**Надёжность и безопасность операционных систем**

**различной архитектуры**

*Назаров Станислав Викторович, Московский научно-исследовательский телевизионный институт, mniti.ru, Москва, 105094, Гольяновская ул., 7А, стр. 1*

**Аннотация.** Введение широкого пакета санкций привело российскую экономику к переориентации на использование отечественных программных продуктов, в том числе и операционных систем. Очевидным преимуществом [отечественных](https://www.cnews.ru/book/%D0%98%D0%BC%D0%BF%D0%BE%D1%80%D1%82%D0%BE%D0%B7%D0%B0%D0%BC%D0%B5%D1%89%D0%B5%D0%BD%D0%B8%D0%B5_-_%D0%98%D0%BC%D0%BF%D0%BE%D1%80%D1%82%D0%BE%D0%BD%D0%B5%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%8C_-_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%86%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%BE%D0%B9_%D1%81%D1%83%D0%B2%D0%B5%D1%80%D0%B5%D0%BD%D0%B8%D1%82%D0%B5%D1%82_-_%D1%80%D0%B5%D0%B6%D0%B8%D0%BC_%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BD%D1%8B%D1%85_%D0%B7%D0%B0%D0%BA%D1%83%D0%BF%D0%BE%D0%BA_%D0%BE%D1%82%D0%B5%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D1%85_%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B9_%D0%B8_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D0%B9) ОС являете то, что они разрабатываются согласно требованиям отечественных регуляторов по безопасности программных продуктов и потенциально могут обеспечить независимость от зарубежного ПО в целом. По итогам опроса Министерство цифрового развития, связи и массовых коммуникаций РФ определило три наиболее перспективные для господдержки отечественные операционные системы (ОС), включенные в реестр российского программного обеспечения: Astra Linux (ГК «Астра»), ОС «Альт» («БазАльт СПО») и «Ред ОС» («Ред Софт»). Все они базируются на Linux. Главный недостаток Linux в том, что после взлома монолитного ядра злоумышленник получит максимальные привилегии. Этот факт приводит к сомнениям – хороша ли архитектура ОС с монолитным ядром? В каждом ли ее применении будет обеспечена требуемая надежность и безопасность функционирования системы? В связи этим встает вопрос, насколько надежна и безопасна ОС с монолитным ядром по сравнению с системами другой архитектуры? В данной статье предлагаются простые модели работы ОС различной архитектуры, основанные на теории марковских процессов, описываемых системами дифференциальных уравнений Колмогорова. Разрабатываются и исследуются три модели систем: операционная система с классическим монолитным ядром (Linux), микроядерная операционная система Э. Таненбаума (Minix3) и кибериммунная операционная система KasperskyOS. Оцениваются и сравниваются надежность и безопасность функционирования систем. Предложенный подход к построению моделей архитектур операционных систем может быть использован для начального исследования надежности функционирования конкретных архитектур операционных систем.

**Ключевые слова:** надежность, безопасность, операционная система, архитектура, ядро, микроядро, модель, исследование.

**Abstract.** The introduction of a wide package of sanctions led the Russian economy to reorient itself to the use of domestic software products, including operating systems. The obvious advantage [of domestic](https://e.mail.ru/BOZm9ooI36KbvgA%3D%26egid=j8sJN%2FwTA3ekmqFM3Lflh9D3L7iI5UkKivMj5qUnDTE%3D%26url=https%3A%2F%2Fclick%D0) OS is that they are developed in accordance with the requirements of domestic software safety regulators and can potentially ensure independence from foreign software as a whole. According to the results of the survey, the Ministry of Digital Development, Communications and Mass Media of the Russian Federation identified the three most promising domestic operating systems (OS) for state support included in the register of Russian software: Astra Linux (Astra Group of Companies), Alt OS (BazAlt SPO) and Red OS (Red Soft). All of them are based on Linux. This is a universal system with a monolithic kernel that adapts well to various applications for servers, desktops, mobile devices. Nevertheless, there are so many system calls in it, and the modules inside the kernel have such a complex chain of connections that you cannot be sure of the reliability of checking all interactions with the security tools built into the kernel. The main drawback of Linux is that after hacking a monolithic kernel, an attacker will receive maximum privileges. This fact leads to doubts - is the OS architecture good with a monolithic core? Will each application ensure the required reliability and safety of the system? In this regard, the question arises, how reliable and safe is an OS with a monolithic core compared to systems of a different architecture? This article proposes simple models of operating systems of various architectures based on the theory of Markov processes described by systems of Kolmogorov differential equations. Three models of systems are being developed and investigated: an operating system with a classic monolithic kernel (Linux), a microkernel operating system by E. Tanenbaum (Minix3) and a cyberimmune operating system KasperskyOS. Reliability and safety of systems operation are evaluated and compared. The proposed approach to building models of operating system architectures can be used for an initial study of the reliability of the operation of specific operating system architectures.

**Keywords**: reliability, security, operating system, architecture, core, microkernel, model, research.

1. **Введение. Свойства, качество и эффективность ОС**

В рамках исследования данной статьи представляется необходимым остановиться на трех основных свойствах операционных систем: надежность, безопасность, эффективность. Под надёжностью операционной системы понимается ее свойство сохранять во времени в установленных пределах значения всех параметров, характеризующих способность системы выполнять требуемые функции в заданных режимах и условиях применения (ГОСТ 27.02 - 2015). Надёжность – комплексное свойство, которое в зависимости от назначения системы и условий ее эксплуатации включает в себя свойства безотказности, долговечности, ремонтопригодности и сохраняемости, а также определённое сочетание этих свойств. В практике применения с надежностью ОС тесно связано другое ее свойство – безопасность. Это состояние защищённости информационной среды системы, которое достигается наличием в структуре ОС определенных средств защита информации или специальными архитектурными решениями, принятыми разработчиками операционной системы. Немаловажным фактором обеспечения безопасности является деятельность по предотвращению утечки защищаемой информации, и защита от несанкционированных и непреднамеренных воздействий. Под эффективностью ОС и эффективностью вообще любой системы понимается степень соответствия системы своему назначению, ее техническое совершенство и экономическая целесообразность [7-10]. На показатели эффективности ОС влияет много различных факторов, среди которых основными являются архитектура ОС, многообразие ее функций, качество программного кода, аппаратная платформа (компьютер) и др. Эти вопросы рассмотрены ниже. В работах автора [7,12,13,17] достаточно подробно рассмотрены вопросы формализации понятия эффективность функционирования применительно к сложным техническим (программно-аппаратным) системам. Определены понятия: параметры системы, свойства, качество системы, показатель эффективности, критерий эффективности. Показана их связь и зависимости. Применительно к рассматриваемой в данной работе задаче эта связь и зависимость может быть представлена как показано на рис. 1.

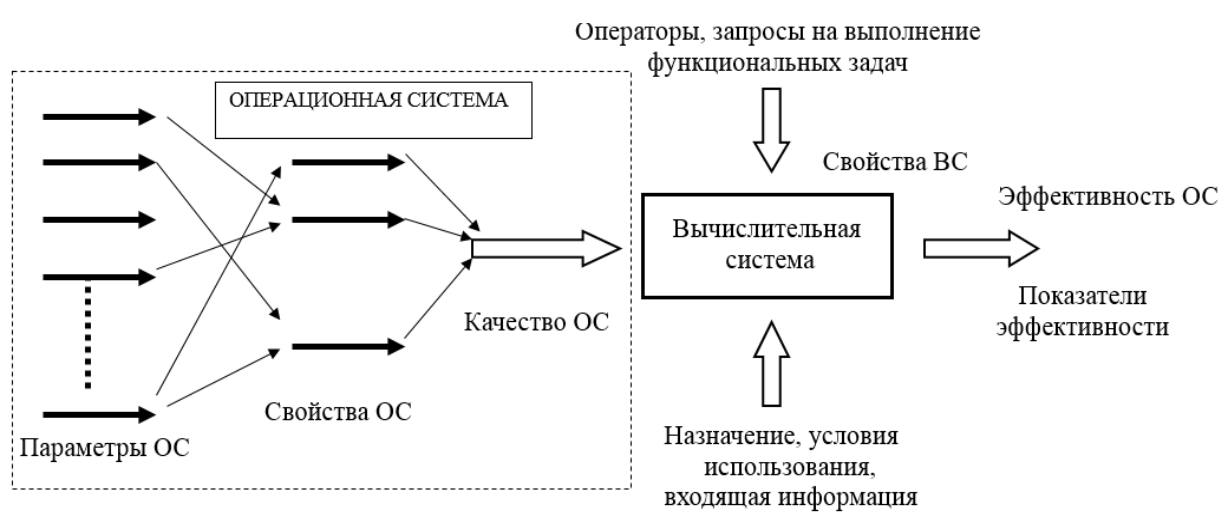


Рис. 1. Связь между параметрами, свойствами и показателями эффективности ОС

Качество операционной системы представляется совокупностью свойств, характеризующих ее использование по конкретному назначению. В то же время эффективность, в соответствии с положениями современной теории эффективности, свойством системы не является. Это обусловлено следующими причинами:

• оценка эффективности (уровня эффективности) связана не только со свойствами операционной системы, но и (даже, возможно, в большей степени) со свойствами результатов ее функционирования и ресурсами, затраченными на достижение этих результатов;

• эффективность функционирования операционной системы определяется не только ее свойствами, но и способом использования системы по целевому назначению (например, режимом работы, входящими потоками данных, характеристиками и требованиями функциональных задач к ресурсам и т.п.);

• рассмотрение эффективности как свойства операционной системы вступает в противоречие с понятием ее качества.

