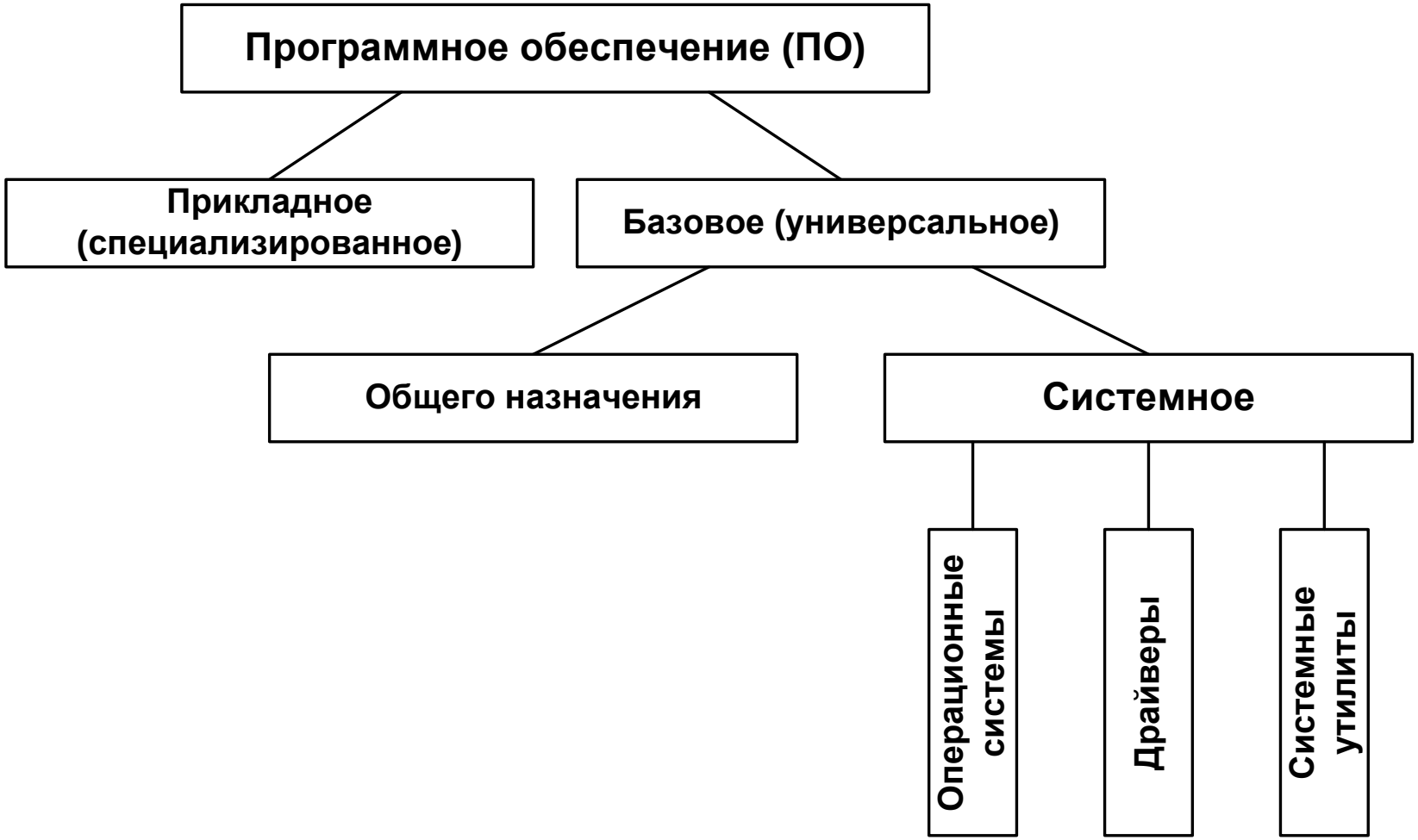
1 Основные понятия и определения

1.1 Общие сведения о системном ПО

Вычислительная система – совокупность программных и аппаратных средств, которые функционируют совместно и согласованно для решения поставленных задач.

В нашем контексте – ***программное*** и ***аппаратное*** (***техническое***) ***обеспечение***, ***ПО*** и ***АО***.



Виды программного обеспечения

Уровни программного обеспечения

Системное ПО – выполняет общие (инвариантные) для всех прикладных задач функции по управлению вычислительной системой.
Системное ПО не участвует непосредственно в решении задач пользователя, но обеспечивает работу прикладных программ.

Операционная система (***ОС***) – занимает центральную часть в системном ПО, обеспечивая согласованное функционирование программных и аппаратных средств вычислительной системы. В ОС концентрируется большинство системных функций.

