OS LEC 04

isagila

Собрано 05.02.2024 в 20:38

Содержание

1. Лекции

1.1. Лекция 24.??.??.

Эволюция понятия «операционная система»

Def 1.1.1. Операционная система это базовое системное программное обеспечение, управляющее работой вычислительного узла и являющееся посредником между аппаратным обеспечением, прикладным программным обеспечением и пользователем.

Принципы архитектуры фон Неймана:

1. Однородность памяти (и код, и данные находятся в единой памяти).
2. Адресность (линейная система адресов) и произвольный доступ к ячейкам памяти.
3. Принцип программного управления.
4. Принцип кодирования (для всего используется двоичное кодирование).

I. Программы—диспетчеры

Одной из задач, решаемых на этом этапе, была задача повторного использования кода, автоматизации загрузки и линковки. Было замечено, что некоторые участки кода часто повторяются, и чтобы не переписывать их заново, была предложена идея выделить в оперативной памяти некоторый участок, в который заранее положить необходимый код, а в основной программе лишь ссылаться на эти функции. В рамках этой идеи удобно использовать программы, которые подставят нужные адреса (слинкуют) заготовленные функции в основной программный код. Также необходимо неким образом обеспечить передачу параметров в эти функции и возможность взаимодействовать с их результатом. Отсюда и возникла потребность в программном обеспечении, которое будет автоматически решать эти задачи.
Следующая задача, которая возникла на этом этапе, это задача оптимизации взаимодействия с устройствами хранения и ввода-вывода. Дело в том, что не всегда все данные можно сохранить в оперативной памяти, а иногда это просто невыгодно или ненужно. Таким образом, нужно неким образом подгружать необходимые данные в оперативную память (из хранилища или с потокового ввода) и выгружать обработанные данные (в хранилище или в потоковый вывод). На тот момент это можно было делать только используя центральный процессор, т.к. в противном случае может возникнуть рассинхронизация: например, операция записи в оперативную память завершится позже, чем требуется, и попытка работать с этими не до конца загруженными данными приведет либо к сбою, либо к некорректной обработке этих данных.
Однако понятно, что это не самый оптимальный способ. Оптимальнее было бы сделать например так: процессор работает с одним блоком памяти, в то время как другой, независимый блок оперативной памяти подгружается из хранилища (или освобождается). Чтобы применить эту идею, введем дополнительно контроллер, который свяжем и с хранилищем, и с оперативной памятью. Разрешим этому контроллеру независимо от ЦП заниматься откачкой и подкачкой данных между хранилищем и оперативной памятью.
Но тогда возникает другая проблема: мы не можем прогнозировать время выполнения этих операций, т.к. большинство хранилищ используют технологии, которые это не позволяют. Чтобы решить эту проблему нужно связать (синхронизировать) работу контроллера и работу центрального процессора. Таким образом появляется механизм прерываний: сначала ЦП дает управляющую команду контроллеру на закачку определенного блока данных. По окончании этой операции контроллер инициирует прерывание, процессор приостанавливает выполнение текущей операции и обрабатывает это прерывание. В ходе обработки прерывания можно (например) изменить некоторый флаг, с помощью которого основная программа поймет о том, что данные готовы к обработке. Данная модель получила название spooling (обеспечение взаимодействия с периферийным устройством параллельно с работой основного вычислительного процесса).

Def 1.1.2. Прерывание это сигнал, поступающий от внешнего устройства к центральному процессору, сообщающий о наступлении некоторого события, в результате которого процессор приостанавливает выполнение текущего набора команд и передает управление подпрограмме — обработчику прерывания.

Последняя задача в рамках этапа программ—диспетчеров это появление однопрограммной пакетной обработки. Т.к. программы становились больше и иногда возникала потребность в том, чтобы переиспользовать некоторые специфичные части кода, то появилось разделение на пакеты. Теперь в память загружалась не просто программа, а некоторое количество пакетов: пока один пакет выполнялся, другой пакет (или несколько пакетов) подгружались в память. Появляется понятие очереди из пакетов, и возникает необходимость в управлении этой очередью: например, некоторые пакеты по тем или иным причинам должны получить приоритет.
Итого, задачи решаемые на данном этапе:

1. Задача повторного использования кода, автоматизации загрузки и линковки.
2. Задача оптимизации взаимодействия с устройствами хранения и ввода-вывода.
3. Задача однопрограммной пакетной обработки.

Разные программы имеют разные требования к ЦП и к контроллеру: есть программы, которые используют много памяти, но мало процессорного времени, а есть программы, которые наоборот, требуют мало памяти, но сильно нагружают процессор. Однако вычислительный узел должен быть универсален и не должен подстраиваться лишь под одну задачу. Итого получается, что в зависимости от текущей программы либо процессор, либо контроллер будет простаивать. Это неэффективно, из-за чего появляется идея сделать так, чтобы пока одна программа использует ресурсы ЦП, другая могла бы использовать контроллер, чтобы погрузить необходимые ей данные в оперативную память. Таким образом ни ЦП, ни контроллер не будут простаивать впустую, а эффективность работы узла будет определяться тем, сколько программ одновременно смогут использовать необходимые им ресурсы.
Однако такой подход влечет за собой массу проблем: если раньше была единая очередь и порядок выполнения команд был очевиден, то теперь необходимо переключаться между программами, следить за тем, чтобы программы не использовали чужие участки памяти, организовывать работу с хранилищем, следить за уровнем доступа к ресурсам и так далее. Решением этих проблем является операционная система, которая занимает позицию между аппаратным обеспечением, программным обеспечением и пользователями.

