Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра «Информационные системы»

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Конспект лекций
для студентов специальности 5В070300-Информационные системы

Алматы 2014

Составитель: Б.Р. Абсатарова. Операционные системы. Конспект лекций для студентов специальности 5В070300-Информационные системы - Алматы: АУЭС, 2014. – 71 с.

Конспект лекций составлен согласно рабочей программе дисциплины «Операционные системы».

В данную методическую разработку включены следующие сведения: основные понятия, классификация, архитектура операционных систем, организация памяти компьютера, файловая система, система управления вводом-выводом, сетевые операционные системы.

Основные теоретические материалы дисциплины разбиты на пятнадцать тем, и каждая из этих тем представлена в виде конспекта отдельной лекции.

Конспекты лекций предназначены для студентов специальности 5В070300 - Информационные системы.

Ил.3, библиогр. – 25 назв.

Рецензент: кандидат технических наук, доцент Ни А.Г.

Печатается по плану издания НАО «Алматинский университет энергетики и связи» на 2014 г.

Введение

Современный компьютер - универсальное, многофункциональное автоматическое устройство обработки информации. Сегодня компьютеры принимают на себя основную часть функций по обработке данных. По историческим меркам компьютерные технологии все еще молоды и находятся в самом начале развития, преобразуя или вытесняя традиционные информационные технологии.

Программное обеспечение (ПО) компьютера разделяется на общесистемное и прикладное.

Операционная система (ОС), являясь основой общесистемного ПО, обеспечивает функционирование и взаимосвязь всех компонентов компьютера и предоставляет пользователю свободный доступ к его аппаратным возможностям.

Прикладное программное обеспечение в свою очередь делится на средства разработки и приложения. Средства разработки — это инструменты программиста, включающие алгоритмические языки программирования (ЯП), а также трансляторы (компиляторы). Приложения - программные продукты, предназначенные для решения задач в конкретной предметной области.

В конспекте лекции «Операционные системы» изучаются основополагающие принципы устройства операционных систем, возможности применения фундаментальных концепций от достигнутого технологического уровня и специфических требований к конкретной реализации, их взаимосвязь с различными новациями в этой области, а также с современными направлениями развития операционных систем.

1 Лекция № 1. Операционная система, основные функции. Эволюция развития операционных систем

Цель лекции: ознакомиться с назначением, основными функциями и классификацией операционных систем, а также получить представление о процессе эволюции операционных систем.

Содержание лекции: назначение и функции, эволюция развития, классификация ОС.

Операционная система - (англ. *operating system*) - базовый комплекс компьютерных программ, обеспечивающий управление аппаратными средствами компьютера, работу с файлами, ввод и вывод данных, а также выполнение прикладных программ и утилит.

При включении компьютера операционная система загружается в память раньше остальных программ и затем служит платформой и средой для их работы. Помимо вышеуказанных функций, ОС может осуществлять и другие, например, предоставление пользовательского интерфейса, сетевое взаимодействие и т. п.

С 1990-х наиболее распространёнными операционными системами для персональных компьютеров и серверов являются ОС семейства Microsoft Windows и Windows NT, Mac OS и Mac OS X, *системы* класса UNIX (особенно GNU/Linux).

Основные функции:

- загрузка приложений в оперативную память и их выполнение;
- стандартизованный доступ к периферийным устройствам (устройства ввода-вывода);
- управление оперативной памятью (распределение между процессами, виртуальная память);
- управление доступом к данным на энергонезависимых носителях (таких как жёсткий диск, компакт-диск и т. д.), как правило, с помощью файловой системы;
- пользовательский интерфейс;
- сетевые операции, поддержка стека протоколов.

Дополнительные функции:

- параллельное или псевдопараллельное выполнение задач (многозадачность);
- взаимодействие между процессами;
- защита самой системы, а также пользовательских данных и программ;
- от злонамеренных действий пользователей или приложений;
- разграничение прав доступа и многопользовательский режим работы (аутентификация, авторизация).

Известно, что компьютер был изобретен английским математиком Чарльзом Бэбиджем в конце восемнадцатого века. Его "аналитическая

машина" так и не смогла по-настоящему заработать, потому что технологии того времени не удовлетворяли требованиям по изготовлению деталей точной механики, которые были необходимы для вычислительной техники. Известно также, что этот компьютер не имел операционной системы. Некоторый прогресс в создании цифровых вычислительных машин произошел после второй мировой войны. В середине 40-х были созданы первые ламповые вычислительные устройства. В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. Не было никакого другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм.

С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы - полупроводниковых элементов. Компьютеры второго поколения стали более надежными, теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Именно в этот период произошло разделение персонала на программистов и операторов, эксплуатационников и разработчиков вычислительных машин. В эти годы появились первые алгоритмические языки, а следовательно, и первые системные программы - компиляторы. Стоимость процессорного времени возросла, что потребовало уменьшения непроизводительных затрат времени между запусками программ. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом. В ходе реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий.

Следующий важный период развития вычислительных машин относится к 1965-1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало гораздо большие возможности новому, третьему поколению компьютеров. Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством

программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/производительность. Вскоре идея программно-совместимых машин стала общепризнанной. Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными "монстрами". Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, и содержали тысячи ошибок, вызывающих нескончаемый поток исправлений. В каждой новой версии операционной системы исправлялись одни ошибки и вносились другие. Однако несмотря на необозримые размеры и множество проблем, OS/360 и другие ей подобные операционные системы машин третьего поколения действительно удовлетворяли большинству требований потребителей. Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования [1,2].

Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку, и наступила эра персональных компьютеров. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11, но вот цена у них существенно отличалась. Если миникомпьютер дал возможность иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер сделал это возможным для отдельного человека. Компьютеры стали широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения, это положило конец кастовости программистов. На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС UNIX доминировала в среде "неинтеловских" компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров. В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС. В сетевых ОС пользователи должны быть осведомлены о наличии других компьютеров и должны делать логический вход в другой компьютер, чтобы воспользоваться его ресурсами, преимущественно файлами. Каждая машина в сети выполняет свою

собственную локальную операционную систему, отличающуюся от ОС автономного компьютера наличием дополнительных средств, позволяющих компьютеру работать в сети.

Рассмотрев назначение ОС и выполняемые ими функции, можно классифицировать все многообразие существующих ОС, взяв за основу наиболее общие классификационные принципы. По количеству одновременно существующих программных процессов ОС делятся на однопрограммные и мультипрограммные. В мультипрограммных ОС, в отличие от однопрограммных, вычислительный процесс организуется таким образом, что в памяти компьютера находятся одновременно несколько программ, попеременно выполняющихся на одном процессоре.

По числу пользователей, осуществляющих доступ к вычислительной системе, различают *однопользовательские* и *многопользовательские* ОС. Многопользовательские системы предоставляют возможность одновременного доступа к вычислительной системе нескольким пользователям. При этом каждый из них работает за своим терминалом, однако все вычисления производятся на одном компьютере.

По назначению ОС делятся на *универсальные* и *специализированные*. Специализированные ОС работают с фиксированным набором программ.

По способу загрузки можно выделить *загружаемые* ОС и системы, *постоянно находящиеся в памяти* вычислительной системы. Последние, как правило, используются для управления работой специализированных устройств. По особенности области использования ОС подразделяются на системы пакетной обработки, системы разделения времени и системы реального времени.

Системы пакетной обработки предназначаются в основном для решения задач вычислительного характера, не требующих быстрого получения результата.

Системы разделения времени организуют вычислительный процесс таким образом, что каждой задаче выделяется квант процессорного времени, вследствие чего ни одна задача не занимает процессор надолго, и это дает возможность пользователю вести диалог со своей программой.

Системы реального времени используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно.

2 Лекция № 2. Архитектура операционных систем

Цель лекции: ознакомиться с основными положениями, преимуществом и недостатками архитектуры ОС с микроядром.

Содержание лекции: «классическая» и многослойная архитектура ОС, основные положения, преимущества и недостатки архитектуры ОС с микроядром.

Ядро — центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, оперативная память, внешнее оборудование. Обычно предоставляет сервисы файловой системы.

Как основополагающий элемент ОС, ядро представляет собой наиболее низкий уровень абстракции для доступа приложений к ресурсам системы, необходимым для его работы. Как правило, ядро предоставляет такой доступ исполняемым процессам соответствующих приложений за счет использования механизмов межпроцессного взаимодействия и обращения приложений к системным вызовам ОС. Описанная задача может различаться в зависимости от типа архитектуры ядра и способа ее реализации.

Монолитное ядро предоставляет богатый набор абстракций оборудования. Все части монолитного ядра работают в одном адресном пространстве. Старые монолитные ядра требовали перекомпиляции при любом изменении состава оборудования. Большинство современных ядер позволяют во время работы подгружать *модули*, выполняющие части функции ядра.

Достоинства: скорость работы, упрощённая разработка модулей.

Недостатки: поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы.

В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется монолитным ядром (monolithic kernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме.

Таким образом, монолитное ядро - это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой. Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список

оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция - это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность операционной системы в целом.

Монолитным ядром является еще и Linux. Оно оптимизировано для более высокой производительности с минимальными контекстными переключениями. Такая архитектура упрощает поддержку кода ядра для разработчиков, но требует перекомпиляции ядра при добавлении новых устройств. Следует отметить, что описанные здесь различия являются «классическими», на практике монолитные ядра могут поддерживать модульность (что зачастую и происходит), а микроядра могут требовать перекомпиляции. *Примеры:* Традиционные ядра UNIX, такие как BSD и Linux; ядро MS-DOS.

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними (*многоуровневые системы* (Layered systems)), так чтобы объекты уровня N могли вызывать только объекты уровня N-1.

Нижним уровнем в таких системах обычно является hardware, верхним уровнем - интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни: интерфейс пользователя, управление вводом-выводом, драйвер устройств связи оператора и консоли, управление памятью, планирование задач и процессов, Hardware.

Главной особенностью подхода архитектуры построения ОС с микроядром является то, что в привилегированном режиме остается работать только очень малая часть ОС, называемая соответственно *микроядром*. Микроядро защищено от остальных частей ОС и пользовательских приложений. Набор входящих в состав микроядра функций, как правило, соответствует слою базовых механизмов обычного ядра, хотя в состав микроядра включаются далеко не все базовые функции ядра, а только функции управления процессами, обработки прерываний, управления виртуальной памятью, пересылки сообщений, управления устройствами ввода-вывода. Выполнение таких функций ОС практически невозможно реализовать в пользовательском режиме. Все машинно-зависимые модули ОС также включаются в микроядро. Не вошедшие в состав микроядра высокоуровневые функции и модули ядра оформляются в виде обычных приложений, работающих в пользовательском режиме.

