Определения

Расширяемость – простота подключения новых устройств.

Масштабируемость – система может подключать дополнительное оборудование для увеличения производительности.

Маршаллинг – преобразования объекта в памяти в формат данных, пригодный для хранения или передачи. Обычно применяется, когда данные необходимо передавать между различными частями одной программы или от одной программы к другой. (c) Wiki

(альтернативное определение: проход по уровням чего-либо сверху-вниз с целью сбора данных)

Симоненко несколько раз приводил пример маршаллинга как упаковки данных в пакеты при пересылке по сети.

Демаршаллинг – обратный процесс. Как бы распаковка данных для того, чтоб их можно было поместить в память.(c) Wiki

(альтернативное определение: проход по уровням чего-либо снизу-вверх с целью сбора данных)

Кэш-промах – если при обращении к кешу, в нём не оказалось запрашиваемых данных, то происходит кеш-промах, в процессе которого необходимое значение подгружается в кеш.

Однопрограммный режим работы – если все ресурсы машины отданы 1 задаче, которая не может быть прервана (она завершается либо сама запланировано, либо аварийно)

Многопрограммный режим работы – если в системе находится несколько задач на разных стадиях исполнения, каждая из которых может быть прервана и восстановлена в любой момент времени.

Пробуксовка системы – процессор нагружен на 100%, а реальной работы не выполняется.

Спин-блокировка – возникает когда процесс требует ресурс, в то время как этот ресурс захвачен другим процессом. В таком случае этот процесс постоянно опрашивает ресурс занимая все процессорное время.

Задание – внешняя единица работы системы, на которую система ресурсы не выделяет. Работа, которую мы хотим дать ОС, чтоб она её выполнила. Задания накапливаются в буферах. В задании должна быть указана программа и данные. Как только появляются ресурсы, задание расшифровывается. Формируется task control block (TCB), он же process control block (PCB).

Задача – внутренняя единица системы, для которой система выделяет ресурсы (активизирует задание).

Процесс – любая выполняемая программа системы; это динамический объект системы, которому она выделяет ресурсы; траектория процессора в адресном пространстве машины. ОС всегда должна знать, что происходит с процессом.

Программное обеспечение – совокупность программ и инструкций по их использованию.

Математическое обеспечение – совокупность программ, инструкций по их использованию и описание математического аппарата, использованного для создания программы.

Ресурсы – то, что может понадобиться для выполнения программы (задачи, процесса). Все ресурсы можно разделить условно на две категории: Физические устройства в составе компа. Внутренние ресурсы пользовательских программ (данные, подпрограммы и т. д.).

I семестр

Этапы развития вычислительной техники

Таненбаум выделяет 4 этапа развития компьютеров/ОС:

I поколение (1945-1955) Лампы, коммутационные панели

Использовались лампы. Составные блоки машины соединялись посредством коммутационных панелей. ОС еще не было. Только прямые численные расчеты (sin, cos, ln)

II поколение (1955-1965) транзисторы, системы пакетной обработки

Компьютеры стали более надежными и быстродействие увеличилось. С перфокарт задания (силами маломощного компьютера) переносились на магнитную ленту, с которой работал основной (мощный компьютер), так как заданий на ленте было много, нужно было управлять ими (начинать, заканчивать и тд) – появился прообраз ОС. Выходные данные так же записывались на магнитную ленту.

III поколение (1965-1980) интегральные схемы и многозадачность

IBM создала совместимую линейку машин разной мощности и одну операционную систему, которая работала на этих компьютерах (очень сложную). Были созданы средства аппаратной поддержки многозадачности и системы разделения времени (MULTICS, из которого берет свои корни Unix).

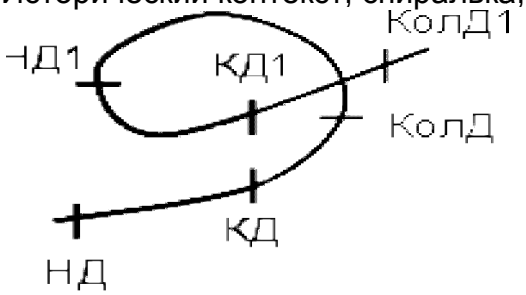
Так же в это время начали развиваться относительно дешевые компьютеры, которые на специфических заданиях почти не уступали по производительности большим мейнфреймам.

IV поколение (1980-наши дни) персональные компьютеры

В конспекте этапы другие.

Методы доступа пользователя к ресурсам

Исторический контекст, спиралька, в которой всё повторяется.



НД – Непосредственный доступ. Все ресурсы одному пользователю, машина работала медленно, поэтому человеческой реакции вполне хватало.

КД (первый прообраз ОС) – Косвенный доступ. Скорость работы машины увеличилась, человек стал тормозить. Появился так называемый однопрограммный режим работы, программы загружали операторы.

КолД – Коллективный доступ. Скорость проца так увеличилась, что, пока тупит один человек, проц может решать другую задачу, появились терминалы – многопрограмный режим работы.

НД1 – Непосредственный доступ 1. Большинство пользователей сидит за своими персоналками – персональные компьютеры.

КД1 – Косвенный доступ 1. Программы начали отправлять на сервер.

КолД1 – Коллективный доступ 1. Клиент-серверный – сервер обрабатывает множество запросов. Возможно, это cloud computing? или “тонкие клиенты”.

Организация многопрограммного режима работы

Система работает в многопрограммном режиме, если в ней находится несколько задач в разной стадии исполнения, и каждая из них может быть прервана другой с последующим возвратом.

Многопрограммный режим в однопроцессорной системе – имеем один набор оборудования (проц) и много процессов. ОС планирует выполнение процессов во времени, синхронизируя их работу. Важно для ОС поддерживать защиту данных процессов друг от друга. О синхронизации заботится программист.

Многопрограммный режим в многопроцессорной системе – это планирование во времени и в пространстве. ОС распределяет процессы на процессоры, синхронизирует их работу во времени (см. статическое и динамическое планирование). ОС заботится об эффективной нагрузке на ресурсы (процы).

По конспекту выделяем:

1. классическое мультипрограммирование

Режим работы истинного совмещения, когда разные блоки могут заниматься разными задачами одновременно. Проц, выполняющий пользовательскую задачу, может работать одновременно с УВВ.

2. параллельная обработка

Режим кажущегося параллелизма (ну, вы знаете, задачи делятся по квантам и выполняются на проце по очереди кванты маленькие, создаётся впечатление параллелизма).

3. режим разделения времени

Совмещает классическое мультипрограммирование и параллельную обработку + доступ привилегированных пользователей к ресурсам машины. Одна машина и много терминалов доступа. При этом часть задач (таких как ввод или редактирование данных оператором) могла исполняться в режиме диалога, а другие задачи (такие как массивные вычисления) – в пакетном режиме.

4. работа в реальном времени

Время ответа соответствует заранее заданным характеристика (определено внешними факторами). Если есть хоть один процесс, который невозможно прервать – уже не режим реального времени. Если процесс не может выполниться за отведённое ему время, должен быть зафиксирован сбой в его работе. Операционная система должна за предсказуемое время отреагировать на непредсказуемое появление внешних событий.