1.2. Лекция 24.??.??.

Как уже было сказано на предыдущей лекции, для эффективного использования ресурсов потребовалось одновременно выполнять (или по крайней мере удерживать в памяти) несколько программ. Однако количество ядер меньше, чем количество процессов, которые необходимо исполнять (на тот момент речь шла вообще об процессорах с одним ядром, которые могут выполнять одновременно только одну программу). Соответственно, возникает задача об обеспечении разделения времени процессора. Таким образом, мы сначала даем одной программе (пакету) некоторое время использовать процессор, потом выполняем переключение (которое тоже занимает какое-то время) и далее предоставляем ресурс процессора другой программе. Время, которое мы предоставляем каждой программе, может быть одинаковым, а может и различаться — об этом мы поговорим позже.
Для реализации этих новых механизмов потребовались некоторые дополнительные вещи. В частности, потребовался таймер, который мог бы вызывать еще один вид прерываний (до этого существовали только прерывания ввода-вывода). Стоит отметить, что механизм прерываний по таймеру не так прост: если в результате обработки этого прерывания было принято решение переключиться на исполнение другого процесса, то необходимо неким образом сохранить текущее состояние регистров процессора, чтобы потом иметь возможность вернуться к этому процессу и продолжить его выполнять.
Однако ЦП это не единственный ресурс. Вторая серьезная задача это обеспечение разделения памяти, т.к. если несколько программ находятся в памяти одновременно, то необходимо каким-то образом обеспечить корректную адресацию, ведь на этапе разработки программного обеспечения нельзя знать, как и где оно будет размещено в памяти. Решением этой проблемы является виртуализация памяти — механизм виртуальной памяти. Каждая программа внутри себя отсчитывает адреса от некоторого виртуального нуля, а операционная система предоставляет механизм пересчета этих виртуальных адресов в физические, причем этот пересчет может осуществляться как в момент загрузки программы в оперативную память, так и при каждом обращении (или не при каждом — стратегия пересчета может быть и другой).
Задача обеспечения разделения памяти повлекла за собой задачу обеспечения защиты программ от деятельности других программ. Для решения этой задачи появляется еще одно прерывание, которое обеспечивает защиту памяти и прерывает работу процессора, если происходит попытка обратиться к памяти, принадлежащей другому приложению. Стоит отметить, что защищать от других программ нужно не только память, но и другие ресурсы.
Следующей задачей является планирование использования ресурсов и исполнения программ. Данная задача является более сложной, нежели планирование очереди пакетов. Помимо этого стоит учитывать, что планирование исполнения программы тесно связано с планированием ресурсов, которые необходимо выделить этой программе. Также нужно не забывать про синхронизацию: некоторые ресурсы являются неразделяемыми, и поэтому нельзя давать к ним доступ поочередно. Например, если две программы хотят что-то напечатать на принтере, то ЦП нельзя просто переключаться между ними и печатать по несколько символов от каждой программы.
Еще одной задачей на этапе мультипрограммных операционных систем является задача обеспечение универсального доступа к устройствам хранения. Для решения этой задачи появляется файлово-каталожная система и модель прав доступа к разным файлам и каталогам.
Итого, основные задачи данного этапа:

1. Обеспечение разделения времени процессора.
2. Обеспечение разделения памяти.
3. Защита программ от действий других программ.
4. Планирование выполнения и синхронизация выполнения.
5. Обеспечение универсального доступа к устройствам хранения.

Появляется механизм виртуальной машины. Каждое приложение работает как будто на своей виртуальной версии вычислительного узла и ничего не знает про другие приложения и про физические ресурсы вычислительного узла. Операционная система становится тем уровнем абстракции, который разделяет аппаратное обеспечение и программное обеспечение, позволяя им взаимодействовать только через нее, но не напрямую. Чтобы поддерживать эту абстракцию требуются некоторые строгие интерфейсы взаимодействия. Интерфейсом взаимодействия с аппаратным уровнем становится механизм привилегированного режима (только код ядра операционной системы имеет доступ к управлению физическими ресурсами вычислительного узла). В обратную стороны мы получаем механизм прерываний. С точки зрения программного обеспечения также появляется механизм взаимодействия — механизм системных вызовов (это «просьба» к ядру операционной системы о выполнении некоторой привилегированной операции или предоставлении некоторого аппаратного ресурса).
Одной из первых операционных систем считается операционная система MCP (1963 год).

