y;
**x              →    10
```

Для улучшения читаемости двойных указателей и других более сложных типов в языке Си имеется возможность определения синонимов типов **typedef**. После объявления

```
typedef int* INT_POINTER;
```

типы **int*** и **INT_POINTER** являются эквивалентными и можно объявить новую переменную так

```
INT_POINTER w = y;
```
.

Для объявления двойного указателя можно поступить так

```
typedef int** INT_DOUBLE_POINTER;
```

или так

```
typedef INT_POINTER* INT_DOUBLE_POINTER;
```
.

Синонимы типов ведут себя также как и встроенные типы и могут использоваться в тех же самых контекстах.

Особым видом указателей являются указатели на функции. Значением таких указателей считается адрес первой команды в теле функции. Обратите внимание на синтаксические и семантические различия определения, объявления функции и объявления указателя на функцию:

```
int func() { return 42; }     ;
int func();                   ;
int (*func)();                .
```

При обработке определения функции компилятором выделяется память для хранения команд, составляющих тело функции, в примере это оператор **return 42;** . На функцию можно сослаться далее по тексту программы.

При обработке объявления функции память не выделяется, так как предполагается, что она будет или уже выделена при обработке определения функции. На функцию можно сослаться далее по тексту программы.

При обработке определения указателя функции память выделяется только для хранения адреса первой инструкции функции. Используя этот адрес, далее по тексту можно сослаться на функцию, адресом которой проинициализирован указатель.

Перед использованием указатель нужно проинициализировать адресом некоторой известной функции **int some_func();**

```
func = &some_func;
```

Имя функции в выражении понимается как адрес её первой инструкции, поэтому применение оператора **&** не является обязательным. После инициализации можно использовать указатель для вызова функции. Для этого применяется оператор разадресации ***** в следующей форме

```
int result = (*func)();
```

Однако допустим и обычный синтаксис вызова функции, как если бы идентификатор **func** обозначал функцию, а не переменную-указатель на функцию.

Для выделения памяти под переменные в куче в языке Си нет специальных синтаксических конструкций. В отличие от большинства языков для этого используются библиотечные функции. Например, синтаксическая конструкция языка С++

```
int* a = new int[5];
```

соответствует такому коду на языке Си:

```
int* a = (int*)malloc(sizeof(int)*5);
```

Обратите внимание, что в приведенном выше коде на Си не производится заполнение нулями выделяемой памяти, а приведение типа не является обязательным. Функция **malloc()** выделяет в куче блок памяти такого размера, чтобы в нее помещались пять переменных типа **int**. Размер одной переменной типа **int** возвращает специальный оператор **sizeof**. Оператор позволяет определить размер, требуемый для типа или переменной, указанных в качестве его параметра.

Функция **malloc()** возвращает указатель специального типа **void***. Это особый тип указателей, не допускающих разадресацию. Разадресация возможна только после явного приведения указателя к конкретному адресному типу, после чего компилятор понимает, как интерпретировать содержимое памяти, на которую указывает такой указатель.

После инициализации переменная **a** из примера выше указывает на первую переменную типа **int** в массиве из пяти последовательно расположенных переменных типа **int**. Для того, чтобы указать на любую из переменных в данном массиве, а не только на первую, также чтобы получить доступ к значениям переменных массива используется адресная арифметика. Например, выражение **a+1** обозначает адрес второй переменной в массиве, а выражение ***(a+1)** обозначает саму вторую переменную в данном массиве. Имеют смысл операции сложения/вычитания адреса с целым значением, инкремента/декремента, разности адресов, которые и называются операторами адресной арифметики. Бессмысленной в любых контекстах является операция сложения адресов.

Операции адресной арифметики можно интерпретировать графически (рис. 10), если представить, что указатель указывает не на одну переменную, а на ячейку на ленте памяти, содержащей кроме данной ячейки множество ячеек такого же типа как в области старших, так и в области младших адресов. Перемещение на следующую ячейку в область старших адресов для указателя типа **T*** соответствует увеличению адреса на **sizeof(T)** байт.

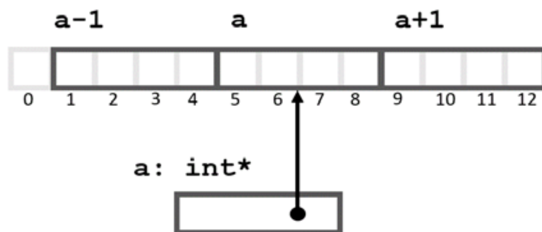

Рис. 10. Графическое представление семантики операций с указателями

Операции с массивами в динамической памяти в языке Си синтаксически унифицированы с операциями с массивами, выделяемыми в статической или автоматической памяти. Например, для обращения ко второму элементу массива можно использовать эквивалентное выражение **`a[1]`** . Однако следует иметь ввиду, что смысл объявления массива **`int a[5];`** с точки зрения использования памяти совершенно другой, чем при рассмотренном выше случае выделения памяти в куче, как показано на рис. 11. Имя массива не является **`lvalue`** и не допускает присваивание. Заметим, что для целей унификации выражения **`a &a &a[0]`** обозначают адрес первой ячейки в массиве **`int a[5];`**.

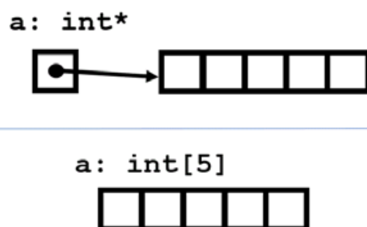

Рис. 11. Схема размещения указателя на массив и массива в памяти

Из-за того, что память, выделяемая под автоматические переменные, высвобождается при возврате из функций, а память, выделенная в кучи, напротив не освобождается; также потому, что не га-

рантируется правильное значение указателя в момент его разадресации при работе с указателями в языке Си возможны следующие состояния ошибок.

При работе с некоторыми компиляторами Си, которые не выдают ошибку отсутствия инициализации, может произойти разадресация неинициализированного указателя, так как инициализация переменных при объявлении в языке Си не является обязательной. Например, данный код может компилироваться, но при выполнении записывать число 5 по произвольному адресу.

```
void wild_ptr(){int *x; *x = 5;}
```

Другие два примера иллюстрируют случаи «висячего» указателя.

```
void dangling_ptr_1()
{ int *x=malloc(100); free(x); *x=6; }
```

Память была выделена в куче, затем освобождена при помощи вызова специальной функции освобождения памяти **free()**. После такого вызова, несмотря на то, что указатель **x** хранит определенное значение адреса, использовать этот адрес для записи значений уже нельзя, так как область памяти стала недоступна. Возможно непредсказуемое нарушение целостности программы.

Другой пример висячего указателя, когда из области видимости выходит переменная в автоматической памяти, адрес которой был запомнен в указателе.

```
int* dangling_ptr_2()
{ int x[5]; return x; }
```

В данном случае адресное значение, возвращенное функцией, нельзя разадресовывать, так как в функции, вызвавшей **dangling_ptr_2()**, переменной **int x[5]** уже не существует.

Еще одной проблемой является «утечка памяти». Когда указатель на память в куче выходит из области видимости, память на которую он указывает может оказаться недоступной. По мере работы

программы объем недоступной из-за утечек памяти растет. В некоторый момент запрос на выделение новой памяти в кучи завершается ошибкой.