Функции операционной системы

1) Обеспечить пользователя средствами для решения его конкретных задач

2) Обеспечение эффективного управления оборудованием при выполнении основных функций

(Таненбаум выделяет несколько иные (но в целом похожие) цели:

а) ОС как виртуальная машина (то есть расширение функций компьютера),

б) Управление ресурсами)

Основные функции:

1. Выполнение элементарных (низкоуровневых) действий, которые являются общими для большинства программ и часто встречаются почти во всех программах (ввод и вывод данных, запуск и остановка других программ, выделение и освобождение дополнительной памяти и др.).
2. Управление процессами
3. Управление оперативной памятью
4. Организация файловой системы
5. Обеспечение пользовательского интерфейса.
6. Сетевые операции, поддержка стека сетевых протоколов.

Ссылочки на вики про ОС и про Пуллинг

Классификация операционных систем

1. Однопрограммные (в однопрограмных системах обычно ОС нет вообще)
2. Многопрограммные
3. Сетевые (операционная система со встроенными возможностями для работы в компьютерных сетях) (c) Wiki
4. Распределённые (должна поддерживаться функция прозрачности (по времени, месторасположению, etc.), масштабируемости (если не хватает производительности - подключаются новые ресурсы), расширяемости)
5. Метавычисления
6. Кластерные вычисления (объединение составных систем по определенном признаку)
7. GRID система

Распределённые системы

Распределённые вычислительные системы

Среда, в которой компоненты системы или ресурсы: процессоры, память, принтеры, графические станции, программы, данные и т.д., связанны вместе посредством сети, которая позволяет пользователям представлять ВС как единую вычислительную среду и иметь доступ к ее ресурсам.

Особенности:

1. Ограниченность связана с тем, что количество узлов в сети ограничено и они являются независимыми компонентами сети.
2. Идентификация. Каждый из ресурсов сети должен однозначно идентифицироваться (именоваться, например).
3. Распределенное управление. Каждый узел должен иметь возможность и средства (hardware, software) для управления сетью. Но с т.з. надёжности нежелательно давать каким-то узлам больше привилегий в управлении.
4. Гетерогенность. Система должна работать на разнородных узлах с различной длиной слов и байтовой организацией. Это касается программного, прикладного и системного обеспечения узлов.

Пользователь не должен знать особенности аппаратного обеспечения сети.

Распределённые операционные системы

РОС строятся на РВС.

Свойства распределенных операционных систем:

1. Надо поддерживать когерентность файлов.

2. Распределенная система распределяет выполняемые работы в узлах системы, исходя из соображений повышения пропускной способности всей системы;

3. Распределенные системы имеют высокий уровень организации параллельных вычислений

Принципы построения распределенных ОС

1. Прозрачность (для пользователя и программы)

Прозрачность сети требует, чтобы детали сети были скрыты от конечных пользователей.

1. расположения – пользователь не должен знать, где расположены ресурсы
2. миграции – ресурсы могут перемещаться без изменения их имен
3. размножения – пользователь не должен знать, сколько копий существует
4. конкуренции – множество пользователей разделяет ресурсы автоматически
5. параллелизма – работа может выполняться параллельно без участия пользователя
6. именование – имя должно быть уникальным в глобальном смысле, и не имеет значения в каком месте системы оно будет использовано

2. Гибкость (не все еще ясно - потребуется менять решения)

Использование монолитного ядра ОС или микроядра.

3. Надежность

Доступность, устойчивость к ошибкам (fault tolerance).

Секретность.

4. Производительность

Грануллированность. Мелкозернистый и крупнозернистый параллелизм (fine-grained parallelism, coarse-grained parallelism). Устойчивость к ошибкам требует дополнительных накладных расходов.

5. Масштабируемость – система может подключать дополнительное оборудование для увеличения производительности.

Плохие решения:

1. централизованные компоненты (один почтовый сервер);
2. централизованные таблицы (один телефонный справочник);
3. централизованные алгоритмы (маршрутизатор на основе полной информации).

Только децентрализованные алгоритмы со следующими чертами:

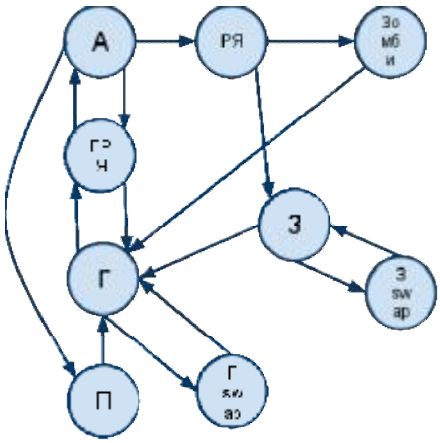
1. ни одна машина не имеет полной информации о состоянии системы;
2. машины принимают решения на основе только локальной информации;
3. выход из строя одной машины не должен приводить к отказу алгоритма;
4. не должно быть неявного предположения о существовании глобальных часов.

Отличие распределённой ОС от сетевой

В сетевой операционной системе пользователи знают о существовании многочисленных компьютеров, могут регистрироваться на удаленных машинах и копировать файлы с одной машины на другую. Каждый компьютер работает под управлением локальной операционной системы и имеет своего собственного локального пользователя. Сетевые операционные системы несущественно отличаются от однопроцессорных операционных систем. Ясно, что они нуждаются в сетевом интерфейсном контроллере и специальном низкоуровневом программном обеспечении, поддерживающем работу контроллера, а также в программах, разрешающих пользователям удаленную регистрацию в системе и доступ к удаленным файлам.

Распределенная операционная система, напротив, представляется пользователям традиционной однопроцессорной системой, хотя она составлена из множества процессоров. При этом пользователи не должны беспокоиться о том, где работают их программы или расположены файлы; все это должно автоматически и эффективно обрабатываться самой ОС.

Состояние процесса



Есть два рассказа о состоянии процессов. Простой и сложный. Для сложного используется картинка выше (9 состояний). Для простого - ниже. Симоненко больше любит сложный (а шо поделать).

Итак. Сложный вариант.

П – подготовка. На этом этапе лежат задания. Т.е. программа (что делать) и данные (над чем делать). Заданию ещё не выделили ресурсы. Когда ему планировщик выделит ресурсы, задание станет процессов и перейдёт в готовое состояние.

Г – готовность. Процесс размещен в ОП, ему выделены ресурсы, сформирован PCB. Так может случиться, что процесс своппируют на диск вместе с его ресурсами (ну, разве что ОП из ресурсов вычеркнут).

Гswap – процесс готов, но своппирован на диске.

ГРя – готовность в режиме ядра. Это некое абстрактное состояние, когда процесс уже имеет ресурсы, но ещё не допущен к процессору. С этого состояния процесс может или получить долгожданный доступ к процессору, или вернуться в очередь готовых.

