Архитектура ОС. Классическая архитектура ядра

Архитектура ОС. Микроядерная архитектура. Достоинства и недостатки микроядерной архитектуры

Аппаратная зависимость и переносимость ОС

Основные задачи и функции подсистемы управления процессами

Подсистема управления процессами. Состав процесса

Подсистема управления процессами. Четыре вида таблиц с информацией по каждому объекту управления

Планирование процессов и потоков. Критерии планирования и требования к алгоритмам планирования

Планирование процессов и потоков. Основные виды ресурсов

Планирование процессов и потоков. Состояние потоков (процессов)

Планирование процессов и потоков. Статическое и динамическое планирование потоков в ОС

Планирование процессов и потоков. Типы планирования: долгосрочное, среднесрочное и краткосрочное

Подсистема управления процессами. Местоположение процесса. Управляющий блок процесса. Образ процесса

Диспетчеризация процессов и задач. Дисциплины диспетчеризации, основанные на приоритетах

Диспетчеризация процессов и задач. Дисциплины диспетчеризации, основанные на квантовании

Диспетчеризация процессов и задач. Смешанные дисциплины диспетчеризации (на примере ОС Windows)

Диспетчеризация процессов и задач. Смешанные дисциплины диспетчеризации (на примере ОС OS/2)

Диспетчеризация процессов и задач. Смешанные дисциплины диспетчеризации (на примере ОС UNIX)

Управление памятью в операционных системах. Простое непрерывное распределение и распределение с перекрытием

Управление памятью в операционных системах. Распределение памяти статическими разделами

Управление памятью в операционных системах. Распределение памяти динамическими разделами

Понятие виртуальной памяти. Механизм преобразования виртуального адреса в физический при страничной организации памяти

Понятие виртуальной памяти. Сегментное распределение

Понятие виртуальной памяти. Страничное распределение

Синхронизация процессов и потоков. Необходимость взаимодействия процессов. Два механизма взаимодействия процессов

Синхронизация процессов и потоков. Взаимное исключение. Требования к взаимным исключениям

Синхронизация процессов и потоков. Критическая секция

Синхронизация процессов и потоков. Аппаратная поддержка взаимоисключения

Синхронизация процессов и потоков. Семафоры Дейкстры

Проектирование параллельных процессов. Алгоритм Деккера. Алгоритм Петерсона

Типовые задачи, требующие организации параллельных вычислительных процессов. Задача «Писатель-читатель»

Типовые задачи, требующие организации параллельных вычислительных процессов. Задача «Поставщик-потребитель»

Типовые задачи, требующие организации параллельных вычислительных процессов. Задача «Обедающие философы»

Тупики

Понятия и определения локальных, сетевых и распределенных операционных систем

Организация многопроцессорных операционных систем. Симметричная схема

Организация многопроцессорных операционных систем. Схема ведущий–ведомый

Организация многопроцессорных операционных систем. Схема с раздельными ядрами

Планирование в многопроцессорных системах. Задачно-независимые алгоритмы планирования

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Круговой алгоритм планирования

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Алгоритм копланирования

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Алгоритм SNPF

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Динамическое разделение

Миграция процессов. Процедура и концепции миграции процессов

Миграция процессов. Стратегии миграции процессов

Синхронизация в распределенных системах. Синхронизация часов

Синхронизация в распределенных системах. Распределенные алгоритмы синхронизации

Синхронизация в распределенных системах. Алгоритмы голосования

Синхронизация в распределенных системах. Централизованные алгоритмы синхронизации

Методы борьбы с тупиками в распределенных операционных системах

Архитектура операционной системы Android

Архитектура операционной системы QNX

Архитектура операционной системы Windows

Архитектура ОС. Классическая архитектура ядра

Общие положения

Наиболее общим подходом к структуризации ОС является разделение всех ее модулей на 2 группы:

- модули, выполняющие основные функции ОС – ядро;
- модули, выполняющие вспомогательные функции ОС.

Функции модуля ядра:

- управление памятью
- процессами
- устройствами ввода/вывода
- переключение контекстов
- загрузка/выгрузка страницы
- обработка прерываний
- создание прикладной программной среды

Приложения обращаются к ядру с запросами через системные вызовы. Скорость работы ядра определяет производительность всей системы.

Вспомогательные модули ОС обычно подразделяются на следующие группы:

1. утилиты – это программы решающие отдельные задачи управления и сопровождения вычислительной системы;
2. системные обрабатывающие программы, к ним относятся текстовые и графические редакторы, компиляторы, компоновщики и отладчики;
3. прикладное ПО;
4. библиотеки процедур различного назначения, упрощающие разработку приложений.

Привилегированный режим для ОС:

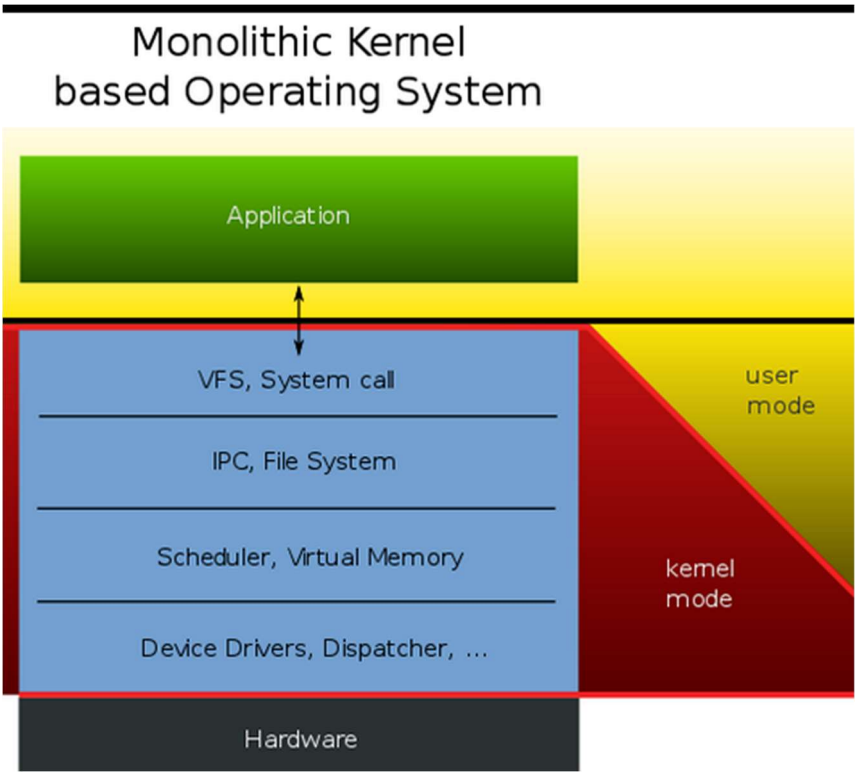
- надежное управление ходом выполнения приложений
- распределение ресурсов между процессами
- защита самой ОС от других процессов

Аппаратная поддержка режимов

- user mode – непривилегированный режим
- kernel mode – привилегированный режим

Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению, переключение из привилегированного в пользовательский.

Классическая архитектура ядра



Система состоит из иерархии слоев, каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс.

Основные слои:

1. средства аппаратной поддержки ОС – средства, которые прямо участвуют в организации вычислительных процессов, это

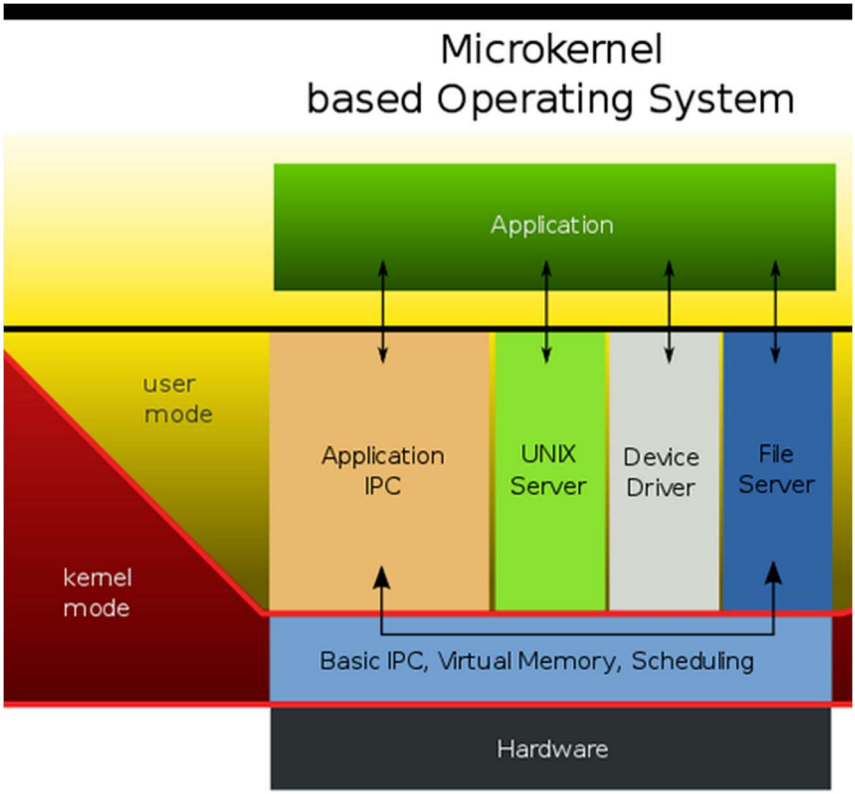
средства поддержки привилегированного режима, прерываний, переключение контекстов, защиты и распределения памяти;
2. машинно-зависимые компоненты ОС – этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера, этот слой должен полностью экранировать вышележащие слои ядра от особенностей аппаратуры;
3. базовые механизмы ядра – примитивные операции ядра (программное переключение контекстов процессов, диспетчеризация прерываний, перемещение страниц из памяти на диск и т. д); исполнительными механизмами для модулей верхних слоев;
4. менеджеры ресурсов – управление основными ресурсами вычислительной системы (менеджеры процессов, менеджеры ввода вывода, файловой системы, менеджеры оперативной памяти и т. д.);
5. интерфейс системных вызовов – этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами; прикладной программный интерфейс ОС.

Для взаимодействия между слоями существуют строгие правила, но между модулями внутри слоя связи могут быть произвольными. Отдельный модуль может выполнить свою работу либо самостоятельно, либо обратиться к другому модулю своего слоя, либо обратиться за помощью к нижележащему слою через межслойный интерфейс.

Многослойная организация системы существенно упрощает разработку, так как позволяет сначала определить сверху вниз функции слоев и межслойные интерфейсы, а затем при детальной реализации постепенно наращивать мощность функций слоев двигаясь снизу вверх. Кроме того, при модернизации системы можно изменять модули внутри слоя без необходимости производить какие-либо изменения в остальных слоях.

Архитектура ОС. Микроядерная архитектура. Достоинства и недостатки микроядерной архитектуры

Микроядерная архитектура



Суть состоит в том, что в привилегированном режиме остается работать только небольшая часть ОС, называемая микроядром.

Микроядро защищено от остальных частей ОС и приложений.

Состав микроядра:

- Базовые функции по обработке прерываний
- Управление памятью
- Inter-Process Communication - IPC
- Управление устройствами ввода/вывода (чтение/запись в порты)

Микроядро <=> слой базовых механизмов монолитного ядра.

Все остальные более высокоуровневые функции ядра формируются в виде приложений, работающих в пространстве пользователя.

Для взаимодействия между процессами-частями ОС необходим механизм вызова процедур одного процесса из другого (IPC).

Менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС

Достоинства:

- Высокая степень переносимости
- Расширяемость ОС
- Хорошо подходит для модели распределенных вычислений
- Повышенная надежность (сервер упал – сервер перезапустили)

Недостаток: низкая производительность (особенно при взаимодействии серверов ОС: приложение - микроядро - сервер – микроядро – приложение).

Аппаратная зависимость и переносимость ОС

Типовые средства аппаратной поддержки ОС:

1. средства поддержки привилегированного режима. Основаны на системном регистре процессора. В обязанности средств поддержки привилегированного режима входит выполнение проверки допустимости выполнения активной программой инструкций процессора при текущем уровне привилегированности
2. средства трансляции адресов. Выполняют преобразования виртуальных адресов, которые содержаться в кодах процесса, в адреса физической памяти
3. средства переключения процессов. Предназначены для быстрого сохранения контекста приостанавливаемого процесса и восстановления контекста процесса, который становиться активным. Содержимое контекста обычно включает содержимое РОН, регистра флагов, системных регистров и указателей
4. система прерываний
5. системный таймер
6. средства защиты областей памяти

Машинно-зависимые компоненты ОС

Объем машинно-зависимых компонентов ОС зависит от того, насколько велики отличия в аппаратных платформах, для которых разрабатывается ОС. Одно из наиболее очевидных отличий – несовпадение системы команд процессоров – преодолевается просто. ОС программируется на языках высокого уровня.

Для уменьшения количества машинно-зависимых модулей производители ОС обычно ограничивают универсальность машинно-независимых модулей.

Для компьютеров на основе процессора Intel разработка машинно-зависимого слоя ОС упрощается за счет встроенной в ПЗУ базовой системы ввода-вывода – BIOS.

Переносимость ОС

Если код ОС может быть сравнительно легко перенесен с процессора одного типа на процессор другого типа и с аппаратной платформы одного типа на аппаратную платформу другого типа, то такую ОС называют переносимой. Для того чтобы обеспечить свойство мобильности ОС, разработчики должны следовать следующим правилам:

1. Большая часть кода должна быть написана на языке, трансляторы которого имеются на всех машинах, куда предполагается переносить систему
2. Объем машинно-зависимых частей кода, которые непосредственно взаимодействуют с аппаратными средствами, должен быть по возможности минимизирован. Так, например, следует всячески избегать прямого манипулирования регистрами и другими аппаратными средствами процессора. Необходимо исключить возможность использования по умолчанию стандартных конфигураций аппаратуры и их характеристик.
3. Аппаратно-зависимый код должен быть надежно изолирован в нескольких модулях, а не быть распределен по всей системе. Изоляции подлежат все части ОС, которые отражают специфику как процессора, так и аппаратной платформы в целом. Для снятия платформенной зависимости, возникающей из-за различий между компьютерами разных производителей, построенными на одном и том же процессоре, должен быть введен хорошо локализованный программный слой машинно-зависимых функций (Hardware Abstraction Layer).

В идеале слой машинно-зависимых компонент ядра полностью экранирует остальную часть ОС от конкретных деталей аппаратной платформы. В результате происходит подмена аппаратуры некой унифицированной виртуальной машиной, одинаковой для всех вариантов аппаратной платформы.

Основные задачи и функции подсистемы управления процессами

Подсистема управления процессами:

- Планирует выполнение процессов;
- Создает и уничтожает процессы;
- Обеспечивает процессы необходимыми системными ресурсами;
- Поддерживает взаимодействие между процессами.

Для этого в памяти поддерживаются специальные информационные структуры, в которые система записывает, какие ресурсы выделены каждому процессу.