Драйверы – системные программы, предназначенные для управления отдельными внешними устройствами. Обычно драйвер создает логическое устройство, с которым могут взаимодействовать программы. Использование драйверов обеспечивает гибкость системы и независимость от особенностей реализации аппаратных средств.

Системные утилиты – выполняемые обычным образом программы, решающие не прикладные, а общесистемные задачи: сервисные программы (обслуживание дисков, администрирование), средства программирования (трансляторы, отладчики), средства обеспечения безопасности, и т.д.

1.2 Принципы построения и функционирования операционных систем

Операционная система:

– расширение машины, унифицированная надстройка, виртуальная машина

– менеджер ресурсов вычислительной системы

В ОС концентрируется большинство системных функций:

– управление процессами;

– управление ресурсами;

– управление вводом-выводом (в том числе файлами и файловой системой);

– обеспечение интерфейса с пользователем;

– общее управление и синхронизация.

Предпосылки появления операционных систем:

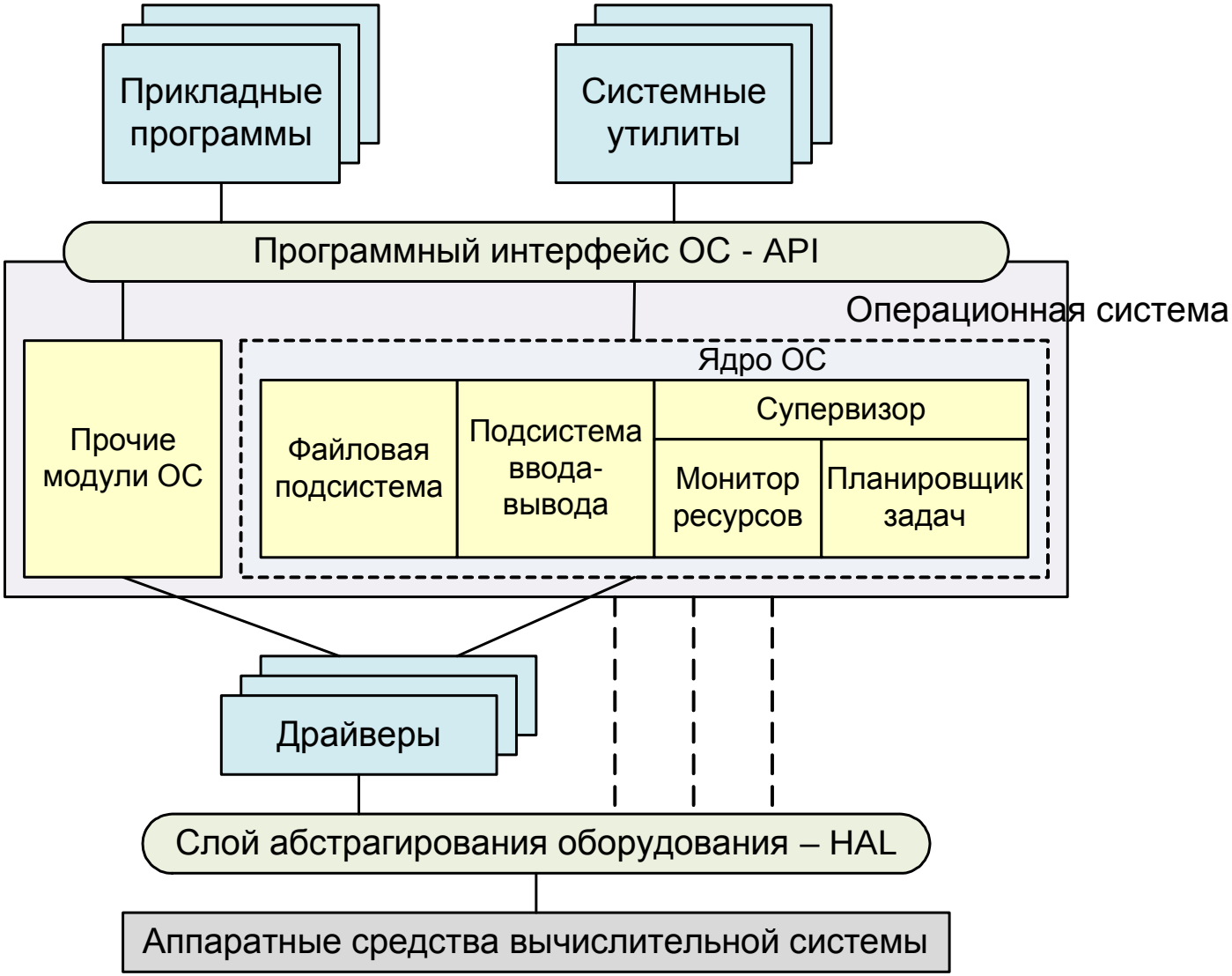
Постепенное удешевление и распространение вычислительной техники, увеличение ее разнообразия

Потребность в удешевлении системного программного обеспечения за счет повторного его использования на различных аппаратных платформах

Потребность в удешевлении прикладного и системного программного обеспечения за счет унификации и переносимости

Поколения операционных систем в целом соответствуют поколениям вычислительной техники.