Работающие в пользовательском режиме менеджеры ресурсов имеют принципиальные отличия от традиционных утилит ОС и системных обрабатывающих программ ОС, хотя при микроядерной архитектуре все эти программные компоненты также оформлены в виде приложений. Утилиты и обрабатывающие программы вызываются в основном пользователями. Ситуации, когда одному приложению требуется выполнение функции другого приложения, возникают крайне редко, поэтому в ОС с классической архитектурой отсутствует механизм, с помощью которого одно приложение могло бы вызвать функции другого. Принципиально другая ситуация возникает в том случае, если в форме обычного приложения оформляется часть ОС. По определению, основным назначением такого приложения является обслуживание запросов других приложений, например создание процесса, выделение памяти, проверка прав доступа к ресурсу и т.д. Вследствие этого вынесенные в пользовательский режим менеджеры ресурсов являются *серверами ОС*, то есть модулями, основное назначение которых - обслуживание запросов локальных приложений и других модулей ОС. Совершенно очевидно, что при реализации микроядерной архитектуры необходимо обеспечить наличие в ОС удобного и эффективного способа вызова процедур одного процесса из другого. Поддержка такого механизма является одной из главных задач, возложенных на микроядро ОС.

Микроядро работает в привилегированном режиме и никогда не удаляется из памяти. Обратиться к нему можно только посредством прерывания [3].

Защищенные подсистемы Windows NT работают в пользовательском режиме и создаются Windows NT во время загрузки ОС. Среди защищенных подсистем можно выделить подкласс, называемый подсистемами окружения.

Операционные системы основанные на концепции микроядра, во многом удовлетворяют требованиям, предъявляемым к современным ОС, обладая переносимостью, расширяемостью, надежностью, и создают хорошие предпосылки для поддержки распределенных приложений. Но, к сожалению, все эти достоинства ОС приобретает в ущерб производительности.

Как уже было сказано ранее, микроядерной ОС присуща изолированность всего машинно-зависимого кода в микроядре, вследствие чего для переноса системы на новую аппаратную платформу требуется минимальное количество изменений, причём все они логически сгруппированы. Эта особенность ОС с микроядром является их несомненным достоинством.

Особенности архитектуры микроядерных ОС обеспечивают их высокую степень расширяемости по сравнению с другими ОС. Добавление очередной подсистемы требует только лишь разработки нового приложения и никоим образом не затрагивает целостность микроядра. Микроядерная структура позволяет не только увеличивать, но и сокращать число компонентов ОС, что также бывает необходимо.

При микроядерном подходе конфигурируемость не вызывает никаких проблем и не требует особых мер, достаточно изменить файл с настройками начальной конфигурации системы или же остановить ненужные больше серверы в ходе работы обычными для остановки приложений средствами.

Использование микроядерной модели повышает надежность ОС. Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти и таким образом защищен от других серверов ОС, чего не наблюдается а традиционной ОС, где все модули ядра влияют друг на друга. И если отдельный сервер терпит крах, то он может быть перезапущен без останова или повреждения остальных серверов ОС. Другим потенциальным источником повышения надежности ОС является уменьшенный объем кода микроядра по сравнению с классическим ядром - это снижает вероятность появления ошибок программирования.

Модель с микроядром хорошо подходит для поддержки распределенных вычислений, так как использует механизмы, аналогичные сетевым, - взаимодействие клиентов и серверов путем обмена сообщениями. Серверы микроядерной ОС могут работать как на одном, так и на разных компьютерах. При получении сообщения от приложения микроядро обработает его самостоятельно и передаст, локальному серверу или же перешлет его по сети микроядру, работающему на другом компьютере. Переход к распределенной обработке требует минимальных изменений в работе ОС - просто локальный транспорт заменяется на сетевой.

Все рассмотренные ранее свойства ОС с микроядерной архитектурой свидетельствовали о предпочтительности ее использования разработчиками. Однако, как уже говорилось, все эти качества приобретаются за счет снижения производительности ОС. Дело в том, что при классической организации ОС выполнение системного вызова сопровождается двумя переключениями режимов, а при микроядерной организации - четырьмя.

Таким образом, ОС на основе микроядра при прочих равных условиях всегда будет менее производительной, чем ОС с классическим ядром. Именно поэтому микроядерный подход не получил такого широкого распространения, которое ему предрекали.

3 Лекция №3. Совместимость операционных систем

Цель лекции: рассмотреть виды и способы реализации совместимости.

Содержание лекции: виды совместимости и способы реализации совместимости.

Конкретные архитектурные и функциональные особенности любой ОС непосредственно должны касаться лишь системных программистов и могут совершенно не быть известны рядовому пользователю. В то время как

некоторые идеи (например, объектно-ориентированный подход) находятся в ведении только разработчиков и лишь косвенно влияют на конечного пользователя, *концепция множественных прикладных сред* приносит пользователю долгожданную возможность выполнять на своей ОС программы, написанные для других ОС и процессоров. Свойство ОС, характеризующее возможность выполнения в ОС приложений, написанных для других ОС, называется *совместимостью***.**

Существует два принципиально отличающихся вида совместимости, которые не следует путать: *совместимость на двоичном уровне* и *совместимость на уровне исходных текстов.* Приложения обычно хранятся в компьютере в виде исполняемых файлов, содержащих двоичные образы кодов и данных. Двоичная совместимость достигается в том случае, если можно взять исполняемую программу, работающую в среде одной ОС, и запустить ее на выполнение в среде другой ОС.

Совместимость на уровне исходных текстов требует наличия соответствующих компиляторов в составе программного обеспечения компьютера, на котором предполагается использовать данное приложение, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция исходных текстов программ в новые исполняемые модули.

Таким образом, совместимость на уровне исходных текстов наиболее важное значение имеет для разработчиков приложений, в распоряжении которых находятся эти исходные тексты. Для конечных же пользователей практическое значение имеет только двоичная совместимость, так как только в этом случае они могут без специальных навыков и умений использовать программный продукт, поставляемый в виде двоичного исполняемого кода, в различных операционных средах и на разных компьютерах. Для пользователя, купившего в свое время пакет программ для MS-DOS, важно, чтобы он мог запускать этот привычный ему пакет без каких-либо изменений или ограничений на своей новой машине, работающей, например, под управлением Windows NT. Множественные прикладные среды как раз и обеспечивают совместимость данной ОС с приложениями, написанными для других ОС и процессоров, на двоичном уровне, а не на уровне исходных текстов.

Каким типом совместимости - двоичной или совместимостью исходных текстов обладает ОС, зависит от многих факторов. Самый значительный из них - архитектура процессора, на котором работает ОС. Чтобы достичь двоичной совместимости, достаточно соблюсти несколько следующих условий:

- вызовы функций API, которые содержит приложение, должны поддерживаться дайной ОС;
- внутренняя структура исполняемого файла приложения должна соответствовать структуре исполняемых файлов данной ОС.

Несравнимо сложнее достигнуть двоичной совместимости операционным системам, предназначенным для выполнения на процессорах, имеющих различающиеся архитектуры. Кроме соблюдения приведенных выше условий, необходимо также организовать эмуляцию двоичного кода. Для того чтобы компьютер смог интерпретировать машинные инструкции, которые ему изначально непонятны, на нем должно быть установлено специальное программное обеспечение - *эмулятор*.

Назначение эмулятора состоит в том, чтобы последовательно выбирать каждую двоичную инструкцию процессора, например, Intel, программным способом дешифровать ее, чтобы определить, какие действия она задает, а затем выполнять эквивалентную подпрограмму, написанную в инструкциях процессора, например, Motorola.

Тем не менее, существует несколько другой, гораздо более эффективный выход из описанной ситуации - использование так называемых прикладных программных сред. Одной из составляющих, формирующих программную среду, является набор функций интерфейса прикладного программирования API, которым ОС обеспечивает свои приложения. Для сокращения времени выполнения чужих программ прикладные среды имитируют обращения к библиотечным функциям. Эффективность данного подхода определяется тем, что большинство современных программ работают под управлением графических интерфейсов пользователя (GUI) типа Windows, UNIX при этом приложения, как правило, наибольшую часть времени тратят на выполнение, некоторых хорошо предсказуемых действий. Они непрерывно осуществляют вызовы библиотек GUI для манипулирования окнами и для других, связанных с GUI, действий. Именно эта особенность приложений позволяет прикладным средам компенсировать большие затраты времени, потраченные на покомандное эмулирование программы. Тщательно спроектированная программная среда имеет в своем составе библиотеки, имитирующие внутренние библиотеки GUI, но написанные на «родном» коде данной ОС. Таким образом, достигается существенное ускорение выполнения программ с API другой операционной системы. Для того чтобы отличить такой подход от более медленного процесса эмулирования кода по одной команде за раз, его называют *трансляцией*.

В настоящее время дополнительное программное обеспечение позволяет пользователям некоторых ОС запускать «чужие» программы (например, Mac OS и UNIX позволяют выполнять программы для DOS и Windows). Но в современном поколении ОС средства для выполнения «чужих» программ становятся стандартной частью системы. Выбор ОС не слишком сильно ограничивает выбор прикладных программ. Хотя столкновение пользовательских интерфейсов программ для Mac OS, Windows и UNIX на одном и том же экране заставит пользователя немного потрудиться, но все же множественные прикладные среды ОС скоро станут такими же стандартными, как мыши и меню.

При реализации множественных прикладных сред разработчики сталкиваются с противоречивыми требованиями. С одной стороны, задачей каждой прикладной среды является выполнение программы по возможности так, как если бы она выполнялась на «родной» ОС. Но потребности этих программ могут входить в конфликт с конструкцией данной ОС. Специализированные драйверы устройств не всегда отвечают требованиям безопасности, возможны конфликты между схемами управления памятью и оконными системами. Но самой большой потенциальной проблемой является производительность - прикладная среда должна выполнять программы с приемлемой скоростью. Этому требованию не могут удовлетворить широко используемые ранее эмулирующие системы. Для сокращения времени на выполнение чужих программ прикладные среды используют имитацию программ на уровне библиотек.

Стандартная многоуровневая структура ОС является основой наиболее очевидного варианта реализации множественных прикладных сред (см. рисунок 1). Операционная система ОС1 поддерживает кроме свои «родных» приложений приложения операционных систем ОС2 и ОС3. Для этого в ее составе имеются специальные приложения - прикладные программные среды, которые транслируют интерфейсы «чужих» операционных систем API ОС2 и API ОС3 в интерфейс своей «родной» операционной системы API ОС1. Так, например, в случае, если бы в качестве ОС2 выступала операционная система UNIX, а в качестве О - OS/2, для выполнения системного вызова создания процесса fork() в UNIX-приложении программная среда должна была обратиться к ядру операционной системы OS/2 с системным вызовом DosExecPgm().