Таким образом, понятие эффективность операционной системы следует понимать, как эффективность ее функционирования, т.е. свойства ее использования по назначению (т.е. свойство операции). Эффективность ОС зависит от ее качества, которое в свою очередь определяется свойствами системы, условиями использования системы, а также свойствами надсистемы, т.е. вычислительной системы в целом.

**Причины сложности, ненадежности и небезопасности ОС**

С развитием теории и практики создания операционных систем сложность и объем кода операционных систем только возрастают, чего нельзя сказать о надежности, качестве и безопасности операционных систем. Причины такого положения дел обусловлены как объективными, так и субъективными факторами. Главный объективный фактор – сложность и трудоемкость разработки, желание учесть максимально возможный перечень требований пользователей. Субъективные факторы в основном связаны с конкуренцией на рынке ОС. Основные причины сложности создания надежных безопасных и эффективных операционных систем, по мнению специалистов, достаточно обозначились на практике [5, 6, 11, 14, 15-17]. Прежде всего надо отметить, что операционные системы – это класс больших программных систем, который характеризуются широкой функциональностью, высокой трудоемкостью и сложностью, рисками и длительностью процесса разработки. Например, создание OS/360 в шестидесятые годы прошлого век потребовало 5000 человеко-лет. Характерная черта больших программных систем – высокая сложность и большой объем программного кода, например, ОС Windows 2000 содержит примерно 30 миллионов строк кода, а следующие версии намного больше. Понятно, что ни один разработчик не может даже надеяться понять целиком эту систему. Следствие гигантского объема – большое количество программных ошибок, которые выявляются долгие годы.

Инерция и желание сохранить “обратную” совместимость требует наличия в новой версии системы интерфейса и возможностей, присутствующих в старой версии, в результате чего ранее разработанные программы и пользователи могут работать с новой версией без переделки (или переучивания). Это усложняет и удорожает систему, а желание сделать систему “более дружественной по отношению к пользователю”, т.е. такой, которая не предъявляет к пользователю особых требований в части знания компьютера и программирования, как правило, приводит к снижению производительности компьютера, а отсюда необходимо принятие разработчикам специальных мер, компенсирующих этот фактор, как, например, реализация графического интерфейса Windows в режиме ядра ОС.

Негативный эффект привносит стратегия производителей, направленная на повышение продаж. Разработчики зачастую не имеют четкого представления, как будет использоваться их система. Это приводит к тому, что они вынуждены увеличивать ее универсальность, добавляя множество разнообразных функций. К этому надо добавить агрессивную рыночную и рекламную политику производителей-конкурентов, периодически объявляющих о разработке новых более совершенных версий программного продукта. В такой ситуации компании-производители снимают поддержку и сопровождение популярных распространенных версий систем, чтобы вынудить потребителей системы перейти на новую версию. При этом необходимо обеспечить максимальную совместимость с предыдущими версиями.

Сложность системы возрастает с течением времени и в связи с необходимым усложнением организации мультипрограммного вычислительного процесса, что связано с прогрессом микроэлектроники и технологии построения вычислительных устройств и с желанием их максимального использования. Отсюда возрастает сложность управления большим числом параллельных заданий, процессов, потоков, волокон в сложных аппаратных многопроцессорных и многоядерных структурах. Большой проблемой является и достижение требуемой надежности и безопасной работы операционной системы. Особенность программных систем в отличие от аппаратуры вытекает из факта их неизменности по мере наработки. Если какие-то изменения происходят, особенно на начальных этапах работы, то это связано с установкой «заплаток» и сервисных пакетов. Однако в целом изменения в ОС происходят только в результате устранения программных ошибок. При разработке системы должен быть предусмотрен учет возможных непреднамеренных ошибок легальных пользователей или их незаконных действий, направленных на получение каких-либо преимуществ и даже хищений. Нужно учитывать также действия потенциально враждебных пользователей, желающих вмешаться в работу системы, выполняя незаконные действия.

Дополнительный вклад в возможные причины ненадёжности и безопасности операционной системы вносит необходимость обеспечения информационной и программной совместимости с другими системами, требования переносимости на другие платформы и поддержки широкой номенклатуры внешних устройств, которые проектируются независимо друг от друга и часто независимо от какой-либо ОС. Развитие технологии виртуализации несколько сглаживает эту проблему, но возникает другая – разработка эффективных гипервизоров. Современные операционные системы – долгожители (первая версия Unix живет с 1970 года и поныне, Windows – c 1985 года). Проектировщики и разработчики ОС должны представлять, как могут измениться компьютеры и приложения в будущем и как к этому подготовиться. Понятно, что только учет всех перечисленных факторов позволит создавать эффективные операционные системы, однако существенно осложняет решение вопросов надежности и безопасности операционных систем.

Основная причина аварийных отказов операционных систем кроется в двух принципиальных дефектах разработки, свойственных всем этим системам: наличие слишком большого числа привилегий и отсутствие адекватной изоляции сбоев. Практически все операционные системы состоят из многочисленных модулей, скомпонованных в одном адресном пространстве и образующих единую бинарную программу, которая выполняется в режиме ядра. Ошибка в любом модуле может легко привести к разрушению структур данных в каком-либо другом, не связанным с ним модуле, и к мгновенному выходу системы из строя. Причиной, по которой все модули компонуются в единое адресное пространство без поддержки какой-либо защиты между модулями, является улучшенная производительность за цену большего числа отказов системы. Тесно связанный вопрос относится к первопричине аварийных отказов.

Если бы каждый модуль был безупречным, то не возникала бы потребность в изоляции сбоев между модулями, поскольку не было бы самих сбоев. По утверждению авторов исследования [ ], большая часть сбоев возникает из-за ошибок программирования, вследствие чрезмерной сложности и использования чужого кода. Исследования показывают, что в программном обеспечении в среднем содержится от одной до шестнадцати ошибок на тысячу строк кода [27, 22, 2], и что верхняя граница этого диапазона явно занижена, поскольку учитывались только те ошибки, которые, в конце концов, удавалось обнаружить. Очевидным заключением является то, что в большем объеме кода содержится большее число ошибок. По мере развития программного обеспечения в каждой его новой версии появляется все больше возможностей (и, соответственно, больший объем кода), и часто новая версия является менее надежной, чем предыдущая. В [22] показано, что число ошибок на тысячу строк кода стремится к стабилизации по мере роста числа выпущенных версий, но асимптотически этот показатель отличается от нуля. Наличие некоторых из этих ошибок позволяет злоумышленникам применять вирусы и черви для заражения и повреждения системы. Так что некоторые якобы наличествующие проблемы «безопасности» не имеют ничего общего с нарушениями мер безопасности, а вызываются всего лишь ошибками в коде программ (например, переполнения буферов позволяют выполнять внедренный код).

Вторая проблема состоит в привнесении в операционную систему чужого кода. Наиболее искушенные пользователи никогда бы не позволили сторонней организации вставить незнакомый код в ядро операционной системы, хотя, когда они покупают новое периферийное устройство и инсталлируют соответствующий драйвер, они именно это и делают. Драйверы устройств обычно пишутся программистами, работающими на изготовителей периферийных устройств, и контроль качества их продукции обычно ниже, чем у поставщиков. В тех случаях, когда драйвер относится к open-source, его часто пишет благонамеренный, но не обязательно опытный доброволец, и контроль качества обеспечивается на еще более низком уровне. Например, в Linux частота появления ошибок в драйверах устройств от трех до семи раз выше, чем в других частях ядра [7]. Даже компания Microsoft, у которой имеются стимулы и ресурсы для применения более серьезного контроля качества, не может добиться намного лучших результатов: 85% всех аварийных отказов в ранних версиях Windows обусловливаются наличием ошибок в коде драйверов. Имеются публикации, посвященные изоляции драйверов устройств с использованием аппаратуры MMU [26] и виртуальных машин [19]. Эти методы концентрируются на решении проблем в унаследованных операционных системах. В отличие от этого, в работе [ ] надежность достигается путем разработки новой облегченной операционной системы (такой как, например, Миникс 3) с правильной изоляцией сбоев.