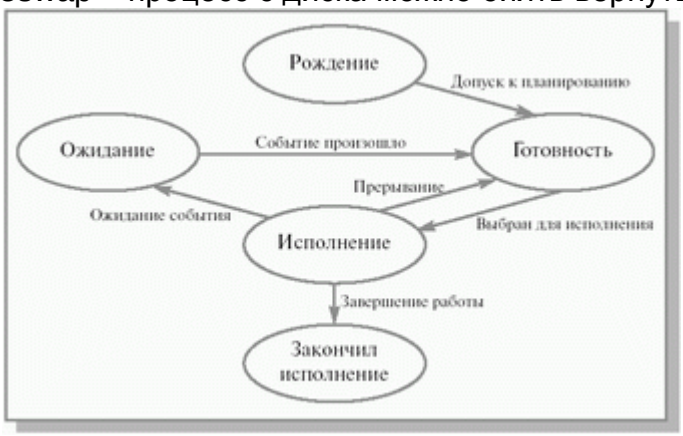
А – активность. Процесс занял процессор и, собственно, выполняется. Если у процесса окончились выделенные ему кванты времени, он возвращается в ГРя. Если внезапно окончились его ресурсы – в очередь подготовленных. Если произошёл системный вызов – в Ря.

Ря – режим ядра. Сюда попадает процесс, пока выполняется системный вызов (например, ядро открывает файл). Процесс может поступить к заблокированным, или стать зомби.

Зомби – процесса в системе нет, но его PCB ещё есть. Такая ситуация возникает, когда процесс завершился, а породивший его процесс ещё не знает об этом.

Заблокированные процессы находятся в состоянии ожидания. Кому не повезёт – освободят ОП и будут своппированны на диск.

Зswap – процесс с диска можно опять вернуть в ОП в очередь заблокированных.



Если система определила необходимость активизации процесса и выделяет нужные ему ресурсы, кроме времени процессора, то она переводит его в готовое состояние.

Если процессор освободился, то первый (наиболее приоритетный) процесс из очереди готовых процессов получает время процессора и переходит в активное состояние. Выделение времени процессора процессу осуществляет диспетчер. В состоянии исполнение происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

1. операционная система прекращает его деятельность;
2. он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание;

3. в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность

4. процесс не выполнился за выделенный ему квант времени, тогда он переходит снова в готовое состояние.

Из состояния ожидание процесс попадает в состояние готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Если процесс требует действия или ресурса, которых в данный момент операционная система не может выполнить, он переводится в подготовленное состояние и считается приостановленным.

В общем случае, система должна следить за процессами в очередях готовых и заблокированных процессов, чтобы не было бесконечно ожидающих процессов или процессов, монопольно удерживающих выделенные им ресурсы.

Для любознательных: линк на интуит

Управление процессами, или Всё о PCB

Выполнение функций ОС, связанных с управлением процессами, осуществляется с помощью специальных структур данных, образующих окружение процесса, среду исполнения или образ процесса. Образ процесса состоит из двух частей: данных режима задачи и режима ядра. Образ процесса в режиме задачи состоит из сегмента кода программы, которая подчинена процессу, данных, стека, библиотек и других структур данных, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи, которые используются ядром для управления процессом. Оно содержит различную вспомогательную информацию, необходимую ядру во время работы процесса.

Каждому процессу в ядре операционной системы соответствует блок управления процессом (PCB – process control block). Вход в процесс (фиксация системой процесса) –это создание его блока управления (PCB), а выход из процесса – это его уничтожение, т. е. уничтожение его блока управления.

Таким образом, для каждой активизированной задачи система создает свой PCB, в котором, в сжатом виде, содержится используемая при управлении информация о процессе.

PCB – это системная структура данных, содержащая определённые сведения о процессе со следующими полями:

1. Идентификатор процесса (имя);
2. Идентификатор родительского процесса;
3. Текущее состояние процесса (выполнение, приостановлен, сон и т.д.);
4. Приоритет процесса;
5. Флаги, определяющие дополнительную информацию о состоянии процесса;
6. Список сигналов, ожидающих доставки;
7. Список областей памяти выделенной программе, подчиненной данному процессу;
8. Указатели на описание выделенных ему ресурсов;
9. Область сохранения регистров;
10. Права процесса (список разрешенных операций);

Ядро ОС

Все операции, связанные с процессами, осуществляются под управлением ядра ОС, которое представляет лишь небольшую часть кода ОС в целом. Ядро ОС скрывает от пользователя частные особенности физической машины, предоставляя ему все необходимое для организации вычислений:

1. само понятие процесса, а значит и операции над ним, механизмы выделения времени физическим процессам;
2. примитивы синхронизации, реализованные ядром, которые скрывают от пользователя физические механизмы перестановки контекста при реализации операций, связанных с прерываниями.

Ядро ОС содержит программы для реализации следующих функций:

1. обработка прерываний;
2. создание и уничтожение процессов;
3. переключение процессов из состояния в состояние;
4. диспетчеризацию заданий, процессов и ресурсов;

5. приостановка и активизация процессов;
6. синхронизация процессов;
7. организация взаимодействия между процессами;
8. манипулирование PCB (process control block – см. выше);
9. поддержка операций ввода/вывода;
10. поддержка распределения и перераспределения памяти;
11. поддержка работы файловой системы;
12. поддержка механизма вызова-возврата при обращении к процедурам;
13. поддержка определенных функций по ведению учета работы машины;

Одна из самых важных функций, реализованная в ядре – обработка прерываний.

Классификация ОС по типу ядра

1. Монолитное ядро. Написание ядра системы по принципу “big mess”. Ядро ОС состоит из большого количества отдельных процедур с определенными интерфейсами. Все эти процедуры компилируются и собираются в один большой объектный файл.

Процессор имеет два режима работы: пользовательский и системный. Когда происходит системный вызов, параметры помещаются в соответствующие регистры, процессор переходит в системный режим, находится соответствующая вызову процедура и вызывается.

Процедуры разделены по 3 уровням: Основная процедура (предоставляет интерфейс всем системным вызовам), Сервисные процедуры (реализуют системные вызовы) и Утилиты (используются сервисными процедурами).

2. Многоуровневая (Кольцевая, Иерархическая). Схема построения системы по слоям (кольцам). Каждый слой (кольцо), реализует некоторую функциональность и все слои которые находятся над ним, уже не должны заботится о ней. Примером может служить система THE, в которой были следующие слои:

5. Оператор
4. Пользовательские программы
3. Управление вводом-выводом
2. Взаимодействие оператор-процесс
1. Управление памятью и барабаном
0. Выделение процессора и многозадачность

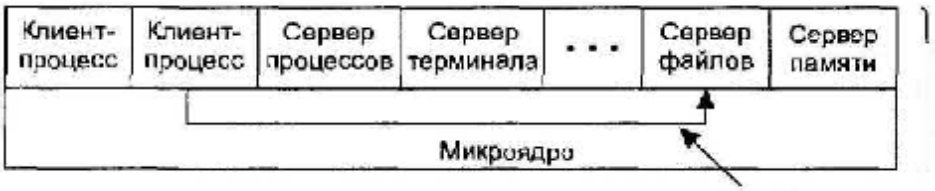
В системе MULTICS была реализована кольцевая структура.