Можно назначить процессу ресурсы:

- В единоличное пользование;
- В совместное пользование с другими ресурсами.

Ресурсы могут выделяться процессу:

- При его создании;
- Динамически, по запросам во время выполнения.

Ресурсы могут быть приписаны процессу:

- На все время его жизни;
- На определенный период.

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

- Управление памятью;
- Подсистема ввода вывода;
- Файловая система.

Подсистема управления процессами. Состав процесса

Общие положения

Процесс – это абстракция, описывающая выполняющуюся программу. Для ОС процесс представляет собой единицу работы, заявку на потребление системных ресурсов.

Подсистема управления процессами:

- планирует выполнение процессов
- занимается созданием и уничтожение процессов
- обеспечивает процессы необходимыми системными ресурсами
- поддерживает взаимодействие между процессами

В ОС, где существуют процессы и потоки, процесс рассматривается ОС как заявка на потребление всех видов ресурсов кроме процессорного времени.

Процесс - это контейнер для набора ресурсов, используемых потоками. Процессорное время распределяется между потоками, которые представляют собой последовательности команд.

В Windows NT/2000/XP/7 процесс включает:

- Закрытое виртуальное адресное пространство - диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- Используемую программу - начальный код и данные, проецируемые на виртуальное адресное пространство процессора;
- Список открытых описателей различных системных ресурсов - семафоров, коммуникационных портов, файлов и других объектов, доступных всем потокам в данном процессе;
- Контекст защиты, называемый маркером доступа и идентифицирующий пользователя, группы безопасности и привилегии, сопоставленный с процессом;
- Идентификатор процесса - уникальное значение, которое идентифицирует процесс в рамках операционной системы;

Каждый процесс имеет минимум один поток. Поток - некая сущность внутри процесса, получающая процессорное время для выполнения.

Без потока программа процесса не может выполняться.

Поток включает следующие наиболее важные элементы:

- Набор регистров процессора, отражающих состояние процессора;
- Два стека, один из которых используется потоком при выполнении в режиме ядра, а другой в пользовательском режиме;
- Закрытую область памяти, называемую локальной памятью потока и используемую подсистемами библиотеками исполняющих систему;
- Уникальный идентификатор потока.

Подсистема управления процессами. Четыре вида таблиц с информацией по каждому объекту управления

Поскольку задачи операционной системы входит управление процессами и ресурсами, она должна располагать информацией о текущем состоянии процессов и ресурса. Операционная система создает и поддерживает четыре различных вида таблиц с информацией по каждому объекту управления:

- Для памяти;
- Устройств ввода-вывода;
- Файлов;
- Процессов.

Таблицы для памяти содержат:

- Объем основной памяти, отведенной процессу;
- Объем вторичной памяти, отведенной процессу;
- Все атрибуты защиты блоков основной или виртуальной, как, например указание, какой из процессов имеет доступ к той или иной совместно используемой области памяти;
- Вся информация, необходимая для управления виртуальной памятью.

Таблицы ввода-вывода:

- Используется операционной системой для управления устройствами ввода вывода и каналами компьютерной системы;
- В каждый момент времени устройство ввода вывода может быть либо свободно, либо отдано в распоряжение какому-то определенному процессу;
- Если выполняется операция ввода вывода, операционная система должна иметь информацию о ее состоянии и о том, какие адреса основной памяти задействованы в этой операции в качестве источника вывода или места, куда передаются данные при вводе.

Таблицы файлов:

- Информация о существующих файлах;
- Об их расположении на магнитных носителях;
- О текущем состоянии и других атрибутах.

Таблицы процессов:

- Так как управление памятью, устройствами ввода-вывода и файлами осуществляется для того, чтобы могли выполняться процессы, поэтому в таблицах процессов должны быть явные или неявные ссылки на эти ресурсы;
- Сами таблицы должны быть доступны для операционной системы, поэтому место для них выделяется системой управления памятью.

Планирование процессов и потоков. Критерии планирования и требования к алгоритмам планирования

Критерии планирования процессов

- Равномерная загрузка ЦП.
- Пропускная способность процессора, измеряемая количеством процессов, которые выполняются в единицу времени.
- Время оборота или полное время выполнения — это интервал времени от момента появления процесса во входной очереди до момента его завершения. Это время включает в себя время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения на процессоре, время ввода-вывода.
- Время ожидания, под которым понимается суммарное время нахождения процесса в очереди готовых процессов.
- Время отклика — это время, прошедшее от момента попадания процесса во входную очередь до момента первого обращения к терминалу.

Планирование потоков включает в себя решение 2 задач:

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

Планирование процессов и потоков. Основные виды ресурсов

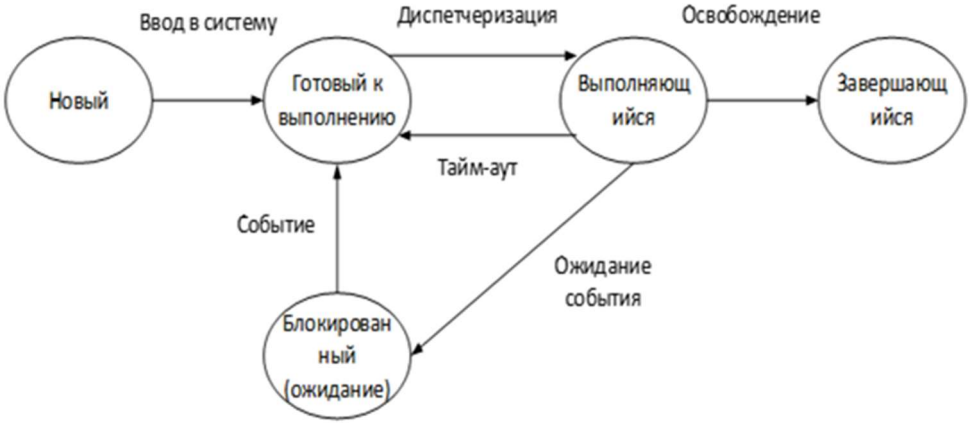
Основные ресурсы:

- ЦП - является не разделяемым ресурсом, процессорное время предоставляется по переменно и ОС должна организовать очереди к процессору
- ОП - разделяемый ресурс - он предоставляется нескольким программам, но дело в том что за тот квант времени что предоставляется программе, работа идет с небольшим участком памяти
- Внешняя память
    - Сама память - разделяемый ресурс
    - Доступ к памяти - разделяется попеременно
- Внешние устройства
    - Разделяемый ресурс если есть механизм прямого доступа
    - Неразделяемый при последовательном доступе
- Программные модули
    - Однократно используемые
    - Повторно используемые
        - Привилегированные
        - Непривилегированные
        - Реентерабельные (при прерывании модуля, его состояние сохраняется для последующего восстановления)
        - Повторно выводимые (модуль делиться на секции, которые выполняются в привилегированном режиме)
- Информационные ресурсы
    - Переменные и файлы
- Разделяемый ресурс если только для чтения
- Не разделяемый при изменении

Планирование процессов и потоков. Состояние потоков (процессов)

Процесс – это абстракция, описывающая выполняющуюся программу. Для ОС процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, т. е. распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожение процессов, обеспечивает процессы

необходимыми системными ресурсами, поддерживает взаимодействие между процессами.



Планирование процессов и потоков. Статическое и динамическое планирование потоков в ОС

Цель планирования состоит в распределении во времени процессов, выполняемых процессором таким образом, чтобы удовлетворить таким требованиям системы как время ожидания, пропускная способность и эффективность работы процессора.

В многопоточных ОС планирование выполнения потоков осуществляется независимо от того, принадлежат ли они одному или разным процессам.

Планирование потоков включает в себя решение двух задач:

1. Определение момента времени для системы текущего активного потока;
2. Выбор для выполнения потока из очереди готовых потоков.

Динамическое планирование

В большинстве ОС универсального назначения планирование осуществляется динамически, то есть решения принимаются во время работы системы на основе анализа текущей ситуации.

ОС работает в условиях неопределенности, так как потоки и процессы появляются в случайные моменты времени и также непредсказуемо завершаются.

Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации.

Статическое планирование

Может быть использован в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее (в системах реального времени).

Планировщик называется статическим, если принимает решение о планировании не во время работы системы, а заранее (предварительный анализ).

Результатом работы статического планировщика является таблица, в которой указывается к какому потоку или процессу, когда и на какое время должен быть предоставлен процессор.

Для построения такой таблицы планировщику нужны полные предварительные знания о характеристиках набора задач. После того, как таблица готова, она используется ОС для переключения потоков и процессов. При этом накладные расходы ОС сводятся лишь к диспетчеризации потоков или процессов.

Планирование процессов и потоков. Типы планирования: долгосрочное, среднесрочное и краткосрочное

Во многих ОС планирование разбивается на три отдельные функции: долгосрочное, среднесрочное и краткосрочное планирование.

Основное отличие между долгосрочным и краткосрочным планированием заключается в частоте запуска.

Долгосрочное планирование осуществляется при создании нового процесса и представляет собой решение о добавлении нового процесса к множеству готовых в настоящий момент процессов в случае освобождения ресурсов памяти.

Желательно, чтобы долгосрочный планировщик создавал неоднородную мультипрограммную очередь, то есть чтобы в очереди находились процессы, ориентированные на преимущественно работу с процессором.

Среднесрочное планирование является частью свопинга и представляет собой решение о добавлении процесса к множеству частично расположенных в основной памяти.

Краткосрочное планирование является решением о том, какой из готовых к выполнению процессов будет выполняться следующим.

Планирование потоков осуществляется на основе информации, хранящейся в дескрипторах процессов и потоков.

При планировании может приниматься во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и другие факторы.

Подсистема управления процессами. Местоположение процесса. Управляющий блок процесса. Образ процесса

Процессу должен быть выделен такой объем памяти, в котором поместились бы программа и данные, принадлежащие процессу;

При работе программы обычно используется стек, с помощью которого реализуются вызовы процедур и передача параметров.

Местоположение процесса. Процессу должен быть выделен такой объем памяти, в котором поместилась бы программа и данные.

С каждым процессом связано несколько атрибутов, которые используются операционной системой для управления этим процессом. Такой набор атрибутов называется **управляющим блоком процесса** (process control block) (Часто используются другие названия этой структуры данных - блок управления задачей, **дескриптор процесса**, дескриптор задачи);

Множество, в которое входят программа, данные, стек и атрибуты, называются **образом процесса**.

Чтобы операционная система могла управлять процессом, по крайней мере небольшая часть его образа должна находиться в основной памяти;

Чтобы можно было запустить процесс, его образ необходимо полностью загрузить в основную (или виртуальную) память;

Таким образом, операционной системе нужно знать местонахождение каждого процесса на диске, а для тех процессов, которые загружены в основную память - их местонахождение в основной памяти.

Диспетчеризация процессов и задач. Дисциплины диспетчеризации, основанные на приоритетах

Общие положения

Диспетчеризация заключается в реализации найденного в результате планирования решения. То есть, переключение процессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст с тем, чтобы впоследствии использовать эту информацию для последующего возобновления выполнения данного потока. Контекст отражает, во-первых, состояние аппаратуры компьютера в момент прерывания потока: значение IP, содержимое РОН, режим работы процессора, флаги, маски прерываний и др. Во-вторых, контекст включает параметры операционной среды, а именно ссылки на открытые файлы, данные о незавершенных операциях ввода-вывода, коды ошибок выполняемых данным потоком системных вызовов и др.

Диспетчеризация сводиться к следующему:

1. Сохранение контекста текущего потока, который требуется сменить.
2. Загрузка контекста нового потока, выбранного в результате планирования.
3. Запуск нового потока на выполнение.

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

1. прерывание от таймера (истек квант времени);
2. запрос на ввод/вывод или на доступ к ресурсу, который сейчас занят. Задача переходит в состояние ожидания;
3. освобождение ресурса. Планировщик проверяет, не ожидает ли этот ресурс какая-либо задача. Если да, то эта задача переводится из состояния ожидания в состояние готовности;
4. аппаратное прерывание, которое сигнализирует о завершении периферийным устройством операции ввода/вывода,

переводит соответствующую задачу в очередь готовых и выполняется перепланирование;

5. внутренние прерывания, сигнализирующие об ошибке, которая произошла в результате выполнения активной задачи. Планировщик снимает задачу и выполняет перепланирование;
6. запросы приложений и пользователей на создание новой задачи или повышения приоритета существующей задачи.

Дисциплины с приоритетами

Приоритетное обслуживание предполагает наличие потоков, некоторые изначально известные характеристики, у которых определяется порядок их выполнения.

В большинстве ОС, поддерживающих потоки, приоритет потока непосредственно связан с приоритетом процесса, в рамках которого выполняется данный поток.

Приоритет процессу назначается ОС при его создании. Значение приоритета включается в описатель процесса и используется.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменения приоритетов могут происходить по инициативе самого потока, когда обращается с соответствующим вызовом к ОС или по инициативе пользователя, когда он выполняет соответствующую команду.

Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты изменяются динамические, в отличие от неизменяемых фиксированных приоритетов.

Существуют 2 разновидности приоритетного планирования:

- обслуживание с относительными приоритетами;
- обслуживание с абсолютными приоритетами.

В обоих случаях выбор потока на выполнение из очереди готовых осуществляется одинаково – выбирается поток, имеющий наивысший приоритет. Но момент времени выполнения потока выбирается по-разному.

В системах с **относительными приоритетами** поток выполняется до тех пор, пока не закончится квант времени или пока он сам не покинет процессор, перейдя в состояние ожидания.

В системах с **абсолютными приоритетами** выполнение активного потока может прерваться еще и в случае, если очереди готовых потоков появился поток, приоритет которого выше приоритета

выполняемого потока. В этом случае прерванный поток переходит в состояние активности.

Диспетчеризация процессов и задач. Дисциплины диспетчеризации, основанные на квантовании

В основе многих вытесняющих алгоритмов лежит концепция квантования, в соответствии с которой каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени – квант. Смена активного потока происходит если:

1. Поток завершился и покинул систему;
2. Произошла ошибка;
3. Поток перешел в состояние ожидания;
4. Исчерпан квант процессорного времени, отведенный данному потоку.

Кванты, выделяемые потоком, могут быть одинаковыми для всех потоков или различными.

Чем больше потоков в системе, тем больше время ожидания и тем меньше возможности вести одновременную интерактивную работу нескольким пользователям.

В алгоритмах, основанных на квантовании, не используется никакой предварительной информации о задачах. При поступлении задачи на обработку ОС не имеет никаких сведений о том, является ли эта задача короткой или длинной, насколько интенсивными будут запросы к устройствам ввода-вывода, на сколько важно её быстрое выполнение.