Общая структура операционной системы вытекает из ее функций:



Упрощенная структура операционной системы

API – ***Application Programming Interface*** – интерфейс прикладного программирования, фактически программный интерфейс системы, подсистемы, компонента.

HAL – ***Hardware Abstraction Layer*** – слой абстрагирования оборудования

Kernel – ядро операционной системы, содержит основные ее подсистемы:

– подсистема файлов

– подсистема ввода-вывода

– монитор ресурсов (память и др.) – распределение ресурсов вычислительной системы процессам-пользователям

– планировщик (диспетчер) задач – распределение процессорного времени

В идеальном случае, от аппаратных средств зависимы только HAL и драйверы, остальная часть ядра может быть аппаратно-независимой. На практике часть ядра обычно остается аппаратно-зависимой, но эту часть стараются минимизировать.

Ограничение аппаратно-зависимой части ПО и выделение ее обособленные модули – путь к обеспечению **переносимости** ПО. То же справедливо и в отношении независимости от **программной** платформы.

Подходы к реализации операционной системы:

Пассивная ОС – функции ОС встроены в язык программирования (как следствие, также в прикладные программы).

Пример: ROM Basic IBM PC.

Монитор – набор завершенных и готовых к исполнению системных программ (подпрограмм) и данных, разделяемых и выполняемых другими (прикладными) программами. Технологически это можно представить как библиотеку с документированными точками входа, доступную всем программам в системе. Исполнение системного кода инициируется только по запросам приложений.

Абстрактная машина – код системы написан в расчете на определенную **программную модель** вычислительной системы. Система зависима от программной модели, но будет выполняться на всех машинах, реализующих эту программную модель.

Пример: MS DOS.

Виртуальная машина – отдельная аппаратно-зависимая часть системы создает среду («виртуальную машину») для выполнения остальных программ. Для переноса системы достаточно реализовать аналогичную виртуальную машину.