```
void mem_leak()
{ int *x=malloc(100); }
```

Игнорирование данных проблем в дизайне языка Си делает его гибким и быстрым, позволяет использовать некоторые приемы программирования, необходимые в низкоуровневом системном коде. Однако возникающие ошибки являются трудно выявляемыми, так как приводят к повреждению случайного кода и данных программы, а что еще более опасно, к повреждению механизма выполнения программы. В некоторых современных языках программирования, например в языке Rust, выполняется полный контроль со стороны системы программирования и выполнения языка над возникновением такого рода ошибок.

3.3. Изучение синтаксиса

Цель выполнения лабораторной работы, требования к решению, порядок сдачи, перечень вариантов заданий к лабораторной работе.

Целью данной лабораторной работы является изучение синтаксиса языка Си, основных приемов программирования при реализации простого алгоритма. При выполнении работы следует обратить внимание на правильное применение управляющих инструкций и операторов, форматирование кода программы.

Решение оформляется в виде функции main() или сплошным кодом при реализации в онлайн компиляторах. Результирующее приложение – консольная программа, запрашивающая у пользователя входные данные и выдающая результат в текстовом виде. До-

пускается подготовить входные данные программы непосредственно в коде. Задание выполняется индивидуально.

Отчет по заданиям лабораторной работы выполняется путем предъявления кода программы преподавателю и демонстрации ее работы. Могут быть заданы дополнительные вопросы по возможным вариантам реализации алгоритма и применению синтаксических конструкций языка Си. Проверенный и зачтенный код включается в дальнейшем в итоговый отчет по лабораторному практикуму.

Вариант №1. Дан двумерный целочисленный массив A(2,10). Известно, что среди его элементов два и только два равны между собой. Напечатать их индексы. Не брать для сравнения одну и ту же пару элементов дважды.

Вариант №2. Составить программу вывода всех трехзначных десятичных чисел, сумма цифр которых равна данному целому числу M.

Вариант №3. В массиве A(N) каждый элемент равен 0, 1 или 2. Переставить элементы массива так, чтобы сначала располагались все нули, затем все двойки и, наконец, все единицы.

Вариант №4. Напечатать все простые числа, не превосходящие заданное число M.

Вариант №5. В написанном выражении ((((1?2)?3)?4)?5)?6 вместо каждого знака «?» вставить знак одной из четырех арифметических операций +, –, *, / так, чтобы результат вычислений равнялся 35. При делении дробная часть в частном отбрасывается. Достаточно найти одно решение.

Вариант №6. Дан одномерный массив. Все его элементы, не равные нулю, переписать, сохраняя их порядок в начало массива, а нулевые элементы – в конец массива.

Вариант №7. Натуральное число называется совершенным, если оно равно сумме всех своих собственных делителей, включая

1. Напечатать все совершенные числа, меньшие, чем заданное M.

Вариант №8. Заданы три числа D, M, Y, которые обозначают число, месяц и год. Найти номер N этого дня с начала года. Обратите внимание на метод определения високосного года.

Вариант №9. Последовательность чисел определяется следующим образом: первое число – произвольное натуральное число, кратное 3; любое следующее число в последовательности равно сумме кубов всех цифр предыдущего числа. Любая такая последовательность становится постоянной, равной 153. Напишите программу, проверяющую это утверждение.

Вариант №10. Дан одномерный массив положительных вещественных чисел. Преобразовать этот массив следующим образом. Сначала обнуляется минимальный элемент. Затем максимальный элемент из оставшихся элементов. Далее минимальный из оставшихся и так до тех пор, пока не останется единственный элемент. Вывести на экран значение и индекс оставшегося элемента.

3.4. Изучение функций

Цель выполнения лабораторной работы, требования к решению, порядок сдачи, порядок оформления отчета.