Алгоритмы планирования, основанные на квантовании

Простейшая дисциплина диспетчеризации является дисциплина **FCFC**, согласно которой задачи обслуживаются в порядке их выполнения. Те задачи, которые были заблокированы в процессе работы, после перехода в состояние готовности ставятся в ту же очередь или для них организуется новая очередь, задача из которой выполняется раньше тех, которые еще не выполнялись. Эта служба не требует внешних вмешательств в ход вычислений. К достоинствам этой дисциплины относятся простота реализации и малые расходы системных ресурсов на формирование форм задачи.

SJN (shortest job next). Дисциплина SJN требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Необходимость сообщать системе характеристики задач, в которых описывались бы потребности в ресурсах вычислительной системы специальные язык управления заданиями. Система ведет подсчет реальных потребностей и сравнивает заказанное время и время выполнения и, в случае превышения данной оценки в данном ресурсе, задание ставиться в конец очереди.

SRT (shortest remaining time). В этой дисциплине первым на выполнение ставится поток, у которого осталось минимальное время до его завершения.

RR (round robin). Дисциплина предполагает, что каждая задача получает квант времени, по истечении которого, она снимается с процесса и процессорное время передается следующей задаче. Задача ставится в конец очереди задач, готовых к выполнению. Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задаче. Величина кванта времени выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей и накладными расходами на частую смену контекста задач.

Диспетчеризация процессов и задач. Смешанные дисциплины диспетчеризации (на примере ОС Windows)

В ОС Windows NT определено 32 уровня приоритетов и 2 класса потоков:

1. Потоки реального времени (16-31)
2. Потоки с переменными приоритетами (1-15)

При создании процесса он получает по умолчанию базовый приоритет, который может быть понижен или повышен ОС.

Первоначально поток получает значение базового приоритета процесса, в котором был создан. Приоритет ±2.

С течением времени приоритет потока, относящийся к классу потоков с переменными приоритетами, может отклоняться от базового приоритета потока, причем эти изменения могут быть не связаны с

изменениями базового приоритета процесса. ОС может повышать приоритет потока в тех случаях, когда поток не полностью использовал отведенный ему квант или понижать приоритет, если квант был использован полностью. ОС наращивает приоритет дифференцированно, в зависимости от того, какого типа событие не дало потоку полностью использовать квант. В частности, ОС повышает приоритет в большей степени потокам, которые ожидают ввода с клавиатуры и, в меньшей степени, потокам, выполняющим дисковые операции. Начальной точкой для динамического приоритета является значение базового приоритета потока. Приоритет потока не может опуститься ниже его базового приоритета, а верхней его границей является значение 15.

Диспетчеризация процессов и задач. Смешанные дисциплины диспетчеризации (на примере ОС OS/2)

В ОС OS/2 используется квантование и абсолютные динамические приоритеты. В каждом их 4 приоритетных классов имеется 32 приоритетных уровня:

- Критический класс - относятся системные потоки (наивысший приоритет)
- Серверный класс - потоки серверных приложений
- Стандартный класс - потоки пользовательских приложений
- Остаточный класс – самый низкий приоритет, все остальные потоки

Приоритеты могут изменяться планировщиком в следующих случаях:

- Если поток находится в ожидании процессорного времени дольше чем это задано системной переменной max_wait, то его уровень приоритета будет автоматически увеличен ОС; верхняя граница – критический класс
- Если поток ушел на выполнения операции ввода/вывода, то после ее завершения он получит наивысшее значение приоритета своего класса
- Приоритет потока автоматически повышается, когда он поступает на выполнение

ОС OS/2 динамически устанавливает величину кванта, отводимого потоку для выполнения. Величина кванта зависит от загрузки системы и интенсивности подкачки. Параметры настройки ОС позволяют явно

задать границы изменения кванта. В любом случае он не может быть меньше 32 мс и больше 65536 мс. Если поток был прерван до истечения кванта времени, то следующий выделенный ему интервал выполнения будет увеличен на время равное одному периоду таймера (32 мс) и так до тех, пока квант не достигнет заранее заданного при настройке ОС предела.

Диспетчеризация процессов и задач. Смешанные дисциплины диспетчеризации (на примере ОС UNIX)

В ОС Unix каждый процесс относиться к одному из 3 приоритетных классов:

- Системный класс – потоки ядра. Приоритеты фиксированы, назначаются ядром
- Потоки реального времени. Приоритеты фиксированы, пользователь может их изменять. Для каждого уровня свой квант времени.
- Потоки разделения времени – пользовательские потоки. Приоритеты динамические относительные, вычисляются от двух составляющих: пользовательской и системной части. Пользовательская часть может быть изменена администратором и владельцем процесса. Системная составляющая зависит от того, как долго поток занимает процессор, не уходя в состояние ожидания. У процессов, которые потребляют большие периоды процессорного времени без ухода в состояние ожидания, приоритет снижается, а у тех процессов, которые часто уходят в состояние ожидания после короткого периода использования процессора, приоритет повышается. Но процессам с низкими приоритетами даются большие кванты времени, чем процессам с высокими приоритетами (ага, щас, квант фиксирован).

Управление памятью в операционных системах. Простое непрерывное распределение и распределение с перекрытием

Общие положения

Что должны делать менеджеры памяти:

1. Выделение блока памяти.
2. Изменение размера блока, выделенного процессу.
3. Освобождение блока памяти.
4. Защита блока памяти, выделенного процессу.
5. Возможность совместного использования одной и той же области основной памяти несколькими процессами.

Процесс может быть загружен в раздел равного или большего раздела.

Простое непрерывное распределение и распределение с перекрытием

Простое непрерывное распределение — это самая простая схема, согласно которой вся память условно может быть разделена на три части:

- область, занимаемая операционной системой;
- область, в которой размещается исполняемая задача;
- свободная область памяти.

Особенности:

- ОС не поддерживает многозадачность => не возникает проблемы распределения памяти между несколькими задачами
- Программные модули, необходимые для всех программ, располагаются в области самой ОС
- Вся оставшаяся память предоставляется задаче. Эта область памяти непрерывная => облегчает работу программиста
- Привязка виртуальных адресов программы к физическому адресному пространству осуществляется на этапе загрузки задачи в память

Два вида потерь вычислительных ресурсов при такой схеме:

- потеря процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операций ввода/вывода
- потеря самой ОП, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный.

Распределение с перекрытием

Если есть необходимость создать программу, логическое (и виртуальное) адресное пространство которой должно быть больше, чем свободная область памяти, или даже больше, чем весь возможный объем оперативной памяти, то используется распределение с перекрытием (так называемые оверлейные структуры).

Этот метод распределения предполагает, что вся программа может быть разбита на части - сегменты. Каждая оверлейная программа имеет одну главную часть и несколько сегментов, причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько не перекрывающихся сегментов. Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта. Либо он сам обращается к ОС с указанием, какой сегмент должен быть загружен в память следующим. Либо он возвращает управление главному сегменту задачи, и уже тот обращается к ОС с указанием, какой сегмент сохранить (если это нужно), а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти. Сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода на место отработавшего последний можно не сохранять во внешней памяти, в отличие от сегментов данных, которые сохранять необходимо.

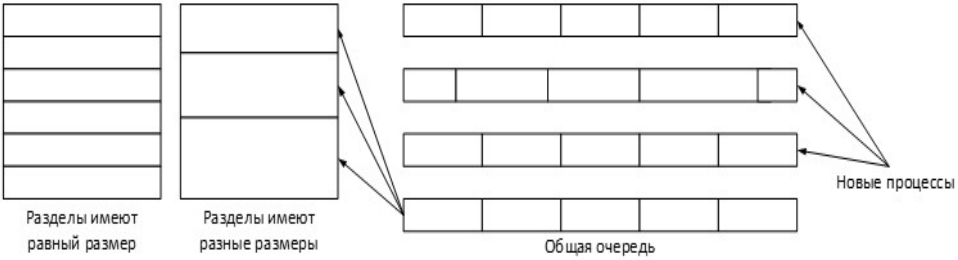
Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к ОС и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор эти вызовы система программирования стала подставлять в код программы сама, автоматически, если в том возникает необходимость.

Управление памятью в операционных системах. Распределение памяти статическими разделами

Статическое распределение - распределение, при котором основная память разделяется на ряд статических разделов во время запуска системы. Процесс может быть загружен в раздел равного или большего размера.

Достоинство - простота реализации, малые системные накладные расходы.

Недостаток - неэффективное распределение памяти.



Очереди могут быть 2 типов:

- Общая очередь - просматриваются свободные блоки и проверяется подойдет или нет
- Организация к каждому типу блоков своих очередей, в этом случае есть недостаток - при большом количестве маленьких программ, и отсутствии больших программ, будет большая очередь несмотря на то, что часть памяти свободна.

Управление памятью в операционных системах. Распределение памяти динамическими разделами

Динамическое распределение – разделы создаются динамически, во время загрузки программы, т. е. каждый процесс загружается в раздел строго необходимого размера.

У данного типа распределения есть проблема фрагментации. Для устранения используются процедуры дефрагментации памяти

(уплотнения памяти), при запуске такой процедуры останавливается работа всей системы. Также такие процедуры сложны так как при переносе в памяти программы следует еще и изменить сегмент состояния программы. Такие процедуры запускаются тогда, когда приходит задача и недостаточно места для ее размещения.

Алгоритмы распределения памяти

Все алгоритмы обладают общим недостатком – через какое-то время все равно придется выполнять фрагментацию памяти.

- **Алгоритм "Первый подходящий"**. Ищет свободные блоки памяти с начала памяти и выбирает первый достаточный по размеру для размещения процесса. При таком алгоритме вся работа находится в начале памяти: первая часть памяти работает эффективно, а до конечных адресов дело не доходит, если система не загружена сильно. Достоинство – не нужно сортировать блоки памяти.
- **Алгоритм "Самый подходящий"**. Блоки сортируются по возрастанию размеров. Система ищет блок, который больше или равен размеру требуемой памяти. Этот метод является наихудшим, так как в нем ищутся блоки, наиболее близкие по размеру к требуемому, а оставшиеся свободными блоки имеют очень маленький размер. Память очень быстро фрагментируется
- **Алгоритм "Самый неподходящий"**. Блоки сортируются по убыванию размеров; выбирается первый в списке блок, в оставшийся новый блок памяти может быть загружен еще один процесс. Данный метод самый эффективный.
- **Алгоритм "Система двойников"**. Вначале всё доступное для распределения пространство рассматривается как единый блок. Если при запросе процессу требуется блок размером больше 1/2 имеющегося, то выделяется весь блок, иначе блок делится на 2 эквивалентных двойника. Этот процесс продолжается до тех пор, пока не будет сгенерирован наименьший необходимый блок. Система двойников представляет собой компромисс для преодоления недостатков схем фиксированного и динамического распределения, но в современной ОС её превосходит виртуальная память, основанная на страничной организации и сегментации.

Понятие виртуальной памяти. Механизм преобразования виртуального адреса в физический при страничной организации памяти

Общие положения

Виртуальная память - видимое (доступное) программисту адресное пространство, представляющее собой адресное пространство оперативной (основной) памяти и быстродействующей дисковой памяти (обычно жесткого магнитного диска).

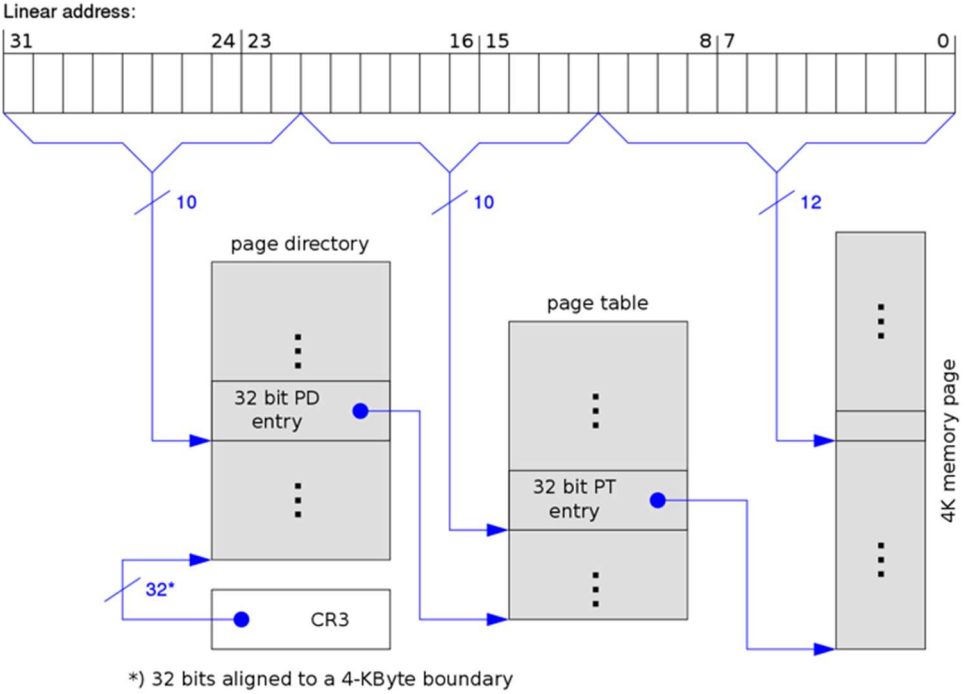
Адреса виртуального адресного пространства называют виртуальными адресами, а адреса ОП - физическими адресами.

Как и при использовании логических адресов, программист пишет программы в виртуальных адресах (ВА), поэтому также требуется процедура преобразования ВА в ФА.

Механизм преобразования виртуального адреса в физический при страничной организации памяти

Для преобразования ВА в ФА можно выделить два метода:

- с использованием таблицы прямого преобразования
- с использованием буфера ассоциативного преобразования (TLB)



Метод таблиц прямого преобразования

Реализуется аппаратно путем преобразования номера страницы в адрес ОП (страничный кадр) через таблицу страниц, хранимую в ОП, объем которой равен числу страниц всего виртуального пространства (может достигать до 4 - 8 млн страниц).

В таблице страниц для каждой страницы хранится ее дескриптор - указатель, описывающий местоположение страницы, разрешенные операции с ней (права доступа к странице) и другая вспомогательная информация:

- бит d достоверности страницы, показывающий, существует или нет данная страница;
- P - бит присутствия, показывающий местонахождение страницы: 1 - в ОП, 0 - на диске;
- бит обновления страницы D (dirty), показывающий, было или нет обновление страницы (запись в страницу) в ОП, и используемый для предотвращения ненужной процедуры перезаписи страницы на диск в случае удаления ее из ОП;
- поле RWX, характеризующее, какие права доступа к странице;
- поле PAR - адрес страничного кадра (базовый начальный адрес страницы) и другая информация.

Так как при одноуровневом преобразовании страничного ВА в ФА объем таблицы страниц может быть значительным (равен максимальному адресуемому числу страниц), в структуру введем регистр CR, в который при инициализации загружается базовый начальный адрес таблицы страниц (ВА PTE – page table entry), что делает ее перемещаемой в адресном пространстве ОП.