3. Виртуальная машина. Этот тип операционных систем берет начало в 70-х годах, когда была очень распространена OS/360, но эта система обеспечивает исключительно пакетную обработку. Поэтому была создана система TSS/370. Эта система создавала несколько виртуальных машин, которые по своим архитектурным возможностям полностью повторяли реальную машину. То есть на каждую из этих виртуальных машин, можно поставить такую же ОС как и на реальную машину.

4. Экзоядерная. Это система в которой ресурсы системы разделяются между несколькими виртуальными системами. Но, в отличии от виртуальных машин, в данном случае каждой системе выдается конкретная часть ресурсов. Если одна система пытается воспользоваться ресурсами, выделенными другой системе, то срабатывает защитным механизм и ее действия пресекаются.

Из конспекта: Многопользовательская система, любой шаг в чужую зону – нарушение безопасности.

5. Микроядерная (модель клиент-сервер) – основная идея: сделать ядро как можно меньше и модульней, а большинство функциональности, которая, обычно, реализуется при монолитной архитектуре прямо в ядре вынести в отдельные модули. Ядро делится на несколько модулей, каждый из которых отвечает за отдельную часть общей функциональности (к примеру, управление файлами, процессами, вводом-выводом). Эти модули запускаются в режиме ядра и обеспечивают минимальную достаточную функциональность. Вся остальная функциональность вынесена в модули, запускаемые в пользовательском режиме. Модули ядра называют серверами, а модули пользовательского режима - клиентами. Такая архитектура сравнительно легко преобразуется для работы в распределенных системах.



Многоуровневые системы

1) Языки визуального программирования
2) Языки высокого уровня
3) Уровень ОС
4) Ассемблер высокого уровня
5) Машинный язык
6) Микропрограммный уровень
7) Аппаратура

Дисциплины обслуживания заявок

Классификация дисциплин обслуживания по:

Классификация по приоритетам:

1. безприоритетные
2. приоритетные.
1. без вытеснения
1. относительные
2. с вытеснением
1. абсолютные
2. выделение равных квантов времени

Если заявка с более высоким приоритетом не прерывает обслуживание заявки с низким приоритетом – относительная ДО без вытеснения, соответственно, если наоборот - абсолютная ДО с вытеснением.

Приоритеты бывают:

1. статические (сразу задаются заявке и не изменяются в процессе)
2. динамические (приоритеты меняются в зависимости от t_ожид. или t_обслуж.).

Классификация по количеству очередей

Одноочередные дисциплины

1. FIFO (первый пришел – первый обслужен). Очередь. Время нахождения в очереди длинных и коротких запросов зависит только от момента их поступления.
2. LIFO (последний пришел – первый обслужен). Стек.
3. Round Robin – круговой циклический алгоритм. Запрос обслуживается в течение кванта времени tk. Если за это время обслуживание не завершено, то запрос передается в конец входной очереди на дообслуживание. Короткие запросы находятся в очереди меньшее время, чем длинные.
4. RAND случайный выбор.

Многоочередные дисциплины

Все новые запросы поступают в очередь 1. Время, выделяемое на обслуживание любого запроса, равно длительности кванта tk. Если запрос обслужен за это время, то он покидает систему, а если нет, то по истечении выделенного кванта времени он поступает в конец очереди i+1. На обслуживание выбирается запрос из очереди i, только если очереди 1,…, i-1 пусты.

Таким образом, длинные запросы поступают сначала в очередь 1, затем постепенно доходят до очереди N и здесь обслуживаются до конца либо по дисциплине FIFO, либо по круговому циклическому алгоритму.

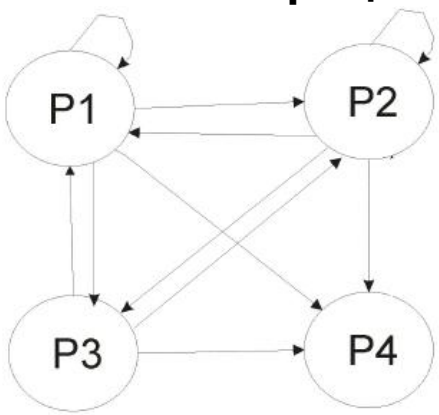
Все остальные многоочередные дисциплины обслуживая - это разные варианты базового алгоритма. Например, время обслуживая может увеличиваться соответственно с номером очереди.

В смешанном алгоритме каждая очередь обслуживается по-своему (FIFO, RR), дисциплины могут чередоваться.

В приоритетных системах каждая очередь отвечает за свой приоритет.

Для любознательных: линк на интуит

Состояние процессора



P1 – обработка системной программы
P2 – обработка программы пользователя
P3 – дешифрация прерывания
P4 – отключение

Структура программ

1) простая структура;
2) структура с запланированным наложением (оверлейная);
3) с динамической последовательной структурой;
4) с динамической параллельной структурой.

Описание каждой структуры:

1. Программа с простой структурой загружается в основную память и выполняется как отдельный объект. Сегмент программы, помещаемый в основную память, содержит всю программу целиком. (Конкретные команды, выполняемые в этом модуле загрузки, не важны для операционной системы, даже если программа может потребовать услуг операционной системы.)
2. Программа со структурой, в которой запланировано наложение, создается редактором связей как отдельный модуль загрузки. Однако в нем определяются сегменты программы, которые не должны одновременно присутствовать в основной памяти. Таким образом, одна и та же область в основной памяти может быть повторно использована на основе иерархического построения программы. Этот метод, известный как наложение, требует минимальной помощи от управляющей программы для переноса налагаемого сегмента в основную память.
3. Если поведение программы во время их выполнения становится более сложным, появляется тенденция к снижению эффективности и гибкости структур с запланированным наложением. Остальные типы структур программ позволяют использовать при выполнении программы несколько модулей загрузки. В программе с динамической последовательной структурой используется несколько модулей загрузки. Для управления модулями загрузки и установления связей между ними используются следующие четыре макрокоманды:

1. LINK (размещение модуля загрузки в основной памяти и последующее его выполнение),
2. XCTL, LOAD и DELETE/ Макрокоманда LINK, обеспечивает. Управление вызывающей программе возвращает макрокоманда RETURN, которая освобождает область памяти, занятую модулем, но не перераспределяет ее. Позже, если вновь требуется тот же самый модуль загрузки (и этот модуль загрузки имеет необходимые атрибуты), а его копия до сих пор находится в основной памяти и не повреждена, то он используется повторно без обращения к операции «выборки».

3. Макрокоманда XCTL обеспечивает выборку и выполнение модуля загрузки, так же как и макрокоманда LINK-Однако макрокоманда XCTL используется в тех случаях, когда программа выполняется в виде нескольких фаз и выполнение модуля загрузки, содержащего макрокоманду XCTL, заканчивается и больше не возобновляется вновь.