III. Сетевые операционные системы

Узким местом становятся операции ввода-вывода. В связи с развитием качества связи появляется понятия удаленного терминала и многотерминальности. Теперь терминал совмещает в себе как функции ввода, так и функции вывода. Из-за удаленности терминала появляется проблема идентификации: если раньше четко можно было отследить, кто работает с вычислительным узлом (т.к. для этого необходимо было физически находится рядом), то теперь сделать это не так просто. Появляется потребность в дополнительных механизмах идентификации, аутентификации и авторизации. Помимо территориального разделения терминалов появляется идея о том, чтобы каким-то образом связать несколько вычислительных узлов. Эта идея возникла для того, чтобы разделять выполнение задач между несколькими вычислительными узлами и не позволять одному из узлов простаивать в то время как другой узел полностью загружен. Таким образом появляются сетевые операционные системы.

IV. Универсальные (мобильные открытые) операционные системы

На данном этапе, т.к. почти под каждый вычислительный узел нужна своя операционная система, то возникает плохая переносимость: ПО, разработанное для одной операционной системы, не всегда может работать на другой операционной системе. Таким образом от операционной системы требуется универсальность, а значит она должна поддерживать разработку приложений на языке высокого уровня, что позволит абстрагироваться от прямого доступа к ресурсам. Для того, чтобы это осуществить, необходимо, чтобы сама операционная система была написана на языке высокого уровня. Решение этой проблемы было найдено в 1969 году (Томсон, Керниган и Ритчи). Им стал язык `C` и операционная система `UNICS`. Первая редакция этой операционной системы пишется на ассемблере и не имеет встроенного компилятора языка высокого уровня. Одновременно с этим разрабатывается интерпретируемый язык `B`, и к 1972 году на этом языке переписывается `UNICS` (вторая редакция). Также разрабатывается компилируемый язык `C`, а код, написанный на `B`, постепенно переписывается на `C`. Итого к 1973 году появляется редакция `UNICS` с встроенным компилятором `C`. Далее, в конце 1973 года, появляется четвертая редакция, в которой ядро полностью написано на `C`, а к 1975-ому году и все утилиты также переписываются на `C` — это и становится пятой редакцией переименованной операционной системы `Unix`. Последняя редакция выходит в 1978 году. Далее уже появляются операционные системы называемые `*nix` системами в знак того, что они многое унаследовали от `Unix`.
Одной из систем на основе `Unix` является `BSD`, которая в некотором виде дожила и до наших дней (`FreeBSD`, `OpenBSD`, `NetBSD` и т.д.). На основе `BSD` была сделана операционная система `SunOS`, которая позже станет `Solaris`, а далее и `OpenSolaris`. Также на основе `Unix` появляются и проприетарные решения такие как `HP-UX`, `AIX`, `IRIX` и другие. Помимо этого стоит отметить `SystemV`, которая появилась как попытка связать `Unix`, развивающийся на тот момент `Solaris` и ветку `BSD` решений.
В 1983 году появляется проект `GNU`. Идея этого проекта заключается в том, чтобы создать свободное ПО и свободную операционную систему, на которой это ПО будет работать. Основатель `GNU` Ричард Столлман выделяет четыре свободы:

1. Свобода использовать программное обеспечение.
2. Свобода изучать и адаптировать программное обеспечение.
3. Свобода распространять программное обеспечение.
4. Свобода улучшать и публиковать программное обеспечение.

Из последнего пункта вытекает идея copyleft-а. Эта лицензия говорит о том, что если данное ПО интегрируется в другой проект или модифицируется, то этот проект также должен иметь copyleft лицензию. Итого все производные проекты и производные от них проекты также будут свободными (т.е. с copyleft лицензией).
Акроним `GNU` рекурсивно расшифровывается как Gnu is Not Unix. В рамках этого проекта создается компилятор `gcc`, пишутся и переписываются библиотеки языка `C`, однако остается проблема с ядром: написать с нуля ядро новой операционной системы оказывается сложной задачей. В 1991 публикуется операционная система `Linux`, которая несмотря на то, что была основана на операционной системе `Minix`, обладала собственным, концептуально новым ядром. Эндрю Таненбаум (автор `Minix`) высказывает резкую критику в сторону новой операционной системы, отмечая монолитность ядра и невозможность переноса на другие архитектуры помимо 8086. Для развития `Linux` все чаще начинает использоваться ПО, разработанное в рамках проекта `GNU`. Таким образом появляется операционная система, которая сейчас называется `GNU/Linux`.

Еще одним примером *nix системы является появившаяся в 1989 году операционная система NeXTSTEP. В 1997 году она интегрируется вместе с некоторыми наработками из FreeBSD в проект Darwin, который позже становится родоначальником операционных систем семейства macOS.

1.3. Лекция 24.??.??.

Цель существования современной операционной системы заключается в том, что она должна обеспечить производительность, надежность и безопасность выполнения пользовательских программ, эксплуатации аппаратного обеспечения, хранения и доступа к данным (в том числе по сети) и диалога с пользователем.
Т.к. операционная система это сложное и комплексное ПО, то чтобы лучше его понять, сначала надо поговорить об его архитектуре. Выделяют несколько уровней архитектуры программного обеспечения:

1. Функциональная архитектура.

   Этот уровень описывает всю совокупность функций, выполняемых операционной системой.

2. Системная архитектура.