Виртуальный адрес состоит из двух полей:

- номера страницы
- смещение внутри страницы

По адресу, сформированному как сумма ВА PTE и номера страницы, выполняется обращение к таблице страниц и выбирается дескриптор страницы. Если страница достоверна (d=1) и находится в ОП (P=1), то ФА формируется операцией конкатенации базового адреса страницы PAR [s] (адреса страничного кадра) и смещения [b] из ВА.

Основными недостатками данного метода являются:

1. низкое быстродействие, так как для доступа к операнду необходимо два обращения к памяти: к таблице страниц за дескриптором и к ОП за операндом;
2. таблица страниц может занимать до нескольких Мбайт памяти;
3. в мультипрограммном режиме работы возникает проблема перераспределения номеров страниц между задачами, так как внутри каждой задачи страницы имеют номера с нулевого, а в общей таблице страниц номера могут совпадать, т. е. номер задачи должен входить в формат ВА, что увеличивает размер таблицы страниц, либо для каждой задачи необходимо хранить их начальные ВА, загружаемые при переключении задач в специальный регистр настройки, а номер страницы для текущей задачи будет определяться как сумма номера страницы ВА и содержимого регистра настройки.

Метод ассоциативного преобразования

Позволяет существенно сократить время преобразования ВА в ФА. В структуру ПР вводится быстродействующая ассоциативная память небольшой емкости (8-128 дескрипторов), в которой хранятся дескрипторы наиболее часто используемых страниц.

При первой попытке обращения к странице преобразование выполняется с помощью таблицы страниц из ОП, а все остальные через АЗУ. Замещение дескрипторов в АЗУ обычно осуществляется по алгоритму LRU, а сам ассоциативный буфер строится на основе частично- ассоциативного распределения. Ассоциативная память называется ассоциативным буфером преобразования (TLB).

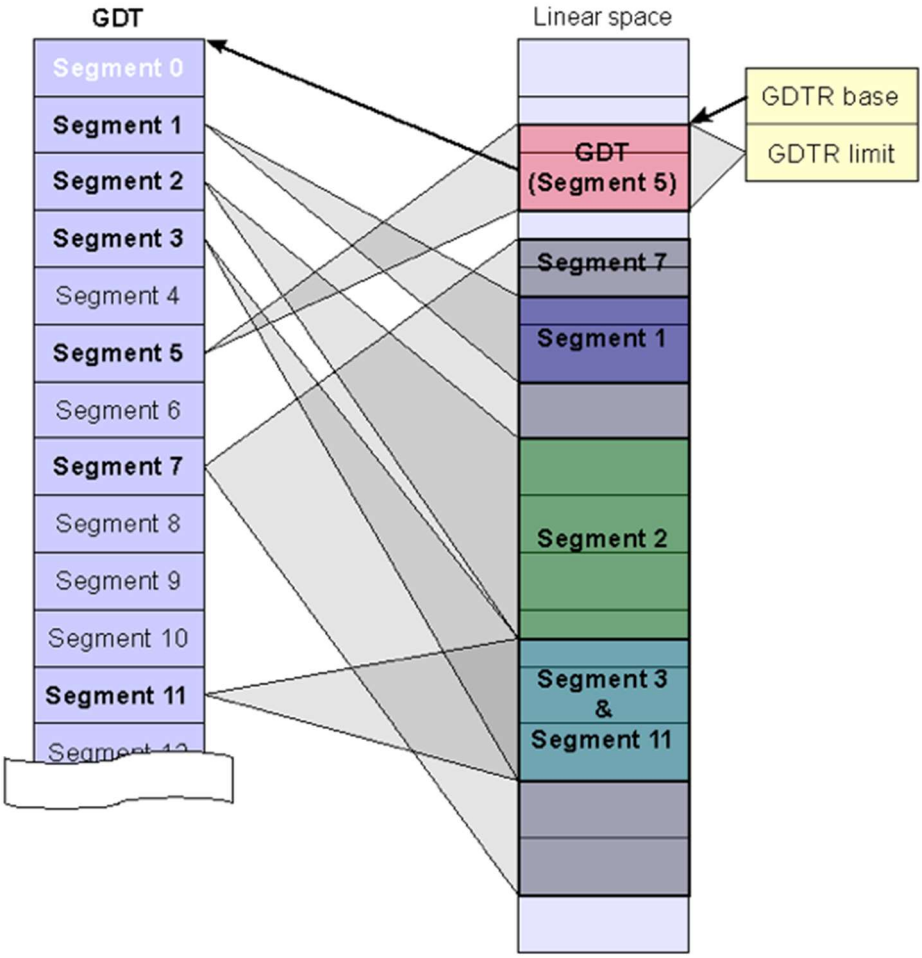
Параллельно с формированием ФА в ПР (процессоре) выполняется контроль по полям прав доступа к странице:

- Бит присутствия в дескрипторе страницы P=0 => формируется прерывание особого случая неприсутствия страницы в ОП и выполняется процедура замещения страницы с диска в ОП
- Бит обновления страницы D=1 => в данную страницу была выполнена хотя бы одна запись и предварительно данная страница удаляется из ОП и переписывается на диск, а в ее дескрипторе изменяются соответствующие поля (P=0, D=0, PAR=адрес местоположения страницы на диске). Далее выполняется замещение страницы, в дескрипторе устанавливаются соответствующие поля (d=1, P=1, PAR=базовый адрес ОП страницы).

В мультипрограммной страничной виртуальной памяти каждой задаче отводится свое линейное виртуальное пространство. Отличие от однозадачной виртуальной памяти заключается в том, что для каждой задачи отводится своя локальная таблица страниц. В структуру ПР вводится дополнительный регистр-указатель, в который при переключении задач из быстродействующей памяти загружается базовый (начальный) адрес таблицы страниц для данной задачи (так-то лучше).

Такой подход имеет и один существенный недостаток: нет возможности использования несколькими задачами одних и тех же страниц.

Понятие виртуальной памяти. Сегментное распределение



При сегментной организации памяти единицей загрузки в ОП задачи является сегмент. При этом задача может состоять из нескольких сегментов. В свою очередь сегмент задачи состоит из кодовой части (команд), данных и стековой области, которые разделяются по своим одноименным сегментам: сегменты кода, сегменты данных и сегмент стека. В отличие от страниц сегменты представляют собой совокупность линейных адресов переменной длины.

Виртуальный адрес определяется номером сегмента и адресом внутри сегмента. Для преобразования ВА в ФА используется сегментная таблица. Процедура преобразования виртуального сегментного адреса в ФА выполняется аналогично преобразованию виртуального страничного адреса в ФА.

Основные отличия:

- разрядность поля смещения в ВА при сегментной организации памяти имеет существенно большую разрядность, чем страница, размер сегмента не является фиксированным;
- формирование ФА осуществляется не путем операции конкатенации смещения и базового адреса сегмента, а их сложением, так как базовый адрес сегмента не является кратным смещению.

Замещение сегментов при их отсутствии в ОП выполняется аналогично страницам. При доступе к сегментам выполняется дополнительный контроль по размеру при обращении к таблице сегментов (размер таблицы сегментов значительно меньше таблицы страниц, определяется при инициализации системы, может быть переменным и задается в поле L регистра базового адреса таблицы сегментов CR), а при обращении в дескрипторе сегмента в поле M задается его размер.

В первом варианте используется метод преобразования ВА в ФА через общую для всех задач таблицу дескрипторов сегментов (GDT - global descriptor table).

Данному методу присущи серьезные недостатки:

- Проблема перераспределения номеров сегментов между задачами, так как сегменты в таблице GDT имеют сквозную нумерацию от 0 до k для всех задач.
- Невозможно организовать эффективную защиту сегментов задач от случайного или преднамеренного доступа со стороны других пользователей.
- После выполнения нескольких замещений сегментов с диска в ОП возникает проблема фрагментации, так как сегменты могут иметь различный размер, что приводит к неэффективному использованию адресного пространства ОП.

При втором варианте используется метод, аналогичный страничной организации памяти через разделение таблицы сегментов между задачами (использование локальных таблиц сегментов задач LDT). С номером задачи однозначно связана своя таблица сегментов задач, в каждой из которых сегменты распределены с нулевого номера.

В данном методе устранены недостатки 1 и частично 2, но проблема фрагментации памяти остается.

Понятие виртуальной памяти. Страничное распределение

Внешняя дисковая память и программа разбиваются на части равной величины, называемые страницами. В начальный момент времени все страницы, представляющие программу, хранятся на дисковой памяти и по мере необходимости затребованные страницы помещаются в ОП. Размер страницы обычно выбирают в пределах 2-4 Кбайт.

Таким образом, ВП является линейной, разбитой на страницы одинакового размера, что исключает фрагментацию ОП при дозагрузке страниц из дисковой памяти в ОП. Если при запросе доступа к странице она отсутствует в ОП, то страница загружается из дисковой памяти в ОП. Если в ОП нет свободной области памяти для дозагрузки страниц, то необходимо предварительно освободить область ОП путем удаления одной из страниц из ОП на дисковую память. Удаляемая страница определяется алгоритмом замещения страниц.

Синхронизация процессов и потоков. Необходимость взаимодействия процессов. Два механизма взаимодействия процессов

Общие сведения

Потребность в синхронизации потоков (и/или процессов) возникает в мультипрограммной ОС и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков

- При обмене данными между потоками;
- При разделении данных;
- При доступе к процессору;
- При доступе к устройствам ввода-вывода.

Любое взаимодействие процессов или потоков связано с их синхронизацией, которая заключается в согласовании их скоростей

путем приостановки потока до наступления некоторого события и последующей его активации при наступлении этого события.

Механизмы взаимодействия процессов

- Процессы взаимодействуют для выполнения общей задачи. Процессу может потребоваться возможность запрашивать сервисы у другого процесса и ждать выполнения соответствующих операций. Процессы могут также синхронизироваться друг с другом. Это происходит, когда один из процессов достигает точки, в которой ему нужно убедиться в выполнении другим процессом какой-либо операции и лишь затем продолжить работу. Процессу требуется операция «Wait» для синхронизации с другим процессом или аппаратным обеспечением. Кроме того, необходимы механизмы сигнализации о завершении операции, когда процесс ждет, что-то должно инициировать продолжение его работы. Причем взаимную синхронизацию необходимо осуществлять независимо от того, имеется ли у процессов общее адресное пространство или нет.
- Процессы «соревнуются» за монопольное использование сервисов или ресурсов. Многие клиенты могут параллельно запрашивать некоторый сервис. Несколько процессов могут пытаться одновременно использовать некоторый ресурс. Система должна всем этим управлять, в частности она должна устанавливать порядок, согласно которому процессы, в случае необходимости, организуются в очередь в ожидании тех ресурсов, который можно использовать только последовательно, а также осуществлять выбор между процессами, запросы которых совпали по времени. «Соревнующимся» процессам нужно ждать получения общего ресурса и сигнализировать о завершении его использования.

Для предотвращения проблем, возникающих, когда два или более процессов используют общую структуру данных, необходим специальный механизм, управляющий доступом к этой структуре и не позволяющий нескольким процессам обращаться к ней, либо изменять ее одновременно.

Синхронизация процессов и потоков. Взаимное исключение. Требования к взаимным исключениям

Взаимное исключение – «соревнующиеся» процессы должны ждать освобождения общего ресурса и сигнализировать об окончании его использования.

Требования к взаимным исключениям:

1. взаимоисключения должны выполняться принудительно: одновременно внутри критической секции должно находиться не более одного потока.
2. процесс, завершивший работу вне критического раздела, не должен влиять на другие процессы.
3. не должна возникать ситуация бесконечного ожидания доступа к критическому разделу.
4. когда в критическом разделе нет ни одного процесса, любой процесс, запросивший возможность входа в него, должен немедленно его получить.
5. не делается никаких предположений о количестве процессов или их относительной скорости работы.
6. процессы остаются в критической секции только в течение ограниченного времени.

Имеется ряд способов удовлетворить перечисленным условиям:

1. Одним из них является передача ответственности за соответствие требованиям самому процессу, который должен выполняться параллельно. Таким образом, процесс, независимо от того является ли он системной программой или приложением, должен координировать свои действия с другими процессами для работы взаимоисключений без поддержки со стороны ОС.
2. Другой подход включает использование машинных команд специального назначения. Достоинство этого подход заключается в снижении накладных расходов, но не решает проблему в общем случае.
3. Еще один подход заключается в предоставлении определенного уровня поддержки со стороны ОС или языка программирования.

Если у процессов имеется общее адресное пространство, планировщик системы может использовать общие объекты данных для их взаимодействия и управления соревнованием за ресурсы.

Системы, где процессы не имеют общей памяти, нуждаются в механизме передачи информации между процессами и синхронизации по принципу «ждать-сигнализировать». Такая передача данных внутри системы вполне возможна, но могут возникнуть следующие проблемы:

- процессы, участвующие в обмене данными, должны знать имена друг друга; в единой централизованной системе пространство имен процессов можно расширить для организации взаимодействия последних.
- на копирование данных между адресными пространствами уходит много времени; необходимо минимизировать это время с помощью аппаратного обеспечения управления памятью.
- на переключение между процессами с разделяемым адресным пространством также требуется много времени.

Синхронизация процессов и потоков. Критическая секция

Для предотвращения проблем, возникающих, когда два или более процессов используют общую структуру данных, необходим специальный механизм, который:

- Управляет доступом к этой структуре
- Не позволяет нескольким процессам обращаться к ней, либо изменять её одновременно

Критическая секция — это часть программы, в которой осуществляется доступ к разделяемым данным. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс.

Существует ряд механизмов, гарантирующих, что в каждый конкретный момент времени критическая секция, связанная с определенным элементом данных, будет выполняться только одним процессом.

Эти механизмы обеспечивают взаимное исключение и монопольный доступ к общим данным.

Синхронизация процессов и потоков. Аппаратная поддержка взаимоисключения

Запрет прерываний

Простейший способ обеспечить взаимоисключения – это позволить процессам, находящимся в критической секции запрещать все прерывания. Эта возможность может быть обеспечена в форме примитивов, определяемых системным ядром для запрета и разрешения прерываний.

Раздел не может быть прерван – взаимоисключение гарантируется. Эффективность работы при этом может заметно снизиться. Кроме того, опасно доверять управление системой пользовательскому процессу. Он может надолго занять процессор, а при сбое процесса в критической секции, крах потерпит вся система, так как прерывания никогда не будут разрешены.

Такой подход не будет работать в многопроцессорной системе.

Использование специальных машинных команд

На уровне аппаратуры обеспечивается обращения к ячейке памяти, исключающее любое другое обращение к той же ячейке. Такие команды атомарно выполняют над ячейками два действия: чтение и запись или чтение и проверку значения. Так как эти действия выполняются в одном цикле, на них не в состоянии повлиять никакие другие инструкции.