Целью данной лабораторной работы является изучение синтаксиса и семантики языка Си, связанных с функциями. При выполнении работы следует обратить внимание на правильное разделение кода на функции, отличие объявлений и определений функций, на правильное применение объявлений функций. Также следует поэкспериментировать со способами передачи и возврата значений из функций.

Решение оформляется в виде кода в одном файле, но включаю-

щем несколько функций. Результирующее приложение – консольная программа, запрашивающая у пользователя входные данные и выдающая результат в текстовом виде. Допускается подготовить входные данные программы непосредственно в коде. Задание выполняется индивидуально.

Отчет по заданиям лабораторной работы выполняется путем предъявления кода программы преподавателю и демонстрации ее работы. Могут быть заданы дополнительные вопросы по работе с функциями в языке Си. Проверенный и зачтенный код в дальнейшем включается в итоговый отчет по лабораторному практикуму.

В качестве варианта задания к лабораторной работе берется ранее выполненный вариант лабораторной работы п. 3.3. Код программы перерабатывается с целью осмысленного разделения его на функции.

3.5. Работа с динамической памятью

Цель выполнения лабораторной работы, требования к решению, порядок сдачи, порядок оформления отчета, перечень вариантов заданий к лабораторной работе.

Целью данной лабораторной работы является изучение методов работы с динамической памятью (кучей) средствами библиотеки времени выполнения языка Си, а также операторов языка C++. При выполнении лабораторной работы следует обратить внимание на определение размера запрашиваемой в куче памяти, синтаксис приведения типов указателей, методы конструирования многомерных массивов, способы обращения к элементам массивов, очистке (освобождению) динамической памяти, когда она больше не используется в программе.

Решение оформляется в виде кода в одном файле. Желательно оформить код, выделяющий и освобождающий динамическую память, в виде функций. Результирующее приложение – консольная программа, запрашивающая у пользователя входные данные и выдающая результат в текстовом виде. Допускается подготовить входные данные программы непосредственно в коде. Задание выполняется индивидуально.

Отчет по заданиям лабораторной работы выполняется путем предъявления кода программы преподавателю и демонстрации ее работы. Могут быть заданы дополнительные вопросы по работе с указателями, по разадресации указателей на элементы массива. Проверенный и зачтенный код включается в дальнейшем в итоговый отчет по лабораторному практикуму, а также используется в следующей лабораторной, посвященной взаимодействию с программным интерфейсом операционной системы для запроса динамической памяти.

Вариант №1. Найти наибольший и наименьший элементы в динамическом массиве размерностью N×M.

Вариант №2. Необходимо каждый элемент строки разделить на сумму элементов строки в динамическом массиве размерностью N×M.

Вариант №3. Необходимо каждый элемент строки разделить на наибольший элемент строки в динамическом массиве размерностью N×M.

Вариант №4. По динамическому массиву из M вещественных чисел необходимо рассчитать выборочное среднее и выборочную дисперсию.

Вариант №5. Динамический массив размерности N×M необходимо дополнить (M+1)-й строкой и (N+1)-м столбцом, в кото-

рых записать суммы элементов соответствующих строк и столбцов. В элементе (M+1, N+1) должна храниться сумма всех элементов массива.

Вариант №6. В динамическом массиве размерности N×M необходимо в каждой строке найти элемент с наименьшим значением, а затем среди этих чисел найти наибольшее. На экран вывести индексы этого элемента.

Вариант №7. Транспонировать матрицу, размещенную в динамическом массиве размерности N×N, не используя дополнительного массива.

Вариант №8. В динамическом массиве размерности N×M необходимо найти номер строки и номер столбца, в которых находится наименьший элемент.

Вариант №9. Создать динамический массив из M строк по N символов каждая. Необходимо вывести только те строки, которые являются палиндромами, то есть читаются одинаково слева направо и справа налево.