1. **Архитектуры операционных систем**

Под архитектурой операционной системы понимают структурную и функциональную организацию ОС на основе некоторой совокупности программных модулей. На архитектуру ранних операционных систем обращалось мало внимания, отсутствовал опыт разработки больших программных систем. Первые ОС разрабатывались как монолитные системы без четко выраженной структуры. Для построения монолитной системы необходимо было скомпилировать все отдельные процедуры, а затем связать их вместе (примерами могут служить ранние версии ядра UNIX или Novell NetWare). Такой подход был несовместим с расширением программной системы. Уже ОС OS/360 содержала более 1 млн строк кода, а система Maltics содержала к 1975 году 20 млн строк [1]. Стало ясно, что разработка таких систем должна вестись на основе модульного программирования. Большинство современных ОС представляют собой хорошо структурированные модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо единой унифицированной архитектуры ОС не существовало. Принципиально важными универсальными подходами к разработке архитектуры ОС являются [2–14]: модульная организация, функциональная избыточность, функциональная избирательность, параметрическая универсальность, концепция многоуровневой иерархической организации и др.

Классической стала считаться архитектура ОС, основанная на концепции иерархической многоуровневой машины, привилегированном ядре и пользовательском режиме работы транзитных модулей. Модули ядра выполняют базовые функции ОС: управление процессами, памятью, устройствами ввода-вывода и т.п. Ядро составляет сердцевину ОС, без которой она является полностью неработоспособной и не может выполнить ни одну из своих функций. В ядре решаются внутрисистемные задачи организации вычислительного процесса, недоступные для приложения. Функции ядра, которые могут вызываться приложениями, образуют интерфейс прикладного программирования – API (Application Programming Interface). Для обеспечения высокой скорости работы ОС значительная часть модулей ядра является резидентной и работает в привилегированном режиме. Этот режим, во-первых, должен обезопасить работу самой ОС от вмешательства приложений, и, во-вторых, должен обеспечить возможность работы модулей ядра с полным набором машинных инструкций, позволяющих собственно ядру выполнять управление ресурсами компьютера, в частности, переключение процессора с задачи на задачу, управлением устройствами ввода-вывода, распределением и защитой памяти и др. Остальные модули ОС выполняют не столь важные функции, как ядро, и являются транзитными.

В концепции иерархической машины структура ОС представляется рядом слоев. При такой организации каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс. На основе этих функций следующий верхний по иерархии слой строит свои функции – более сложные и более мощные и т.д. Такая организация системы существенно упрощает ее разработку, т.к. позволяет сначала «сверху-вниз» определить функции слоев и межслойные интерфейсы, а при детальной реализации, двигаясь «снизу-вверх», – наращивать мощность функции слоев. Кроме того, модули каждого слоя можно изменять без необходимости изменений в других слоях (но не меняя межслойных интерфейсов!) [1, 2, 12, 13, 15, 16].

Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению – обратное переключение. За счет этого возникает дополнительная задержка в обработке системного вызова. Однако такое решение стало классическим и используется во многих ОС (UNIX, VAX, VMS, IBM OS/390, OS/2, Windows др.). Многослойная классическая многоуровневая архитектура ОС не лишена своих проблем. Дело в том, что значительные изменения одного из уровней могут иметь трудно предвидимое влияние на смежные уровни. Кроме того, многочисленные взаимодействия между соседними уровнями усложняют обеспечение безопасности.

Поэтому, как альтернатива классическому варианту архитектуры ОС, часто используется микроядерная архитектура ОС [1,18 - 22]. Суть этой архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть ОС, называемая микроядром. Микроядро защищено от остальных частей ОС и приложений. В его состав входят машинно-зависимые модули, а также модули, выполняющие базовые механизмы обычного ядра. Все остальные более высокоуровневые функции ядра оформляются как модули, работающие в пользовательском режиме. Таким образом, в архитектуре с микроядром традиционное расположение уровней по вертикали заменяется горизонтальным. Внешние по отношению к микроядру компоненты ОС реализуются как обслуживающие процессы. Между собой они взаимодействуют как равноправные партнеры с помощью обмена сообщениями, которые передаются через микроядро. Поскольку назначением этих компонентов ОС является обслуживание запросов приложений пользователей, утилит и системных обрабатывающих программ, менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС. Схема смены режимов при выполнении системного вызова в микроядерной ОС сопровождается как минимум четырьмя переключениями режимов, в то время как в классической архитектуре – двумя. Следовательно, производительность ОС микроядерной архитектуры при прочих равных условиях будет ниже, чем у ОС с классическим ядром (наиболее критикуемый недостаток). В то же время признаны следующие достоинства микроядерной архитектуры [1, 18- 20]:

* единообразные интерфейсы;
* простота расширяемости;
* высокая гибкость;
* возможность переносимости;
* высокая надежность работы (наиболее важное преимущество);
* поддержка распределенных систем;
* поддержка объектно-ориентированных ОС.

По многим источникам вопрос масштабов потери производительности в микроядерных ОС является спорным [18- 20]. Многое зависит от размеров и функциональных возможностей микроядра. Избирательное увеличение функциональности микроядра приводит к снижению количества переключений между режимами системы, а также переключений адресных пространств процессов. Может быть, это покажется парадоксальным, но есть и такой подход к микроядерной ОС, как уменьшение микроядра.

1. **Метод и ограничения, принятые при разработке моделей архитектур операционных систем**

Разрабатываемые далее модели архитектур операционных систем основаны на предположении, что процесс функционирования компьютерной системы под управлением ОС является марковским. Его особенность заключается в том, что состояния системы изменяются во времени случайным непредсказуемым образом, причем для каждого момента времени t0 вероятность любого состояния для t1>t0 (в будущем) зависит только от вероятности состояния в момент t0 (в настоящем) и не зависит от вероятностей состояний при t <t0 (в прошлом). Другими словами, свойство марковского процесса заключается в том, что вероятности достижений будущих состояний не зависят от "предыстории" процесса. Если система меняет свое состояние скачкообразно и переходы из одного состояния в другое обладают марковским свойством, то случайный процесс называется марковской цепью [10, 11]. Удобно переходы из одного состояния в другое отображать в виде графа, в котором вершины представляют собой возможные состояния системы, а дуги графа отражают переходы из одного состояния в другое.

Поскольку переход из одного состояния в другое для СМО возможен в любой момент времени, определяемый появлением заявки во входном потоке, то для изучения систем массового обслуживания (в нашем случае операционной системы) применяются непрерывные марковские цепи. Одна из важнейших задач теории марковских процессов заключается в нахождении вероятностей состояний цепи. Эти вероятности для непрерывных марковских цепей определяются с помощью дифференциальных уравнений Колмогорова [23 - 27]. Так как предельные вероятности состояний системы постоянны, то, заменяя в уравнениях Колмогорова их производные нулевыми значениями, можно перейти к системе линейных алгебраических уравнений, описывающих стационарный режим. Систему этих уравнений можно составить непосредственно по размеченному графу состояний, если руководствоваться правилом, согласно которому слева в уравнениях стоит предельная вероятность данного состояния pi, умноженная на суммарную интенсивность всех потоков, ведущих из данного состояния, а справа – сумма произведений интенсивностей всех потоков, входящих в i-е состояние, на вероятности тех состояний, из которых эти потоки исходят. Построить граф состояний системы, управляемой ОС некоторой архитектуры, не представляет особой сложности (с некоторыми допущениями, не снижающими адекватность модели цели исследования). Большую сложность представляет собой определение интенсивностей потоков, переводящих систему из одного состояния в другое [28].

Задачей разработки моделей архитектур ОС в данной работе является оценка надежности работы компьютерной системы, которую обеспечивает операционная система той или иной архитектуры. При этом исходим из предположения, что надежность аппаратуры является абсолютной, и в целом надежность функционирования компьютерной системы определяется надежностью программного обеспечения (ПО), в которое входит операционная система, системные и пользовательские приложения. Если рассматривать отказавшее ПО без учёта его восстановления, а также случайный характер и независимость отказов в программах, то основные показатели надёжности в этом случае не отличаются от тех, которые характерны для аппаратуры компьютера. Таким образом, основными показателями надёжности ПО являются [29]:

• вероятность безотказной работы программы p(t), представляющая собой вероятность того, что ошибки программы не проявятся в интервале времени (0, t);

• вероятность отказа программы q(t) или вероятность события отказа до момента времени t;

• интенсивность отказов программы λ(t);

• средняя наработка программы на отказ T, являющаяся математическим ожиданием временного интервала между последовательными отказами.