Test and Set

Простая неразрывная (атомарная) процессорная инструкция, которая копирует значение переменной в регистр, и устанавливает некое новое значение. Во время исполнения данной инструкции процессор не может прервать её выполнение и переключиться на выполнение другого потока. Если используется многопроцессорная архитектура, то, пока один процессор выполняет эту инструкцию с ячейкой памяти, другие процессоры не могут получить доступ к этой ячейке, что может достигаться путём кратковременного блокирования шины памяти. x86: TSL

Compare and Swap

Атомарная инструкция, сравнивающая значение в памяти с одним из аргументов, и в случае успеха записывающая второй аргумент в память. x86: LOCK:CMPXCHG

Load-link/store-conditional

Пара инструкций, используемых в синхронизации. Инструкция load-link загружает значение из памяти, а store-conditional инструкция пишет новое значение в тот же участок памяти только в том случае, если в промежутке между этими двумя инструкциями не было записи в этот участок памяти.

Преимущества:

1. этот подход очень простой, поэтому легко проверяем;
2. применим к любому количеству процессов;
3. может использоваться для поддержки множества критических разделов, каждый из которых может быть определен при помощи своей собственной переменной.

Недостатки:

1. используется активное ожидание, т. е. процессы, находящиеся в ожидании доступа к критическому разделу, продолжают потреблять процессорное время;
2. возможно голодание, т. е. выбор ожидающего процесса произволен и может оказаться, что какой-то из процессов будет ожидать входа в критический раздел бесконечно;
3. возможна взаимоблокировка.

Блокировка памяти

Это средство запрещает использование двух или более команд, которые обращаются к одной и той же ячейке памяти.

Так как в некоторой ячейке памяти хранится значение разделяемой переменной, то получить доступ к ней может только один процесс, несмотря на возможное совмещение выполнения команд во времени. Механизм блокировки памяти предотвращает одновременный доступ к разделяемой переменной, но не предотвращает чередование доступа.

Синхронизация процессов и потоков. Семафоры Дейкстры

Семафоры

Это объект с операциями SimWait() и SimSignal(). Его атрибутом обычно является целое число и очередь, а его операции определены следующим образом.

SimWait() – если счетчик больше 0, уменьшить его на единицу и позволить процессу продолжить работу, в противном случае приостановить процесс, пометить его, как блокированный у данного семафора.

SimSignal() – если ни один процесс не ждет у данного семафора, увеличить целое число на единицу, в противном случае освободить один процесс и продолжить его работу с инструкции, следующей за инструкцией «ждать».

При использовании семафорного механизма процесс, которому отказано в доступе к критической ресурсу, находится в состоянии пассивного ожидания.

Способы использования семафоров

1. Взаимоисключения – с помощью семафора, инициализированного значением 1, можно обеспечить монопольный доступ к общему ресурсу.
2. Условная синхронизация – два взаимосвязанных процесса должны иметь возможность синхронизировать свои действия. Процессы могут быть связаны следующим образом: при достижении процессом А определенной точки, он не может продолжить работу, пока процесс В не выполнит некоторую задачу.
3. Использование более одного экземпляра ресурса. В ОС типичной бывает ситуация, когда несколько процессов имеют возможность обращаться к ресурсу одновременно, но количество таких обращений должно быть ограничено.

Проектирование параллельных процессов. Алгоритм Деккера. Алгоритм Петерсона

Общие положения

Пусть имеется два или более циклических процесса в которых есть абстрактные критические секции. Необходимо реализовать их

взаимное исключение. Причем взаимоисключение будет реализовано за счет доступа к общим переменным.

Алгоритм Деккера

Деккер предложил использовать 3 семафорные переменные, по одной на каждый процесс и общую очередь. Очередь должна указывать на то, чье право сделать попытку входа в критическую секцию, при условии, что каждый из процессов желает ее выполнить. Алгоритм исключает возможность бесконечного откладывания процесса.

Псевдокод

```
flag[0] := false
flag[1] := false
turn := 0   // or 1
```

p0:

```
flag[0] := true
while flag[1] = true {
    if turn = 1 {
        flag[0] := false
        while turn = 1 {}
        flag[0] := true
    }
}

// критическая секция
...
turn := 1
flag[0] := false
// конец критической
секции
...
```

p1:

```
flag[1] := true
while flag[0] = true {
    if turn = 0 {
        flag[1] := false
        while turn = 0 {}
        flag[1] := true
    }
}

// критическая секция
...
turn := 0
flag[1] := false
// конец критической
секции
...
```

Процессы объявляют о намерении войти в критическую секцию; это проверяется внешним циклом «while». Если другой процесс не заявил о таком намерении, в критическую секцию можно безопасно войти (вне зависимости от того, чья сейчас очередь). Взаимное исключение всё равно будет гарантировано, так как ни один из процессов не может войти в критическую секцию до установки этого флага (подразумевается, что, по крайней мере, один процесс войдёт в цикл «while»). Это также гарантирует продвижение, так как не будет

ожидания процесса, оставившего «намерение» войти в критическую секцию. В ином случае, если переменная другого процесса была установлена, входят в цикл «while» и переменная turn будет показывать, кому разрешено войти в критическую секцию. Процесс, чья очередь не наступила, оставляет намерение войти в критическую секцию до тех пор, пока не придёт его очередь (внутренний цикл «while»). Процесс, чья очередь пришла, выйдет из цикла «while» и войдёт в критическую секцию.

Алгоритм Петерсона

Псевдокод

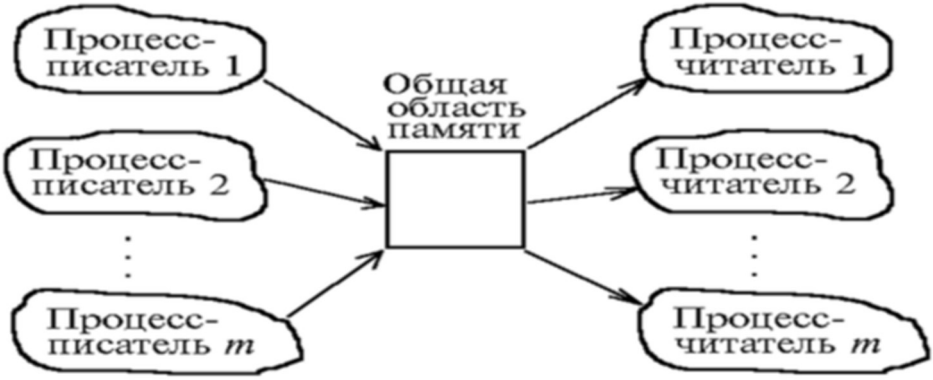
```
want[0] := false
want[1] := true
waiting := 0

p0:
want[0] := true
waiting := 0
while (want[1] = true and waiting = 0) {}

// критическая секция
...
want[0] := false

p1:
want[1] := true
waiting := 1
while (want[0] = true and waiting = 1) {}

// критическая секция
…
want[1] := false
```

Потоки 0 и 1 никогда не могут попасть в критическую секцию в один момент времени: если 0 вошёл в секцию, он установил want[0] в true. Тогда либо want[1] == false (тогда поток 1 не в критической секции), либо waiting == 1 (тогда поток 1 пытается войти в критическую секцию и крутится в цикле), либо поток 1 пытается войти в критическую секцию после установки want[1] == true, но до установки waiting и цикла. Таким образом, если оба процесса находятся в критической секции, должно быть want[0] && want[1] && waiting == 0 && waiting == 1, но такого не может быть одновременно и мы пришли к противоречию.

Для того, чтобы оба процесса находились в ожидании, необходимы противоположные значения переменной waiting, что невозможно.

Типовые задачи, требующие организации параллельных вычислительных процессов. Задача «Писатель-читатель»

Например, в системе, резервирующей билеты, множество конкурирующих процессов читают одни и те же данные. Однако, когда один из процессов начинает модифицировать данные, ни какой другой процесс не должен иметь доступа к этим данным, даже для чтения.

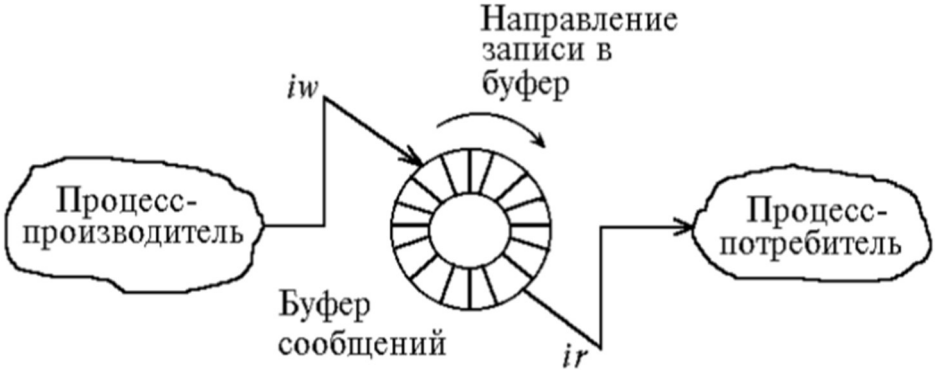


Задачу «Читатель-писатель» можно решать, исходя из следующих предположений:

1. Наибольший приоритет имеет читатель: в этом случае любому из процессов – писателей запрещается вход в критическую секцию до тех пор, пока в ней находится хотя бы один читающий процесс. Процесс, ожидающий доступа по записи, будет ждать до тех пор, пока все читающие процессы не окончат работу, и если в это время появляется новый читающий процесс, он тоже беспрепятственно получит доступ. Чтобы этого избежать, можно модифицировать алгоритм.
2. Наибольший приоритет имеет писатель. Тогда в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получают доступ к ресурсу, а ожидают, когда процесс-писатель обновит данные. Отрицательная сторона данного решения заключается в том, что оно несколько снижает производительность процессов-читателей, так

как вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

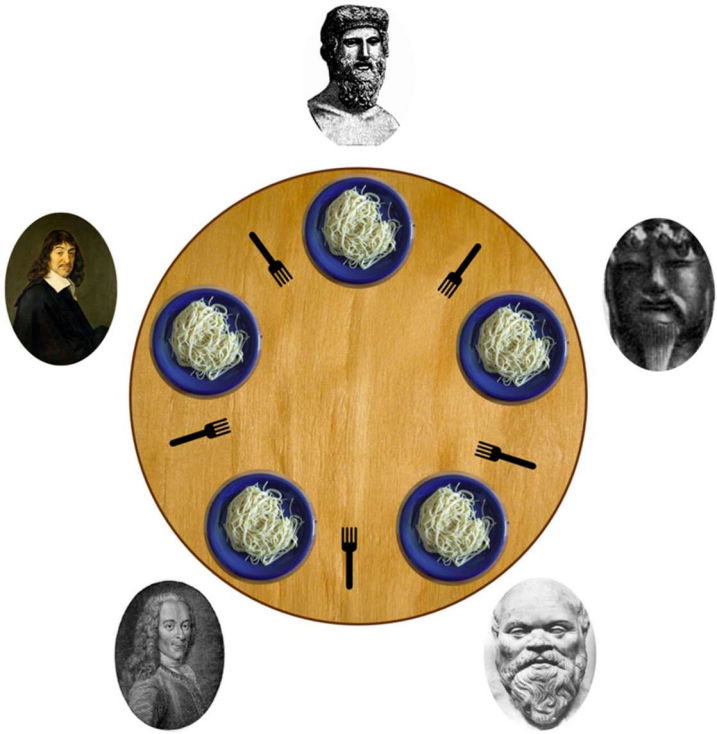
Типовые задачи, требующие организации параллельных вычислительных процессов. Задача «Поставщик-потребитель»



Процессы взаимодействуют через некоторую обобщенную область памяти – буфер сообщений. В эту область процесс-поставщик помещает очередное сообщение, а процесс-потребитель – считывает сообщение из этой области. В общем случае буфер способен хранить несколько сообщений. Необходимо согласовать работу процессов при одностороннем обмене сообщениями, чтобы удовлетворять следующие требования:

1. Выполнять взаимоисключение по отношению к критическому ресурсу (буферу сообщений);
2. Учитывать состояние буфера, характеризующее возможность записи или чтения очередного сообщения. Процесс-поставщик при попытке записи в полный буфер должен быть заблокирован. Попытка процесса-потребителя чтения из пустого буфера также должна быть блокирована.

Типовые задачи, требующие организации параллельных вычислительных процессов. Задача «Обедающие философы»



За круглым столом расставлены пять стульев, на каждом из которых сидит определенный философ. В центре стола – большое блюдо спагетти, на столе лежат пять вилок – каждая между двумя соседними тарелками. Каждый философ может находиться только в двух состояниях: либо он размышляет, либо ест спагетти. Начать думать философу ничего не мешает. Чтобы начать есть, нужны две вилки (одна в правой и одна в левой руке). Закончив еду, философ кладет вилки справа и слева от своей тарелки и начинает размышлять до тех пор, пока снова не проголодается.

В представленной задаче имеются две опасные ситуации: ситуация голодной смерти и ситуация голодания отдельного философа. Ситуация голодной смерти возникает в случае, когда философы

одновременно проголодаются и одновременно попытаются взять, например, свою левую вилку. В данном случае возникает тупиковая ситуация, так как никто из них не может начать есть, не имея второй вилки. Для исключения ситуации голодной смерти были предложены различные стратегии распределения вилок.

Ситуация голодания возникает в случае заговора двух соседей слева и справа против определенного философа. Заговорщики поочередно забирают вилки то слева, то справа от него. Такие согласованные действия злоумышленников приводят жертву к вынужденному голоданию, так как он никогда не сможет воспользоваться обеими вилками.

Существует простейшее решение с использованием семафоров: когда один из философов захочет есть, он берет вилку слева от себя, если она в наличии, а затем – вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места.

Другим решением может быть алгоритм, который обеспечивает доступ к вилкам только четырем из пяти философов. Тогда всегда среди четырех философов по крайней мере один будет иметь доступ к двум вилкам. Данное решение не имеет тупиковой ситуации.

Тупики

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы. Так как все процессы находятся в состоянии ожидания, ни один из них не будет причиной какого-либо события, которое могло бы активировать любой процесс в группе, все процессы продолжают ждать до бесконечности.

В большинстве случаев событие, которого ждет каждый процесс, является возвратом какого-либо ресурса, в данный момент занятого другим участником группы. Иначе говоря, каждый участник в группе процессов, зашедший в тупик, ждет доступа к ресурсу, принадлежащему заблокированному процессу. Ни один из процессов не может работать, ни один из них не может освободить какой-либо ресурс и ни один из них не может возобновиться. Кол-во процессов, а также кол-во и вид ресурсов, имеющихся из запрашиваемых здесь не важны. Результат остается тем же самым для любого вида ресурсов, и аппаратных, и программных.