4. Макрокоманда LOAD обеспечивает загрузку модуля загрузки, но не его выполнение. В дальнейшем он используется с помощью обычной команды BRANCH (УСЛОВНЫЙ ПЕРЕХОД). Модуль загрузки, загруженный с помощью макрокоманды LOAD, может быть удален из основной памяти.

5. Программа с динамической параллельной структурой использует макрокоманду ATTACH для создания подзадачи, которая выполняется совместно с породившей ее задачей. Во всех предыдущих случаях существуют только одна задача и только один блок управления задачей (TCB), при динамической параллельной структуре для каждой выполняемой макрокоманды ATTACH создается дополнительная задача. Каждый модуль загрузки, хранящийся в библиотеке программ, может быть одного из трех типов: однократно используемый, повторно используемый и реентерабельный.

1. Однократно используемый модуль загрузки вызывается из библиотеки всякий раз, когда к нему обращаются.

2. Повторно используемый модуль загрузки является самовосстанавливающемся, так что его команды и константы, измененные при предыдущем выполнении, восстанавливаются перед его повторным выполнением.

3. Реентерабельный модуль загрузки не изменяется в ходе своего выполнения. Системные задачи часто разрабатываются в виде реентерабельных модулей и имеют нулевой ключ памяти.

Модули

Модуль в программировании представляет собой функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом или поименованной непрерывной его части, предназначенный для использования в других программах. Модули позволяют разбивать сложные задачи на более мелкие в соответствии с принципом модульности. Обычно проектируются таким образом, чтобы предоставлять программистам удобную для многократного использования функциональность (интерфейс) в виде набора функций, классов, констант. Модули могут объединяться в пакеты и, далее, в библиотеки.

Модули могут быть обычными, т. е. написанными на том же языке, что и основная программа, в которой они используются, либо модулями расширения, которые пишутся на отличном от языка основной программы языке. Модули расширения обычно пишутся на более низкоуровневом языке.

Модульное программирование может быть осуществлено, даже когда синтаксис языка программирования не поддерживает явное задание имён модулям.

Программные инструменты могут создавать модули исходного кода, представленные как части групп — компонентов библиотек, которые составляются программой компоновщиком.

Свойства модулей:

1. стандартная внутренняя структура
2. взаимная независимость
3. параметрическая универсальность
4. функциональная независимость

Планирование

Статические алгоритмы планирования

Планирование выполняется на другом оборудовании, чем выполнение, сначала создаётся план, потом система его выполняет. Например,

RMS – Rate Monotonic Scheduling. Использует приоритетное вытесняющее планирование. Приоритет присваивается каждой задаче до того, как она начала выполняться. Преимущество отдается задачам с самыми короткими периодами выполнения.

Динамические алгоритмы планирования

Планируются на том же оборудовании, что и выполняется. Должно быть быстрым даже в ущерб качеству планирования. Например,

EDF – Earliest Deadline First Scheduling. Приоритет задачам присваивается динамически, причем предпочтение отдается задачам с наиболее ранним предельным временем начала (завершения) выполнения.

Балансное планирование заключается в стремлении к балансировке нагрузки узлов системы (при перегрузке уменьшается нагрузка на узел за счет миграции процессов).

Планирование бывает долгосрочным (планирование заданий) и краткосрочным (планирование использования процессора). Иногда выделяют среднесрочное планирование (своппинг).

Задачи планирования:

1. Справедливость (между пользователями)
2. Эффективность – полностью занять процессор
3. Сокращение полного времени выполнения (turnaround time)
4. Сокращение времени ожидания (waiting time)
5. Сокращение времени отклика (response time)

Планирование бывает для

1. однопроцессорных систем (где 1 процессор и надо распределить время между кучей задач). Используются знакомые уже алгоритмы FCFS (FIFO), RR. И, возможно кому-то незнакомые:
   1. SJF (Shortest-Job-First) – ставит короткие задачи к началу очереди, в результате чего растет производительность. Бывают вытесняющие и невытесняющие.
   2. Гарантированное планирование – каждому пользователю гарантируется равный объем ресурсов.

   В перечисленные алгоритмы могут добавляться приоритеты и дополнительные очереди.

   Планировщики называются временными.
2. многопроцессорных систем

Многопроцессорные системы бывают однородные и неоднородные (в неоднородных системах разное оборудование, соответственно одна и та же заявка на разных узлах может быть обслужена с разной скоростью).

В таких системах могут быть явно указаны пары задача-узел (т.е. конкретная задача может решаться только на конкретных узлах), а могут быть не указаны.

Время решения задачи зависит от производительности узла, где она решается, и от времени пересылок между узлами.

Алгоритмы оптимизируют по времени выполнения (так расположить задачи на узлах, чтобы уменьшить время их решения и кол-во пересылок) и по эффективности загрузки системы (чтобы все узлы были более-менее одинаково загружены).

Алгоритмы: венгерский, Шаркара, Янга, ветвей и границ.

Планировщики называются пространственными или пространственно-временными.

Так, братва, выкладываю кое-что по планированию из того, что делал лично я. Если кто любознательный и хочет сверкнуть силой разума перед Симоном - может поможет:

Алгоритмы планирования заданий можно разделить на две основные группы – алгоритмы, основанные на эвристике, и алгоритмы, основанные на случайном поиске. Эвристические алгоритмы можно глобально разделить на три подкатегории – алгоритмы планирования по спискам, алгоритмы кластеризации и алгоритмы дублирования задач.

1.1 Алгоритмы планирования по спискам

Все алгоритмы планирования по спискам создают список всех заданий графа, упорядоченный но их приоритету. Обычно они разделяются на две фазы – фаза выбора задачи, во время которой выбирается задача с наивысшим приоритетом, и фаза выбора процессора, во время которой выбирается такой процессор, который минимизирует заданную целевую функцию. Среди подобных алгоритмов следует выделить алгоритм Модифицированного Критического Пути (MCP), алгоритм Планирования Динамических Уровней (DLS), алгоритм «Меньшее Время Первое» (ETF), алгоритм Динамического Критического Пути (DCP), а также алгоритм Минимизации Времени Окончания (HEFT). Большинство из подобных алгоритмов рассчитаны на полносвязные однородные системы. Алгоритмы планирования по спискам обычно более производительны и обеспечивают более высокое качество расписаний, чем алгоритмы других категорий.

1.2 Алгоритмы кластеризации

Алгоритмы этого класса присваивают вершины заданного графа неограниченному числу кластеров. На каждом шаге выбираются задачи для кластеризации. На каждой итерации алгоритм улучшает расписание слиянием двух кластеров. Задания, находящиеся в одном кластере, будут выполняться на одном процессоре. Наиболее яркими представителями этой группы можно назвать такие алгоритмы как алгоритм Кластеризации Доминантной Последовательности (DSC), Метод Линейной Кластеризации (LCM), и систему Планирования и Кластеризации (CASS).

1.3 Алгоритмы дублирования заданий

Идея алгоритмов дублирования заданий лежит в том, чтобы при планировании графа приложения некоторые из заданий планировать более чем один раз, что приводит к уменьшению межпроцессорного взаимодействия. Алгоритмы дублирования различаются в зависимости от принятой стратегии дублирования задач. Алгоритмы этой категории обычно предназначены для составления расписаний для неограниченного числа идентичных процессоров и в основном имеют высокую вычислительную сложность.