Формальные методы оценки надежности программных систем не позволяют получить числовые значения этих показателей в зависимости от числа возможных программных ошибок без испытания программной системы. При этом характер изменения этих показателей во времени будет зависеть от модели надёжности ПО [30, 31]. Эти модели учитывают тот факт, что возникающие при работе программ ошибки устраняются, количество ошибок уменьшается и, следовательно, интенсивность их появления понижается, а наработка на отказ программы увеличивается. Так как в нашем случае такой возможности определения наработки программы на отказ нет, примем следующие ограничения и допущения, позволяющие определить исходные данные для моделей функционирования компьютерных систем с ОС различных архитектур. Основное ограничение касается числа состояний исследуемой модели. Оно не должно быть значительным (не более 4 – 7), в противном случае усложняется поиск решения. Далее будем считать, что 1000 строк программного кода содержат от 4 до 16 ошибок. Это подтверждается практикой и рядом публикаций [1, 2, 17, 18]. Для драйверов число ошибок в 3 – 7 раз больше. Примем далее, что микроядерная ОС (подобная Minix) содержит 6000 строк, драйвер – 100000 строк, основная часть ядра многоуровневой модульной ОС (подобной Linux) содержит 10 млн. строк и вспомогательная часть – 20 млн. строк. Примем допущение, что микроядерное ядро Minix с числом ошибок, равным 6 (1 ошибка на 1000 строк кода), имеет наработку на отказ T=100000 час. Другие ограничения и допущения будут вводиться в конкретных моделях ОС.

1. **Модель операционной системы с многоуровневым модульным ядром**

На рис. 2 представлен граф состояний и переходов компьютерной системы с многоуровневой модульной операционной системой (типа Linux). Перечислим состояния и соответственно вероятности нахождения системы в этих состояниях: P1 – вероятность работы ядра операционной системы ОСЯ в привилегированном режиме; P2 – вероятность работы модулей операционной системы ОСП в пользовательском режиме; P3 – вероятность работы пользовательских приложений ПП; P4 – вероятность отказа системы.

Определим интенсивности переходов системы из одного состояния в другое (здесь и далее параметры моделей выбраны на основе измерений в реальных операционных системах [28]): λ1 – интенсивность системных вызовов со стороны прикладных программ, примем значение λ1 = 10000 1/c; λ2 – интенсивность системных вызовов со стороны прикладных программ, в выполнении которых участвуют устройства системы с соответствующими драйверами; пусть каждый 200-й системный вызов требует участия драйверов устройств, тогда значение λ2 = 50 1/c; λ3 – интенсивность выполнения системных вызовов драйверами устройств. Пусть на выполнение одного вызова требуется 5 мс, тогда λ3 = 200 1/c; λ4 – интенсивность передачи выполненных системных вызовов в прикладные программы. Эти вызовы выполняются только ОСЯ (назовём их короткими) или совместно с ОСП или ОСЯ передает результат работы ОСП (назовем их длинными). Примем среднее время передачи выполненных системных вызовов 0, 1 мс. Тогда значение λ4 =10000 1/c; λ5 – интенсивность отказов ОСЯ. Выше было принята интенсивность отказов ядра Minix равной 100000 часов при 6 программных ошибках. Размер ОСЯ был принят равным 10 млн. строк. Число программных ошибок в данном случае следует принять равным не менее 3 ошибок на 1000 строк. Таким образом, ОСЯ содержит не менее 30000 ошибок. Считая, что наработка на отказ обратно пропорциональна числу ошибок, получаем наработку на отказ для ОСЯ равную 20 час. При таких допущениях λ5 = 0,000014 1/c. λ6 – интенсивность отказов ОСП. Размер ОСП был принят равным 20 млн. строк. Число программных ошибок в данном случае следует принять равным не менее 10 на 1000 строк кода (как отмечено выше число ошибок в драйверах не чем в 3 раза больше, чем в программах ядра).

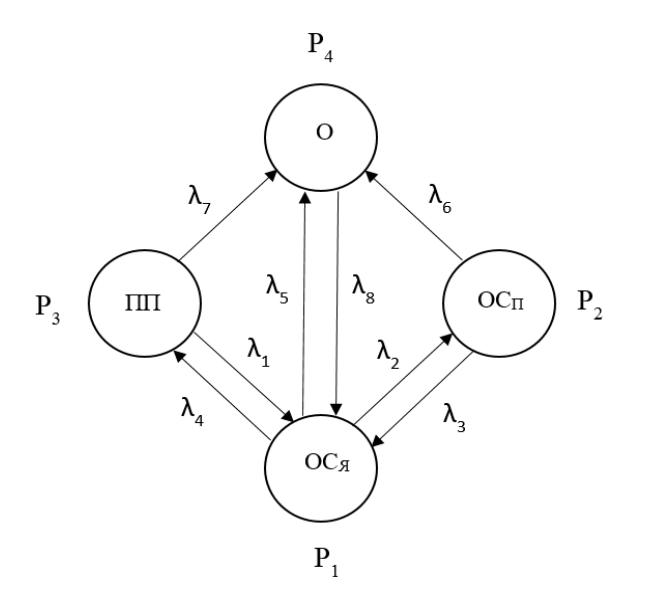


Рис. 2. Граф состояний и переходов системы с многослойным модульным ядром

Число ошибок в ОСП при принятых допущениях не менее 300000. Однако следует заметить, реально используется работе не более 1/10 возможностей ОСП. С учетом этого допущения можно принять λ6 = 0,000014 1/c. λ7–интенсивность отказов ПП. Считая средний размер пользовательского приложения 20000 строк, можно считать, что в нем (при 20 ошибках на 1000 строк) порядка 400 ошибок. Отсюда λ7 = 0,0000047 1/c. λ8 – интенсивность перезагрузки операционной системы. Примем время перезагрузки равное трем минутам, тогда λ8 = 0,0055 1/c. По графу состояний и переходов системы, представленному на рис. 2, можно составить систему уравнений, руководствуясь следующим правилом: слева в уравнениях стоит предельная вероятность данного состояния Pi, умноженная на суммарную интенсивность всех потоков, ведущих из данного состояния, а справа – сумма произведений интенсивностей всех потоков, входящих в i-е состояние, на вероятности тех состояний, из которых эти потоки исходят [12].

Полученная система уравнений имеет следующий вид:

(1)

В данном случае имеем четыре уравнения в системе (1) при четырех неизвестных. Уравнения однородны (не имеют свободного члена) и определяют неизвестные только с точностью до произвольного множителя. Но можно воспользоваться так называемым нормировочным условием и с его помощью решить систему. При этом одно (любое) из уравнений можно отбросить (оно вытекает как следствие из остальных). Отбросим первое уравнение и, таким образом, получаем следующую систему уравнений:

(2)

Подставив определенные выше значения коэффициентов при переменных и записав систему уравнений в матричной форме, удобной для решения средствами Excel, получим систему (2) в следующем виде:

= 0;

= 0;

= 0; (3)

= 1.

На решении этой системы уравнений остановимся позже, после построения всех предлагаемых в статье моделей.

1. **Модель мультисерверной микроядерной операционной системы**

В качестве примера такой операционной системы возьмем Unix-подобную структуру MINIX 3, предложенную Э. Таненбаумом более 15 лет назад. В данном случае предлагается иметь несколько небольших модулей (микроядро – МЯ и сервер реинкарнации – СР), работающих в режиме ядра, остальная часть операционной системы представляет собой набор полностью изолированных серверов (серверных процессы – СП) и драйверов (Д), работающих в режиме пользователя. В операционной системе MINIX 3 микроядро обрабатывает прерывания, обеспечивает основные механизмы для управления процессами, реализует межпроцессные взаимодействия и производит планирование процессов. Оно также предоставляет небольшой набор вызовов ядра для авторизованных драйверов и серверов, например, для чтения части заданного пользовательского адресного пространства или записи в авторизованные порты ввода-вывода. В адресном пространстве микроядра работает драйвер таймера, но он планируется как отдельный процесс. Никакие другие драйверы в режиме ядра не работают. Над уровнем микроядра находится уровень драйверов устройств. Для каждого устройства ввода-вывода имеется собственный драйвер, выполняемый в виде отдельного процесса в собственном адресном пространстве, защищенном аппаратурой устройства управления памятью.