   Реализация операционной системы это программно-аппаратный комплекс: есть аппаратные компоненты, которые являются неотъемлемой частью операционной системы, а есть программные компоненты, которые могут быть как свободными, так и проприетарными библиотеками.

3. Программная архитектура.

   Т.к. операционная система содержит некоторые программные компоненты, то нужно учитывать, что эти компоненты также обладают собственной структурой.

4. Архитектура данных.

   Предыдущий пункт говорит нам о сложности организации кода, а значит этот код работает с не менее сложноорганизованными данными, поэтому имеет смысл говорить об архитектуре данных.

Функциональная архитектура. Функции операционной системы

1. Управление разработкой и исполнением пользовательского ПО.
   (a) Предоставление возможности и API для написания программного обеспечения, совместимого с этой операционной системой.
   (b) Предоставление возможности загрузить и выполнить написанное приложение, а также обеспечить ему доступ к требуемым в процессе исполнения ресурсам.
   (c) Обнаружение и обработка ошибок, возникающих в ходе выполнения написанного ПО.
   (d) Высокоуровневый доступ к устройствам ввода-вывода.
   (e) Управление хранилищем данных: обеспечение высокоуровневого доступа к данным и их безопасности.
   (f) Мониторинг ресурсов.

2. Оптимизация использования ресурсов.

   Обозначим $k_1, k_2, \ldots$ — критерии оптимальности использования соответствующего ресурса. Т.к. в распоряжении вычислительного узла обычно находится не один, а множество ресурсов, то чаще всего оптимизировать все критерии одновременно невозможно. Из-за этого операционные системы используют разные принципы оптимизации, например:

   (a) Суперкритерий (свертка).

   Пусть $\widehat{k} = \alpha k_1 + \beta k_2 + \gamma k_3 + \ldots$ при условии, что $\alpha + \beta + \gamma + \ldots = 1$, т.е. используется взвешенная сумма критериев. Выбирается та стратегия, у которой суперкритерий $\widehat{k}$ максимален.

   (b) Условный критерий.

   В некоторых ситуациях значения какого-либо критерия (или нескольких критериев) обязательно должны находится в некотором диапазоне. Тогда сначала ищется «область» в которой выполнены требуемые условия к критериям, а потом уже оптимизируются остальные критерии.

   Стоит помнить, что операционная система это открытая система: пользователи открывают новые приложения, закрывают старые, запускают на обработку большие объемы данных или наоборот, бездействуют — в общем, помимо того, что требуется решать сложную задачу многокритериальной оптимизации, также нужно учитывать текущий контекст. Операционные системы обычно для решения этой проблемы используют ту или иную реализацию цикла Деминга (PDCA). Данный цикл состоит из четырех этапов:

   (a) Планирование. На этом этапе формируются некоторые значения коэффициентов $\alpha, \beta, \gamma, \ldots$.
   (b) Выполнение. На этом этапе операционная система принимает решения согласно выбранному плану.

(c) Проверка. На данном этапе происходит проверка сделанных решений на соответствие некоторым целевым показателям.

(d) Действия. На данном этапе нужно каким либо образом исправить несоответствие полученного результата плану.

Далее система уходит на новый цикл: на новом этапе планирования может быть выбрана другая стратегия, т.к. ситуация поменялась, или та же самая, если она хорошо себя зарекомендовала.

3. Поддержка администрирования и эксплуатации вычислительного узла.

   Операционная система должна предоставлять средства для диагностики, системного администрирования (восстановление после сбоев), восстановления поврежденных файлов (резервное копирование).

4. Поддержка развития самой операционной системы.

   Операционные системы, как чрезвычайно сложное ПО, проектируются и создаются в течение очень долго времени, а значит и использоваться будут также длительное время (процесс перехода на новую ОС связан с риском и определенными затратами). Это значит, что ОС должна быть открыта к изменениям: за время ее использования появятся новые программные и аппаратные решения, будет написано новое ПО, и ОС должна быть спроектирована так, чтобы у этих продуктов была возможность работать в рамках этой ОС (либо должна быть возможность добавить их поддержку). В современных ОС эта функция обычно реализуется с помощью средств автоматического обновления.

Функциональные подсистемы

1. Подсистема управления процессами.
   (a) Планировщики.
   (b) Структуры данных, отвечающие за хранение данных о процессах.
2. Подсистема управления памятью.
   (a) Механизм виртуализации памяти.
   (b) Защита памяти одного процесса от других процессов.
   (c) Распределение данных по памяти с разной скоростью доступа.
3. Подсистема управления файлами.
   (a) Преобразование символьных имен файлов в адреса их физического хранения.
   (b) Механизм управления каталогами.
4. Подсистема управления внешними устройствами.
5. Подсистема защиты данных и администрирования.
   (a) Идентификация, аутентификация, авторизация пользователя.
   (b) Аудит операционной системы, действий пользователя, поведения приложений, сетевой активности и т.д.
6. API.
7. Подсистема пользовательского интерфейса.
   (a) Интерфейс командной строки.
   (b) Графический пользовательский интерфейс (GUI).

1.4. Лекция 24.??.??.