1.4 Алгоритмы случайного направленного поиска

Алгоритмы случайного направленного поиска используют случайный выбор для направления поиска в пространстве решений. Эти технологии комбинируют знания, полученные на предыдущих шагах с определённым случайным выбором для получения нового результата. Генетические алгоритмы (ГА) являются самыми популярными и широкоиспользуемыми алгоритмами на основе случайного направленного поиска. ГА генерируют хорошие расписания заданий, однако время их работы значительно выше, чем в эвристических алгоритмах. Кроме того, для получения оптимальных результатов необходимо тщательное исследование параметров алгоритма.

Кроме ГА можно отметить еще методы симулирования отжига и методы локального поиска, которые также входят в эту группу.

2. Краткий обзор эвристических алгоритмов для неоднородных систем

В данном разделе представлен краткий обзор двух наиболее известных эвристических алгоритмов статического планирования для неоднородных систем – алгоритм Нормированного - Минимального Времени и алгоритма минимизации времени окончания.

2.1 Алгоритм Нормированного - Минимального Времени

Это двухэтапный алгоритм. На первом этапе задания группируются на основе параметра «уровень», т.е. в одну группу попадают задания, которые могут выполняться параллельно. На втором этапе алгоритм присваивает каждую задачу самому быстрому из доступных процессоров. Задание в более низком уровне имеет более высокий приоритет, чем задание в более высоком уровне. В пределах уровня наивысший приоритет имеет задание с наибольшим временем выполнения. Каждое задание присваивается такому процессору, который минимизирует сумму времени вычисления задания и суммарного времени взаимодействия задания с заданиями из предыдущего уровня. Алгоритм имеет временную сложность O(v2*q2) где v – количество вершин а q – количество процессоров.

2.2 Алгоритм Минимизации Времени Окончания (HEFT)

Данный алгоритм также является представителем класса алгоритмов планирования по спискам. Алгоритм разделяется на две части. На первом этапе каждой задаче присваивается приоритет, на втором этапе каждая вершина из списка присваивается такому процессору, который минимизирует время окончания её выполнения. Есть также разновидность этого алгоритма, которая известна как разновидность Критического Наследника (CC modification). В качестве критического наследника для текущей вершины выбирается такая вершина, которая является наследницей для текущей и имеет наивысший рейтинг (приоритет). Текущая вершина присваивается такому процессору, который минимизирует время завершения критического наследника текущей вершины. Очевидно, такой алгоритм имеет большую временную сложность, поскольку на каждом шаге производится оптимизация на шаг вперед.

По ряду исследований этот алгоритм даёт лучшие результаты планирования при фиксированной временной сложности и служит некоторым эталоном для разработчиков алгоритмов планирования. Поэтому именно этот алгоритм был выбран для реализации в представленной модели.

Для любознательных: линк1 и линк2, а так же вспомните, кто делал, курсач.

Планировщики

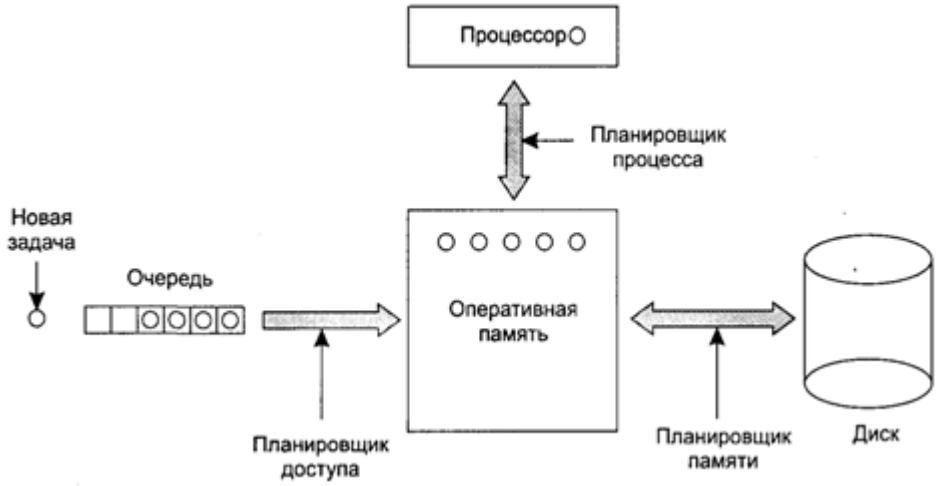
В однопроцессорной системе есть три уровня планировщиков:

1. 1 уровень. планировщик заданий, для долгосрочного планирования процессов (отвечает за прохождение новый процессов в систему и следит за количеством процессов, одновременно находящихся в системе)
2. 2 уровень. промежуточный или среднесрочный планировщик (по сути это своппинг, то есть перекачка всего (или части) процесса на диск (или с диска)).
3. 3 уровень. планировщик использования процессора, для краткосрочного планирования процессов (выбор готового процесса и перемещение его в разряд активных, в

следствии того, что предыдущий процесс обратился к устройству ввода/вывода или у него закончился квант времени)

Структура прохождения задачи через ВычСистему

Описана трёхуровневая модель.



1 уровень. Задание (см. определения выше) кто-то формирует. Программист или система. В первом случае, если задание правильно оформлено (синтаксис), оно накапливается в очереди входных заданий.

Для выбора заданий из очереди может использоваться алгоритм, в котором устанавливается приоритет коротких задач перед длинными. Впускной/входной планировщик волен придержать некоторые задания во входной очереди, а пропустить задание, поступившее позже остальных.

Т.е. на 1 уровне происходит фильтрация заданий, но ресурсы к ним не определяются. В задании должна быть указана программа и данные. В результате, имеем очередь входных заданий.

2 уровень. Если в системе есть свободные ресурсы (см. определение выше), то всплывает планировщик второго уровня. Он выбирает из очереди задание, определяет, можно ли его решить с помощью ресурсов, что есть в системе. Если можно, то вызывает инициализатор/инициатор, который как только определит ресурсы и расшифрует задание, сформирует блок управления задачей/процессом (PCB).

Так исторически сложилось, что сначала был TCB (task control block), а потом появился многопрограммный режим работы, появились процессы и вместе с ними PCB.

Таким образом, задание превращается в задачу/процесс. Возможна ситуация, когда процессов слишком много и они все в памяти не помещаются, тогда некоторые из них будут выгружены на диск. Второй уровень планирования определяет, какие процессы можно хранить в памяти, а какие — на диске. Этим занимается планировщик памяти.

Для оптимизации эффективности системы планировщик памяти должен решить, сколько и каких процессов может одновременно находиться в памяти. Кол-во процессов, одновременно находящихся в памяти, называется степенью многозадачности.