Вариант №10. Реализовать код для перемножения двух матриц, размещенных в динамических массивах размерности N×N.

3.6. Задачи

Задачи на самопроверку знания основ языка Си. Решение записывается от руки или в онлайн-компиляторе языка Си или C++, например, *https://www.onlinegdb.com/*, *https://cpp.sh/*, *https://coliru.stacked-crooked.com/*.

1. Имеется переменная типа double. Напишите выражение для вычисления первой цифры справа от десятичной точки, считая, что значение переменной записано в формате с фиксированной точкой.
2. Имеется переменная типа double. Напишите выражение для вычисления второй цифры слева от десятичной точки в представлении значения этой переменной в формате с фиксированной

точкой. Результат запишите в переменную типа char. Если такой цифры в представлении числа нет, то присвойте переменной значение '_'.

3. Найдите все простые числа от 2 до 1000.
4. Найдите все простые делители числа в диапазоне от 2 до 1000.
5. Запишите код для вычисления произведения двух матриц.
6. Напишите функцию, вычисляющую максимальный элемент в массиве. Запишите различные способы передачи массива в функцию и доступа к элементам в самой функции.
7. Создайте в динамической памяти и заполните значением 1.0 нижнетреугольную матрицу размером 10×10, выделив память только под используемые элементы.
8. Напишите функцию, выполняющую сортировку произвольного массива. Критерий упорядочения элементов передать как параметр функции.

Примеры тестовых экзаменационных задач на проверку знаний указателей и адресной арифметики. Требуется выбрать один правильный ответ из числа приведенных в вопросе. При устном ответе на экзаменационные задачи далее нужно пояснить, почему остальные ответы являются ошибочными. При наличии возможности для пояснений используются онлайн-компиляторы языков Си или С++. Так же для пояснений ответа можно использовать графические обозначения переменных-указателей в виде прямоугольников, а адресов в виде стрелок.

Вопрос 1	Имеется фрагмент кода на языке Си { …; x = *y; …}. Выберите правильный вариант объявления и инициализации переменных x и y.	Балл
1	int x=0, *y=0;	
2	int x=0, *y=&x;	
3	int *x=0, y=0;	
4	int x=0, y=0;	

Вопрос 2	Имеется фрагмент кода на языке Си {…; int x[5]={0},*p; p=x+1; …} . Какой смысл имеет значение переменной p	Балл
1	Ноль	
2	Единица	
3	первый элемент массива x	
4	указатель на второй элемент массива x	

Вопрос 3	Имеется фрагмент кода на языке Си { …; *x = y; …}. Выберите правильный вариант объявления и инициализации переменных x и y.	Балл
1	int x=0, *y=0;	
2	int *x=0, y=0;	
3	int *x=(int*)malloc(sizeof(int)), y=0;	
4	int x=0, *y=&x;	

Вопрос 4	Имеется фрагмент кода на языке Си {…; int x; int y[3]={1,2,3}; x=*(y+2); …} . Какое значение имеет переменная x после его выполнения.	Балл
1	0	
2	1	
3	2	
4	3	

Вопрос 5	Имеется фрагмент кода на языке Си { …; x = &y; …}. Выберите правильный вариант объявления и инициализации переменных x и y.	Балл
1	int x, *y=0;	
2	int *x, *y=0;	
3	double *x, y=1.0;	
4	double x, *y=1.0;	

Вопрос 6	Имеется фрагмент кода на языке Си {…; int **x, *y,z=0; x=&y; y=&z; …} . Какое значение имеет переменная x после его выполнения.	Балл
1	указатель на ноль	
2	указатель на указатель, указывающий на ноль	
3	указатель на x	
4	двойной указатель на x	

Вопрос 7	Имеется фрагмент кода на языке Си { …; *x = *y; …}. Выберите правильный вариант объявления и инициализации переменных x и y.	Балл
1	int z=0,*x=&z,*y=&z;	
2	int x,y=0;	
3	int z=0,*x=&z,*y=&x;	
4	int *x,*y=0;	

Вопрос 8	Имеется фрагмент кода на языке Си {…; char* s=”123”;char c=*(s+3); …} . Какое значение примет переменная с после его исполнения.	Балл
1	'1’	
2	‘2’	
3	‘3’	
4	0	

Задачи на знание базовых свойств указателей и операторов, используемых для работы с ними. Дана функция, записанная на языке Си, в которой пропущены некоторые фрагменты. Требуется восстановить пропущенные фрагменты таким образом, чтобы получившийся код был синтаксически и семантически корректным.

а)