Системная архитектура

Т.к. требования, предъявляемые к операционной системе, противоречивы и операционная система является открытой системой, то в области операционных систем нет «идеального» решения, которое бы удовлетворило всем требованиям — всегда приходится искать компромисс. Исходя из этого существуют разные архитектурные решения, имеющие разные преимущества и недостатки. Главным моментом, определяющим архитектуру ОС, является ответ на вопрос: что и как будет выполняться в ядре, а что будет вынесено за его пределы.

Def 1.4.1. Ядром операционной системы называется та часть ее кода, которая отличается двумя характеристиками присущими только ей (в совокупности) и никому больше: резидентность и привилегированный режим. Причем если резидентность может быть присуща другому ПО, то привилегированный режим это характеристики только ядра операционной системы.

Def 1.4.2. Резидентность ядра означает то, что его код находится в оперативной памяти всегда, в течении всего периода эксплуатации операционной системы, и как правило в неизменных адресах.

Def 1.4.3. Код, выполняемый в привилегированном режиме, не ограничивается проверками на доступ к адресам памяти.

Таким образом с одной стороны хочется, чтобы код ядра был как можно меньше: так он будет требовать меньше оперативной памяти, а также повысится надежность, ведь чем больше кода имеет доступ к привилегированному режиму, тем больше вероятность возникновения ошибки, а ошибка на таком уровне будет стоить очень дорого. С другой стороны встает вопрос безопасности: чем меньше будет ядро, тем больше функций придется выполнять вне привилегированного режима, а значит придется переключаться между режимами, чтобы обеспечить взаимодействие компонентов ОС. Если же бóльшая часть компонентов ОС помещена в ядро, то они могут взаимодействовать между собой напрямую, что увеличивает быстродействие системы. Помимо этого чем меньше код ядра, тем больше существует возможностей вмешаться в работу той части ОС, которая не защищена привилегированным режимом.
Помимо ядра существуют и другие принципы, которые влияют на архитектуру операционных систем:

1. Принцип модульной организации.

   Операционная система, как и любое сложное ПО, должна быть представлена в виде совокупности модулей с изолированной функциональностью.

2. Принцип функциональной избыточности.

   Операционная система должна обладать функционалом бóльшим, чем нужно каждому конкретному пользователю здесь и сейчас.

3. Принцип функциональной избирательности.

   Архитектура операционной системы должна позволять нам выбирать между функциями, предоставляемыми этой ОС. Причем должна быть возможность делать выбор на разных уровнях: например, какую-то функцию можно временно отключить, чтобы не расходовать на нее ресурсы, а какую-то функциональность можно получить, если поставить дополнительный пакет.

4. Принцип параметрической универсальности.

   Операционная система должна выносить во внешнюю среду как можно больше своих параметров управления.

5. Концепция многоуровневой иерархической вычислительной системы.

   Операционная система обычно делится на слои, и, обеспечив интерфейс взаимодействия между слоями, мы получаем возможность изменять один из слоев, не затрагивая остальные.

6. Принцип разделения модулей операционной системы.

   Модули операционной системы делятся на модули ядра и модули, относящиеся к вспомогательным функциям.

Архитектуры операционных систем

I.a Монолитная архитектура

Несмотря на название в монолитной архитектуре можно выделить три слоя (не всегда явно, но обычно это так): главная программа, сервисы и утилиты. Главная программа представлена одним модулем и умеет взаимодействовать с сервисами, а сервисы уже взаимодействуют с утилитами по принципу «многие-ко-многим», т.е. один сервис может взаимодействовать с несколькими утилитами и одна утилита может использоваться несколькими сервисами. Для чего же нужно такое разделение?
Утилиты обеспечивают нам работу с аппаратной частью, каждая их них реализует некоторый протокол взаимодействия с контроллером соответствующего экземпляра аппаратного обеспечения. Главная программа же является интерфейсом для взаимодействия с пользовательским ПО — ее задачей является получение системных вызовов. Механизм системных вызовов работает следующим образом: программа помещает в некоторой (но не произвольной) части выделенной ей памяти идентификатор системного вызова и необходимые аргументы и инициирует программное прерывание. Далее управление передается ядру операционной системы (в данном случае слою главной программы) и происходит анализ: какая программа инициировала системный вызов, какие аргументы были переданы и т.д. Из-за того, что утилиты работают с аппаратной частью, то существует еще слой сервисов, которые умеют принимать решения и исполнять их с помощью утилит. Главная программа после «расшифровки» системного вызова вызывает один или несколько сервисов, которые уже непосредственно занимаются его выполнением. Результат работы сервиса передается сначала главной программе, а потом и инициировавшему системный вызов приложению.
К преимуществам этой архитектуры относится быстродействие (сервисы могут вызывать утилиты без переключения режима), безопасность (все решения принимаются на уровне ядра). Из недостатков можно отметить проблемы с надежностью и повышенные затраты памяти.