Планировщик памяти периодически просматривает процессы, находящиеся на диске, чтобы решить, какой из них переместить в память. Среди критериев, используемых планировщиком, есть следующие:

1. Сколько времени прошло с тех пор, как процесс был выгружен на диск или загружен с диска?
2. Сколько времени процесс уже использовал процессор?
3. Каков размер процесса (маленькие процессы не мешают)?
4. Какова важность процесса?

3 уровень. Как только процессор освобождён (либо естественно либо с вытеснением), срабатывает планировщик 3-го (верхнего уровня, он же планировщик процессора) и из готовых процессов/задач он должен выбрать процесс/задачу для выполнения и занять время выполнение в процессоре.

4 уровень. Собственно, его можно не выделять отдельно. На последнем этапе результаты выполнения могут быть сохранены или выведены на внешний носитель.

Планирование в параллельных системах

Для планирования в параллельных системах добавляется транспортный планировщик. Он служит для распараллеливания заданий, синхронизации процессов по данным – обеспечения поступления требуемых данных и поддержки связей между вычислительными узлами при реализации связи по данным.

к ПП добавляется функция адаптирования, при выполнении которой задания распределяются соответственно особенностям данной системы (например: специальные схемы для систем гиперкуб, транспьютерных систем или дополнительная схема для неоднородной среды).

к ПН добавляется функция балансирования в случае реконфигурации (отказа некоторых элементов) системы.

В результате мы приходим к семиуровневой модели.

Семиуровневая модель

Полную систему планирования в этом случае можно представить в виде семиуровневой модели, где каждый уровень часто рассматривают как отдельную задачу. Порядок выполнения уровней может быть изменен в зависимости от особенности системы и целей задачи планирования.

1. (низкий уровень) Предварительное (входное) планирование исходного потока заявок (задача фильтрации)
2. (низкий уровень) Структурный анализ взаимосвязи входного потока заявок по ресурсам с определением общих ресурсов (Анализ)
3. (транспортный) Структурный анализ заявок и определение возможности распараллеливания (Задача распараллеливания)
4. (промежуточный) Адаптация распределения работ соответственно особенностям ВС (Задача адаптирования)
5. (промежуточный) Составление плана – расписания выполнения взаимосвязанных процедур. Оптимизация плана по времени решения и кол-ву ресурсов (Задача Оптимизации)
6. (промежуточный) Планирование потока задач претендующих на захват времени процессора на каждый процессор – задача распределения.
7. (высокий) Выделение процессорного времени, активизация задач. Перераспределение работ в ВС, при отказе оборудования (задача распределения – перераспределения).

Загрузчик программы в ОП

Загрузчик – программа, которая подготавливает объектную программу к выполнению и инициирует ее выполнение.

Более детально функции загрузчика следующие:

1. распределение: выделение места для программ в памяти. Для размещения программы в ОП должно быть найдено и выделено свободное место в памяти. Для выполнения этой функции загрузчик обычно обращается к ОС, которая выполняет его запрос на выделение памяти в рамках механизма управления памятью.
2. загрузка: фактическое размещение команд и данных в памяти. Считывание образа программы с диска (или другого внешнего носителя) в оперативную память.
3. связывание: разрешение символических ссылок между объектами. (Связывание с динамическими библиотеками) Компоновка программы из многих объектных модулей. У модулей из-за трансляции по одному, адреса процедур и данных не актуальны и не соответствуют друг другу. Загрузчик же "видит" все объектные модули, входящие в состав программы, и он может вставить в обращения к внешним точкам правильные адреса. Загрузчики, которые выполняют функцию связывания вместе с другими функциями, называются связывающими загрузчиками. Выполнение функции связывания может быть переложено на отдельную программу, называемую редактором связей или компоновщиком.
4. перемещение: настройка всех величин в модуле, зависящих от физических адресов в соответствии с выделенной памятью. Программа разрабатывается в некотором виртуальном адресном пространстве, в котором адресация ведется относительно начала программы. При размещении в памяти программе выделяется свободный участок с реальными адресами. Все величины в программе, которые должны быть привязаны к реальным адресам, должны быть настроены с учетом адреса, по которому программа загружена.
5. инициализация: передача управления на входную точку программы.

Не обязательно функции загрузчика должны выполняться именно в той последовательности, в какой они описаны.

Виды загрузчиков

1. Абсолютный. Выполняет только связывание. Как часть загрузчика может рассматриваться редактор связей (хотя он выполняется отдельно) Абс. загрузчик выделяет загружаемой программе память в пределах памяти, выделенной процессу
2. Настраиваемый. До этапа выполнения программы в модулях описываются векторы переходов (внутри – и внешнемодульные), и константы, которые нужно переопределить. Загрузчик при просмотре этих записей определяет абсолютные адреса настраиваемых величин.
3. Непосредственно связывающий. Динамически выполняет все 4 функции. Выделяет память постранично, выполняет связывание программы динамически, по мере необходимости. Для этого к модулю прикрепляются 2 таблицы: ESD и RLD (таблицы векторов переходов и адресных констант).

Редактор связей

Редактор связей выполняет только функцию связывания – сборки программы из многих объектных модулей и формирование адресов в обращениях к внешним точкам. На выходе редактора связей мы получаем загрузочный модуль.

Загрузчик ОС. Схема загрузки ОС

Загрузчик BIOS

Выполняет:

1. инициализацию основных компонентов материнской платы;
2. обслуживает системные прерывания (как аппаратные, так и программные);
3. из Main Boot Record считывает первые 512 байт (бутовый/начальный загрузчик) в ОП. Передает ему управление.

Бутовый (первичный) загрузчик

Определяет активный раздел. Обращается к месту на жестком диске где записан основной загрузчик (обычно к 0 разделу 0 дорожки) и загружает основной загрузчик в память.

Основной (вторичный) загрузчик

Основной загрузчик системы инициализирует некоторые из подсистем (система ввода/вывода => файл конфигурации). Он может загружать несколько операционных систем, давая пользователю выбирать, какую из систем нужно загружать в конкретном случае. После выбора загружаемой системы, загрузчик проводит необходимые приготовления (к примеру, переводит процессор в защищенный режим работы) и начинает загрузку частей ядра в ОП (программы обработки прерываний, управление памятью, управление процессами). После этого загрузчик передает управление ядру (при этом возможна передача параметров. К примеру, при загрузке Linux ядру можно передавать настройки графического и режима и другие параметры).

После формирования ядра начинает работу программа инициализации системы. Подгружается командный интерпретатор (он грузится последним потому, что мы можем указать свой собственный интерпретатор).

Библиотеки

Понятие библиотек появилось еще в первом поколении машин. Но “настоящие” библиотеки появились при разработке компиляторов.

Структура:

1. библиотека
2. каталог
3. управляющая программа

Динамические библиотеки

Часть основной программы, которая загружается в ОС по запросу работающей программы в ходе её выполнения (Run-time), т.е. динамически (Dynamic Link Library, DLL в Windows). Один и тот же набор функций (подпрограмм) может быть использован сразу в нескольких работающих программах, из-за чего они имеют ещё одно название — библиотеки общего пользования (Shared Library). Если динамическая библиотека загружена в адресное пространство самой ОС (System Library), то единственная копия может быть использована множеством работающих с нею программ.

