УДК 537.322.2

С. Л. Бабичев, К. А. Коньков, А. К. Коньков

Московский физико-технический институт (государственный университет)

Использование пула вычислительных потоков со статическим планированием для оптимизации подсистемы защищённых виртуальных носителей модифицированной защищённой среды

Излагаются аспекты реализации модифицированной защищённой среды, используемой для уменьшения уязвимости в современных операционных системах. Рассматривается подсистема защищённых виртуальных носителей и вопросы эффективности реализации данной подсистемы, в частности максимально полное использование вычислительных ядер. Для этого предлагается механизм пула вычислительных потоков, реализация данного механизма и доказывается отсутствие тупиков в реализации.

Ключевые слова: модифицированная защищённая среда, защищённые виртуальные носители, пул вычислительных потоков, сети Петри, статическое планирование, отсутствие тупиков.

Модифицированная защищённая среда

Система безопасности современных операционных систем имеет ряд ограничений, позволяющих, например, в ряде случаев получить доступ к конфиденциальным данным при наличии физического доступа к носителям информации. Для частичного устранения данного недостатка в среду функционирования операционной системы может быть добавлен компонент, называемый модифицированной защищённой средой (МЗС). Это – программный слой между приложениями операционной системы и оборудованием, позволяющий добавить в обычную среду такие компоненты, как усиленные механизмы аутентификации и авторизации, а также использовать добавочные механизмы защиты от неправомочного применения.

Подсистема защищённых виртуальных носителей – одно из таких средств. Термин *контейнер* будет использоваться для обозначения области внешней памяти, которая требуют дополнительных данных (*ключа*) для предоставления к ней доступа. Для *контейнеров* определяются операции *подключения* и *отключения*. Операция *подключения* устанавливает связь между *контейнером* и доступными объектами файловой системы (защищённый виртуальный носитель). В операционной системе подключенный контейнер представлен в виде виртуального носителя, жёсткого диска или защищённой директории. Операция *отключения* разрывает эту связь. При операциях ввода/вывода на защищённый виртуальный носитель входящие в состав подсистемы компоненты осуществляют криптографическое преобразование данных, основанное на *ключе*, в результате чего *контейнер* содержит только закодированную информацию. Степень защищённости информации, находящейся на контейнере, всецело определяется криптографической стойкостью применяемых алгоритмов.

Подсистема МЗС спроектирована таким образом, что она допускает использование внешних по отношению к подсистеме криптографических провайдеров, работающих в пользовательском контексте вычислительной системы (в отличие от контекста ядра). Данный подход позволяет использовать в контексте ядра только минимально необходимый набор операций, связанных только с управлением вводом/выводом, перенося криптографические операции в пользовательский контекст (рис. 1).

Подсистема МЗС использует модель обработки данных, которая не изменяет количества запросов и не переупорядочивает их. Влияние модели на производительность может быть выражено в первом приближении с использованием двух основных факторов – латентности (независимого от размера запроса времени задержки при прохождении запроса через систему) и пропускной способности (зависимой от размера запроса компоненты).

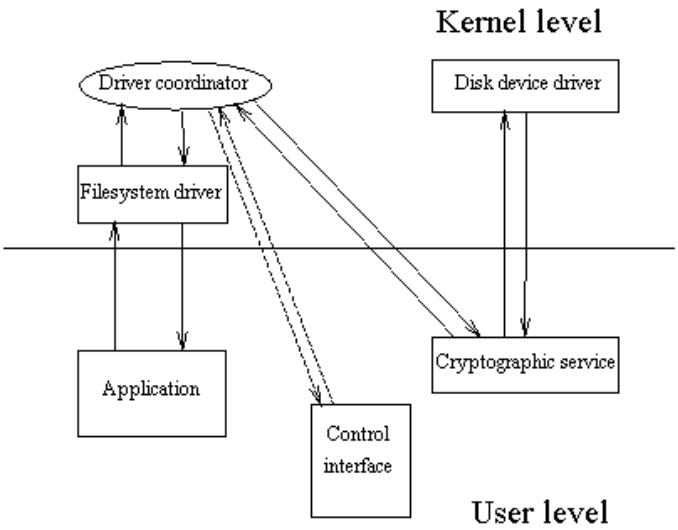

Рис. 1. Архитектура подсистемы защищённых виртуальных носителей модифицированной защищённой среды

Латентность – константная для данной вычислительной системы и алгоритмов обработки данных величина, которая определяется накладными расходами на инициацию и завершение операций преобразования, временем передачи информации внутри системы от одних компонент к другим.

Пропускная способность зависит как от алгоритмов преобразования, количества вычислительных ядер, так и от алгоритмов обработки данных.

Для наименьшего влияния на среду требуется минимизировать фактор латентности и максимизировать пропускную способность.