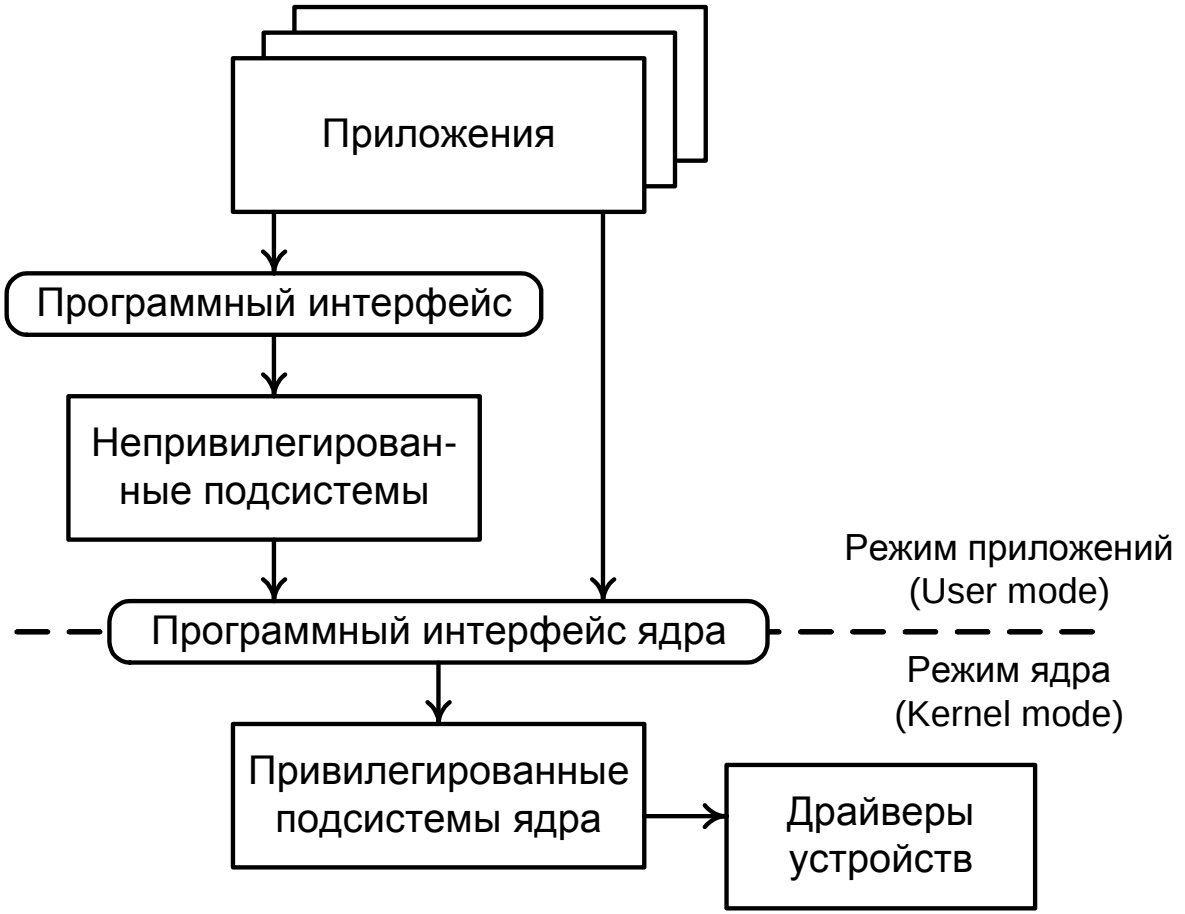
Уровни выполнения кода

Одноуровневые ОС – код всех программ всегда выполняется на одном и том же уровне привилегий (простые однозадачные или специализированные системы)

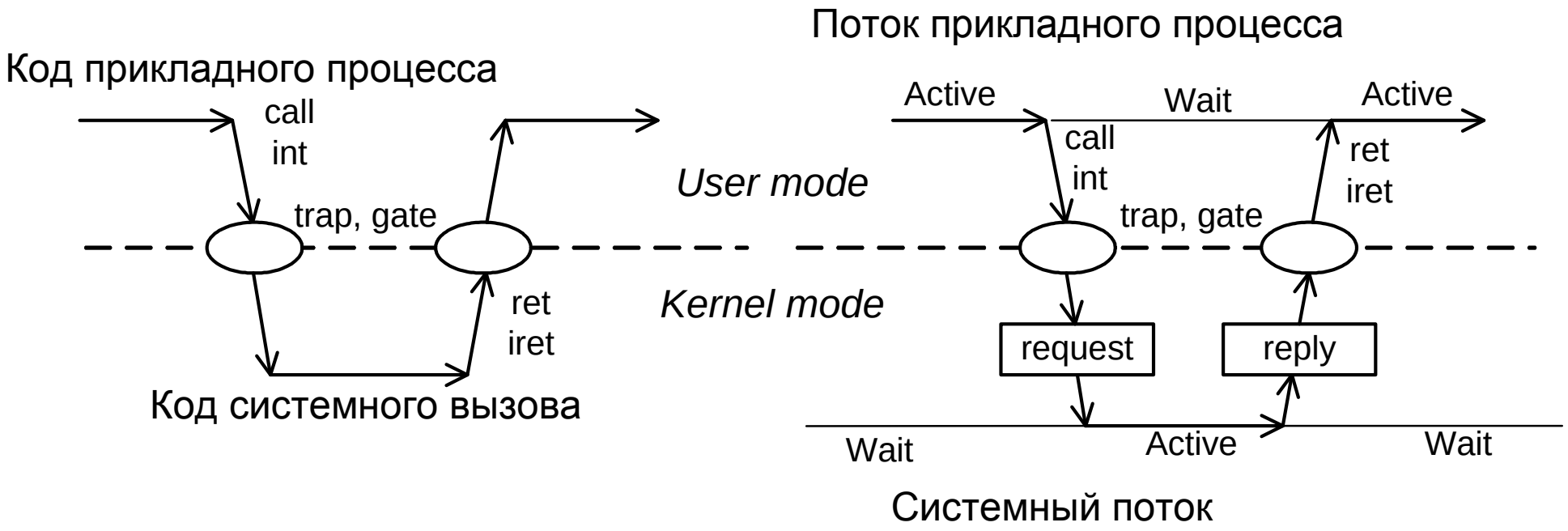
Двухуровневые ОС – выделяются 2 уровня (режима) выполнения:

– Режим ***приложения*** (***User Mode***) – ограниченный в правах

– Режим ***ядра*** (***Kernel Mode***) – привилегированный



Уровни выполнения программ



а) переключение контекстов б) переключение потоков

Переключение уровня выполнения кода

Для переключения уровня выполнения кода системные вызовы оформляются как «ловушки», обработка которых приводит к переключению уровня привилегий кода, и код системного вызова выполняется уже на привилегированном уровне (временная смена контекста текущего выполняющегося процесса).

В более сложном случае формируется сообщение с запросом, описывающим требуемую функцию. Сообщение передается на выполнение одному из системных потоков, который возвращает результат запроса. Прикладной поток на время работы системного находится в состоянии ожидания либо продолжает работать параллельно. Системный поток в отсутствие запросов остается в состоянии ожидания.