Драйверы работают в пользовательском режиме и не могут исполнять привилегированные команды, а также читать из портов компьютера или писать в них. Для получения последней возможности они должны производить вызовы ядра. Такая конструкция повышает надежность, хотя и порождает небольшие дополнительные расходы. Поверх уровня драйверов устройств располагается уровень серверов. Файловый сервер является небольшой (4500 строк исполняемого кода) программой, которая принимает запросы от пользовательских процессов по обработке Posix-совместимых вызовов, относящихся к файлам (read, write, lseek и stat) и выполняет их. На этом уровне находится и менеджер процессов, который поддерживает управление процессами и памятью и выполняет Posix-совместимые и другие системные вызовы, такие как fork, exec и brk.

Сервер реинкарнации является родительским процессом всех других серверов и всех драйверов. Если драйвер или сервер аварийно или по собственной инициативе завершается, либо не отвечает на периодические запросы отклика, то сервер реинкарнации принудительно завершает его, если это требуется, и перезапускает из копии на диске или в основной памяти. В число других серверов входит сервер сети, поддерживающий весь стек TCP/IP; простой сервер имен, используемый всеми остальными серверами; и информационный сервер, способствующий отладке. Наконец, над уровнем серверов находятся пользовательские процессы (ПП). Для пользователей единственным отличием мультисерверной системы от других Unix-систем является то, что библиотечные процедуры для системных вызовов выполняют свою работу путем посылки сообщений серверам. Во всем остальном это обычные пользовательские процессы, в которых может использоваться API Posix. Межпроцессные взаимодействия (IPC) в MINIX 3 поддерживаются на основе передачи сообщений фиксированной длины с использованием принципа рандеву: система копирует сообщение напрямую от отправителя к получателю, когда оба они к этому готовы.

Кроме того, поддерживается механизм асинхронного уведомления о событиях. С системой передачи сообщений интегрирована обработка прерываний. Обработчики прерываний используют механизм уведомлений для сигнализации о завершении ввода-вывода. Этот механизм позволяет обработчику установить бит в битовой шкале "необработанных прерываний" драйвера и продолжить выполнение без блокировки. Когда драйвер становится готовым к получению прерывания, ядро преобразует его в обычное сообщение. Среди других особенностей, способствующих повышению надежности, наиболее важным является свойство самовосстановления. Если драйвер производит запись по неверному указателю, впадает в бесконечный цикл или неправильно ведет себя каким-либо другим образом, то сервер реинкарнации автоматически заменит его, часто без влияния на другие процессы. В соответствии с рассмотренной архитектурой микроядерной мультисерверной операционной системой ее граф состояний и переходов может быть представлен, как показано на рис. 3.

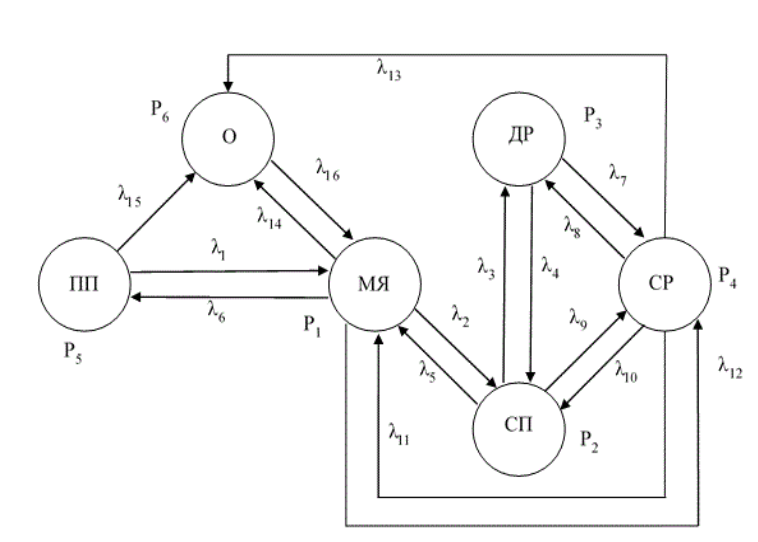


Рис. 3. Граф состояний и переходов мультисерверной

микроядерной операционной системы

Перечислим состояния и соответственно вероятности нахождения системы в этих состояниях: P1 – вероятность работы микроядра операционной системы МЯ в привилегированном режиме; P2 – вероятность работы модулей сервисных процессов (СП) в пользовательском режиме; P3 – вероятность работы драйверов ДР в пользовательском режиме; P4 – вероятность работы сервера реинкарнации (СР) в пользовательском режиме; P5 – вероятность работы пользовательских приложений ПП; P6 – вероятность отказа системы.

Аналогично тому, как это было сделано в предыдущей модели, определим интенсивности переходов системы из одного состояния в другое, сохраняя при этом характеристики выполняемых в системе программ пользователей ПП: λ1 – интенсивность системных вызовов со стороны прикладных программ, примем значение λ1 = 10000 1/c; λ2 – интенсивность системных вызовов со стороны прикладных программ, в выполнении которых участвуют сервисные процессы (СП) без устройств компьютерной системы и, следовательно без драйверов. Учитывая (как принято в модели 1), что каждый 200-й системный вызов требует участия драйверов устройств, тогда значение λ2 = 50 1/c; λ3 – интенсивность системных вызовов со стороны прикладных программ, требующих участия в их выполнении драйверов устройств; λ3 = 50 1/c; λ4 – интенсивность выполнения системных вызовов драйверами устройств; пусть на выполнение одного вызова требуется 5 мс, тогда λ4 = 200 1/c; λ5 – интенсивность передачи результатов системных вызовов, выполненных самим микроядром МЯ и драйверами совместно с сервисными процессами, в прикладную программу ПП. Примем среднее время передачи 0,01 мс. Тогда значение λ5 =100000 1/c; λ6 – интенсивность передачи результатов завершения выполнения системных вызовов драйверами совместно с сервисными процессами, в пользовательские процессы ПП (по сути, это – разблокировка процессов). Примем среднее время передачи 0,01 мс. Тогда значение λ6 =100000 1/c.

Выше было принята наработка на отказ ядра Minix равной 100000 часов при 6 программных ошибках. Примем размер драйверов 1 млн. строк. Число программных ошибок в данном случае следует принять равным не менее 20 на 1000 строк кода. Таким образом, драйверы содержит не менее 20000 ошибок. Считая, что наработка на отказ обратно пропорциональна числу ошибок, получаем наработку на отказ для Д равную 30 час. При таких допущениях λ7 = 0,00000923 1/c. λ8 – интенсивность перезапуска драйверов сервером реинкарнации. Перезапуск может происходить из оперативной памяти или (если в данный момент там есть его копия) или с диска (в противном случае). Примем среднее значение времени перезапуска драйвера 0,01 мс. Таким образом, λ8 = 100000 1/c; λ9 – интенсивность отказов серверных процессов (СП). Будем считать, надежность работы СП примерно такая же, как и драйверов. В этом случае λ9 = 0,00000923 1/c. λ10 – интенсивность перезапуска серверных процессов сервером реинкарнации. Можно считать это действие аналогичным действию перезапуска драйверов. Тогда λ10 = 100000 1/c; λ11–интенсивность отказов сервера реинкарнации (СР). Следует считать, что СР также надежен, как и микроядро. В противном случае пропадает сама суть микроядерной операционной системы. Следовательно, наработка на отказ СР равна 100000 час. и λ11 = 0,0000000028 1/c; λ12 – интенсивность перезапуска сервера реинкарнации.

Предполагая, что этот процесс аналогичен перезапуску Д и СП, будем считать, что λ12 = 100000 1/c; λ13 – интенсивность фатальных отказов СР (восстановление СР с помощью МЯ невозможно). Примем, что интенсивность таких отказов на порядок ниже интенсивности отказов СР, когда восстановление возможно, тогда λ13 = 0,0000000028 1/c; λ14 – интенсивность фатальных отказов МЯ. Считая, что надежности СР и МЯ эквивалентны, λ14 = 0,0000000028 1/c; λ15 – интенсивность фатальных отказов пользовательских приложений. Примем это значение, равным соответствующему значению в модели 1, т.е. λ15 =0,0000047 1/c. λ16 – интенсивность перезагрузки операционной системы. Примем время перезагрузки равное трем минутам, тогда λ16 = 0,0055 1/c. Аналогично тому, как это сделано в модели по рис. 2, составляем систему уравнений для модели по рис. 3:

(4)

Исключаем из системы (4) первое уравнение. Учитывая установленные выше значения коэффициентов при переменных и записав систему уравнений в матричной форме, удобной для решения средствами Excel, получим систему (4) в следующем виде:

; (5)

Также, как и после системы уравнений, описывающих первую модель ОС, на решении этой системы уравнений остановимся позже, после построения всех предлагаемых в статье моделей.