При рассмотрении проблемы тупиков целесообразно понятие ресурсов системы обобщить и разделить их все на два класса:

- Повторно используемые (системные ресурсы), SR
- Расходоумые ресурсы, CR

Повторно используемый ресурс — это конечное множество идентичных единиц со следующими свойствами:

- Число единиц ресурсов постоянно
- Каждая единица ресурса или доступна или распределена одному и только одному процессу. Здесь разделение либо отсутствует, либо не принимается во внимание, т. к. не оказывает влияния на распределение ресурсов, а значит и на возникновение тупиковой ситуации
- Процесс может освободить единицу ресурса, т. е. сделать ее доступной, только если он ранее получил эту единицу. Т. е. никакой процесс не может оказывать какое-либо влияние ни на один ресурс, если он ему не принадлежит

Перечисленные свойства обычных системных ресурсов существенны для изучения проблемы тупиков.

Расходуемый ресурс характеризуется следующими свойствами:

- Число доступных единиц некоторого ресурса типа CR изменяется по мере того, как приобретаются (расходуются) и освобождаются (производятся) отдельные их элементы выполняющимися процессами. И такое число единиц ресурса является потенциально неограниченным. Процесс-производитель увеличивает число единиц ресурса, освобождая одну или более единиц, которые он создал
- Процесс-потребитель уменьшает число единиц ресурса, сначала запрашивая, а затем приобретая одну или более единиц. Единицы ресурса, которые приобретены, в общем случае не возвращаются ресурсу, а расходуются

Эти свойства потребляемых ресурсов присущи многим синхронизирующим сигналам, сообщениям и данным, порождаемым как аппаратурой, так и программным обеспечением, и могут рассматриваться как ресурсы типа CR, при изучении тупиков. В их число входят:

- Прерывания от таймеров и устройств ввода/вывода
- Сигналы синхронизации процессов
- Сообщения, содержащие запросы на различные виды обслуживания или данные, а также соответствующие ответы

Для возникновения ситуации взаимоблокировки должны выполняться четыре условия:

- Условие взаимного исключения, при котором процессы осуществляют монопольный доступ к ресурсам
- Условие ожидания. Характеризуется тем, что процессы в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы
- Условия отсутствия перераспределения. У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими должен сам освободить эти ресурсы.
- Условие циклического ожидания. В данном случае должна существовать круговая последовательность из двух или более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим процессом последовательности

Для того чтобы произошла взаимоблокировка, должны выполняться все четыре условия. Если хоть одно из них отсутствует, тупиковая ситуация невозможна.

Направления изучения тупиковых ситуаций:

- Метод предотвращения тупиков
- Обходы тупиков
- Обнаружение тупиков
- Восстановление после тупиков

Предотвращение тупиков

Предотвращение тупиков основывается на предположении о чрезвычайно высокой его стоимости. Поэтому лучше потратить доп. ресурсы системы, чтобы исключить вероятность возникновения тупика при любых обстоятельствах. Этот подход используется в наиболее ответственных системах, чаще в realtime-системах. Дисциплина, предотвращающая тупик, должна гарантировать, что какое-либо из 4 условий, необходимых для его наступления, не может возникнуть.

Условия взаимного исключения можно подавить путем разрешения неограниченного разделения ресурсов. Это удобно для повторно вводимых программ и ряда драйверов, но совершенно неприемлемо совместно используемым переменным в критических интервалах.

Условие ожидания можно подавить, предварительно выделяя ресурсы. При этом процесс может начинать исполнение только получив все необходимые ресурсы заранее. Предварительное выделение может привести к снижению эффективности работы вычислительной системы в целом.

Условие отсутствия перераспределения можно исключить, позволяя ОС отнимать у процесса ресурсы. Для этого в ОС должен быть

предусмотрен механизм, запоминания состояния процесса и ресурсов с целью последующего восстановления.

Условие кругового ожидания можно исключить, предотвращая образование цепи запросов. Это можно обеспечить с помощью принципа иерархического выделения ресурсов. Все ресурсы образуют некоторую иерархию и процесс, затребовавший ресурс на одном уровне может затем потребовать ресурсы на следующем, более высоком уровне. Он может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях. После того, как процесс получил, а потом освободил ресурсы данного уровня, он может запросить ресурсы на том же самом уровне. Иерархическое выделение ресурсов часто не дает никакого выигрыша, если порядок использования ресурсов, определенный в описании процессов, отличается от порядка уровней иерархии. Тогда ресурсы будут использоваться крайне неэффективно.

Обход тупиков (алгоритм банкира, Дейкстра)

Состояние ОС безопасно, если система может обеспечить завершение всех задач в течение какого-то определённого промежутка времени. См. методичку.

Обнаружение тупиков с последующим восстановлением

Это простой алгоритм, который изучает граф и завершается или когда находит цикл, или когда показывает, что циклов в этом графе не существует. Он использует одну структуру данных. Во время работы алгоритма на ребрах графа будет ставиться метка, говорящая о том, что их уже проверяли. Для каждого узла N в графе выполняется 5 шагов, где является начальным узлом:

1. Задаются начальные условия: L - пустой список, все ребра не маркированы
2. Текущий узел добавляется в конец списка L и проверяется кол-во появлений узла в списке. Если узел присутствует в двух местах, значит граф содержит, и работа алгоритма завершается
3. Для заданного узла определяется выходит ли из него хотя бы одно немаркированное ребро. Если да, то выполняется переход к следующему шагу.
4. Случайным образом выбирается любое исходящее немаркированное ребро, которое помечается. Затем по этому ребру выполняется переход к новому текущему узлу и выполняется переход к шагу 2. Если на 4-м шаге не обнаружено ни одного немаркированного ребра, последний узел из списка

удаляется и выполняется к предыдущему узлу. Если это первоначальный узел, значит граф не содержит циклов и алгоритм завершается

Алгоритм обнаружения взаимоблокировок при наличии нескольких ресурсов каждого типа основан на сравнении векторов: сравнивается вектор доступных ресурсов и строки матрицы запросов. Алгоритм обнаружения тупиков состоит из следующих шагов:

1. Ищем немаркированный процесс Pi, для которого i-я строка матрицы R меньше или равна вектору A
2. Если такой процесс найден, i-я строка матрицы C прибавляется к вектору A, процесс маркируется и выполняется переход к шагу 1
3. Если таких процессов не существует, работы алгоритма заканчивается. Завершение алгоритма означает, что все немаркированные процессы попали в тупик.

На первом шаге алгоритм ищет процесс, который может доработать до конца. Такой процесс характерен тем, что все требуемые для него ресурсы должны находиться среди доступных в данный момент ресурсов. Тогда выбранный процесс проработает до своего завершения и после этого вернет ресурсы, которые он занимал. Затем процесс маркируется как законченный. Если окажется, что все процессы могут работать, то взаимоблокировки нет.

После обнаружения взаимоблокировок необходимо применить меры для восстановления системы:

1. Восстановление при помощи принудительной выгрузки ресурсов. Всегда можно временно отобрать ресурс у владельца и отдать его другому процессу. Способность забирать ресурс у процесса, отдавать его другому процессу, а затем возвращать назад, так, что исходный процесс этого не замечает, в значительной мере зависит от свойств ресурса
2. Восстановление через откат. Если есть вероятность появления взаимоблокировок, можно организовать работу таким образом, чтобы процессы периодически создавали контрольный точки. Это означает, что состояние процесса записывается в файл и в последствии он может быть возобновлен из этого файла. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, т. е. информацию о том, какие ресурсы в данный момент предоставлены процессу. При обнаружении взаимоблокировки достаточно понять какие ресурсы нужны процессам, и чтобы выйти из тупика процесс, занимающий необходимый ресурс, откатывается к тому моменту времени,

перед которым он получил данный ресурс, для чего запускается одна из его контрольных точек

3. Восстановление путем уничтожения процессов.

Двухфазное блокирование

Для предотвращения блокировок во многих системах баз данных, когда часто выполняются операции блокирования нескольких записей, используется подход двухфазного блокирования. В первой фазе процесс пытается заблокировать все требуемые записи по одной. Если операция успешна, процесс переходит ко второму этапу, выполняя обновление и освобождение блокировок. Никакой работы не совершается. Если во время выполнения первой фазы какая-либо необходимая запись оказывается заблокированной, процесс просто освобождает все свои блокировки и начинает первую фазу заново.

Понятия и определения локальных, сетевых и распределенных операционных систем

В зависимости от того, какой виртуальный образ создает операционная система для того, чтобы подменить им реальную аппаратуру компьютерной сети, различают сетевые ОС и распределенные ОС.

Сетевая ОС предоставляет пользователю некую виртуальную вычислительную систему, работать с которой гораздо проще, чем с реальной сетевой аппаратурой. В то же время эта виртуальная система не полностью скрывает распределенную природу своего реального прототипа, т. е. является виртуальной сетью.

При использовании ресурсов компьютеров сети пользователь сетевой ОС всегда помнит, что имеет дело с сетевыми ресурсами. Для доступа к ресурсам нужно выполнять особые операции, например, просматривать список разделяемых ресурсов компьютеров сети, подключать к файловой системе удаленный разделяемый каталог на вымышленную букву локального дисковода (подключить сетевой диск) или ставить перед именем каталога еще и имя компьютера, на котором тот расположен.

Пользователи сетевой ОС обычно должны быть в курсе того, где хранятся их файлы, и должны использовать явные команды передачи файлов для перемещения файлов с одной машины в сети на другую.

Распределенная ОС, динамически и автоматически распределяя работы по различным машинам сети для обработки, заставляет набор машин работать как виртуальный универсальный процессор. Пользователь распределенной ОС, вообще говоря, не имеет сведений о том, на какой машине выполняется его работа.

Распределенная ОС существует как единая операционная система в масштабах вычислительной системы (сети). Каждый компьютер сети, работающий под управлением распределенной ОС, выполняет часть функций этой глобальной ОС, которая объединя-ет все компьютеры сети для эффективного использования всех сетевых ресурсов.

Организация многопроцессорных операционных систем. Симметричная схема

Общие положения

ОС могут классифицироваться в зависимости от отсутствия или наличия в них средств поддержки многопроцессорной обработки. Многопроцессорные ОС в свою очередь могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой. Выбор ОС для многоядерной архитектуры способен как значительно уменьшить, так и увеличить усилия, которые необходимы для решения проблем распределения ресурсов между ядрами и переноса программного кода в ОС с поддержкой многоядерности.

Существуют три основных схемы многопроцессорной обработки:

1. Ведущий-ведомый
2. Симметричная
3. Раздельные ядра

Симметричная схема

Самые сложные. Здесь ОС распоряжается набором одинаковых процессоров, любой из которых может обращаться к любому устройству ввода-вывода и к любому устройству хранения. Поскольку ОС может выполняться одновременно на множестве процессоров нужно гарантировать взаимное исключение при работе с общими структурами данных.

Многопроцессорные системы симметричной схемы являются самыми отказоустойчивыми. Если один из процессоров выходит из строя, ОС

удаляет его из набора доступных процессоров, выполняющийся на этом процессоре процесс, передается любому процессору.

Одним из недостатков многопроцессорных схем симметричной схемы является конкуренция при доступе к ресурсам ОС. Важно продумать организацию глобальных структур, чтобы предотвратить ненужное блокирование данных, исключающее выполнение ОС на нескольких процессорах сразу.

Организация многопроцессорных операционных систем. Схема ведущий–ведомый

Схема «ведущий-ведомый»

Организация многопроцессорной системы по принципу ведущий-ведомый делает один процессор главным, а остальные – подчиненными.

В ведущем процессоре исполняется собственно ОС, на ведомых - только пользовательские программы. При высокой интенсивности ввода-вывода ведомые процессы будут часто обращаться к ведущему процессору.

С точки зрения отказоустойчивости вычислительная эффективность падает при отказе одного из ведомых процессоров, но система остается работоспособной. Отказ ведущего процессора приводит к остановке системы.

Системы с организацией ведущий-ведомый являются тесно связанными, поскольку все ведомые процессоры зависят от ведущего.

Организация многопроцессорных операционных систем. Схема с раздельными ядрами

Схема с раздельными ядрами

На каждом процессоре выполняется своя ОС. Процесс, запущенный на определенном процессоре выполняется на этом процессоре до завершения.

В ОС есть несколько структур, содержащих информацию глобального характера, доступ к которым должен контролироваться механизмами

взаимного исключения. Система, построенная по схеме с раздельными ядрами является слабосвязанной.

Эта схема более отказоустойчивая, чем схема “ведущий-ведомый”. Если откажет один процессор, система остается работоспособной, но процессы, выполнявшиеся на отказавшем процессоре должны быть перезапущены заново.

В схеме с раздельными ядрами каждый процессор работает с ресурсами доступными только ему. Такая схема выигрывает от минимальной конкуренции при доступе к ресурсам ОС. Однако, процессоры не кооперируются при выполнении отдельных процессов и некоторые процессоры могут простаивать в то время как на других выполняются многопоточные процессы.

Планирование в многопроцессорных системах. Задачно-независимые алгоритмы планирования

Общие положения

Планирование в многопроцессорных системах (МПС) также как в однопроцессорных должно обеспечивать максимальную производительность за минимальное время реагирования для всех процессов.

В отличие от алгоритма планирования, используемых в однопроцессорных системах, алгоритмы для МПС должны определять не только порядок выполнения процессов, но и на каких процессорных эти процессы должны выполняться. Когда нужно определить на каком процессоре должен выполняться процесс, планировщик должен принять во внимание несколько моментов.

Например, в некоторых стратегиях планирования целью является максимальное распараллеливание работы, которая позволяет повысить производительность. Другие стратегии основаны на привязке задач к процессорам, т.е. взаимосвязи процесса и процессора, его локальной памяти и кэша. Алгоритмы планирования с пространственным разделением направляют взаимодействующие процессы на один процессор. Алгоритмы планирования в многопроцессорных системах обычно делятся на задачно-ориентированные и задачно-независимые.

Задачно-независимые алгоритмы не пытаются добиться максимального распараллеливания или воспользоваться привязкой

задачи к процессору, в то время как задачно-ориентированные оценивают свойство каждой задачи и пытаются достичь максимального ее распараллеливания или мягкой привязки к процессору.

Задачно-независимые алгоритмы

FCFS (First Come First Served)

Этот алгоритм помещает поступающие процессы в глобальную очередь выполнения. Когда появляется свободный процессор, программа планирования размещает на этом процессоре первый процесс из очереди выполнения и выполняет его до тех пор, пока процесс не освободит процессор.

Достоинство: простота реализации, обработка процессов в порядке поступления, нет опасности бесконечного ожидания

Недостаток: коротким процессам необходимо очень долго ждать

RR (Round Robin)

Круговой алгоритм планирования процессов помещает все готовые к выполнению процессы в глобальную очередь выполнения.

Этот алгоритм в многопроцессорных системах работает почти так же, как и в однопроцессорных — процесс выполняется в течение максимум одного полного кванта времени, а затем приходит очередь другого процесса. Ранее выполнявшийся процесс помещается в конец глобальной очереди выполнения.

Достоинство: предотвращает бесконечное откладывание выполнения процесса

Недостатки: не обеспечивает высокую степень параллельности или хорошей привязки к процессорам, поскольку игнорирует связи между процессами

SPF (Shortest Process First)

При использовании этого алгоритма планирования первым запускается процесс, на полное выполнение которого требуется меньше всего времени.