Более сложная иерархия уровней выполнения задач:

– прикладные программы

– исполнительная подсистема

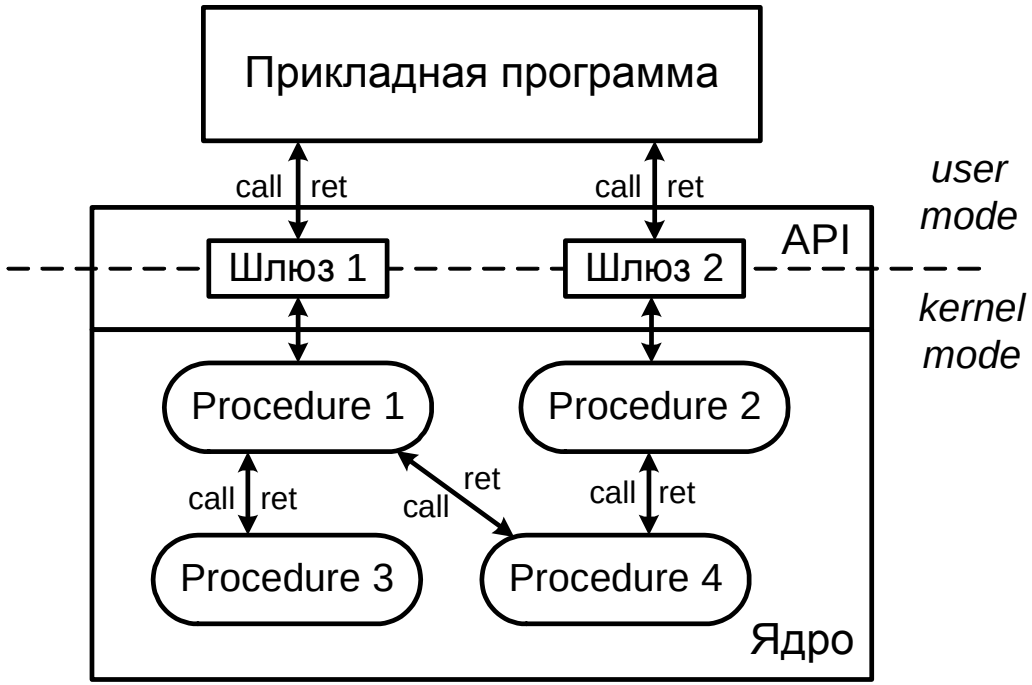
– супервизор

– ядро

1.3 Архитектуры ядра операционных систем

1.3.1 Монолитное ядро

Представляет собой единую программу, состоящую из набора подпрограмм, служащих обработчиками системных вызовов и аппаратных событий.
Всё ядро целиком функционирует на едином уровне привилегий, и каждая его подпрограмма может (потенциально) вызвать любую другую.



Монолитное ядро

Преимущества: простота и естественность реализации, минимальные затраты на выполнение вызовов.

Недостатки: сложность проектирования, большая вероятность конфликтов и побочных эффектов внутри ядра, требует полной пересборки.

1.3.2 Модульное ядро

Отличается от монолитного лишь структурированием – делением на отдельные модули, которые объединяются посредством выделенных и документированных интерфейсов. Обычно в модуль объединяются группы подпрограмм, соответствующие тем или иным подсистемам: диспетчер системных функций, обслуживание API, сервисные процедуры и т.д.

Структурирование способствует гибкости и надежности, возможна частичная замена модулей, частичная пересборка.

«Настоящее» монолитное ядро фактически не встречается.

С распространением микроядерной архитектуры монолитное или модульное ядро стали называть ***макроядром***.