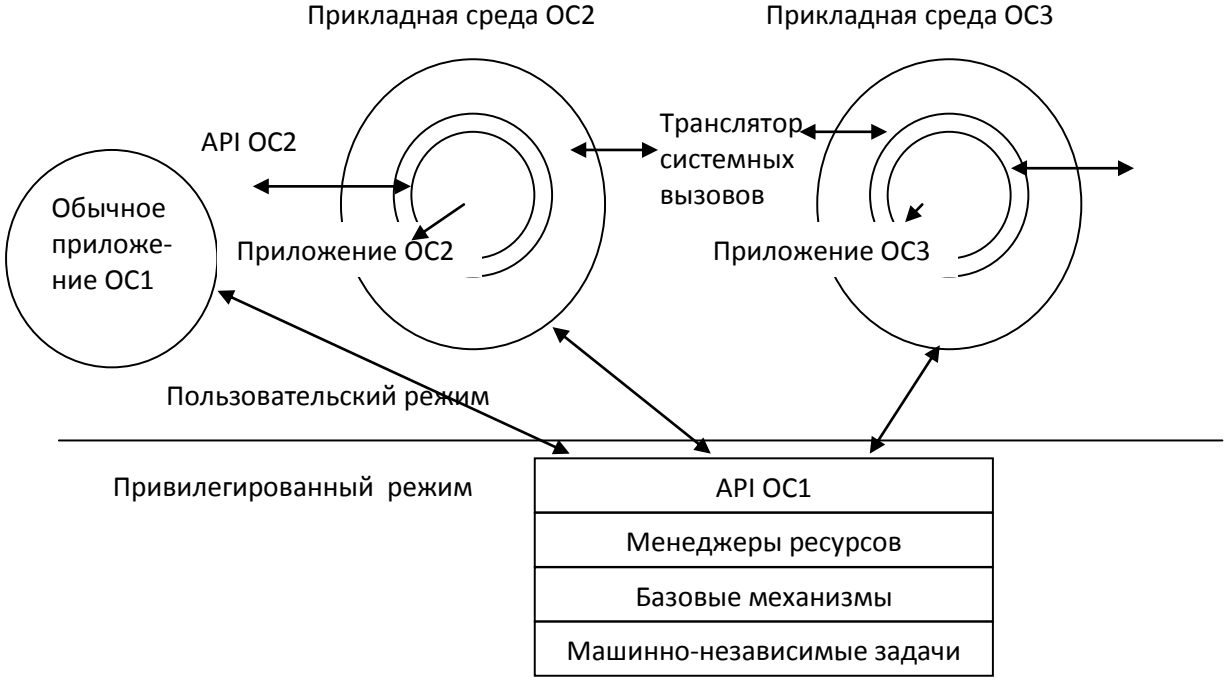

Рисунок 1 - Трансляция системных вызовов с помощью прикладных программных сред

Трудности при такой реализации возникают вследствие того, что поведение почти всех функций, составляющих API одной ОС, как правило, существенно отличается от поведения соответствующих (если они вообще есть) функций другой ОС. Например, чтобы функция создания процесса в OS/2 DosExecPgm() полностью соответствовала функции создания процесса fork() в UNIX-подобных системах, ее нужно изменить таким образом, чтобы она поддерживала возможность копирования адресного пространства родительского процесса в пространство процесса-потомка, хотя при нормальном поведении этой функции память процесса-потомка инициализируется на основе данных нового исполняемого файла.

Другой вариант реализации множественных прикладных сред подразумевает наличие в ОС нескольких равноправных прикладных программных интерфейсов. Это достигается путем размещения непосредственно в пространстве ядра системы прикладных программных интерфейсов всех этих ОС: API ОС1, API ОС2 и API ОС3. В этом варианте функции уровня API обращаются к функциям нижележащего уровня ОС, которые должны поддерживать все три в общем случае несовместимые прикладные среды.

В соответствии с микроядерной архитектурой все функции ОС реализуются микроядром и серверами пользовательского режима. Важно заметить, что каждая прикладная среда оформляется в виде отдельного сервера пользовательского режима и не включает базовых механизмов. Приложения, используя API, обращаются с системными вызовами к соответствующей прикладной среде через микроядро. Прикладная среда обрабатывает запрос, выполняет его (возможно, обращаясь для этого за помощью к базовым функциям микроядра) и отсылает приложению результат. В ходе выполнения запроса прикладной среде приходится, в свою очередь, обращаться к базовым механизмам ОС, реализуемым микроядром и другими серверами ОС.

Такому подходу к конструированию множественных прикладных средств присущи все достоинства и недостатки микроядерной архитектуры, в частности:

- очень просто можно добавлять и исключать прикладные среды, что является следствием хорошей расширяемости микроядерных ОС;
- надежность и стабильность выражаются в том, что при отказе одной из прикладных сред все остальные сохраняют работоспособность;
- низкая производительность микроядерных ОС сильно сказывается на скорости работы прикладных сред, а значит, и на скорости выполнения приложений [4].

4 Лекция № 4. Представления процесса в операционной системе

Цель лекции: ознакомиться с понятием и состоянием процесса.

Содержание лекции: понятие и состояния процесса.

Рассмотрим следующий пример. Если запустить программу извлечения квадратного корня, то, извлекая значения корня с разными исходными данными, компьютерная система должна заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. С точки зрения пользователя, запущена одна и та же программа, но сформированы два различных задания. Теперь другой пример. Пусть оба пользователя извлекают корень квадратный из 1, то есть они сформировали идентичные задания, но загрузили их в вычислительную систему со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения и ждет окончания операции ввода-вывода, второе только начинает исполняться.

Следовательно, «программа» и «задание» в пользовательском смысле не может применяться для описания, происходящего в вычислительной системе. Это происходит потому, что термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов «программа» и «задание» мы будем использовать новый термин – *«процесс»***.**

Процесс характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами.

Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в

непредусмотренных ситуациях (например, при обработке внешних прерываний).

При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди. Как видим, каждый процесс может находиться, как минимум, в двух состояниях: процесс исполняется, и процесс не исполняется. Процесс, находящийся в состоянии «процесс исполняется», через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние процесс не исполняется. Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии процесс не исполняется, и переводит его в состояние процесс исполняется. Новый процесс, появляющийся в системе, первоначально помещается в состояние процесс не исполняется. Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние процесс не исполняется на два новых состояния: готовность и ожидание (см. рисунок 2).

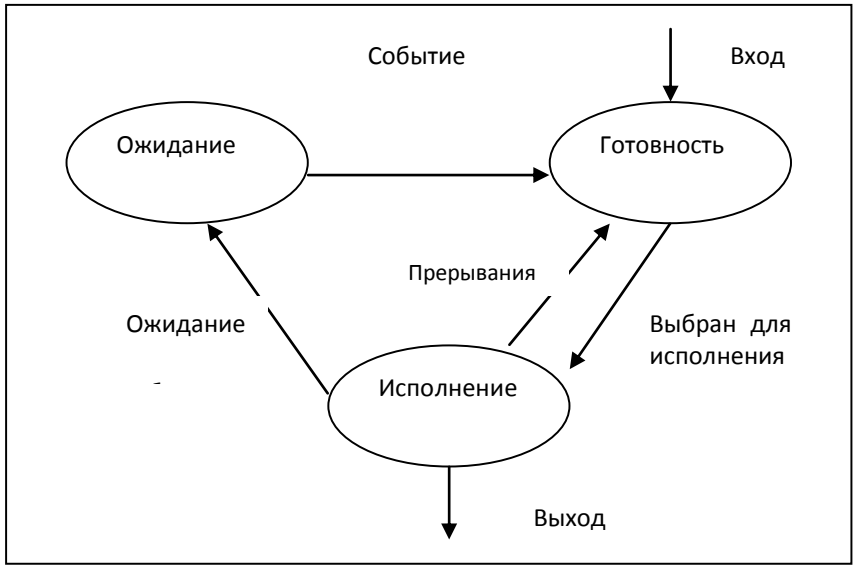

Рисунок 2 - Более подробная диаграмма состояний процесса

Всякий новый процесс, появляющийся в системе, попадает в состояние готовность. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние исполнение. В состоянии исполнение происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

- операционная система прекращает его деятельность;
- он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание;
- в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность.

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: рождение и закончил исполнение (см. рисунок 3).

Теперь для появления в вычислительной системе процесс должен пройти через состояние рождение. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние «готовность». При завершении своей деятельности процесс из состояния «исполнение» попадает в состояние «закончил исполнение».

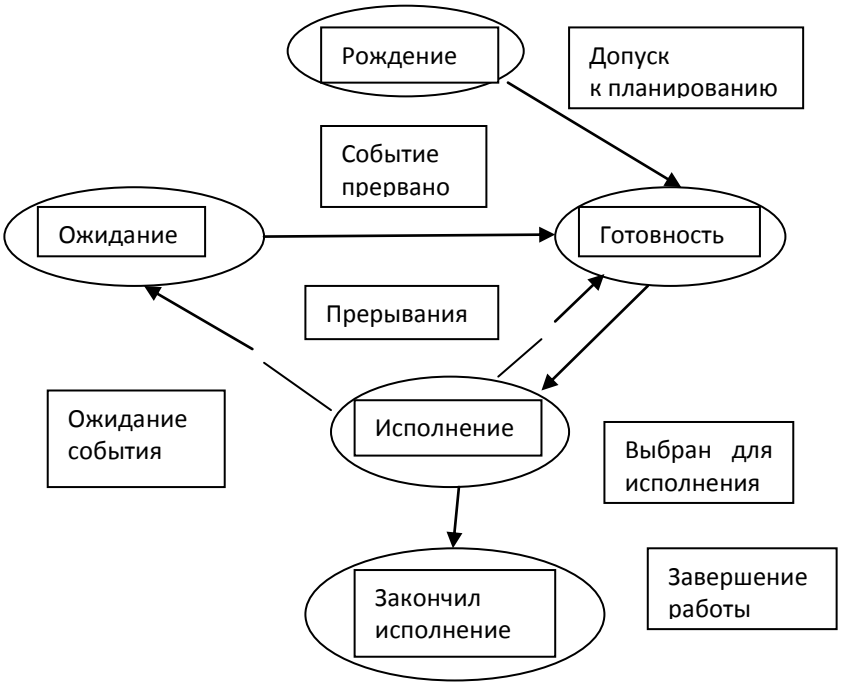

Рисунок 3 - Диаграмма состояний процесса, принятая в курсе

В конкретных операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9. Тем не менее, так или иначе, все операционные системы подчиняются изложенной выше модели.

5 Лекция №5. Операции над процессами и связанные с ними понятия

Цель лекции: рассмотреть Process Control Block и контекст процесса, а также операции над процессами.