1. **Модель операционной системы Касперского**

По словам главы «Лаборатории Касперского» Евгения Касперского [ ], сегодня существуют три глобальные проблемы безопасности, с которыми государствам и бизнесу приходится сталкиваться и которые «Лаборатория Касперского» решает, благодаря разработанной концепции кибериммунности и собственной операционной системе. Первая проблема – массовая киберпреступность. Успехи лаборатории в этой области широко известны. Свыше 400 миллионов пользователей и 270 тысяч корпоративных клиентов во всем мире уже доверяют свою защиту продуктам «Лаборатории Касперского». Вторая проблема – профессиональная киберпреступность, которая становится всё более подготовленной, консолидированной и технически обеспеченной. Бороться с ней можно с помощью многофакторной эшелонированной защиты, включающей в себя защиту периметра, сети и трафика и др. Третья проблема – атаки на промышленную и критическую инфраструктуру, которые характеризуются наивысшей степенью риска и в случае успеха наносят наиболее сильный ущерб.

В критической инфраструктуре риски зашкаливают, и решить эту проблему только добавленными средствами безопасности невозможно. Именно поэтому лаборатория разработала концепцию кибериммунитета. Кибериммунной Е. Касперский называет такую систему, стоимость организации атаки на которую выше, чем возможный ущерб. Достичь этого позволяет операционная система Kaspersky OS, с микроядерной архитектурой, где все связи и взаимодействия проходят уровень безопасности. Причем это разрешительная система безопасности, а не запретительная. Если какой-то ее элемент будет поражен, вредоносное ПО не проникнет дальше. В этом главное отличие Kaspersky OS от традиционных операционных систем. Важно, что этот принцип может быть реализован не только в операционных системах для ПК. На рынке уже представлено несколько устройств и комплексных решений, работающих на Kaspersky OS.

Понятие кибериммунитета основано на концепции Secure by Design (безопасность при разработке), ключевой принцип которой состоит в том, что безопасность должна являться неотъемлемой частью любой разрабатываемой системы, присутствовать в каждом ее компоненте и сопровождать весь цикл разработки, начиная с этапа проектирования. Таким образом, у системы как бы появляется «врожденный иммунитет» к различного рода киберугрозам – как существующим, так и новым. Kaspersky OS – это инструмент, который помогает сторонним разработчикам быстрее создавать более безопасные программные и аппаратные решения. В этом смысле можно определить кибериммунитет как инновационную комбинацию инструментов, методологий и способов разработки программного обеспечения по принципу Secure by Design.

Остановимся на особенностях архитектуры Kaspersky OS. Система [полностью создана](https://securelist.ru/features-of-secure-os-realization/30181/) с чистого листа специалистами «Лаборатории Касперского», является оригинальной разработкой компании и зарегистрирована в реестре российского ПО как рекомендованная для приобретения отечественными организациями и государственными структурами. Основой ОС служит надёжное микроядро, которое допускает только строго определённый способ взаимодействий системных процессов и обмена данными, тем самым обеспечивая полноту контроля доступа. Будучи компактным, оно может портироваться на различные аппаратные платформы. KasperskyOS построена на базе концепции MILS (Multiple Independent Levels of Security — «множественные независимые уровни защиты/безопасности»), определяющей строгие принципы изоляции системных процессов и политик управления информационными потоками. Одним из наиболее важных компонентов ОС [является](https://os.kaspersky.ru/2019/03/26/kasperskyos-immunnyy-podkhod-k-zashhite-informaci/) модуль контроля доступа Kaspersky Security System (KSS), отслеживающий все межпроцессорные взаимодействия и исключающий доступ приложений к защищённым областям памяти с критически важными процессами и данными. KSS поддерживает широкий набор правил и политик доступа и может быть настроен в соответствии с конкретными требованиями заказчика. Любое действие, не предусмотренное политиками безопасности, запрещено по умолчанию. И это, пожалуй, самое важное отличие Kaspersky OS от популярных сегодня операционных систем, в которых всё, что не запрещено, по умолчанию разрешено.

Микроядро KasperskyOS — собственная разработка «Лаборатории Касперского». Как дизайн, так и реализация всех частей микроядра отвечают главной задаче — созданию безопасной операционной системы. В KasperskyOS всего три системных вызова и только один интерфейс межпроцессного взаимодействия. Благодаря этому поверхность атаки минимальна. Микроядро KasperskyOS спроектировано так, чтобы по максимуму использовать возможности Kaspersky Security System (KSS). И ее микроядро — действительно «микро» (не более тысячи строк кода). В них втиснуты диспетчер процессов, механизм межпроцессного взаимодействия (IPC — inter-process communication), модуль контроля доступа KSS (Kaspersky Security System), который следит за механизмом IPC.

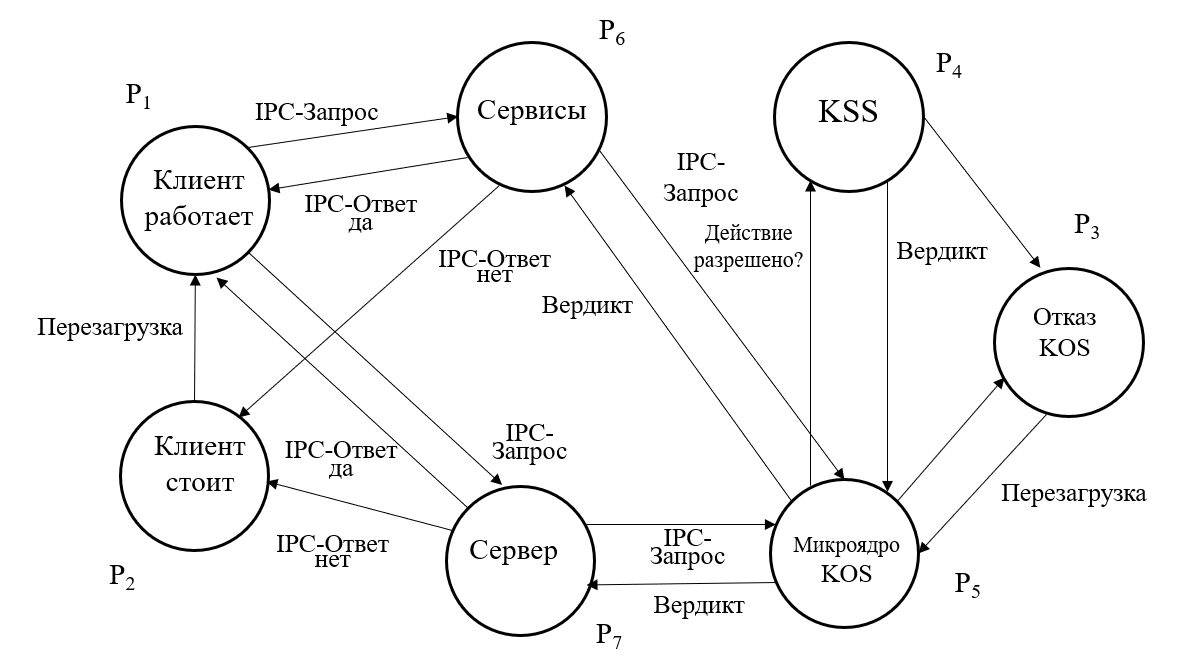
Таким образом, базовый принцип KasperskyOS схож с общим подходом любых микроядерных систем. Процессы взаимодействуют между собой и с функциями ядра системы, отправляя и получая IPC-сообщения. Микроядро предоставляет им эту возможность с помощью механизма IPC. В случае KasperskyOS за этим механизмом следит KSS, которая для каждого IPC-сообщения выносит вердикт «можно» (allow) или «нельзя» (deny). При этом по умолчанию KSS реализует принцип default deny. То есть если программа по какой-то причине не реализует такую модель взаимодействия, то ни отправить, ни получить IPC-сообщения она не сможет. И останется в полной изоляции.