```
void func(){
  /* объявления и инициализация пропущены */
  x=*y;
  /* вывод пропущен */
}
```

б)

```
void func(){
  /* объявления и инициализация пропущены */
  *x=y;
  /* вывод пропущен */
}
```

в)

```
void func()
{
  /* объявления и инициализация пропущены */
  x=&y;
  /* вывод пропущен */
}
```

Задачи на понимание семантики выражений, использующих указатели. Объясните смысл выражений с указателями: воспользовавшись графическим представлением переменных, значений и указателей изобразите состояние памяти после вычисления выражений в каждом случае.

а) `int x[5],*p; p=x+1;`

б) `int x=1; int*y=&x;`

в) `int x;int y[3]={1,2,3}; x=*(y+2);`

г) `int **x,*y,z=0;x=&y;y=&z;`

д) `int r1[2]={11,12}; int r2[2]={21,22}; int *m[2]={r1,r2};`

е) `char* s=”123”;char c=*(s+3);`

Задачи на работу с массивами, размещенными в динамической памяти. Напишите код на языке Си для выделения памяти под двумерный массив чисел типа int в динамической памяти (куче). Далее запишите код функции, на вход которой подаются указатель на этот массив, индексы элемента в массиве. Функция возвращает значение элемента массива, соответствующее переданным индексам. Индексация элементов с нуля. Варианты размещения:

а) построчное размещение элементов в памяти;

б) постолбцовое размещение элементов в памяти;

в) древовидное размещение (массив указателей на одномерные массивы).

———

СПИСОК ЛИТЕРАТУРЫ

1. Гордеев, А.В. Операционные системы: учеб. для вузов по направлению подготовки бакалавров и магистров «Информатика и вычисл. техника» и направлению подготовки дипломированных специалистов «Информатика и вычислительная техника» / А. В. Гордеев. – 2-е изд. – СПб: Питер: 2009. – 416 с.

2. Олифер, В.Г. Сетевые операционные системы: учеб. пособие для вузов по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника» / В. Г. Олифер, Н. А. Олифер. – 2-е изд. – СПб: Питер: Питер Пресс, 2009. – 669 с.

3. Столлингс, В. Операционные системы. Внутреннее устройство и принципы проектирования / В. Столлингс; пер. с англ. – 9-е изд. – М: Диалектика, 2020. – 1264 с.

4. Бэкон, Джин. Операционные системы. Параллельные и распределенные системы / Дж. Бэкон, Т. Харрис; пер. с англ. – СПб: Питер, 2004. – 799 с.

5. Таненбаум, Э. Современные операционные системы / Эндрю Таненбаум, Херберт Бос. – 4-е изд. – СПб: Питер: Питер Пресс, 2022. – 1120 с. – (Классика computer science).

6. Карпов, В.Е. Основы операционных систем: курс лекций: учебное пособие: для вузов по специальностям в области информационных технологий / В. Е. Карпов, К. А. Коньков; под ред. В. П. Иванникова; Интернет-университет информационных технологий. – М.: ИНТУИТ. РУ, 2005. – 536 с.

7. Дейтел, Х.М. Операционные системы: [в 2 т.] / Х.М. Дейтел, П.Д. Дейтел, Д.Р. Чофнес; под ред. С.М. Молявко; пер. с англ. – 3-е изд. – М.: Бином, 2016 – Т. 1: Основы и принципы. – 2016. – 1024 с.

8. Дейтел, Х.М. Операционные системы: [в 2 т.] / Х. М. Дейтел, П. Д. Дейтел, Д. Р. Чофнес; пер. с англ. ред. С. М. Молявко. – 3-е

изд. – М.: Бином, 2016. – Т. 2: Распределенные системы, сети, безопасность. – 2016. – 704 с.

9. Таненбаум, Э. Операционные системы. Разработка и реализация / Э. С. Таненбаум, А. Вудхалл; пер. с англ. – 3-е изд. – СПб.; М.; Нижний Новгород: Питер, 2007. – 704 с.

10. Руссинович, М. Внутреннее устройство Windows / М. Руссинович, Д. Соломон, А. Ионеску; пер. с англ. – 7-е изд. – СПб.; М.; Нижний Новгород: Питер, 2022. – 944 с. – (Классика computer science).

11. Уорд, Б. Внутреннее устройство Linux / Б. Уорд; пер. с англ. – 3-е изд. – СПб.; М.; Нижний Новгород: Питер, 2023. – 480 с.

12. Рихтер, Дж. Windows via C/C++. Программирование на языке Visual C++ / Дж. Рихтер, К. Назар. – М.: Питер, Русская Редакция, 2009. – 896 с.

13. Керниган, Б.У. Язык программирования С / Б.У. Керниган, Д.М. Ритчи. – 2-е изд. – М.: Диалектика, 2020. – 288 с.

14. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – М.: ДМК Пресс, 2016. – 272 с.

Учебное издание

Востокин Сергей Владимирович

АРХИТЕКТУРА ОПЕРАЦИОННЫХ СИСТЕМ

Учебное пособие

Редакционно-издательская обработка
издательства Самарского университета

Подписано в печать 14.04.2023. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 5,25.
Тираж 120 экз. (1-й з-д 1-27). Заказ № . Арт. – 13(Р1УП)/2023.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА».
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.