Содержание лекции: Process Control Block и контекст процесса, одноразовые и многоразовые операции, переключение контекста.

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

- создание процесса - завершение процесса;
- приостановка процесса (перевод из состояния исполнение в состояние готовность) - запуск процесса (перевод из состояния готовность в состояние исполнение);
- блокирование процесса (перевод из состояния исполнение в состояние ожидание) - разблокирование процесса (перевод из состояния ожидание в состояние готовность).

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми. Рассмотрим подробнее, как операционная система выполняет операции над процессами.

Process Control Block и контекст процесса. Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
- содержимое регистров процессора;

- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);

- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);

- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для совершения операций над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее PCB (Process Control Block) или блоком управления процессом. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB. В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть регистровым контекстом процесса, а все остальное - системным контекстом процесса. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно для того, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его пользовательским контекстом. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

Одноразовые операции. Сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть процессом-родителем (parent process), а вновь созданный процесс – процессом-ребенком (child process). Процессы - дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов – генеалогический лес. Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево. При рождении процесса система заводит новый PCB с состоянием процесса рождение и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество битов, для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его освободившийся идентификационный номер может быть повторно использован для другого процесса.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в PCB.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система Unix разрешает порождение процесса только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой пользовательский контекст с помощью специального системного вызова. Операционная система VAX/VMS допускает только второе решение. В Windows NT возможны оба варианта (в различных API).

Многоразовые операции не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

Запуск процесса. Из числа процессов, находящихся в состоянии готовность, операционная система выбирает один процесс для последующего исполнения. Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. Далее состояние процесса изменяется на исполнение, восстанавливаются значения регистров для данного процесса и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

Блокирование процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния исполнение в состояние ожидание.

Разблокирование процесса. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый процесс в состоянии ожидание для данного события, и если находился, переводит его в состояние готовность, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.).

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций,

выполняемых над различными процессами, и сопровождается *переключением процессора* с одного процесса на другой.

При исполнении процессором некоторого процесса возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс, инициировавший запрос на ввод-вывод и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования. Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии исполнение.

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста.

6 Лекция №6. Планирование процессов

Цель лекции: ознакомиться с планированием различных процессов.

Содержание лекции: уровни, критерии, параметры, вытесняющее и невытесняющее планирования.

Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовность могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т. е. будет переведен в состояние исполнение, мы использовали это словосочетание. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида планирования мы будем рассматривать как различные уровни планирования процессов. Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения

ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования – долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20–30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена. Планирование использования процессора применяется в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования – краткосрочное. В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название swapping, что можно перевести на русский язык как «перекачка», хотя в специальной литературе оно употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов – среднесрочным [5,7].

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

1) *справедливость* - гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться;

2) *эффективность* - постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%;

3) *сокращение полного времени выполнения* (turnaround time) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением;

4) *сокращение времени ожидания* (waiting time) - сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки;

5) *сокращение времени отклика* (response time) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами:

- *были предсказуемыми*. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению корня квадратного из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске;

- *были связаны с минимальными накладными расходами.* Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит;

- *равномерно загружали ресурсы вычислительной системы*, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы;

- *обладали масштабируемостью*, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок**.**

Параметры планирования**.** Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям. К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические параметры системы описывают количество свободных ресурсов на данный момент.

К статическим параметрам процессов относятся характеристики, как правило, присущие заданиям уже на этапе загрузки:

- каким пользователем запущен процесс или сформировано задание;
- насколько важной является поставленная задача, т. е. каков приоритет ее выполнения;
- сколько процессорного времени запрошено пользователем для решения задачи;
- каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода;
- какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может использоваться следующая информация:

- сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
- сколько оперативной памяти занимает процесс;
- сколько процессорного времени уже предоставлено процессу.

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название CPU burst, а промежуток времени непрерывного ожидания ввода-вывода – I/O burst.

Процесс планирования осуществляется частью операционной системы, называемой планировщиком. Планировщик может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии готовность в следующих четырех случаях:

- когда процесс переводится из состояния исполнение в состояние завершил исполнение;
- когда процесс переводится из состояния исполнение в состояние ожидание;
- когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера);
- когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии исполнение, не может дальше исполняться, и для выполнения необходимо выбрать новый процесс. В случаях 3 и 4 планирование может не проводиться, процесс, который исполнялся до прерывания, может продолжать свое выполнение после обработки прерывания. Если планирование осуществляется только в случаях 1 и 2, говорят, что имеет место *невытесняющее* (nonpreemptive) планирование. В противном случае говорят о *вытесняющем* (preemptive) планировании.

7 Лекция №7 Алгоритмы планирования

Цель лекции: ознакомиться с различными алгоритмами планирования.

Содержание лекции: алгоритмы планирования (First-Come, First-Served, Round Robin, Shortest-Job-First), гарантированное и приоритетное планирование, многоуровневые очереди, многоуровневые очереди с обратной связью.

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования.

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой *FCFS* по первым буквам его английского названия – *First Come, First Served* (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии «готовность», выстроены в очередь. Когда процесс переходит в состояние «готовность», он, а точнее ссылка на его PCB, помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел). Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

Модификацией алгоритма FCFS является алгоритм, получивший название *Round Robin* (Round Robin – это вид детской карусели в США) или сокращенно *RR*. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта:

- время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново;

- продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Shortest-Job-First (SJF) - алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF-планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания времени очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания.

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который *гарантирует*, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: T_i – время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и τ_i – суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение T_i/N процессорного времени. Если

$$\tau_i << T_i/N,$$

то i-й пользователь несправедливо обделен процессорным временем. Если же

$$\tau_i >> T_i/N,$$

то система явно благоволит к пользователю с номером i. Вычислим для каждого пользовательского процесса значение коэффициента справедливости

$\tau_i N/T_i$

и будем предоставлять очередной квант времени процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом *гарантированного планирования*. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно многопроцессорного времени.

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При *приоритетном планировании* каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет [10,14].

Принципы назначения приоритетов могут опираться как на внутренние критерии вычислительной системы, так и на внешние по отношению к ней. Внутренние используют различные количественные и качественные характеристики процесса для вычисления его приоритета. Это могут быть, например, определенные ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Внешние критерии исходят из таких параметров, как важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени, и других политических факторов. Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут не запускаться неопределенно долгое время. Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии «готовность».

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность. Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем приоритет очередей пользовательских процессов. А приоритет очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться алгоритм FCFS, а для интерактивных процессов - алгоритм RR. Подобный подход, получивший название *многоуровневых очередей (Multilevel Queue)*, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

Дальнейшим развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения.

Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Перемещение процессов из очередей с низкими приоритетами в очереди с высокими приоритетами позволяет более полно учитывать изменение поведения процессов с течением времени.

Многоуровневые очереди с обратной связью (Multilevel Feedback Queue) представляют собой наиболее общий подход к планированию процессов из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа планирования, помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

- количество очередей для процессов, находящихся в состоянии готовность;
- алгоритм планирования, действующий между очередями;
- алгоритмы планирования, действующие внутри очередей;
- правила помещения родившегося процесса в одну из очередей;
- правила перевода процессов из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

8 Лекция №8. Организация памяти компьютера. Простейшие схемы управления памятью

Цель лекции: ознакомиться с физической организацией памяти компьютера и различными схемами управления памятью.

Содержание лекции: физическая организация памяти компьютера, логическая память, связывание адресов, простейшие схемы управления памятью, схема с фиксированными разделами, оверлейная структура, сегментная и сегментно-страничная организация памяти.

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: *основную* (главную, оперативную, физическую) и *вторичную* (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился «простой свопинг» (система по-прежнему размещает каждый

процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько *разделов фиксированной величины*. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции. Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов. Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем. Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов. Наиболее распространены три стратегии. Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел. Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места. Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например, для размещения файлов на диске.

Очевидный *недостаток* этой схемы - число одновременно выполняемых процессов ограничено числом разделов. Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая *оверлей* (*overlay*) или организация структуры с перекрытием. Основная идея - держать в памяти только те инструкции программы, которые нужны в данный момент. Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-

разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами. Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы [4,13].

Схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера. Смежные свободные участки могут быть объединены. Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения. Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации.

В самом простом и наиболее распространенном случае *страничной организации памяти* (или *paging*) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней

фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе - упорядоченная пара (p,d), где p - номер страницы в виртуальной памяти, а d - смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Существуют две другие схемы организации управления памятью: *сегментная и сегментно-страничная*. Сегменты, в отличие от страниц, могут иметь переменный размер. При сегментной организации виртуальный адрес по-прежнему является двумерным и состоит из двух полей – номера сегмента и смещения внутри сегмента. Однако в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек).

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 2^{32} байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

Сегментно-страничная и страничная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных [14].

9 Лекция №9. Виртуальная память. Архитектурные средства поддержки виртуальной памяти

Цель лекции: ознакомиться с понятием виртуальной памяти и архитектурными средствами поддержки виртуальной памяти.

Содержание лекции: понятие виртуальной памяти, архитектурные средства поддержки виртуальной памяти, страничная виртуальная память

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах *виртуальной памяти* у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц. Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс

лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера, тем не менее, поддерживала виртуальную память, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин «виртуальная память» означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. В случае виртуальной памяти система отображения адресных пространств, помимо трансляции адресов, должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

Страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид, структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (page fault) или страничный отказ.

Как и в случае *простой сегментации*, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например, может превышать размер оперативной памяти. При размещении в памяти больших программ приходим к необходимости разбиения сегментов на страницы и поддержки своей таблицы страниц для каждого сегмента.

Организация таблицы страниц - один из ключевых элементов отображения адресов в страничной и *сегментно-страничной* схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бит. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого, запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц [16].

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 2^{64} байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти . Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем $2^{32}/2^{12}=2^{20}$, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц.

Количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее

разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой *многоуровневой таблицы страниц*.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры.

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

Решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, Это устройство называется *ассоциативной памятью,* иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет недостаток - записи в ней не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование *хеш-таблицы виртуальных адресов*. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Обычно размер страницы – это степень двойки от 2^9 до 2^{14} байт. Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только

постранично. При выборе размера нужно учитывать: размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше), объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы. Как правило, размер страниц задается аппаратно, например, в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах таких, как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например, в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее, большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

10 Лекция №10. Управление файлами

Цель лекции: рассмотреть вопросы именования, выявить операции, которые разрешается производить над файлами.

Содержание лекции: основные понятия файловой системы, операции над файлами.

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами.

Перечислим *основные функции* файловой системы;

- идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти;
- распределение внешней памяти между файлами. Для работы с конкретным файлом не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ;
- обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера;
- обеспечение защиты от несанкционированного доступа;
- обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа;
- обеспечение высокой производительности.