Вся остальная функциональность операционной системы, включая драйверы, файловые системы и сетевые стеки, вынесена в режим пользователя. Ядро KasperskyOS гарантирует полную изоляцию компонентов IT-системы. Единственный вид межпроцессных взаимодействий, предоставляемый ядром, - синхронный обмен сообщениями («запросом» и «ответом»). При этом каждое сообщение передается подсистеме Kaspersky Security System для проверки на соответствие заданной политике безопасности. Ядро доставит сообщение, только если это разрешено политикой. Таким образом, базовый принцип KasperskyOS сходен с общим подходом любых микроядерных систем. Процессы взаимодействуют между собой и с функциями ядра системы, отправляя и получая IPC-сообщения. Микроядро предоставляет им эту возможность с помощью механизма IPC (рис. 4). В случае KasperskyOS за этим механизмом следит KSS, которая для каждого IPC-сообщения выносит вердикт «можно» (allow) или «нельзя» (deny). При этом по умолчанию KSS реализует принцип default deny. Если программа по какой-то причине не реализует такую модель взаимодействия, то ни отправить, ни получить IPC-сообщения она не сможет и останется в полной изоляции. Следовательно, ни для вирусов, ни для вредоносного ПО, ни для «криворуко» написанного софта нет возможности нарушить вычислительный процесс.

[](https://xakep.ru/wp-content/uploads/2015/12/1450678756_ff8c_kos-architecture-idea.png)

Рис. 4. Схема межпроцессного взаимодействия (из статьи Е. Лебеденко. Операционная система от “Лаборатории Касперского”. Как устроена Kaspersky OS. 21.12.2015)

Учитывая изложенные особенности построения микроядерной кибериммунной операционной системы KasperskyOS, ее модель графа состояний и переходов можно представить следующим образом (рис. 5).



λ2

λ16

λ9

λ10

λ7

λ6

λ8

λ15

λ14

λ13

λ12

λ11

λ3

λ4

λ5

λ1

Рис. 5. Граф состояний и переходов KOS

Будем считать, что пользователь в своей работе может использовать свою программу, а также пользоваться услугами сервисов и сервера операционной системы Kaspersky OS. Перечислим состояния и соответственно вероятности нахождения системы в этих состояниях: P1 – вероятность того, что пользователь системы занят работой, в этом состоянии система исправна и работоспособна. P2 – пользователь простаивает в связи с отказом со стороны ОС в использовании ее сервисов или сервера (причины: небезопасный код программы пользователя, неправильный запрос на услуги ОС и др.). В этом состоянии возможна перезагрузка программы и попытка дальнейшей работы. Операционная система исправна, поскольку она практически в силу небольшого объёма кода (несколько десятков строк кода) практически безотказна. Для определенности примем наработку на отказ 1000000 час. P3 – состояние отказа KOS по причине проявления ошибки в KSS или микроядре KOS. В этом случае возможна перезагрузка системы и дальнейшая работа пользователя. Дальнейшие состояния и, соответственно вероятности отображают работу компонентов KOS: P4 – работа подсистемы Kaspersky Security System (KSS), P5 – работа микроядра KOS, P6 и P7 – соответственно – работа сервисов и сервера системы.

Перейдем к определению интенсивностей перехода между состояниями системы. Примем на основе данных статьи [ ] интенсивность обращения Клиента к услугам сервисов λ1 =100000 1/c. В любом случае получение требуемой услуги происходит по схеме, приведенной на рис. 6. Как видно из рис. 6, происходит некоторая задержка в обработке запроса Клиента (дважды работает ядро и KSS), однако в нашей модели это можно не учитывать. Поэтому λ3 = λ4 = 100000 1/c. В потоке λ4 передается вердикт, определяющий возможность правильного выполнения запроса. Ответ “да” определяет интенсивность потока λ2, соответственно ответ “нет” определяет интенсивность потока λ5. Будем считать, что с вероятностью, близкой к 1, например, 0,9999999 будет получен ответ “да”, в этом случае вероятность ответа “нет” = 0,00000001. В этих условиях интенсивности потока λ2 = λ1= 9999999,9 1/с, соответственно λ5 = 0,001 1/с. Рассмотрим обращение клиента к услугам сервера. Пусть время выполнения сервиса KOS=0,05мс, а время выполнения драйвера сервера (печать на мониторе) – 5 мс (см. [ ]). Предположим, что на 10000 запросов Клиента приходится одно обращение к драйверу печати. Тогда среднее время обслуживания составит значение

.

Таким образом, получаем, интенсивность потока λ6 = 1/0,0000505 Интенсивности потоков λ8 и λ9 имеет такое же значение. Последний поток содержит значение вердикта. Ответ “да” определяет в этом случае интенсивность потока λ7, соответственно ответ “нет” определяет интенсивность потока λ10. Примем условие, что с вероятностью 0,999999 будет получен ответ “да”, в этом случае вероятность ответа “нет” = 0,000001. В этих условиях интенсивности потока λ7 = λ9 = 19 801,98 1/с, соответственно λ10 = 0,02 1/с.

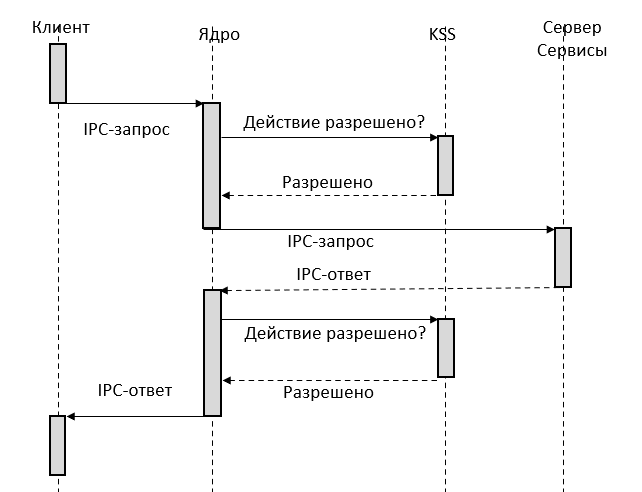


Рис. 6. Обмен сообщениями Клиент-Сервер

Интенсивность потока λ11 определяется суммой интенсивностей λ3 и λ8 и, следовательно равна 119 802 1/c. Такую же интенсивность имеет поток вердиктов λ12. Интенсивность потоков λ13 и λ14 определяется только возможными программными ошибками в KOS. Пересчитывать количество таких ошибок, как это делалось выше, по статистическим данным в зависимости от числа строк программного кода не имеет смысла. Поверим заявлениям разработчиков Kaspersky OS об отсутствии программных ошибок и для определенности будем считать, что наработка на отказ KOS равна 1000000 час. Отсюда можно считать, интенсивности отказов λ13 и λ14 составляют значение, равное 1/1000000 час. Перевод в секунды дает значение λ13 = λ14 = 0, 00000000028 1/с. Что касается интенсивности потоков перезагрузки, примем ее равной этому значению в предыдущих моделях, т.е. λ15 = λ16 = 0,0055 1/c.

По полученному и размеченному графу состояний и переходов в соответствии с правилом, изложенным в разделе 3, составляем систему линейных алгебраических уравнений, описывающих стационарный режим работы KOS:

λ2 λ7 λ16

+

+

λ11 (6)

λ8 λ3 λ12

λ1 λ4

λ6 λ9

Аналогично, как и в предыдущих системах уравнений, исключаем из системы (6) пятое уравнение. Учитывая установленные выше значения коэффициентов при переменных и записав систему уравнений в матричной форме, удобной для решения средствами Excel, получим систему (6) в следующем виде:

99999,9 19801,98 0,0055

(7)

1. **Решение систем уравнений, представляющих модели**

Самым распространенным способом решения системы линейных уравнений в Excel является матричный метод. Он заключается в построении матрицы из коэффициентов уравнений и создании обратной матрицы. Матрицу заполняем числами, которые являются коэффициентами уравнения. Числа должны располагаться последовательно по порядку с учетом расположения корня, которому они соответствуют. Если в выражении отсутствует корень, то коэффициент считается равным нулю. Если коэффициент не обозначен, но соответствующий корень имеется, то считается, что коэффициент равен единице. Полученную таблицу считаем вектором **A**. Далее необходимо получить матрицу, обратную существующей. Для этого можно воспользоваться функцией Excel МОБР. Окно аргументов этой функции и всего одно поле – «Массив», в котором нужно указать адрес таблицы. При работе с массивами после завершения ввода формулы следует произвести набор сочетания клавиш Ctrl+Shift+Enter. После этого программа производит вычисления и на выходе в предварительно выделенной области мы имеем матрицу, обратную данной. Далее нужно умножить обратную матрицу на матрицу **B**, которая состоит из одного столбца значений, расположенных после знака «равно» в выражениях. Для умножения таблиц в Excel есть функцияМУМНОЖ. Активируется окно аргументов функции МУМНОЖ. В поле «Массив1» указываются координаты обратной матрицы. Аналогичное действие проводим для внесения координат в поле «Массив2», выделяя значения колонки **B**. После набираем комбинацию клавиш Ctrl+Shift+Enter. В результате в предварительно выделенной ячейке отобразятся корни уравнения. Решение трех систем уравнений, составленных выше для рассмотренных моделей, показано на рис. 7.