I.b Многослойная архитектура

Со временем из-за большего количества функций, возложенных на операционную систему, количество сервисов довольно сильно увеличилось и появилась идея о разделении среднего слоя сервисов на несколько. Это концепция многослойной архитектуры. **Нельзя считать, что это новая отдельная архитектура** (т.к. даже в рамках монолитной архитектуры можно выделить слои): это скорее концепция, которая является логическим продолжением монолитной архитектуры и далее позволит нам перейти к другим видам архитектур. Причем стоит отметить, что многослойную архитектуру строили по-разному, и она менялась с течением времени. Далее будет рассмотрен лишь один из вариантов ее трактовки. Данную архитектуру удобно представить в виде концентрических окружностей. Тогда если смотреть от центра наружу, то порядок слоев будет такой:

1. Аппаратное обеспечение.
2. Средства аппаратной поддержки ядра.
   (a) Система прерываний.
   (b) Средство для поддержки привилегированного режима.
   (c) Средство поддержки виртуальной памяти.
   (d) Смена контекстовых регистров.
   (e) Системный таймер.
   (f) Защита памяти.
3. Машинно-зависимые модули (hardware abstraction layer, HAL).
4. Базовые механизмы ядра.

   Этот слой отвечает за исполнение решений, принятых менеджерами ресурсов.
5. Менеджеры ресурсов.

   На этом слое реализованы основные алгоритмы принятия решений: модули для составления расписаний (schedulers) и модули, занимающиеся задачами размещения (allocators).
6. Интерфейс системных вызовов (API).

Стоит отметить, что первые два внутренних слоя реализованы на аппаратном уровне. Слои 2 и 3 (их имеет смысл рассматривать в паре) обеспечивают возможность установки ОС на ту или иную платформу.

Замечание 1.4.4. Если все вышеперечисленные слои относятся к ядру, то такая архитектура все еще будет называться монолитной.

Еще одним серьезным монолитного ядра является то, что при любом изменении аппаратного обеспечения (и не только его) требуется перекомпиляция ядра. Даже когда появились решения, требующие не пересборки ядра, а лишь перезапуска ОС, это проблема все осталась актуальной: например, если ОС развернута на сервере, то ее перезапуск приведет к тому, что пользователи некоторое время не будут иметь доступа к серверу (что не всегда допустимо).
Развитием монолитного ядра стало модульное ядро. Его преимуществом является то, что зачастую (но не всегда) можно добавить, удалить или заменить модуль без перезапуска ядра.
Также недостатком монолитного ядра являются проблемы с созданием распределенных решений. Пусть есть несколько физических вычислительных узлов и требуется балансировать нагрузку между ними. В случае монолитной архитектуры у каждого узла будут свои менеджеры ресурсов и, следовательно, будет очень сложно этого достичь.

II. Микроядерная архитектура

Идея заключается в том, чтобы взять многослойную архитектуру и часть внешних слоев вынеси из режима ядра (kernel mode) в пользовательский режим (user mode). В режиме ядра остается слой базовых механизмов и более внутренние слои, т.е. слои, отвечающие за непосредственное исполнение решений, а вот слои, принимающие решения, переходят в пользовательский режим. Если в многослойной архитектуре решения принимали менеджеры ресурсов, то в микроядерной архитектуре это делают серверы (суть та же, но название другое). Теперь если приложению нужно совершить системный вызов, то оно сначала обращается в ядро, ядро отдает запрос соответствующему серверу (или нескольким), возвращаясь в пользовательский режим. После выполнения запроса сервер отвечает ядру, а оно передает этот ответ приложению, которое изначально инициировало системный вызов.
Основным преимуществом является то, что мы уменьшаем объемы памяти, выделяемые для операционной системы. Другим преимуществом является удобство в построении распределенных систем. К недостаткам можно отнести количество переключений между режимами, однако это не главная проблема. Основной проблемой является надежность и безопасность: т.к. серверы находятся в пользовательском режиме, то другое пользовательское ПО может помешать их работе или перехватить какие-либо данные.

Замечание 1.4.5. Гибридное ядро это то ядро, которое можно пересобрать так, что часть функций поменяет свое расположение (перейдет из режима ядра в пользовательский режим или наоборот).

III. Наноядерная архитектура

Идея наноядерной архитектуры заключается в том, чтобы взять микроядерную архитектуру и еще больше функций вынеси из ядра. Как правило при этой архитектуре в ядре остается только обработка прерываний, но иногда там дополнительно реализуются низкоуровневые планировщики с простым алгоритмом планирования. Наноядра в основном используют в гипервизорах для систем виртуализации.

IV. Экзоядерная архитектура

Данная архитектура является развитием идеи распределенной ОС и попыткой построить гетерогенную (т.е. реализованную над разнородным оборудованием) распределенную ОС. В данной архитектуре принятие решений и межпроцессное взаимодействие остаются в режиме ядра, а взаимодействие с оборудованием разрешается напрямую.

1.5. Лекция 24.??.??.

Управление процессами

Def 1.5.1. Процесс это совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и контекста исполнения, находящихся под управлением операционной системы.