Правила именования файлов зависят от ОС. Многие ОС поддерживают имена из *двух частей* (имя+расширение), например progr.c (файл, содержащий текст программы на языке Си) или autoexec.bat (файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения как на используемые в имени символы, так и на длину имени файла. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

Важный аспект организации файловой системы и ОС - следует ли поддерживать и распознавать типы файлов. Если да, то это может помочь правильному функционированию ОС, например, не допустить вывода на принтер бинарного файла.

Основные *типы файлов*: регулярные (обычные) файлы и директории (справочники, каталоги). Обычные файлы содержат пользовательскую информацию. Директории - системные файлы, поддерживающие структуру файловой системы. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками (атрибутами). Мы будем рассматривать директории ниже.

Обычные (или регулярные) файлы реально представляют собой набор блоков (возможно, пустой) на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов - нетекстовые, или *бинарные*, файлы. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый файл в ОС Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл. Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt - ASCII-файлы, файлы с расширениями .exe - выполнимые, файлы с расширениями .obj, .zip - бинарные и т. д.

Кроме имени, ОС часто связывают с каждым файлом и другую информацию, например, дату модификации, размер и т. д. Эти другие характеристики файлов называются *атрибутами*. Список атрибутов в разных ОС может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, тип файла), адресную информацию (устройство,

начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции и информацию об использовании (даты создания, последнего чтения, модификации и др.).

Программист воспринимает файл в виде набора однородных записей**.** *Запись* - это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются - для ввода. ОС поддерживают несколько вариантов структуризации файлов. Простейший вариант - так называемый *последовательный файл*, то есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой *неструктурированную последовательность байтов*. Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewound).

Операционная система должна предоставить в распоряжение пользователя набор *операций для работы с файлами*, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти. Рассмотрим в качестве примера основные файловые операции ОС Unix [Таненбаум, 2002].

Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.

1. Удаление файла и освобождение занимаемого им дискового пространства.

2. Открытие файла. Перед использованием файла процесс должен его открыть.

Цель данного системного вызова - разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания дескриптора или управляющего блока файла. Дескриптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках

программирования, под дескриптором понимается альтернативное имя файла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом. Например, на языке Си операция открытия файла fd=open(pathname,flags,modes); возвращает дескриптор fd, который может быть задействован при выполнении операций чтения (read(fd,buffer,count);) или записи.

Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.

Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать *текущую* позицию.

Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.

Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например, переименование файла, получение атрибутов файла и т. д.

Существует два способа выполнить последовательность действий над файлами.

В первом случае для каждой операции выполняются как универсальные, так и уникальные действия (схема stateless). Например, последовательность операций может быть такой: open, read1 close, ... open, read2, close, ... open, read3, close.,

Альтернативный способ - это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: open, read1, ... read2, ... read3, close.

Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции; поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

11 Лекция №11. Реализация файловой системы

Цель лекции: получить представление о принципах реализации и структуре файловых систем.

Содержание лекции: система хранения, управление внешней памятью, связной список, таблица отображения файлов, индексные узлы, размер блока, структура файловой системы на диске, монтирование файловых систем, целостность файловой системы.

Система хранения данных на дисках может быть структурирована следующим образом:

Нижний уровень - оборудование. Это в первую очередь магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая *системой ввода-вывода*. Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента ОС - файловой системы - используемое дисковое пространство в виде *непрерывной последовательности блоков фиксированного размера*. Система ввода-вывода имеет дело с *физическими* блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с *логическими* блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1. Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию. Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую *индексным узлом* (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла. Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения. Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким

образом, может состоять из четырех основных частей. В начале раздела находится суперблок, содержащий общее описание файловой системы, например: тип файловой системы; размер файловой системы в блоках; размер массива индексных узлов; размер логического блока.

Описанные структуры данных создаются на диске в результате его *форматирования* (например, утилитами format, makefs и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

Массив индексных узлов (ilist) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.

В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

Так же, как файл должен быть открыт перед использованием, и файловая система, хранящаяся на разделе диска, должна быть *смонтирована*, чтобы стать доступной процессам системы.

Функция mount (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция umount (демонтировать) выключает файловую систему из иерархии. Функция mount, таким образом, дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков.

Процедура монтирования состоит в следующем. Пользователь (в Unix это суперпользователь) сообщает ОС имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования).

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о вновь смонтированном устройстве, о его суперблоке и корневом каталоге, а также сведения о точке монтирования. Для устранения потенциально опасных побочных эффектов число линков к каталогу - точке монтирования - должно быть равно 1. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами.

Важный аспект надежной работы файловой системы - контроль ее *целостности*. В результате файловых операций блоки диска могут считываться в память, модифицироваться и затем записываться на диск.

Причем многие файловые операции затрагивают сразу несколько объектов файловой системы. Например, копирование файла предполагает выделение ему блоков диска, формирование индексного узла, изменение содержимого каталога и т. д. В течение короткого периода времени между этими шагами информация в файловой системе оказывается несогласованной.

И если вследствие непредсказуемого останова системы на диске будут сохранены изменения только для части этих объектов (нарушена атомарность файловой операции), файловая система на диске может быть оставлена в несовместимом состоянии. В результате могут возникнуть нарушения логики работы с данными, например, появиться «потерянные» блоки диска, которые не принадлежат ни одному файлу и в то же время помечены как занятые, или, наоборот, блоки, помеченные как свободные, но в то же время занятые (на них есть ссылка в индексном узле) или другие нарушения.

В современных ОС предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

Рассмотрим пример создания жесткой связи для файла. Для этого файловой системе необходимо выполнить следующие операции: создать новую запись в каталоге, указывающую на индексный узел файла; увеличить счетчик связей в индексном узле.

Если аварийный останов произошел между 1-й и 2-й операциями, то в каталогах файловой системы будут существовать два имени файла, адресующих индексный узел со значением счетчика связей, равному 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового. Если же порядок операций изменен и, как прежде, останов произошел между первой и второй операциями, файл будет иметь несуществующую жесткую связь, но существующая запись в каталоге будет правильной. Хотя это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае [5,16].

Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый журнализация. Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может стоить дорого, но она оправданна, так как в случае отказа позволяет реконструировать потерянные данные.

Если же нарушение все же произошло, то для устранения проблемы несовместимости можно прибегнуть к утилитам (fsck, chkdsk, scandisk и др.), которые проверяют целостность файловой системы. Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

Возможны также эвристические проверки. Например, нахождение индексного узла, номер которого превышает их число на диске или поиск в пользовательских директориях файлов, принадлежащих суперпользователю.

К сожалению, приходится констатировать, что *не существует никаких средств, гарантирующих абсолютную сохранность* информации в файлах, и в тех ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим процедурам дублирования.

12 Лекция №12. Система управления вводом-выводом

Цель лекции: ознакомиться с системой управления и устройствами вводом-выводом, структурой контроллера устройств, функцией подсистемы ввода-вывода, а также буферизацией и кэшированием.

Содержание лекции: физические принципы организации ввода-вывода, структура контроллера устройства, опрос устройств и прерывания, прямой доступ к памяти, функции подсистемы ввода-вывода, буферизация и кэширование, spooling и захват устройств.

Функционирование любой вычислительной системы обычно сводится к выполнению двух видов работы: обработке информации и операций по осуществлению ее ввода-вывода. Процессы занимаются обработкой информации и выполнением операций ввода-вывода.

С точки зрения программиста, под «обработкой информации» понимается выполнение команд процессора над данными, лежащими в памяти независимо от уровня иерархии – в регистрах, кэше, оперативной или вторичной памяти. Под *«операциями ввода-вывода»* программист понимает обмен данными между памятью и устройствами, внешними по отношению к памяти и процессору такими, как магнитные ленты, диски, монитор, клавиатура, таймер. С точки зрения операционной системы «обработкой информации» являются только операции, совершаемые процессором над данными, находящимися в памяти на уровне иерархии не ниже, чем оперативная память. Все остальное относится к «операциям ввода-вывода». Чтобы выполнять операции над данными, временно расположенными во вторичной памяти, операционная система, сначала производит их подкачку в оперативную память, и лишь затем процессор совершает необходимые действия.

Контроллеры устройств ввода-вывода весьма различны как по своему внутреннему строению, так и по исполнению (от одной микросхемы до специализированной вычислительной системы со своим процессором, памятью и т. д.), поскольку им приходится управлять совершенно разными приборами. Обычно каждый контроллер имеет, по крайней мере, четыре внутренних регистра, называемых регистрами состояния, *управления, входных*

данных и выходных данных. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов, что для нас не существенно. Для простоты изложения будем считать, что каждому регистру соответствует свой порт. *Регистр состояния* содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. Эти биты индицируют завершение выполнения текущей команды на устройстве (*бит занятости*), наличие очередного данного в регистре выходных данных (*бит готовности данных*), возникновение ошибки при выполнении команды (*бит ошибки*) и т. д. *Регистр управления* получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства. Часть битов в этом регистре может быть отведена под код выполняемой команды, часть битов будет кодировать режим работы устройства, бит *готовности команды* свидетельствует о том, что можно приступить к ее выполнению. *Регистр выходных данных* служит для помещения в него данных для чтения вычислительной системой, а регистр входных данных предназначен для помещения в него информации, которая должна быть выведена на устройство. Обычно емкость этих регистров не превышает ширину линии данных.

Опрос устройств и прерывания. Исключительные ситуации и системные вызовы. В нашей модели для вывода информации, помещающейся в регистр входных данных, без проверки успешности вывода процессор и контроллер должны связываться следующим образом:

- процессор в цикле читает информацию из порта регистра состояний и проверяет значение бита занятости. Если *бит занятости* установлен, то это означает, что устройство еще не завершило предыдущую операцию, и процессор уходит на новую итерацию цикла. Если *бит занятости* сброшен, то устройство готово к выполнению новой операции, и процессор переходит на следующий шаг;
- процессор записывает код команды вывода в порт регистра управления;
- процессор записывает данные в порт регистра входных данных;
- процессор устанавливает *бит готовности команды*.

В следующих шагах процессор не задействован:

- когда контроллер замечает, что *бит готовности команды* установлен, он устанавливает *бит занятости*;
- контроллер анализирует код команды в регистре управления и обнаруживает, что это команда вывода. Он берет данные из регистра входных данных и инициирует выполнение команды;
- после завершения операции контроллер обнуляет *бит готовности команды*;

- при успешном завершении операции контроллер обнуляет *бит ошибки* в регистре состояния, при неудачном завершении команды - устанавливает его;

- контроллер сбрасывает *бит занятости*.