При написании программы программисту достаточно указать транслятору, что следует подключить нужную библиотеку и использовать функцию из неё.

Если библиотеки не окажется в системе, программа может не загружаться.

Статические библиотеки

Могут быть в виде исходного текста, подключаемого программистом к своей программе на этапе написания, либо в виде объектных файлов, присоединяемых (линкуемых) к исполняемой программе на этапе компиляции. В результате программа включает в себя все необходимые функции, что делает её автономной, но увеличивает размер. Без статических библиотек объектных модулей (файлов) невозможно использование большинства современных компилирующих языков и систем программирования: Fortran, Pascal, C, C++ и других.

Прерывания

Прерывание — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

Классы прерываний

В зависимости от источника возникновения сигнала прерывания делятся на:

1. асинхронные или внешние (аппаратные) — события, которые исходят от внешних источников (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание (англ. Interrupt request, IRQ);
2. синхронные или внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение, обращение к недопустимым адресам или недопустимый код операции;
3. программные (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания как правило используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.

Фазы прерывания

Любая особая ситуация, вызывающая прерывание, сопровождается сигналом, называемым запросом прерывания (ЗП). Запросы прерываний от внешних устройств поступают в процессор по специальным линиям, а запросы, возникающие в процессе выполнения программы, поступают непосредственно изнутри микропроцессора.

После появления сигнала запроса прерывания ЭВМ переходит к выполнению программы-обработчика прерывания. Обработчик выполняет те действия, которые необходимы в связи с возникшей особой ситуацией. Например, такой ситуацией может быть нажатие клавиши на клавиатуре компьютера. Тогда обработчик должен передать код нажатой клавиши из контроллера клавиатуры в процессор и, возможно, проанализировать этот код. По окончании работы обработчика управление передается прерванной программе.

Эта операция называется переключением контекста. При реализации переключения используются слова состояния программы (PSW), с помощью которых осуществляется управление порядком выполнения команд. В PSW содержится информация относительно состояния процесса, обеспечивающая продолжение прерванной программы на момент прерывания.

Существуют три типа PSW: текущее, новое и старое PSW. Адрес следующей команды (активной программы), подлежащей выполнению, содержится в текущем PSW, в котором также указываются и типы прерываний, разрешенных и запрещенных в данный момент.

В новых PSW содержатся адреса, по которым резидентно размещаются программы обработки прерываний. При возникновении разрешенного прерывания, в одном из старых PSW система сохраняет содержимое текущего PSW – адрес следующей команды данного процесса, которая должна выполниться по окончании обработки прерывания и передаче управления данному прерванному процессу (т.е. адрес команды, которая следует за текущей в данном процессе и которая должна была бы выполниться при отсутствии сигнала прерывания). Таким образом, обеспечивается корректное возвращение в прерванный процесс после обработки прерывания.

Время реакции – это время между появлением сигнала запроса прерывания и началом выполнения прерывающей программы (обработчика прерывания) в том случае, если данное прерывание разрешено к обслуживанию.

Время реакции зависит от момента, когда процессор определяет факт наличия запроса прерывания. Опрос запросов прерываний может проводиться либо по окончании выполнения

очередного этапа команды (например, считывание команды, считывание первого операнда и т.д.), либо после завершения каждой команды программы.

Первый подход обеспечивает более быструю реакцию, но при этом необходимо при переходе к обработчику прерывания сохранять большой объем информации о прерываемой программе, включающей состояние буферных регистров процессора, номера завершившегося этапа и т.д. При возврате из обработчика также необходимо выполнить большой объем работы по восстановлению состояния процессора.

Во втором случае время реакции может быть достаточно большим. Однако при переходе к обработчику прерывания требуется запоминание минимального контекста прерываемой программы (обычно это счетчик команд и регистр флагов). В настоящее время в компьютерах чаще используется распознавание запроса прерывания после завершения очередной команды.

Время реакции определяется для запроса с наивысшим приоритетом.

Насыщение системы прерываний

Если запрос на прерывание окажется необслуженным к моменту прихода нового запроса от того же источника, то возникает так называемое насыщение системы прерываний. В этом случае предыдущий запрос прерывания от данного источника будет машиной утрачен, что недопустимо. Быстродействие ЭВМ, характеристики системы прерываний, число источников прерывания и частоты возникновения запросов должны быть согласованы таким образом, чтобы насыщение было невозможным

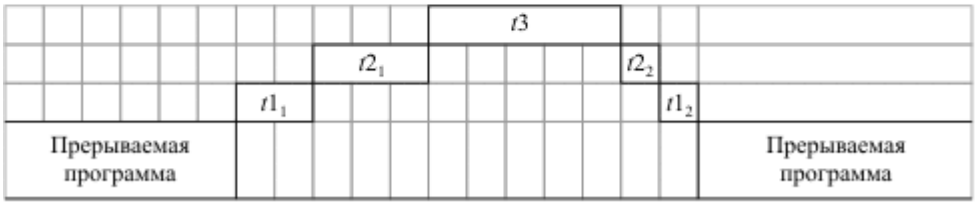
Насыщение системы прерываний возможно при неправильной настройке приоритетов (зацикливание приоритетов), а также при чрезмерном увеличении устройств и процессов, вызывающих прерывания – система постоянно находится в режиме прерывания

Глубина прерывания

Глубина прерывания – максимальное число программ, которые могут прерывать друг друга. Глубина прерывания обычно совпадает с числом уровней приоритетов, распознаваемых системой прерываний. Или же: глубина системы прерывания – это степень вложенности прерываний. Она определяется количеством старых PSW, которые можно поместить в постоянную область памяти или в стек.

Работа системы прерываний при различной глубине прерываний (n) представлена на рис. 14.2. Здесь предполагается, что с увеличением номера запроса прерывания увеличивается его приоритет.

а) $n = 1$



б) $n = 3$

Рис. 14.2. Работа системы прерываний при различной глубине прерываний

Зацикливание прерываний

Это случай когда первая программа прерывает вторую, вторая - третью, третья - первую.

Обработка команд ввода-вывода

Есть три фундаментально различных способа осуществления операций ввода- вывода: программный ввод-вывод, ввод- вывод, управляемый с помощью прерываний, и ввод-вывод, использующий DMA.

Как обрабатывается прерывание ввода/вывода (из шпоры)

1. Обращение к супервизору
2. Сохранение текущего PSW в старый PSW. Выполнение дешифрации прерывания
3. Сохранение нового PSW в текущий
4. Переход по SVC на обработчик прерывания
5. Подготовка операций ввода/вывода

6. В адресное слово канала записывается адрес начала канальной программы
7. Запуск канальной программы
8. Команда SIO
9. Передача ус-вом сигнала об усп./неусп. старте – в слове сост. Канала CSW.
10. Обработчик прерываний запрашивает ответ о успешном старте
11. Ввод-вывод
12. Обработчик анализирует CSW на сигнал о завершении ввода/вывода, записывает старый PSW в текущий

Все. Дальше – продолжение основной программы