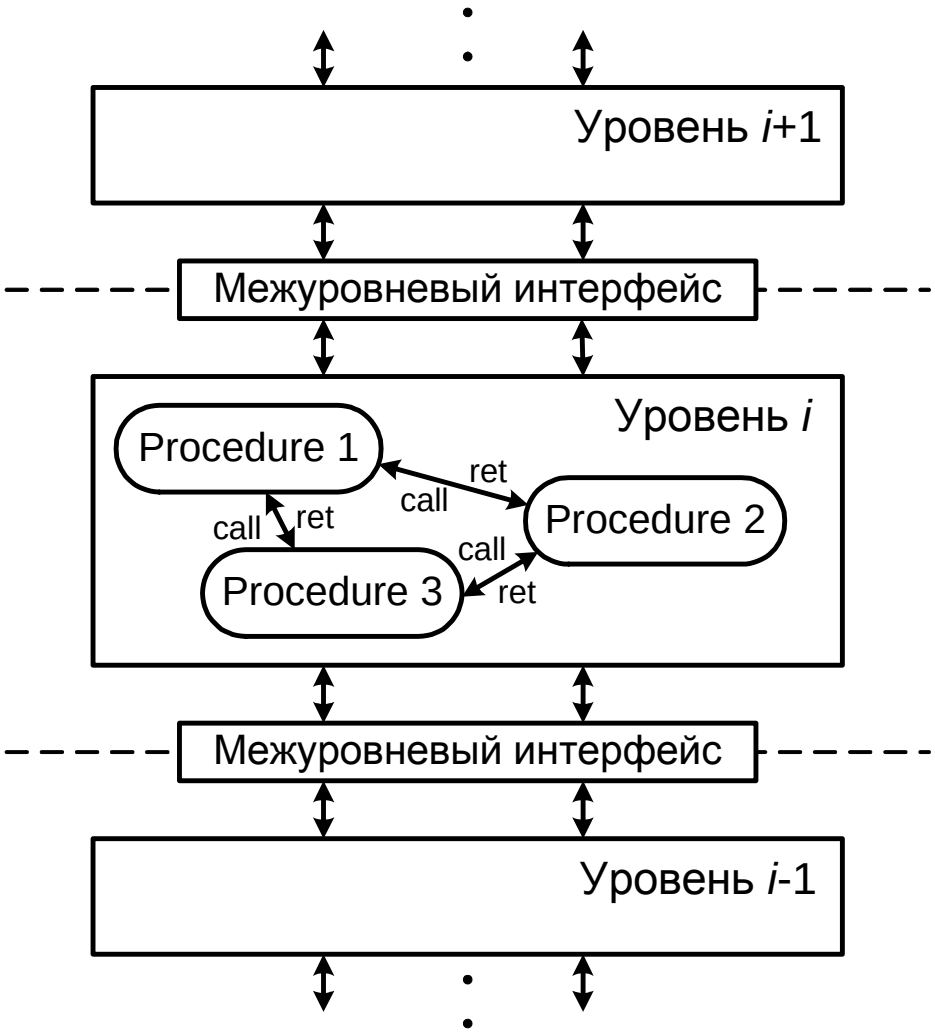
Пример: классическая архитектура Unix.

1.3.3 Многоуровневое (layered) ядро

Упорядоченный систематический подход к структурированию: модули встраиваются в строгую иерархию. Взаимодействие между модулями происходит только через заранее определенные унифицированные интерфейсы, количество точек взаимодействия минимизировано (в идеале только «вход» и «выход» модуля). Функционирование модулей максимально независимо, в т.ч. и за счет передачи данных между ними только ***по значению***, т.е. копированием (требование приводит к существенному росту затрат, поэтому часто не выполняется).

Преимущества: модульность, широкие возможности замены модулей

Недостатки: логическая сложность проектирования, дополнительные затраты на передачу управления между уровнями



Многоуровневая система (показан один уровень и его соседи)

Многоуровневая архитектура не нашла широкого применения для ОС, однако эта концепция оказалась продуктивной для специализированного ПО, например коммуникационного (поддержка протоколов обмена между системами и подсистемами).

Примеры:

– Проект ОС THE (Technishe Hogeschool Eindhoven) – Дейкстра, 1968

– Модель взаимодействия открытых систем (OSI) – 7 уровней

– Стек TCP/IP – 4 уровня

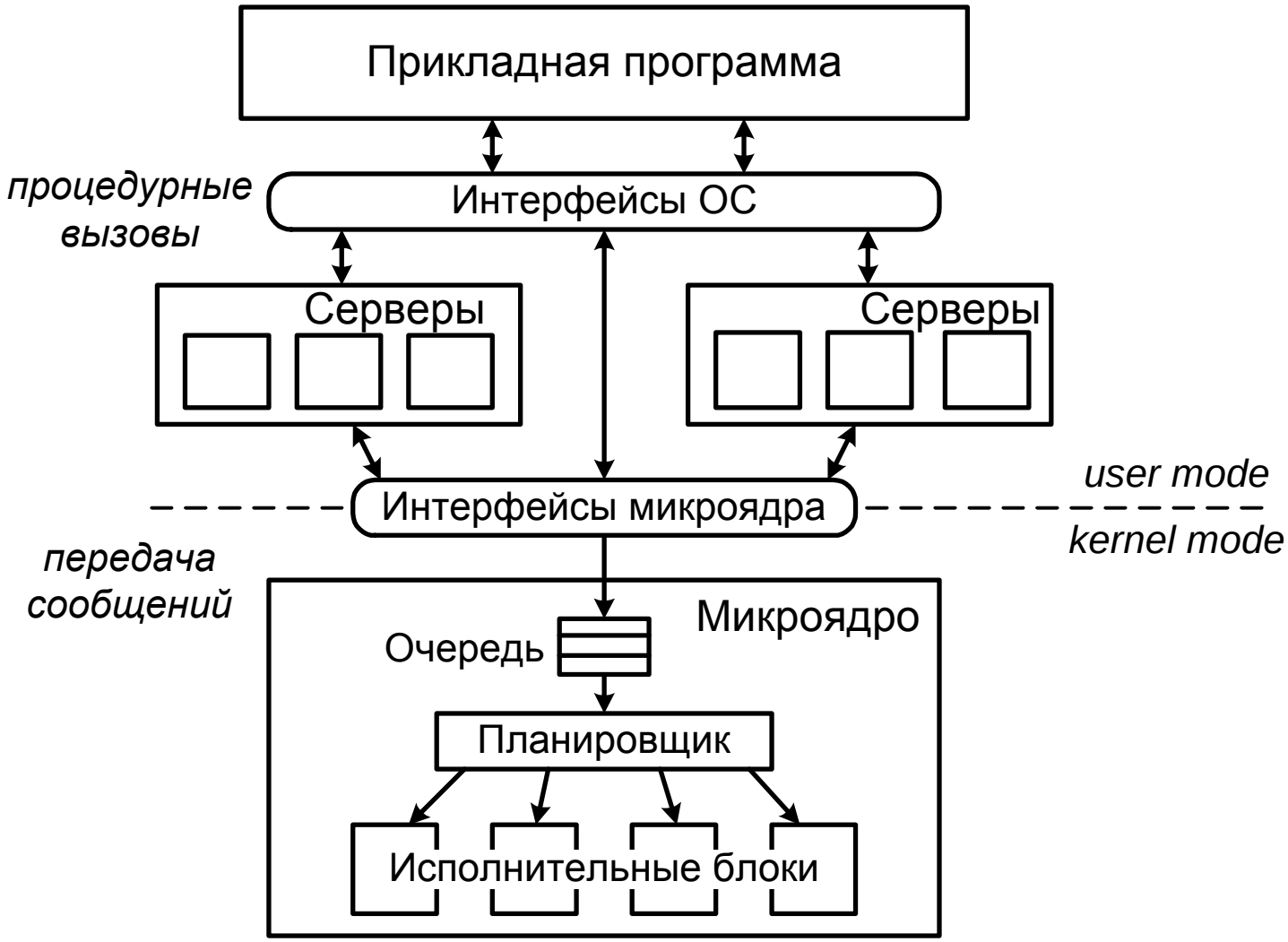
Стремление упростить построение, уменьшая число уровней, приводит к архитектуре ***микроядра***.

1.3.4 Архитектура микроядра

Микроядерная архитектура – вариант многоуровневой с всего двумя уровнями, функции которых разделены изначально: собственно микроядро и сервисы (серверы)

Микроядро – специальный модуль ядра, реализующий наиболее важные, аппаратно-зависимые функции ОС: управление процессами и их взаимодействие, управление ресурсами, включая память, ввод-вывод, первичная обработка прерываний. Код микроядра выполняется только в привилегированном режиме.

Сервисы – системные процессы, выполняющие все остальные функции ОС: работа с файловой системой, логическими устройствами, другими системными объектами. Сервисы выполняются на пользовательском уровне привилегий и в адресном пространстве прикладных задач.



Микроядерная архитектура

Различные подходы к распределению функций между микроядром и сервисами приводит к противоположным тенденциям: «тяжелое» и «легкое» микроядро («наноядро»).

Использование очереди сообщений (запросов) в качестве API позволяет изолировать уровни системы, избавиться от блокировок, решить проблему нереентерабельности системного кода.

Преимущества микроядерной архитектуры: надежность, минимизация аппаратно-зависимого кода (облегчает переносимость)

Недостатки: дополнительные затраты на взаимодействие между уровнями.

Примеры:

– RT-11, OS-9, RSX-11, VMS – ранние реализации

– BSD Unix

– Mac OS X

– QNX

– Windows NT («модифицированная микроядерная архитектура»)

1.3.5 Архитектура виртуальной машины.

Доведенная до логического завершения идея изоляции пользовательского уровня от деталей вычислительной системы: все программы взаимодействуют только с **виртуальной машиной** – функциональным эквивалентом реальной, но реализуемой различными аппаратными и программными средствами, а точнее – выполняются ею. Соответственно, зависимой от аппаратной платформы будет только сама виртуальная машина, прочее же ПО, включая наиболее сложные высокоуровневые алгоритмы ОС, может и должно быть написано переносимым, что существенно упрощает унификацию ПО, создание программно-совместимых семейств ЭВМ.