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов. Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройства. Однако обязанности базовой подсистемы не сводятся к выполнению только действий трансляции общего системного вызова в обращение к частной функции драйвера. Базовая подсистема предоставляет вычислительной системе такие услуги, как поддержка блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление spooling'a и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций.

Все системные вызовы, связанные с осуществлением операций ввода-вывода, можно разбить на три группы по способам реализации взаимодействия процесса и устройства ввода-вывода.

К первой, наиболее привычной для большинства программистов группе, относятся блокирующиеся системные вызовы. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т. е. процесс переводится операционной системой из состояния *исполнение* в состояние *ожидание*. Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния *ожидание* в состояние *готовность*. После того как процесс будет снова выбран для *исполнения*, в нем произойдет окончательный возврат из системного вызова. Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу далее.

Ко второй группе относятся неблокирующиеся системные вызовы. Их название не совсем точно отражает суть дела. В простейшем случае процесс, применивший неблокирующийся вызов, не переводится в состояние *ожидание* вообще. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем, в зависимости от текущей ситуации (состояния устройства, наличия данных и т. д.). В более сложных ситуациях процесс может блокироваться, но условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. Типичным случаем применения неблокирующегося системного вызова может являться периодическая проверка на поступление информации с клавиатуры при выполнении трудоемких расчетов.

К третьей группе относятся асинхронные системные вызовы. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода-вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом. Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокирующийся системный вызов для выполнения операции read вернется немедленно, но может прочитать запрошенное количество байтов, меньшее количество или вообще ничего. Асинхронный системный вызов для этой операции также вернется немедленно, но требуемое количество байтов рано или поздно будет прочитано в полном объеме [10].

Под *буфером* обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Обмен информацией между двумя процессами относится к области кооперации процессов, и мы подробно рассмотрели его организацию в соответствующей лекции. Здесь нас будет интересовать использование буферов в том случае, когда одним из участников обмена является внешнее устройство. Причины, приводящие к использованию буферов в базовой подсистеме ввода-вывода, таковы:

- разные скорости приема и передачи информации, которыми обладают участники обмена;
- разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно;
- необходимость копирования информации из приложений, осуществляющих ввод-вывод, в буфер ядра операционной системы и обратно.

Под словом *кэш* (cache – «наличные») понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, предназначенную для ускорения работы вычислительной системы.

Под словом *spool* мы подразумеваем буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования различными процессами. Правда, в современных вычислительных системах *spool* для ввода данных практически не используется, а в основном предназначен для накопления выходной информации. В некоторых операционных системах вместо использования *spooling* для устранения *race condition* применяется механизм монопольного захвата устройств процессами. Если устройство свободно, то один из процессов может получить его в монопольное распоряжение. При этом все другие процессы при попытке осуществления операций над этим устройством будут либо блокированы (переведены в состояние *ожидание*), либо получат

информацию о невозможности выполнения операции до тех пор, пока процесс, захвативший устройство, не завершится или явно не сообщит операционной системе о своем отказе от его использования. Обеспечение *spooling* и механизма захвата устройств является прерогативой базовой подсистемы ввода-вывода.

13 Лекция №13. Сети и сетевые операционные системы

Цель лекции: ознакомиться с сетевыми операционными системами, понятием протокола и разобраться с проблемами адресации в сети.

Содержание лекции: сетевые и распределенные ОС, понятие протокола, многоуровневая модель построения сетевых вычислительных систем, проблемы адресации в сети.

Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных сетевых средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения существенно не меняют структуру операционной системы. Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся, на локальной или удаленной машине и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети.

Все перечисленные выше цели объединения компьютеров в вычислительные сети не могут быть достигнуты без организации взаимодействия процессов на различных вычислительных системах. Будь то доступ к разделяемым ресурсам или общение пользователей через сеть - в основе всего этого лежит взаимодействие удаленных процессов, т. е. процессов, которые находятся под управлением физически разных операционных систем. Взаимодействие удаленных процессов принципиально отличается от других случаев. Общей памяти у различных компьютеров физически нет. Удаленные процессы могут обмениваться информацией, только передавая друг другу пакеты данных определенного формата (в виде последовательностей электрических или электромагнитных сигналов, включая световые) через некоторый физический канал связи или несколько таких каналов, соединяющих компьютеры. Поэтому в основе всех средств взаимодействия удаленных процессов лежит передача структурированных пакетов информации или сообщений. При взаимодействии локальных процессов и процесс–отправитель информации, и процесс-получатель функционируют под управлением одной и той же операционной системы. Эта же операционная система поддерживает и функционирование промежуточных накопителей данных при использовании непрямой адресации. Для

организации взаимодействия процессы пользуются одними и теми же системными вызовами, присущими данной операционной системе, с одинаковыми интерфейсами. Более того, в автономной операционной системе передача информации от одного процесса к другому, независимо от используемого способа адресации, как правило (за исключением микроядерных операционных систем), происходит напрямую - без участия других процессов - посредников.

Для того чтобы удаленные процессы могли обмениваться данными, необходимо, чтобы сетевые части операционных систем руководствовались определенными соглашениями, или, как принято говорить, поддерживали определенные *протоколы*.

Самым нижним уровнем в сетевых вычислительных системах является уровень, на котором реализуется реальная физическая связь между двумя узлами сети. Для обеспечения обмена физическими сигналами между двумя различными вычислительными системами необходимо, чтобы эти системы поддерживали определенный протокол физического взаимодействия – *горизонтальный протокол*. На самом верхнем уровне находятся пользовательские процессы, которые инициируют обмен данными. Количество и функции промежуточных уровней варьируются от одной системы к другой.

Формальный перечень правил, определяющих последовательность и формат сообщений, которыми обмениваются сетевые компоненты различных вычислительных систем, лежащие на одном уровне, мы и будем называть *сетевым протоколом***.**

Всю совокупность вертикальных и горизонтальных протоколов (интерфейсов и сетевых протоколов) в сетевых системах, построенных по «слоеному» принципу, достаточную для организации взаимодействия удаленных процессов, принято называть *семейством* протоколов или *стеком* протоколов.

Эталоном многоуровневой схемы построения сетевых средств связи считается семиуровневая модель открытого взаимодействия систем (Open System INTerconnection – OSI), предложенная Международной организацией Стандартов (International Standard Organization – ISO) и получившая сокращенное наименование OSI/ISO:

Уровень 1 – физический. Этот уровень связан с работой hardware. На нем определяются физические аспекты передачи информации по линиям связи такие, как: напряжения, частоты, природа передающей среды, способ передачи двоичной информации по физическому носителю, вплоть до размеров и формы используемых разъемов. В компьютерах за поддержку физического уровня обычно отвечает сетевой адаптер.

Уровень 2 – канальный. Этот уровень отвечает за передачу данных по физическому уровню без искажений между непосредственно связанными узлами сети. На нем формируются физические пакеты данных для реальной

доставки по физическому уровню. Протоколы канального уровня реализуются совместно сетевыми адаптерами и их драйверами.

Уровень 3 – сетевой. Сетевой уровень несет ответственность за доставку информации от узла-отправителя к узлу-получателю. На этом уровне частично решаются вопросы адресации, осуществляется выбор маршрутов следования пакетов данных, решаются вопросы стыковки сетей, а также управление скоростью передачи информации для предотвращения перегрузок в сети.

Уровень 4 – транспортный. Регламентирует передачу данных между удаленными процессами. Обеспечивает доставку информации вышележащим уровнем с необходимой степенью надежности, компенсируя, быть может, ненадежность нижележащих уровней, связанную с искажением и потерей данных или доставкой пакетов в неправильном порядке. Наряду с сетевым уровнем может управлять скоростью передачи данных и частично решать проблемы адресации.

Уровень 5 – сеансовый. Координирует взаимодействие связывающихся процессов. Основная задача – предоставление средств синхронизации взаимодействующих процессов. Такие средства синхронизации позволяют создавать контрольные точки при передаче больших объемов информации. В случае сбоя в работе сети передачу данных можно возобновить с последней контрольной точки, а не начинать заново.

Уровень 6 – уровень представления данных. Отвечает за форму представления данных, перекодирует текстовую и графическую информацию из одного формата в другой, обеспечивает ее сжатие и распаковку, шифрование и декодирование.

Уровень 7 – прикладной. Служит для организации интерфейса между пользователем и сетью.

Любой пакет информации, передаваемый по сети, должен быть снабжен адресом получателя. Если взаимодействие подразумевает двустороннее общение, то в пакет следует также включить и адрес отправителя. Существует два подхода к наделению объектов такими сетевыми адресами: одноуровневый и двухуровневый. *При одноуровневом подходе* каждый процесс, желающий стать участником удаленного взаимодействия (при прямой адресации), и каждый объект, для такого взаимодействия предназначенный (при непрямой адресации), получают по мере необходимости собственные адреса (символьные или числовые), а сами вычислительные комплексы, объединенные в сеть, никаких самостоятельных адресов не имеют. *При двухуровневой адресации* полный сетевой адрес процесса или промежуточного объекта для хранения данных складывается из двух частей – адреса вычислительного комплекса, на котором находится процесс или объект в сети (удаленного адреса), и адреса самого процесса или объекта на этом вычислительном комплексе (локального адреса). Уникальность полного адреса будет обеспечиваться уникальностью

удаленного адреса для каждого компьютера в сети и уникальностью локальных адресов объектов на компьютере.

Инициатором связи процессов друг с другом всегда является человек, будь то программист или обычный пользователь. Способ разрешения адресов заключается в том, что на каждом сетевом компьютере создается файл, содержащий имена всех машин, доступных по сети, и их числовые эквиваленты. Обращаясь к этому файлу, операционная система легко может перевести символьный удаленный адрес в числовую форму. Такой подход не является удобным, так как добавление или удаление компонента сети требует внесения изменений в файлы на всех сетевых машинах. Поэтому удобнее использовать метод разрешения адресов, который заключается в частичном распределении информации о соответствии символьных и числовых адресов по многим комплексам сети, так что каждый из этих комплексов содержит лишь часть полных данных. Он же определяет и правила построения символических имен компьютеров. Один из таких способов, используемый в Internet, получил английское наименование domain name service или сокращенно DNS.

В реальных сетевых вычислительных системах обычно используется комбинация рассмотренных подходов. Для компьютеров, с которыми чаще всего приходится устанавливать связь, в специальном файле хранится таблица соответствий символьных и числовых адресов. Все остальные адреса разрешаются с использованием служб, аналогичных службе DNS. Для локальной адресации процессов и промежуточных объектов при удаленной связи обычно организуется новое специальное адресное пространство.