1. **Обсуждение результатов моделирования**

Решение системы уравнений (3) дает следующие значения переменных:

0,090815456; 227039; 0,908154557; 0,001007284. Таким образом, вероятность того, что компьютерная система в установившемся состоянии исправно функционирует, равна сумме +0,909185. Такая низкая надежность ОС с многослойным ядром в нашем случае объясняется заданием в исходных данных большого количества программных ошибок в ядре и той части системы, которая работает в пользовательском режиме (300000 ошибок). Вероятность нахождения системы в состоянии перезагрузки равна 0,090815456. Ситуация весьма похожа на начальные версии Linux и других операционных систем. В процессе устранения ошибок надежность системы, конечно, повысится, но всё-таки не будет достаточно высокой. Радикальный путь – уменьшения ядра ОС.

Решение системы уравнений (6) дает следующие значения переменных: 0,003880570; 0,00384185; 0,952238435; 0,00190285; 0,3805695; 0,00000830. Таким образом, вероятность того, что компьютерная система в установившемся состоянии исправно функционирует, равна сумме +0,999534035. Вероятность того, что система в установившемся состоянии будет находиться на этапе перезагрузки, равна 0,00000830. Таким образом, результаты проведенных вычислений позволяют сделать вывод о значительно более высокой надежности работы микроядерной операционной системы по сравнению с операционной системой с большим многоуровневым модульным ядром.

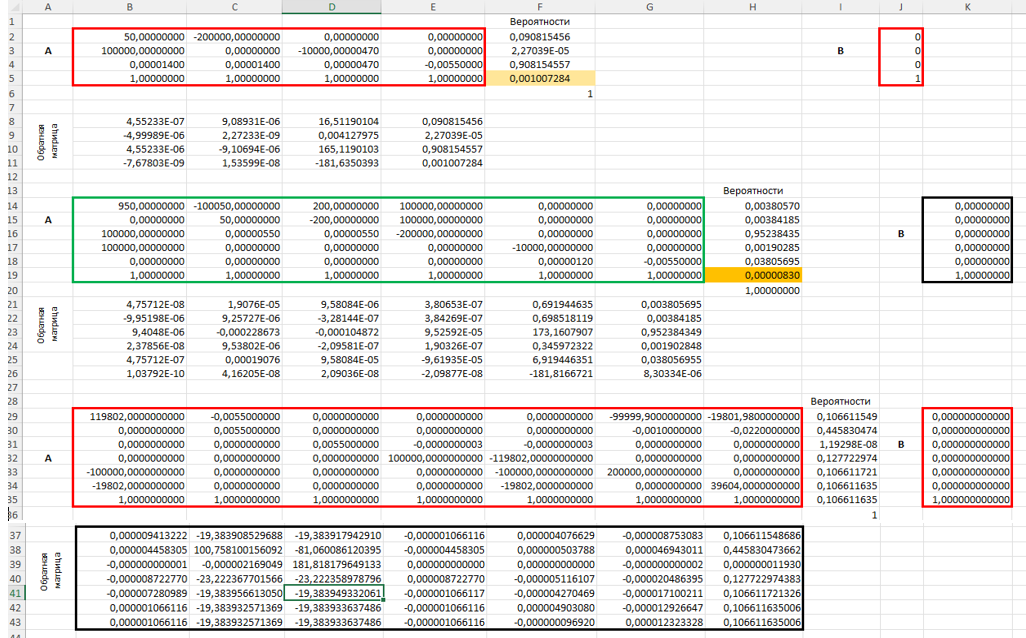


Рис. 7. Решение систем уравнений

И в заключение рассмотрим решение системы уравнений (7). Оно дает следующие значения переменных: 0,106611549; 0,445830474; 0,0000000119298; 0,127722974; 0,106611721; 0,106611635; 0,106611635. Заметим, что в случае Kaspersky OS, клиент может простаивать, получив вердикт deny (нет на рис. 5) на запрошенную услугу. В случае наших исходных данных вероятность такого простоя 0,445830474. Изменив интенсивности потоков ,которые определяют частоту появления вердикта “да” и можно получить требуемое значение вероятностей, определяющих работу и простой клиента. При этом, операционная система работоспособна, что подтверждается низкой вероятностью ее отказа 0,0000000119298 в установившемся режиме. Конечно, в реальности (не в модели) вердикт определяется правильность работы клиента при формировании запросов обращения к услугам сервисов и сервера.

**Заключение**

При решении задач разработки, производства, выбора и эксплуатации программных систем и, операционных систем, в частности, для исследования критических параметров, таких как надежность, безопасность, защищенность часто требуется обратиться к методам моделирования, среди которых широкое применение получили методы имитационного, полунатурного и натурного моделирования. Традиционно под моделированием на ЭВМ понималось лишь имитационное моделирование (ИМ). Можно, однако, увидеть, что и при других видах моделирования компьютер может быть весьма полезен, за исключением разве физического моделирования, где компьютер тоже может использоваться, но, скорее, для целей управления процессом моделирования. Например, при математическом моделировании выполнение одного из основных этапов – построение математических моделей по экспериментальным данным просто немыслимо без компьютера.

Имитационное моделирование представляет собой метод конструирования модели реальной системы и постановки экспериментов на этой модели с целью исследования её поведения, либо оценивания различных стратегий, обеспечивающих функционирование данной системы. Этот метод, конечно, может быть успешно использован для определения критических параметров архитектур операционных систем, их оценки, сравнения и выбора наиболее подходящей для конкретного применения. При этом необходимо создать структуру исследуемой системы и описать ее поведение с помощью состояний и моментов переходов между этими состояниями. Состояние системы в каждый момент времени можно определить как множество значений ее параметров в этот момент времени. Изменение значений параметров можно считать переходом в другое состояние. Внешняя среда задается посредством входных данных. При необходимости моделирования вероятностных систем и процессов в ИМ включается и статистическое моделирование (метод Монте-Карло).

Имитационное моделирование достаточно распространено при исследовании сложных систем благодаря ряду преимуществ. Благодаря ИМ на ранних стадиях предварительного проектирования систем можно быстро получить нужную информацию, пусть и с некоторыми допущениями, о возможном функционировании проектируемой системы. Можно исследовать особенности функционирования системы при любых условиях, в частности тех, которые не могут быть реализованы в натурных экспериментах. При этом параметры системы и окружающей среды можно варьировать в широких границах, воссоздавая произвольные, как реальные, так и гипотетические, ситуации. Главными недостатками метода машинной имитации являются довольно большие затраты времени и средств на построение адекватной модели, а также трудность и даже невозможность учета в модели некоторых важных особенностей реальной системы. По этим причинам машинную имитацию как численный машинный метод решения сложных задач целесообразно применять при следующих условиях:

* непригодность или отсутствие аналитических методов решения задач;
* полная уверенность в успешном создании имитационной модели, которая адекватно описывает исследуемую систему (процесс);
* возможность использовать сам процесс построения имитационной модели для предварительного исследования моделируемой системы.

Натурный эксперимент для определения надёжности функционирования операционных систем различных архитектур практически невозможен в силу неприемлемых временных и финансовых затрат. Поэтому единственный путь решения этой проблемы – разработка простых и достаточно адекватных моделей.

Предложенный подход к построению моделей архитектур операционных систем, основанный на теории марковских процессов, описываемых системами дифференциальных уравнений Колмогорова, в условиях принятия ограничений об абсолютной надёжности аппаратуры, случайном характере и независимости отказов в программах без учёта их восстановления и отсутствии параллельных процессов в работе модулей ОС может быть использован для начального исследования надежности функционирования конкретных архитектур операционных систем. Нельзя сказать, что такой подход не имеет недостатков. Один из них – ограничение числа возможных состояний в модели системы. При числе состояний более 8 – 10 усложняется решение систем линейных алгебраических уравнений. Повышение точности результатов моделирования в подобных моделях связано с уточнением исходных данных, которые определяют значения наработки на отказ модулей исследуемой операционной системы и интенсивности переходов исследуемой системы из одного состояния в другое. Для уточнения характеристик надежности модулей системы необходимо построение моделей изменения надежности при выявлении и устранении программных ошибок и соответствующая статистика разработчика операционных систем. Тем не менее во многих случаях предложенный подход достаточно результативен.