Процесс для операционной системы представлен в виде некоторой структуры данных (process control block (PCB), дескриптор процесса). Стоит отметить, что процесс это не то же самое, что и программа: одна программа может порождать несколько процессов. Это весьма логичный и понятный вариант: допустим в некотором комплексном ПО один процесс занимается обработкой GUI, а другой взаимодействием по сети. Однако обратная ситуация также возможна: несколько программ могут исполняться в рамках одного процесса. Допустим, мы запустили некоторое приложение, и оно выполнило системный вызов. Тогда часть времени в рамках одного процесса выполнялся непосредственно код приложения, а часть времени — код ядра. Т.к. ядро это отдельная программа (даже скорее несколько программ), то получается, что в рамках одного процесса выполнялось несколько программ.
Однако одного понятия процесса недостаточно для эффективного манипулирования ресурсами. Предположим, что у нас есть большая растровая картинка, которую надо каким-либо образом обработать, причем обработка каждого конкретного пикселя может происходить независимо от обработки других пикселей. В этом случае эффективно обрабатывать данную картинку параллельно и по частям, однако концепция процессов мешает этому: картинка должна быть помещена в адресное пространство только одного процесса, а другие процессы не должны иметь к ней доступ. Для решения этой проблемы появилась следующая идея: пусть внутри одного процесса будет разрешено создавать несколько наборов команд и связанных с ними контекстов.

Def 1.5.2. Пара из набора команд и связанного с ним контекста в рамках одного процесса называется потоком (thread).

Итого в рамках одного процесса может существовать несколько потоков, причем все потоки имеют доступ в общему адресному пространству процесса. Стоит отметить, что в современных ОС единицей диспетчеризации является именно поток, а не процесс. Однако в связи в этим возникают некоторые проблемы, а именно: переключение между потоками возможно только через переключение в режим ядра, т.к. диспетчеризацией потоков занимается ОС (и делает она это в режиме ядра). Причем т.к. за переключение потоков отвечает ОС, то она может делать это не так, как задумано разработчиком ПО. Это в свою очередь может негативно сказаться на производительности.
Для решения этой проблемы появился еще один уровень иерархии (ниже потоков), который называется волокно (fiber) и предоставляет пользовательскую многопоточность вне инструментария ОС. Теперь любой поток представлен в виде множества волокон.

Замечание 1.5.3. Существуют разночтения в термине fiber. Помимо «волокна» он иногда трактуется как «легковесный поток», а иногда как «green thread» (зеленый поток).

Как и в случае потока, каждое волокно содержит в себе некоторый набор команд и контекст их выполнения, но разница в том, что управление переключением между волокнами (и его планирование) берет на себя код этого потока, а не ОС. Плюсами такого подхода является решение проблем, описанных выше, а вот к минусам можно отнести то, что приходится самостоятельно реализовывать алгоритмы планирования и переключения между волокнами.
У операционной системы есть механизм прерываний по таймеру, который позволит переключать потоки, а у волокон нет такого механизма. Основным решением этой проблемы стало решение кооперативной многозадачности: его идея заключается в том, что само волокно в какой-то момент отдаст управление следующему волокну, либо волокну, являющемуся диспетчером. Также стоит отметить, что у волокон может быть поддержка на уровне ОС: не с точки зрения планирования, а точки зрения предоставления API для разработки приложений с использованием этого механизма.
Однако даже этих трех уровней иерархии оказалось недостаточно, чтобы эффективно управлять процессами в операционной системе. Представим себе работу браузера: если для каждой вкладки сделать отдельный поток, то возникнет проблема с безопасностью. Т.к. потоки в рамках процесса браузера имеют общее адресное пространство, то одна вкладка будет иметь возможность получить доступ к данным другой вкладки (причем это может быть как преднамеренно, так и вследствие некоторой ошибки). Отсюда возникает другая идея: выстроим иерархию процессов,

где будет корневой процесс, а все вкладки будут им порождены. Но тогда возникает следующая проблема: пусть мы открыли какое-то одно приложение и 99 вкладок в браузере, тогда с точки зрения операционной системы 99% процессов это процессы браузера и лишь 1% это процессы другого приложения. Однако планировщик учитывает их всех на одном уровне.
Отсюда возникает идея о том, что нужно научиться ограничивать доступ к ресурсам для некоторых групп процессов. Для этого в иерархии над процессами появляется еще один уровень. В разных операционных системах он называется по-разному: в Windows это job (задание, работа), в Linux это cgroup (контрольная группа). Суть заключается в том, что в рамках этой группы процессов можно установить определенные квоты на использование тех или иных ресурсов.

Основные функции подсистемы управления процессами

1. Создание.

   В операционной системе любой процесс порождается другим процессом, он не создается абстрактно извне. Таким образом любой процесс имеет родительский процесс. Как именно появляется первый процесс мы не будем рассматривать в рамках этого курса, лишь скажем что какое-то решение есть, и он как-то появляется. Отсюда получается, что в операционной системе есть некоторая иерархия процессов (и в разных ОС она будет отличаться), где каждый процесс представлен некоторой структурой данных. В разных ОС она будет разной, но в общем виде эта структура содержит следующие поля:

   (a) Информация по идентификации процесса. Сюда входят PID (уникальный идентификатор процесса), PPID (идентификатор родительского процесса), UID (идентификатор пользователя, запустившего процесс).

   (b) Информация по состоянию процесса (статус и контекст).