Таким образом, полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, порт>. Подобная пара получила наименование socket (в переводе – «гнездо» или, как стали писать в последнее время, сокет), а сам способ их использования – *организация связи с помощью* сокетов. В случае непрямой адресации с использованием промежуточных объектов сами эти объекты также принято называть сокетами. Поскольку разные протоколы транспортного уровня требуют разных адресных пространств портов, то для каждой пары надо указывать, какой транспортный протокол она использует, – говорят о разных типах сокетов.

В современных сетевых системах числовой адрес обычно получает не сам вычислительный комплекс, а его сетевой адаптер, с помощью которого комплекс подключается к линии связи. При наличии нескольких сетевых адаптеров для разных линий связи один и тот же вычислительный комплекс может иметь несколько числовых адресов. В таких системах полные адреса удаленного адресата (процесса или промежуточного объекта) задаются парами <числовой адрес сетевого адаптера, порт> и требуют доставки информации через указанный сетевой адаптер.

Базой для взаимодействия локальных процессов служит организация общей памяти, в то время как для удаленных процессов – это обмен физическими пакетами данных.

Организация взаимодействия удаленных процессов требует от сетевых частей операционных систем поддержки определенных протоколов. Сетевые средства связи обычно строятся по «слоеному» принципу. Формальный перечень правил, определяющих последовательность и формат сообщений, которыми обмениваются сетевые компоненты различных вычислительных систем, лежащие на одном уровне, называется сетевым протоколом. Каждый уровень слоеной системы может взаимодействовать непосредственно только со своими вертикальными соседями, руководствуясь четко закрепленными соглашениями - вертикальными протоколами или интерфейсами. Вся совокупность интерфейсов и сетевых протоколов в сетевых системах, построенных по слоеному принципу, достаточная для организации взаимодействия удаленных процессов, образует *семейство* протоколов или *стек* протоколов [16,17].

Удаленные процессы, в отличие от локальных, при взаимодействии обычно требуют двухуровневой адресации при своем общении. Полный адрес процесса состоит из двух частей: удаленной и локальной.

Для удаленной адресации используются символьные и числовые имена узлов сети. Перевод имен из одной формы в другую (разрешение имен) может осуществляться с помощью централизованно обновляемых таблиц соответствия полностью на каждом узле или с использованием выделения зон ответственности специальных серверов. Для локальной адресации процессов применяются порты. Упорядоченная пара из адреса узла в сети и порта получила название socket.

Для доставки сообщения от одного узла к другому могут использоваться различные протоколы маршрутизации.

С точки зрения пользовательских процессов обмен информацией может осуществляться в виде датаграмм или потока данных.

14 Лекция №14. Основные понятия информационной безопасности

Цель лекции: ознакомиться с основными понятиями информационной безопасности и использованием криптографических алгоритмов.

Содержание лекции: угрозы безопасности, криптография как одна из базовых технологий безопасности ОС.

Любое потенциальное действие, которое направлено на нарушение конфиденциальности, целостности и доступности информации, называется *угрозой.* Реализованная угроза называется атакой. Конфиденциальная (confidentiality) система обеспечивает уверенность в том, что секретные

данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются авторизованными). Под доступностью (availability) понимают гарантию того, что авторизованным пользователям всегда будет доступна информация, которая им необходима. И наконец, целостность (integrity) системы подразумевает, что неавторизованные пользователи не могут каким-либо образом модифицировать данные.

Защита информации ориентирована на борьбу с так называемыми умышленными угрозами, то есть с теми, которые, в отличие от случайных угроз (ошибок пользователя, сбоев оборудования и др.), преследуют цель нанести ущерб пользователям ОС. Умышленные угрозы подразделяются на активные и пассивные.

Пассивная угроза - несанкционированный доступ к информации без изменения состояния системы, активная - несанкционированное изменение системы. Пассивные атаки труднее выявить, так как они не влекут за собой никаких изменений данных. Защита против пассивных атак базируется на средствах их предотвращения. Можно выделить несколько типов угроз. Наиболее распространенная угроза - попытка проникновения в систему под видом легального пользователя, например, попытки угадывания и подбора паролей. Более сложный вариант - внедрение в систему программы, которая выводит на экран слово login. Для защиты от подобных атак ОС запускает процесс, называемый аутентификацией пользователя.

Угрозы другого рода связаны с нежелательными действиями легальных пользователей, которые могут, например, предпринимать попытки чтения страниц памяти, дисков и лент, которые сохранили информацию, связанную с предыдущим использованием. Защита в таких случаях базируется на надежной системе авторизации. В эту категорию также попадают атаки типа отказ в обслуживании, когда сервер затоплен мощным потоком запросов и становится фактически недоступным для отдельных авторизованных пользователей.

Наконец, функционирование системы может быть нарушено с помощью программ-вирусов или программ - «червей», которые специально предназначены для того, чтобы причинить вред или недолжным образом использовать ресурсы компьютера.

Общее название угроз такого рода - вредоносные программы (malicious software). Обычно они распространяются сами по себе, переходя на другие компьютеры через зараженные файлы, дискеты или по электронной почте. Наиболее эффективный способ борьбы с подобными программами – соблюдение правил «компьютерной гигиены». Проблема информационной безопасности оказалась настолько важной, что в ряде стран были выпущены основополагающие документы, в которых регламентированы основные подходы к проблеме информационной безопасности. Наиболее известна оранжевая (по цвету обложки) книга Министерства обороны США. В этом

документе определяется четыре уровня безопасности – D, C, B и A. По мере перехода от уровня D до A к надежности систем предъявляются все более жесткие требования. Уровни C и B подразделяются на классы (C1, C2, B1, B2, B3). Чтобы система в результате процедуры сертификации могла быть отнесена к некоторому классу, ее защита должна удовлетворять определенным требованиям.

Субъект безопасности - активная системная составляющая, к которой применяется политика безопасности, а объект - пассивная. Примерами субъектов могут служить пользователи и группы пользователей, а объектов - файлы, системные таблицы, принтер и т. п.

Формируя политику безопасности, необходимо учитывать несколько базовых принципов. Так, Зальтцер (Saltzer) и Шредер (Schroeder) (1975) на основе своего опыта работы с MULTICS сформулировали следующие рекомендации для проектирования системы безопасности ОС.

Проектирование системы должно быть открытым. Нарушитель и так все знает (криптографические алгоритмы открыты).

Не должно быть доступа по умолчанию. Ошибки с отклонением легитимного доступа будут обнаружены скорее, чем ошибки там, где разрешен неавторизованный доступ.

Нужно тщательно проверять текущее авторство. Так, многие системы проверяют привилегии доступа при открытии файла и не делают этого после. В результате пользователь может открыть файл и держать его открытым в течение недели и иметь к нему доступ, хотя владелец уже сменил защиту.

Давать каждому процессу минимум возможных привилегий.

Защитные механизмы должны быть просты, постоянны и встроены в нижний слой системы, это не аддитивные добавки (известно много неудачных попыток «улучшения» защиты слабо приспособленной для этого ОС MS-DOS).

Важна физиологическая приемлемость. Если пользователь видит, что защита требует слишком больших усилий, он от нее откажется. Ущерб от атаки и затраты на ее предотвращение должны быть сбалансированы.

Многие службы информационной безопасности такие, как контроль входа, в систему, разграничение доступа к ресурсам, обеспечение безопасного хранения данных и ряд других, опираются на использование *криптографических алгоритмов*.

Шифрование - процесс преобразования сообщения из открытого текста (plaintext) в шифротекст (ciphertext) таким образом, чтобы его могли прочитать только те стороны, для которых оно предназначено, проверить подлинность отправителя (аутентификация), гарантировать, что отправитель действительно послал данное сообщение.

В алгоритмах шифрования предусматривается наличие ключа. Ключ - это некий параметр, не зависящий от открытого текста. Результат применения алгоритма шифрования зависит от используемого ключа. В криптографии

принято правило Кирхгофа: «Стойкость шифра должна определяться только секретностью ключа». Правило Кирхгофа подразумевает, что алгоритмы шифрования должны быть открыты.

В методе шифрования с *секретным* или симметричным ключом имеется один ключ, который используется как для шифрования, так и для расшифровки сообщения. Такой ключ нужно хранить в секрете. Это затрудняет использование системы шифрования, поскольку ключи должны регулярно меняться, для чего требуется их секретное распространение. Наиболее популярные алгоритмы шифрования с секретным ключом: DES, TripleDES, ГОСТ и ряд других.

Часто используется шифрование с помощью односторонней функции, называемой также хеш - или дайджест - функцией. Применение этой функции к шифруемым данным позволяет сформировать небольшой дайджест из нескольких байтов, по которому невозможно восстановить исходный текст. Получатель сообщения может проверить целостность данных, сравнивая полученный вместе с сообщением дайджест с вычисленным вновь при помощи той же односторонней функции. Эта техника активно используется для контроля входа в систему. Например, пароли пользователей хранятся на диске в зашифрованном односторонней функцией виде. Наиболее популярные хеш-функции: MD4, MD5 и др.

В системах шифрования с *открытым* или *асимметричным ключом* (public/ assymmetric key) используется два ключа. Один из ключей, называемый открытым, несекретным, используется для шифрования сообщений, которые могут быть расшифрованы только с помощью секретного ключа, имеющегося у получателя, для которого предназначено сообщение. Иногда поступают по-другому. Для шифрования сообщения используется секретный ключ, и если сообщение можно расшифровать с помощью открытого ключа, подлинность отправителя будет гарантирована (система электронной подписи). Этот принцип изобретен Уитфилдом Диффи (Whitfield Diffie) и Мартином Хеллманом (Martin Hellman) в 1976 г.

Использование открытых ключей снимает проблему обмена и хранения ключей, свойственную системам с симметричными ключами. Открытые ключи могут храниться публично, и каждый может послать зашифрованное открытым ключом сообщение владельцу ключа. Однако расшифровать это сообщение может только владелец открытого ключа при помощи своего секретного ключа, и никто другой. Несмотря на очевидные удобства, связанные с хранением и распространением ключей, асимметричные алгоритмы гораздо менее эффективны, чем симметричные, поэтому во многих криптографических системах используются оба метода.

Среди несимметричных алгоритмов наиболее известен RSA, предложенный Роном Ривестом (Ron Rivest), Ади Шамиром (Adi Shamir) и Леонардом Эдлманом (Leonard Adleman). Идея, положенная в основу метода, состоит в том, чтобы найти такую функцию $y=\Phi(x)$, для которой получение

обратной функции $x=f^{-1}(y)$ было бы в общем случае очень сложной задачей (NP-полной задачей). Например, получить произведение двух чисел $n=p\times q$ просто, а разложить n на множители, если p и q достаточно большие простые числа, - NP - полная задача с вычислительной сложностью ~ n^{10}. Однако если знать некую секретную информацию, то найти обратную функцию $x=f^{-1}(y)$ существенно проще. Такие функции также называют односторонними функциями с лазейкой или потайным ходом.