Достоинство: меньшее время ожидания для коротких процессов

Недостатки: не использует параллелизм и привязку к процессорам, запуск долгого процесса может бесконечно откладываться, если постоянно появляются короткие

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Круговой алгоритм планирования

Круговой алгоритм планирования

Круговой алгоритм планирования задач помещает все готовые к выполнению задачи в глобальную очередь выполнения. Из этой очереди каждая задача назначается для выполнения группе процессоров (хотя и необязательно одной и той же группе каждый раз, когда до задачи доходит очередь выполняться). У каждой задачи есть собственная очередь процессов. Если в системе есть p процессоров, и она использует кванты продолжительности q, то задача получает в общей сложности p x q процессорного времени при ее отправке на выполнение. Обычно в задаче не точно p процессов, каждый из которых расходует целиком по одному кванту (то есть выполнение процесса может блокироваться до истечения кванта времени). Поэтому в алгоритме кругового планирования процессы задачи отправляются на выполнение, пока задача не израсходует в общей сложности p x q времени, пока она не выполнится до конца или пока не заблокируются все ее процессы. Кроме того, алгоритм может поровну разделить p x q времени между всеми процессами задачи, позволяя каждому процессу выполняться, пока он не израсходует свой лимит времени, не завершится или не заблокируется. Альтернатива: если в задаче больше, чем p процессов, то алгоритм может выбрать p процессов для выполнения в течение кванта времени длиной q.

Достоинства: предотвращает бесконечное откладывание выполнения процесса, обеспечивает хороший уровень параллелизма

Недостаток: дополнительные накладные расходы на переключение контекста

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Алгоритм копланирования

Алгоритмы копланирования используют глобальную очередь выполнения с циклическим доступом. Цель этих алгоритмов - выполнять процессы из одной задачи одновременно, вместо того, чтобы обеспечивать их максимальную привязку к процессорам. Есть несколько реализаций алгоритмов копланирования: матричные, непрерывные и неразделяющие.

Неразделяющий алгоритм копланирования помещает процессы из одной задачи в смежные позиции в глобальной очереди выполнения. Планировщик использует окно, размер которого равен количеству процессоров в системе. Все процессы, попадающие в окно, выполняются параллельно в течение максимум одного кванта. Выполнив обработку группы процессов, окно перемещается к следующей группе, которая тоже выполняется параллельно в течение одного кванта.

Если процесс, попавший в окно, не может выполняться, окно расширяется на один процесс вправо, и в течение одного кванта выполняется также следующий процесс в очереди, даже если он не принадлежит к задаче, выполняемой в данный момент.

Достоинства: предотвращает бесконечное откладывание выполнения процесса, процессы одной задачи часто выполняются одновременно

Недостаток: слабая привязка к процессорам

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Алгоритм SNPF

SNPF (Smallest Number of Processes First)

Этот алгоритм может быть как вытесняющим, так и невытесняющим, но в обоих вариантах он использует глобальную очередь выполнения.

Приоритет задачи в такой очереди будет обратно пропорционален количеству процессов в этой задаче. Если задачи с одинаковым количеством процессов конкурируют, пытаясь занять один и тот же процессор, задача, которая дольше ожидала, получает приоритет.

В невытесняющей версии алгоритма SNPF, когда процессор становится доступным, алгоритм выбирает процесс из задачи в начале очереди выполнения и позволяет ему выполняться до завершения. В вытесняющей версии, если в очереди появляется новая задача с малым количеством процессов, она получает высокий приоритет и ее процессы начинают выполняться так скоро, как это возможно.

Достоинства: процессы одной задачи часто выполняются одновременно

Недостаток: выполнение задачи с большим количеством процессов может откладываться бесконечно

Планирование в многопроцессорных системах. Задачно-ориентированные алгоритмы планирования. Динамическое разделение

Динамическое разделение сводит к минимуму потери производительности вследствие промахов в кэше, пытаясь поддерживать максимальную привязку процессов к процессорам. Планировщик поровну распределяет процессоры системы между задачами. Количество процессоров, которое достанется каждой задаче, всегда меньше или равно количеству доступных для выполнения процессов в этой задаче.

Алгоритм можно усовершенствовать так, что каждый процесс будет всегда выполняться на одном и том же процессоре. Если в каждой задаче содержится по одному процессу, то динамическое разделение превратится в обычный круговой алгоритм планирования. Если в систему поступают новые задачи, система динамически изменяет распределение процессоров.

Достоинство: максимальная привязка к процессорам

Недостаток: перераспределение процессоров каждый раз, когда появляется новая задача

Миграция процессов. Процедура и концепции миграции процессов

Миграция процессов представляет собой перенос процесса с одного процессора на другой. Возможность исполнения процесса на любом процессоре предоставляет следующие преимущества:

1. процессы можно переносить на слабо загруженные процессоры и таким образом уменьшать время реагирования и повышать производительность и пропускную способность;
2. возможность миграции процессов повышает отказоустойчивость;
3. возможность миграции позволяет улучшить разделение ресурсов, т. к. в больших системах некоторые ресурсы не продублированы во всех узлах;
4. возможность миграции процессов повышает эффективность взаимодействия. Два процесса тесно взаимодействующие друг с другом, должны выполняться на одном узле или близко расположенных узлах, чтобы уменьшить задержки при взаимодействии. Поскольку каналы связи часто формируются динамически, можно воспользоваться миграцией процессов и сделать размещение процессов тоже динамически.

При миграции процессов используется стандартная последовательность шагов:

1. Сначала узел генерирует запрос миграции к удаленному узлу. В большинстве схем миграцию начинает отправитель запроса поскольку его узел перегружен или выполняющемуся на нем процессу нужен доступ к ресурсу расположенному на удаленном узле. В некоторых схемах слабо загруженный узел может запросить процессы у других узлов
2. Если отправитель и получатель соглашаются осуществить миграцию, отправитель приостанавливает выполнение процесса подлежащего миграции. Кроме того отправитель создает очередь сообщений в которую записывает все сообщения для мигрирующего процесса
3. Затем отправитель собирает данные о состоянии процесса. Для этого выполняется копирование содержимого выделенной процессу памяти, копирование содержимого регистров, состояния открытых файлов и другой связанной с процессом информации. Отправитель передает все собранные данные

получателю и получатель помещает их в специально созданный пустой процесс
4. На следующем шаге сообщения для мигрирующего процесса передаются на принимающий узел. Затем отправитель и получатель уведомляют все остальные узлы о новом месте размещения мигрировавшего процесса
5. Получатель запускает новый экземпляр процесса, а отправитель уничтожает свой экземпляр процесса

Существует различные концепции миграции процессов в разных архитектурах вычислительных систем. Самая длительная часть процедуры миграции процессов это перенос данных в памяти.

Чтобы миграция была успешной мигрирующие процессы должны обладать рядом свойств:

1. мигрирующий процесс должен быть прозрачным, т. е. его миграция не должны приводить к отрицательным последствиям, т. е. процесс при миграции не должен терять адресованных ему сообщений от других процессов или дескрипторов открытых файлов;
2. система должны быть масштабируемой, т. к. как если остаточная зависимость процесса будет расти с каждой миграцией, система может захлебнуться в потоке сетевых сообщений от процессов, обращающихся к страницам на удаленных узлах;
3. необходимо обеспечивать гетерогенную миграцию процессов, т. е. процессы должны иметь возможность миграции между разными архитектурами в распределенных системах, поэтому информация о состоянии процесса должна храниться в платформено-независимом формате.

Миграция процессов. Стратегии миграции процессов

Стратегии миграции процессов должны поддерживать равновесие между потерями производительности при переносе больших объемов данных принадлежащих процессам и выигрышам от минимизации остаточной зависимости процессов.

Полная миграция

В некоторых системах разработчики предполагают, что большая часть адресного пространства мигрировавшего процесса будет восстановлена на новом узле. В таких системах часто используются

полная миграция, при которой во время начальной миграции переносятся все станицы процесса. Это позволяет процессу выполняться на новом узле так же эффективно, как и на старом. Однако если процесс не собирается обращаться к большей части своего адресного пространства, то начальная задержка и большая загрузка каналов связи при полной миграции превращаются в накладные расходы.

Миграция для активных страниц

Чтобы уменьшить начальную стоимость полной миграции используется миграция для активных страниц. При этой стратегии переносу подвергаются только страницы помеченные как используемые. Эта стратегия подразумевает, что в системе есть вторичное устройство хранения, к которому могут обращаться и узел отправитель, и узел получатель. Все неиспользуемые страницы загружаются со вторичного устройства хранения если процесс обращается к ним с узла получателя. Этот подход снижает начальное время перемещения процесса и устраняет остаточную зависимость. Однако при каждом доступе к нерезидентной страницы возникает задержка, отсутствующая при полной миграции.

Миграция с копированием при обращении

Похожа на полную миграцию для активных страниц, но при ее использовании мигрирующий процесс может запрашивать неиспользованные страницы как со вторичного устройства хранения, так и с исходного узла. Эта стратегия обладает всеми преимуществами стратегии полной миграции для активных страниц, но кроме того позволяет диспетчеру памяти выбирать откуда запрашивать страницы. Доступ к памяти на удаленном узле может выполнять быстрее, чем доступ к дисковому накопителю, однако стратегия миграции с копированием при обращении может привести к росту нагрузки на память в узле отправителе.

Миграция с предварительным копированием

Узел отправитель начинает передавать используемые страницы еще до приостановки выполнения процесса. Любая переданная страница, к которой процесс обращается до его миграции, помечается как подлежащая повторной передаче. Чтобы гарантировать, что процесс все-таки мигрирует, система определяет нижний порог количества оставшихся используемых страниц, при котором можно начинать миграцию процесса. Когда этот пород достигается процесс не приходиться приостанавливать на долго по сравнению с другими стратегиями, например полным копированием или копирование при

обращении, а доступ к памяти в узле получателе будет быстрым, поскольку большая часть данных уже скопирована в этот узел. Кроме того эта стратегия обеспечивает минимальную остаточную зависимость. Недостаток этой стратегии заключается в том что в течении короткого промежутка времени рабочий набор процесса оказывается дублированным, т. е. существует на обоих узлах задействованных в миграции.

Синхронизация в распределенных системах. Синхронизация часов

Общие положения

Обычно децентрализованные алгоритмы имеют следующие свойства:

1. Относящаяся к делу информацию распределена среди всех компьютеров;
2. Процессы принимают решения на основе только локальной информации;
3. Не должно быть единственной критической точки, выход из строя которой приводил бы к краху системы;
4. Не существует общих часов или другого источника точного глобального времени.

В распределенных системах сложно достигнуть согласия относительно времени, что может привести к невозможности зафиксировать глобальное состояние для анализа ситуации, например, для обнаружения взаимоблокировок или для организации промежуточного запоминания.

Основные методы синхронизации времени в распределенных системах опираются на два основных требования:

- В определенные моменты системное время узлов системы должно максимально совпадать
- При синхронизации недопустим перевод часов в обратную сторону, т. е. уменьшение системного времени, поскольку это может привести к выходу за начало интервала синхронизации, либо к нарушению работы механизма определения очередности событий

Таким образом можно оперировать только переводом времени вперед, либо замедлением его хода. Если в распределенной системе присутствует узел, имеющий физические часы, либо приемник сигналов точного времени, то задача синхронизации сводится к

необходимости синхронизации остальных узлов системы с данным эталоном.

Алгоритм Кристиана

Периодически каждый узел посылает эталонному узлу запрос его текущего времени. Эталонный узел максимально быстро отвечает на этот запрос. Но следует помнить, что между отправкой сообщения сервера и его получением узлом, проходит некоторое время, которое непостоянно и зависит от текущей загрузки сети передачи данных. Для оценки этого времени, Кристиан предложил принимать время передачи ответа как половину времени между отправкой запроса и получением ответа. Для повышения точности следует производить серию таких замеров.

Достоинства:

- Простота реализации
- Высокая эффективность в небольших сетях и сетях с малой загрузкой
- Поскольку в системе присутствует источник точного времени и синхронизации производится по нему то время в системе в целом соответствует реальному

Недостатки:

- Требует внешнего источника точного времени
- Не имеет встроенной защиты от перевода часов в обратную сторону
- Некорректно работает в сетях с резкими скачками загрузки сети
- Не позволяет корректно устанавливать время в случае если запрос и ответ передаются по разным маршрутам то есть требуется разное время для передачи данных сообщений

Алгоритм Беркли

Данный алгоритм предназначен для случая, когда источник точного времени отсутствует и сервер времени опрашивает все узлы для выяснения их текущего времени. Затем сервер усредняет это значение и рассылает команды на установку нового значения времени, либо замедления часов.

Достоинства:

- Простота реализации
- Высокая эффективность для систем не критичных к отклонениям по времени

Недостатки:

- Требуется выделенный узел - сервер времени
- Поскольку источник точного времени отсутствует, а за общесистемное время отвечает принимается усреднённое время узлов, оно может иметь мало общего с реальным временем, а узлы с замедлением для коррекции времени, оказывают влияние на общесистемное время

Усредняющие алгоритмы

Семейство алгоритмов, имеющих общий принцип работы. С заданной периодичностью все узлы системы производят широковещательную рассылку текущего времени. Затем в течение определенного времени принимают сообщения о текущем времени от других узлов. Когда все сообщения приняты, запускается алгоритм вычисления текущего времени. Простейший вариант такого алгоритма - усреднение полученных значений. Алгоритм может быть усовершенствован путем отброса самых больших и самых маленьких значений для защиты от заведомо неверных значений. Другой путь усовершенствования алгоритма - это попытка оценки времени прохождения сообщения от источника возможно с учетом знаний о топологии сети для получения более точных значений.

Достоинства:

- Не требуется дополнительный узел то есть сервер времени
- Достаточно высокая эффективность подстройки под конкретные задачи

Недостатки:

- Сильная зависимость эффективности от загрузки сети
- Низкая степень синхронизации
- Не обеспеченность установки на всех узлах одинакового времени так как часть сообщений на конкретном узле может быть не получена или получена не вовремя или отброшено как заведомо ложное

Алгоритм Лэмпорта

В 1978 году Лэмпорт предложил алгоритм синхронизации времени при этом он указал, что абсолютной синхронизации не требуется. Если два процесса не взаимодействуют то единого времени им не требуется, кроме того в большинстве случаев согласованное время может не иметь ничего общего с астрономическим временем и в таких случаях можно говорить о логических часах. Для синхронизации логических часов Лэмпорт определил отношение "произошло до" a->b - a произошло до b. Означает что все процессы согласны с тем что

сначала произошло событие a, а затем событие b. Это отношение может быть очевидным в двух случаях:

1. Если оба события произошли в одном процессе;
2. Если событие a операция отправки сообщения в одном процессе, а событие b прием этого сообщения другим процессом.