   (c) История (она сильно зависит не только от типа ОС, но и от ее версии и планировщика).

   Как же создается процесс? Сначала рассмотрим пользовательские процессы в Linux (процессы ядра рассматривать не будем). Они образуют дерево процессов, где корнем является процесс с PID = 1 и PPID = 0. Порождение новых процессов происходит методом клонирования (fork), т.е. полным копированием адресного пространства. PID для нового процесса выдается ОС, PPID определяется как PID процесса, который клонировали, а другие свойства (например UID) наследуются от родительского процесса. Таким образом ни один дочерний процесс не будет иметь больше прав, чем его родитель, что выгодно с точки зрения безопасности.

   После клонирования сегмент кода дочернего процесса заменяется на код необходимого приложения, и мы получаем полноценный новый процесс. После завершения работы дочерний процесс «отчитывается» родителю об этом, и родитель (с помощью системного вызова) может считать и обработать код его завершения. Если же родительский процесс внезапно нештатно завершается (а его дочерние процессы продолжают работать), то новым родителем для «осиротевших» процессов становится корневой процесс, т.к. дерево процессов должно оставаться связным. Есть так же и другая интересная ситуация: если родительский продолжает работать, но не может корректно обработать код завершения дочернего процесса, то такой дочерний процесс становится «зомби»-процессом. Он вроде бы и завершил свою работу, но все равно остается в дереве процессов. Подробнее об этом будет рассказано в следующей лекции.

   В Windows работает другой механизм. Корневым процессом является диспетчер процессов, который несет ответственность за создание всех новых процессов, т.е. если какому-либо из процессов потребовалось создать потомка, то он через системный вызов обращается к диспетчеру процессов и просит создать новый процесс. Таким образом нет никакого дерева процессов: все процессы как бы являются потомками диспетчера процессов. Плюсом такого решения является централизованный тотальный контроль за появлением процессов. С другой стороны права дочернего процесса определяются не родительским процессом, а диспетчером процессов, поэтому формально можно создать процесс с правами выше, чем у родительского процесса. Для предотвращения этого у ОС есть механизмы защиты, но потенциальная возможность их обойти все равно остается.

2. Обеспечение ресурсами.

   Любой процесс с самого начала уже обеспечен некоторыми ресурсами: например, ему выделено адресное пространство. Такие ресурсы называются статическими и будут оставаться с процессом до его завершения. Однако есть еще и динамические ресурсы: например, в процессе своего выполнения процесс может потребовать дополнительную память или доступ к файлу. Даже процессорное время тоже можно считать динамическим ресурсом. Итого задача обеспечения ресурсами сводится к задачам планирования ресурсов (задача о расписании) и аллокации (задача о рюкзаке).

3. Изоляция.

   В последующих лекциях про память будет обсуждаться вопрос об обеспечении изоляции памяти. Также в последующих лекциях при обсуждении синхронизации частично будет затронута тема изоляции.

4. Планирование.

   Планированию далее будет посвящена отдельная лекция, т.к. это весьма многоуровневый процесс: необходимо планировать процессорное время, очереди на ввод-вывод, рождаемость процессов и т.д.

5. Диспетчеризация (переключение процесса между различными состояниями).

   Простейшая диспетчеризация заключается в том, что процесс переключается между состояниями «работает» и «ждет». Смена состояний происходит в три шага:

   (a) Сохранение контекста текущего процесса.

   (b) Загрузка контекст другого процесса.

   (c) Смена состояний этих двух процессов.

   Стоит отметить, что последний шаг должен быть атомарным, в противном случае мы получаем неуправляемое состояние системы: если произойдет прерывание, то у нас одновременно будет существовать либо два работающих процесса, либо вообще ни одного. Для удовлетворения этого требования существуют разные хитрые механизмы, которые будут рассмотрены позднее. Понятно, что у процесса может быть больше двух состояний: об этих состояниях и о переходах между ними мы поговорим на следующей лекции.

6. Взаимодействие.

   В рамках этого курса будем касаться этой темы достаточно слабо, т.к. взаимодействие процессов существенно отличается в разных операционных системах.

7. Синхронизация.

   Этой теме также будет посвящена отдельная лекция (даже чуть больше), в рамках которой мы поговорим про семафоры, мьютексы, тупики и т.п.

8. Уничтожение.

   Уничтожение процесса это не такой простой процесс, как кажется на первый взгляд: процесс мог владеть определенными ресурсами, которые возможно нужно корректно «завершить». Например, процесс что-то записал в файл, но т.к. современные файловые системы чаще всего используют механизм отложенной записи, то ОС нужно принудительно инициировать запись на диск, ведь после уничтожения процесса данные, которые он хотел записать, станут недоступны. Также может быть такое, что процесс владел некоторыми неразделяемыми ресурсами и блокировал доступ к этим ресурсам для других процессов — после уничтожения эти блокировки нужно снять.

   Помимо этого необходимо завершить все дочерние процессы: проверить, что они корректно завершились, и в противном случае каким-либо образом это обработать. Еще нужно отчитаться родительскому процессу о своем завершении и предоставить код завершения.