Применяемые в RSA прямая и обратная функции просты. Они базируются на применении теоремы Эйлера из теории чисел.

15 Лекция №15. Защитные механизмы операционных систем

Цель лекции: рассмотреть защитные механизмы ОС.

Содержание лекции: защитные механизмы операционных систем, простое свойство секретности.

Для начала рассмотрим проблему контроля доступа в систему. Наиболее распространенным способом контроля доступа является процедура регистрации. Обычно каждый пользователь в системе имеет уникальный идентификатор. Идентификаторы пользователей применяются с той же целью, что и идентификаторы любых других объектов, файлов, процессов. Идентификация заключается в сообщении пользователем своего идентификатора. Для того чтобы установить, что пользователь именно тот, за кого себя выдает, то есть что именно ему принадлежит введенный идентификатор, в информационных системах предусмотрена процедура аутентификации (authentication, опознавание, в переводе с латинского означает "установление подлинности"), задача которой - предотвращение доступа к системе нежелательных лиц.

Обычно аутентификация базируется на одном или более из трех пунктов:

- то, чем пользователь владеет (ключ или магнитная карта);
- то, что пользователь знает (пароль);
- атрибуты пользователя (отпечатки пальцев, подпись, голос).

Наиболее простой подход к аутентификации - применение пользовательского пароля. Когда пользователь идентифицирует себя при помощи уникального идентификатора или имени, у него запрашивается пароль. Если пароль, сообщенный пользователем, совпадает с паролем, хранящимся в системе, система предполагает, что пользователь легитимен. Пароли часто используются для защиты объектов в компьютерной системе в отсутствие более сложных схем защиты. Недостатки паролей связаны с тем, что трудно сохранить баланс между удобством пароля для пользователя и его

надежностью. Пароли могут быть угаданы, случайно показаны или нелегально переданы авторизованным пользователем неавторизованному.

Есть два общих способа угадать пароль. Один связан со сбором информации о пользователе. Люди обычно используют в качестве паролей очевидную информацию (скажем, имена животных или номерные знаки автомобилей). Другой способ - попытаться перебрать все наиболее вероятные комбинации букв, чисел и знаков пунктуации (атака по словарю). Несмотря на все это, пароли распространены, поскольку они удобны и легко реализуемы.

Для хранения секретного списка паролей на диске во многих ОС используется криптография. Система задействует одностороннюю функцию, которую просто вычислить, но для которой чрезвычайно трудно (разработчики надеются, что невозможно) подобрать обратную функцию.

При удаленном доступе к ОС нежелательна передача пароля по сети в открытом виде. Одним из типовых решений является использование криптографических протоколов. В качестве примера можно рассмотреть протокол опознавания с подтверждением установления связи путем вызова - CHAP (Challenge Handshake Authentication Protocol). Опознавание достигается за счет проверки того, что у пользователя, осуществляющего доступ к серверу, имеется секретный пароль, который уже известен серверу. Пользователь инициирует диалог, передавая серверу свой идентификатор. В ответ сервер посылает пользователю запрос (вызов), состоящий из идентифицирующего кода, случайного числа и имени узла сервера или имени пользователя. При этом пользовательское оборудование в результате запроса пароля пользователя отвечает следующим ответом, зашифрованным с помощью алгоритма одностороннего хеширования, наиболее распространенным видом, которого является MD5. После получения ответа сервер при помощи той же функции с теми же аргументами шифрует собственную версию пароля пользователя. В случае совпадения результатов вход в систему разрешается. Существенно, что незашифрованный пароль при этом по каналу связи не посылается.

После успешной регистрации система должна осуществлять авторизацию (authorization) - предоставление субъекту прав на доступ к объекту. Средства авторизации контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые были определены администратором, а также осуществляют контроль возможности выполнения пользователем различных системных функций. Система контроля базируется на общей модели, называемой матрицей доступа. Компьютерная система может быть смоделирована как набор субъектов (процессы, пользователи) и объектов (процессор, сегменты памяти, принтер, диски и ленты, файлы, программы, семафоры), то есть все то, доступ к чему контролируется. Каждый объект имеет уникальное имя, отличающее его от других объектов в системе, и каждый из них может быть доступен через хорошо определенные и значимые операции. Операции

зависят от объектов. Желательно добиться того, чтобы процесс осуществлял авторизованный доступ только к тем ресурсам, которые ему нужны для выполнения его задачи [8,15].

Различают дискреционный (избирательный) способ управления доступом и полномочный (мандатный). При дискреционном доступе, определенные операции над конкретным ресурсом запрещаются или разрешаются субъектам или группам субъектов. Полномочный подход заключается в том, что все объекты могут иметь уровни секретности, а все субъекты делятся на группы, образующие иерархию в соответствии с уровнем допуска к информации. Иногда это называют моделью многоуровневой безопасности, которая должна обеспечивать выполнение следующих правил:

Субъект может читать информацию только из объекта, уровень секретности которого не выше уровня секретности субъекта.

Субъект может записывать информацию в объекты только своего уровня или более высоких уровней секретности.

Большинство операционных систем реализуют именно дискреционное управление доступом. Главное его достоинство - гибкость, основные недостатки - рассредоточенность управления и сложность централизованного контроля.

Чтобы рассмотреть схему дискреционного доступа более детально, введем концепцию домена безопасности (protection domain). Каждый домен определяет набор объектов и типов операций, которые могут производиться над каждым объектом. Возможность выполнять операции над объектом есть права доступа, каждое из которых есть упорядоченная пара <object-name, rights-set>. Домен, таким образом, есть набор прав доступа. Например, если домен D имеет права доступа <file F, {read, write}>, это означает, что процесс, выполняемый в домене D, может читать или писать в файл F, но не может выполнять других операций над этим объектом.

Связь конкретных субъектов, функционирующих в операционных системах, может быть организована следующим образом:

- каждый пользователь может быть доменом. В этом случае набор объектов, к которым может быть организован доступ, зависит от идентификации пользователя;
- каждый процесс может быть доменом. В этом случае набор доступных объектов определяется идентификацией процесса;
- каждая процедура может быть доменом. В этом случае набор доступных объектов соответствует локальным переменным, определенным внутри процедуры. Заметим, что когда процедура выполнена, происходит смена домена.

Иногда применяется *комбинированный способ* доступа. Например, в том же Unix на этапе открытия файла происходит анализ ACL (операция open). В случае благоприятного исхода файл заносится в список открытых процессом файлов, и при последующих операциях чтения и записи проверки прав

доступа не происходит. Список открытых файлов можно рассматривать как перечень возможностей.

Существует также схема lock-key, которая является компромиссом между списками прав доступа и перечнями возможностей. В этой схеме каждый объект имеет список уникальных битовых шаблонов (patterns), называемых locks. Аналогично каждый домен имеет список уникальных битовых шаблонов, называемых ключами (keys). Процесс, выполняющийся в домене, может получить доступ к объекту, только если домен имеет ключ, который соответствует одному из шаблонов объекта.

Контроль повторного использования объекта предназначен для предотвращения попыток незаконного получения конфиденциальной информации, остатки которой могли сохраниться в некоторых объектах, ранее использовавшихся и освобожденных другим пользователем. Безопасность повторного применения должна гарантироваться для областей оперативной памяти (в частности, для буферов с образами экрана, расшифрованными паролями и т. п.), для дисковых блоков и магнитных носителей в целом. Очистка должна производиться путем записи маскирующей информации в объект при его освобождении (перераспределении). Например, для дисков на практике применяется способ двойной перезаписи освободившихся после удаления файлов блоков случайной битовой последовательностью.

Даже самая лучшая система защиты рано или поздно будет взломана. Обнаружение попыток вторжения является важнейшей задачей системы защиты, поскольку ее решение позволяет минимизировать ущерб от взлома и собирать информацию о методах вторжения. Как правило, поведение взломщика отличается от поведения легального пользователя. Иногда эти различия можно выразить количественно, например, подсчитывая число некорректных вводов пароля во время регистрации. Основным инструментом выявления вторжений является *запись данных аудита*. Отдельные действия пользователей протоколируются, а полученный протокол используется для выявления вторжений. Аудит заключается в регистрации специальных данных о различных типах событий, происходящих в системе и так или иначе влияющих на состояние безопасности компьютерной системы.

Помимо протоколирования, можно периодически *сканировать с*истему на наличие слабых мест в системе безопасности. Любая проблема, обнаруженная сканером безопасности, может быть ликвидирована как автоматически, так и передана для решения менеджеру системы.

Список литературы

1. Карпов В.Е., Коньков К.А. Основы операционных систем.- Москва: издательство ИНТУИТ, 2005.

2. Дейтел Г. Введение в операционные системы.-М.: Мир, 2011.

3. Курячий Г.В. Операционная система UNIX.- Москва, 2004.

4. Олифер В.Г., Олифер Н.А. Операционные системы.-Спб.: Издательский дом Питер, 2008.

5. Коньков К.А. Устройство и функционирование ОС Windows.- Москва, 2008.

6. http://cne.gmu.edu/pjd/PUBS/bvm.pdf.

7. Таненбаум Э.Современные операционные системы.-СПб.: Издательский дом Питер, 2005.

8. Андреев А. Microsoft Windows XP.- СПб.: Издательский дом Питер, 2006.

9. Олифер В.Г. Сетевые операционные системы.- Спб.: Издательский дом Питер, 2010.

10. Моримото Р. Microsoft Windows Server 2008 R2. -М: Полное руководство, 2011.

11. Айвенс К. Windows Server 2003. -М: Полное руководство, 2004.

12. Дейтел Х.М. Операционные системы. Т.1.-М.: «Бином», 2011.

13. Дейтел Х.М. Операционные системы. Т.2.-М.: «Бином», 2011.

14. Киселев С.В. Операционные системы. –М.: «Академия», 2012.

15. Мартемьянов Ю.Ф. Операционные системы. Концепция построения и обеспечения безопасности. –М.: «Горячая линия – Телеком». 2011.

16. Назаров С.В. Современные операционные системы. –М.: «Интернет-УИТ: Бином», 2011.

17. Сафонов В.О. Основы современных операционных систем. –М.: «БИНОМ», 2011.

Содержание

Сводный план 2014 г. поз.367

Бибигуль Рыскуловна Абсатарова

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Конспект лекций
для студентов специальности 5В070300-Информационные системы

Редактор Л.Т. Сластихина
Специалист по стандартизации Н.К. Молдабекова

Подписано в печать __.__.__.
Тираж 30 экз.
Объем 4,1 уч.-изд. л.

Формат 60х84 1/16
Бумага типографическая №1
Заказ _____. Цена 2050 т.

Копировально-множительное бюро
Некоммерческого акционерного общества
Алматинского университета энергетики и связи
050013 Алматы, Байтурсынова, 126