Примеры:

– CP/CMS (VM/370) – первая ОС на основе виртуальной машины, для IBM 370.

– VMWare

– HAL в архитектуре ОС, в т.ч. Win NT

Своего рода развитием концепции виртуальной машины как архитектуры операционной системы можно считать «браузерные» операционные системы: Web OS, Chrome OS, Firefox OS и т.п., а также Android. В свою очередь, их же можно рассматривать как комбинированные архитектуры, т.к. «виртуальная машина» работает поверх ядра иной ОС (чаще Unix/Linux-подобной).

1.3.6 Нетрадиционные архитектуры

«Экзоядро» – функционал ядра ограничивается средствами обеспечения взаимодействия процессов и безопасного выделения ресурсов, прочие функции реализованы в библиотеках пользовательского уровня (libOS) и выполняются самими процессами.

Достоинства: потенциально большая эффективность при обращении к ресурсам, изоляция процессов, минимизация кода ядра.

Недостатки: логические сложности проектирования, жесткие требования к реализации ядра и процессов.

Примеры:

– Arrakis

1.3.7 Смешанные архитектуры. «Браузерные» ОС

Примеры:

– Chrome OS

– Android

– «Модифицированное» микроядро Win NT

1.4 Некоторые базовые концепции и сущности

1.4.1 Вычислительные процессы и потоки, многозадачность

Вычислительный процесс – выполняющаяся программа вместе с адресным пространством, пространством дескрипторов и контекстом процесса.

Вычислительный процесс, ***Process*** – выполняющаяся программа вместе с выделенными ей ресурсами: адресным пространством и созданными объектами (пространством дескрипторов). Можно считать, что процесс представлен его ***образом*** и ***контекстом***, в котором выполняется содержащийся в образе код процесса.

Образ процесса – код и данные процесса:

– образ во внешней памяти (исполняемый файл на диске)
– образ в оперативной памяти (загруженная программа)

Контекст процесса:

– содержимое регистров
– состояние стека процесса
– информация о процессе в таблицах ОС

Состояния процесса:

– активность
– готовность
– ожидание (блокировка)

Роль процесса (концептуально):

– процесс – обладатель ресурсов
– процесс – контейнер потоков

Вычислительный поток (***программный поток***), ***Thread*** – объект, связанный с выполнением программы процессором. Поток выполняется параллельно с другими потоками всех процессов и делит адресное пространство с другими потоками своего процесса.

Потоки более экономичны по сравнению с процессами и имеют возможность более эффективного и менее затратного взаимодействия друг с другом (в рамках одного процесса). Благодаря этому концепция потоков поддерживается всеми основными современными «универсальными» ОС. Однако реализация может идти различными путями:

– MS Windows (Win 32 и Win 64) – изначально предусмотренный объект ядра

– Unix и Unix-подобные системы – как правило, «облегченный» процесс

В многопоточной системе планировщик управляет именно потоками, поэтому представление о состояниях переносится с процессов на потоки.

1.4.2 Виртуальная память, виртуальное адресное пространство

Абстракция памяти системы, позволяющая решить задачи:

– расширение адресного пространства по сравнению с доступной физической оперативной памятью за счет использования внешней памяти

– более эффективные гибкие алгоритмы управления памятью, в т.ч. упрощение построения адресного пространства процесса

– эффективное разделение (изоляция) адресных пространств процессов

Фактически в системе происходит взаимное отображение физического и виртуального адресных пространств. Это требует аппаратной поддержки центральным процессором и, возможно, дополнительными внешними схемами.

1.4.3 Прерывания, события, событийное управление

Прерывания (***Interrupt***) или ***исключения*** (***exception***) – механизм, обеспечивающий оперативное реагирование на события в системе или в периферийных устройствах путем нарушения естественного порядка выполнения программ с последующим возвратом в точку прерывания. Таким образом, реализуется выход за рамки однозадачности даже в однозадачных системах.

Требуется аппаратная поддержка процессором и, возможно, внешними схемами.

Обработка прерывания предполагает использование специальных структур:

– таблица векторов (дескрипторов) прерываний
– обработчики прерываний – программы, выполняемые на фоне других программ.

Событие – программная надстройка над прерыванием (либо особым состоянием алгоритма). Поскольку события являются программными объектами, использование их не требует специальных привилегий.

Примеры:
– оконные сообщения Window Message (MS Windows)
– сигналы (Unix)

Событийное управление – подход к организации логики программы, когда порядок выполнения ветвей алгоритма определяется внешними либо внутренними событиями. Концепция расширяется на случай управления алгоритмом посредством событий, а не явной передачи управления. Характерно для управляющих систем, хорошо соответствует особенностям функционирования пользовательских интерфейсов и крупных систем.