Если два события a и b случились в разных процессах, которые не обмениваются сообщениями и отношение и a->b и b->a являются не верными и эти события называются одновременными. Логическое время t должно соответствовать условию - если a->b то t(a) <t(b). При обмене данными между взаимодействующими узлами происходит так же обмен сведеньями о их локальном времени, если отметка в сообщении о локальном времени оказывается меньше, чем локальное время принимающего узла, такая ситуация считается штатной так как сообщение было получено раньше чем получено. Если же в сообщении указано более позднее время, чем локальное время узла, то есть сообщение было послано позже чем получено делается вывод о необходимости коррекции системного времени узла. Время устанавливается в значение времени сообщении +1.

Достоинства:

- Не требуется выделенный узел, то есть сервер времени
- Высокая эффективность при малом числе узлов
- Высокая степень синхронизации
- Время переводится только вперед и нет необходимости ждать эффекта от замедления времени на спешащих узлах

Недостатки:

- Не принимается в расчет время пересылки сообщений между узлами
- Поскольку имеет место постоянный перевод часов вперед обще системное время имеет мало общего с реальным временем

Синхронизация в распределенных системах. Распределенные алгоритмы синхронизации

Распределенный алгоритм

Когда процесс хочет войти в критическую секцию, он формирует сообщения, содержащие имя нужной ему критической секции, номер процесса и текущее значение времени. Затем он посылает это сообщение всем другим процессам. Предполагается, что передача сообщения надежна, то есть получение каждого сообщения сопровождается подтверждением.

Когда процесс получает такое сообщение, его действия зависят от того, в каком состоянии он находится по отношению к указанной в сообщении критической секции. Возможны 3 ситуации:

1. Если получатель не находится в критической секции и не собирается в данный момент в нее входить, то он посылает процессу-отправителю сообщение с разрешением.
2. Если получатель уже находится в критической секции, то он не отправляет никакого ответа, а ставит запрос в очередь.
3. Если получатель хочет войти в критическую секцию, но еще не сделал этого, то он сравнивает временную отметку поступившего сообщения со значением времени, которое содержится в его собственном сообщении, разосланном другим процессам. Если время в поступившем к нему сообщении меньше, то есть его собственный запрос возник позже, то он посылает сообщение-разрешение, в противном случае он не посылает ответ, а ставит поступившее сообщение-запрос в очередь.

Процесс может войти в КС только в том случае, если он получил ответное сообщение-разрешение от всех остальных процессов. Когда процесс покидает критическую секцию, он посылает сообщение всем процессам из своей очереди и исключает эти процессы из очереди.

Алгоритм Token Ring

Алгоритм Token Ring создает логическое кольцо, в которое входят все процессы системы и каждый процесс знает номер своей позиции в кольце, а также номер ближайшего к нему следующего процесса. Когда кольцо инициализируется, процессу 0 передается так называемый маркер (token). Токен циркулирует по кольцу. Он переходит от процесса n к процессу n+1 путем передачи сообщения по типу "точка-точка". Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в КС. Если да, он входит в данную КС и только после того, как процесс выйдет из КС, он передает токен дальше по кольцу. Если же процесс, принявший токен от своего соседа, не заинтересован во вхождении в КС, то он сразу отправляет токен в кольцо. Следовательно, если ни один из процессов не желает входить в КС, то в этом случае токен просто

циркулирует по кольцу с высокой скоростью. Ни один процесс не может дважды подряд входить в КС, не отдавая маркер.

Синхронизация в распределенных системах. Алгоритмы голосования

Общие положения

Многие распределенные алгоритмы требуют, чтобы один из процессов был координатором, инициатором или выполнял другую специальную роль. Обычно не важно, какой именно процесс выполняет эти специальные действия, главное, что-бы он вообще существовал.

Если все процессы абсолютно одинаковы и не имеют отличительных характеристик, способа выбрать один из них не существует. Соответственно, считается, что каждый процесс имеет уникальный номер, например сетевой адрес (для простоты считается, что на каждую машину приходится по процессу).

В общем, алгоритмы голосования пытаются найти процесс с максимальным номером и назначить его координатором. Алгоритмы разрешаются способами поиска.

Также считается, что каждый процесс знает номера всех остальных процессов, но не известно, какие из них в настоящее время работают, а какие нет. Алгоритм голосования должен гарантировать, что если голосование началось, то оно, рассмотрев все процессы, решит, кто станет новым координатором.

Алгоритм «забияки»

Когда один из процессов замечает, что координатор больше не отвечает на запросы, он инициирует голосование. Например, процесс P проводит голосование следующим образом:

1. Процесс P посылает всем процессам с большими, чем у него, номерами сообщение типа ГОЛОСОВАНИЕ;

2. Далее возможно два варианта развития событий:

- если никто не отвечает, процесс P выигрывает голосование и становится координатором;
- если один из процессов с большими номерами отвечает, то он становится координатором, а процесс P прекращает опрос и продолжает работу как обычный процесс.

В любой момент процесс может получить сообщение ГОЛОСОВАНИЕ от одного из своих коллег с меньшим номером. По получении этого

сообщения получатель посылает отправителю сообщение ОК, показывая, что он работает и готов стать координатором. Затем получатель сам организует голосование. В конце концов, все процессы, кроме одного, отпадут, этот последний и будет новым координатором. Он уведомит о своей победе посылкой всем процессам сообщения, гласящего, что он новый координатор и приступает к работе.

Если процесс, который находился в нерабочем состоянии, начинает работать, он организует голосование. Если он оказывается процессом с самым большим из работающих процессов номером, он выигрывает голосование и берет на себя функции координатора. Итак, побеждает всегда самый большой парень в городишке, отсюда и название «алгоритм забияки».

Кольцевой алгоритм

Основан на использовании кольца. В отличие от некоторых других кольцевых алгоритмов в этом не требуется маркер. Мы предполагаем, что процессы физически или логически упорядочены, так что каждый из процессов знает, кто его преемник. Когда один из процессов обнаруживает, что координатор не функционирует, он строит сообщение ГОЛОСОВАНИЕ, содержащее его номер процесса, и посылает его своему преемнику. Если преемник не работает, отправитель пропускает его и переходит к следующему элементу кольца или к следующему, пока не найдет работающий процесс. На каждом шаге отправитель добавляет свой номер процесса к списку в сообщении, активно продвигая себя в качестве кандидата в координаторы.

В конце концов, сообщение вернется к процессу, который начал голосование. Процесс распознает это событие по приходу сообщения, содержащего его номер процесса. В этот момент тип сообщения изменяется на КООРДИНАТОР и вновь отправляется по кругу, на этот раз с целью сообщить всем процессам, кто стал координатором (элемент списка с максимальным номером) и какие процессы входят в новое кольцо. После того как это сообщение один раз обойдет все кольцо, оно удаляется и процессы кольца возвращаются к обычной работе.

Синхронизация в распределенных системах. Централизованные алгоритмы синхронизации

Централизованный алгоритм

Для синхронизации выполнения процессов решающих общую задачу или работающих с общими данными или при работе с монопольными ресурсами используются несколько алгоритмов. Централизованный алгоритм предполагает, что один из процессов выбирается в качестве координатора. Когда какой-либо процесс хочет войти в критическую секцию он посылает сообщение с запросом координатору в какую критическую секцию он хочет войти и ждет от него разрешение. Если в этот момент ни один из процессов не находится в критической секции то координатор посылает ответ с разрешением. Если же некоторый процесс уже выполняет критическую секцию связанную с данным ресурсом то ответ не посылается, а запрашивающий процесс ставится в очередь. Этот алгоритм гарантирует взаимное исключение, но вследствие своей централизованной природой обладает низкой отказа устойчивостью.

Для выбора координатора используется несколько алгоритмов и в большинстве случаев координатором назначается процесс с наибольшим идентификатором. Процесс выбора координатора инициируется тогда, когда координатор перестает работать, т. е. не отвечает на запросы.

Методы борьбы с тупиками в распределенных операционных системах

Тупики в РПС подобны тупикам в централизованных системах, но их сложнее обнаружить и предотвратить. Иногда выделяют особый вид тупиков в РПС - коммуникационные тупики. При борьбе с тупиками в РПС используется несколько стратегий:

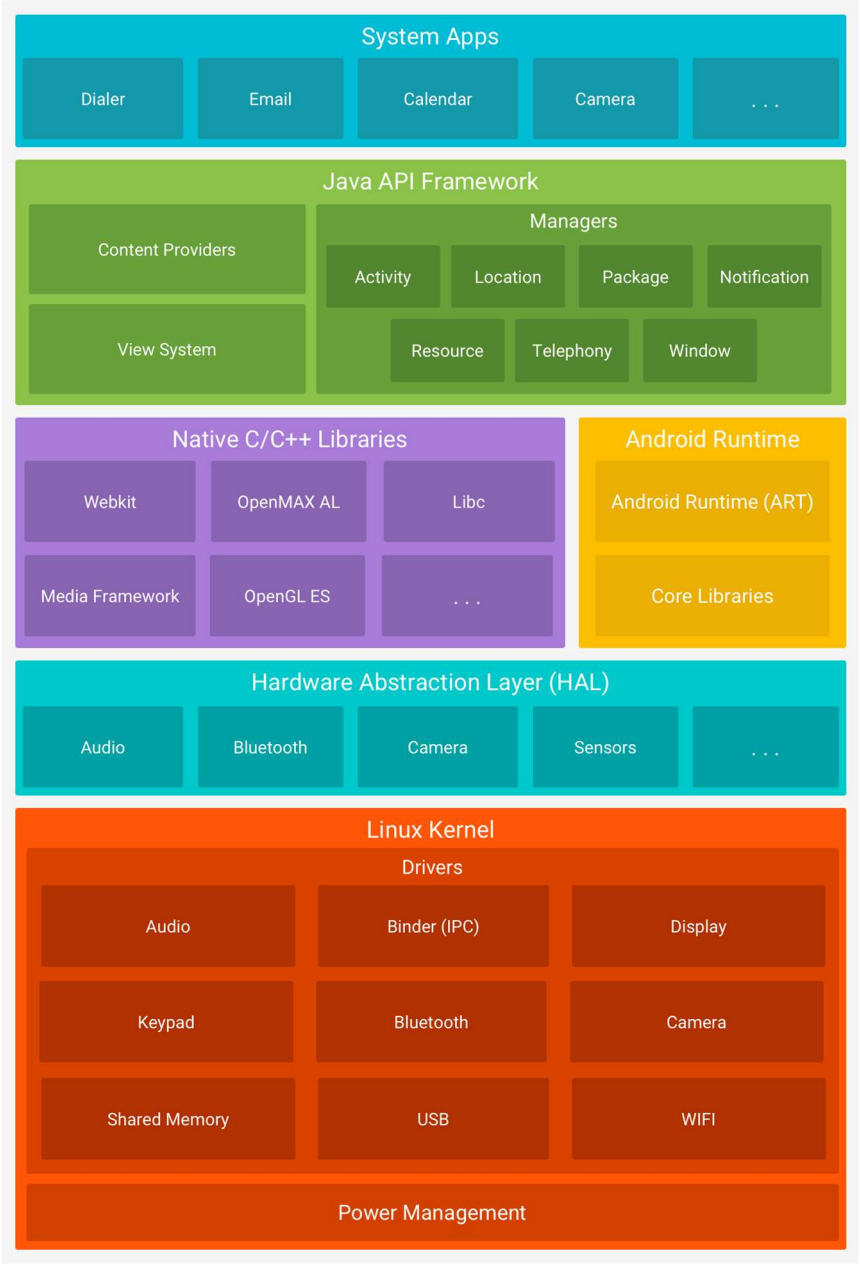
1. Полное игнорирование проблемы
2. Обнаружение тупиков

При обнаружении тупиков используются два основных метода:

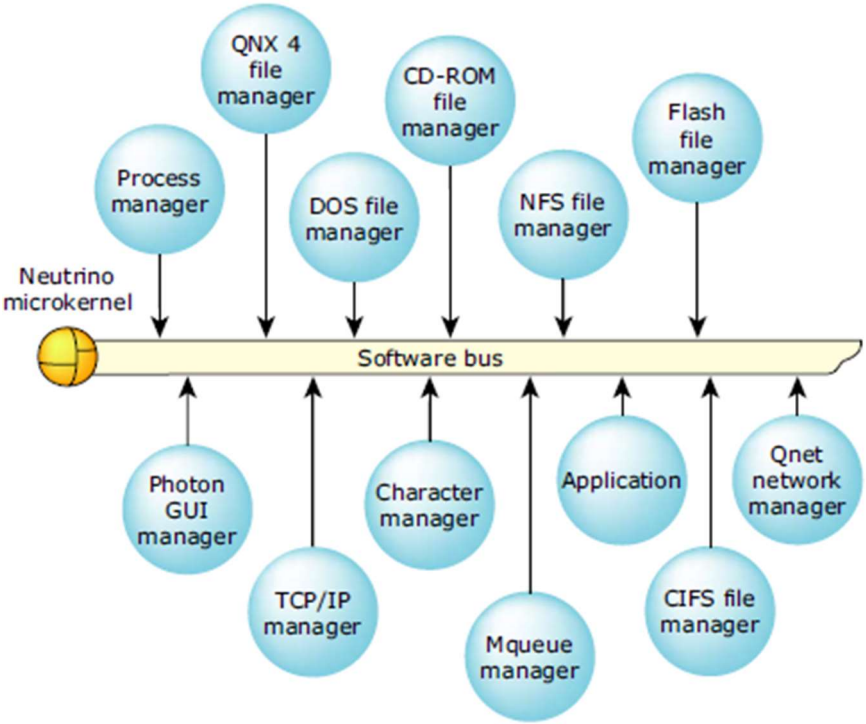
1. Централизованное обнаружение тупиков. Заключается в распространении идеи графа размещения ресурсов на РПС. Координатор строит единый граф для всех узлов системы и на нем выполняет редукцию.
2. Распределенное обнаружение тупиков. Здесь инициирование задачи обнаружения тупика начинает процесс, который подозревает, что система находится в тупике. Он посылает тем процессам, которые держат нужный ему ресурс сообщение из трех информационных полей. Первое - номер процесса, инициировавшего поиск, и оно не меняется при дальнейшей пересылке. Второе и третье поля содержат соответственно номер процесса, который зависит от ресурса и номер процесса, который держит этот ресурс. Если в конце концов сообщение вернется к тому процессу, который инициировал поиск, то он приходит к выводу, что система зациклена и он должен предпринять действия по ликвидации тупика.

Предотвращение тупиков в РПС базируется на идеи упорядочивания процессов по их временным меткам. Допустимой является ситуацией, когда старший процесс ждет ресурсы, которые захватил младший процесс. В противном случае младший процесс должен отказаться от ожидания.

Архитектура операционной системы Android



Архитектура операционной системы QNX



Состав микроядра:

- Сервер потоков POSIX
- Сервер сигналов POSIX
- Сервер передачи сообщений между потоками (IPC)
- Сервер синхронизации потоков
- Сервер планировщика
- Сервер таймеров
- Сервер управления процессами и памятью

Архитектура операционной системы Windows