В данной работе рассматривается способ увеличения пропускной способности системы путём использования нескольких вычислительных ядер в условиях работы с разделяемой памятью, рассмотрен механизм пула потоков вычислений со статическим планированием и доказательство отсутствия тупиков в созданной модели.

Понятие модифицированной защищённой среды подробно рассматривается в работе [7]. В работах [3–6] рассматриваются дополнительные аспекты реализации и функционирования модифицированной защищённой среды.

Архитектура подсистемы ЗВН

Основными компонентами подсистемы ЗВН являются:

- драйвер виртуального устройства, эмулирующий виртуальный носитель на жёстком диске (Д);
- сервис обработки запросов ввода-вывода (СОЗВВ);
- управляющий сервис (УС);
- консоль управления (КУ).

Все компоненты системы, за исключением драйвера виртуального устройства, работают в пользовательском контексте операционной системы.

Оптимизация процесса обработки в обрабатывающем сервисе

Наиболее ресурсоёмкой в обрабатывающем сервисе является операция криптографического запроса ввода-вывода. Латентность обработки постоянна для данной вычислительной системы и определяется только архитектурой механизма передачи информации между компонентой ядра ЗВН (драйвером) и обрабатывающим сервисом, исполняющимся в контексте пользователя.

Таким образом, общее время прохождения запроса ввода-вывода через подсистему ЗВН можно уменьшить, увеличив пропускную способность, то есть скорость криптографического преобразования запроса ввода-вывода.

Для оптимизации данного процесса требуется определить распределение пакетов ввода/вывода по их длинам, соотношение операций чтения и записи, а также типичные скорости обработки пакетов различными криптографическими алгоритмами.

На рисунках 2 и 3 показано распределение частоты размера запросов ввода/вывода для сценариев «Рабочая станция» и «Сервер» соответственно. Информация о распределении была получена с помощью драйвера, входящего в состав ЗВН. Распределение частот по длинам пакетов существенно разное, однако в обоих режимах основную группу составляют запросы размером, большим или равным 32 килобайтам.

Размер запроса	512	≤1024	≤2048	≤4096	≤8192	≤16384	≤32768	≤131072
Частота запроса	0,002	0,013	0,069	0,227	0,310	0,361	0,403	1,000

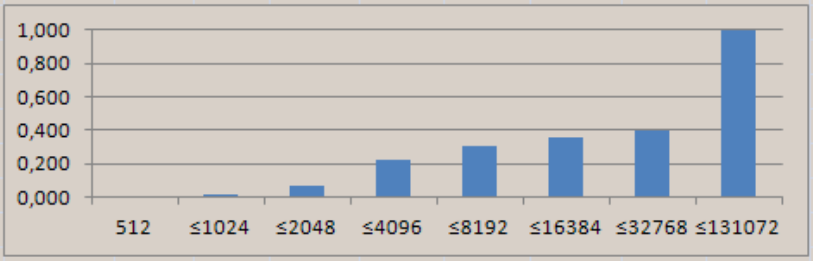

Рис. 2. Распределения размеров запросов ввода/вывода в сценарии *Рабочая станция* по частоте

Размер запроса	512	≤1024	≤2048	≤4096	≤8192	≤16384	≤32768	≤131072
Частота запроса	0,000	0,000	0,002	0,057	0,089	0,127	0,186	1,000

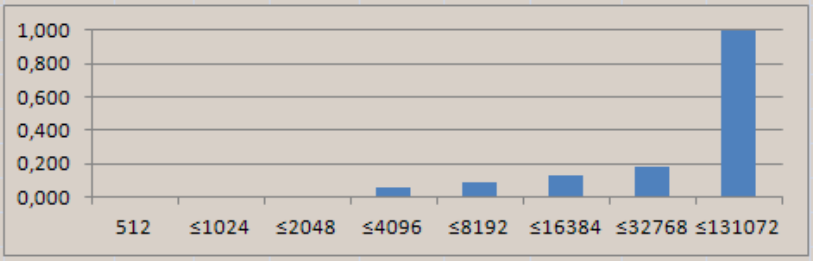

Рис. 3. Распределение размеров запросов ввода/вывода в сценарии *Сервер* по частоте

На рис. 4 приведены приблизительные числа скорости криптографических преобразований по различным алгоритмам, приведённые к одному гигагерцу тактовой частоты одного процессорного ядра процессора Intel Core2.

Алгоритм	AES-128	AES-256	GOST 28147-89
Скорость	39 МБ/с	29 МБ/с	20 МБ/с

Рис. 4. Приблизительная приведённая скорость криптографического преобразования

Из этих таблиц можно оценить приблизительное время обработки одного блока данных в 4 килобайта:

$$\frac{4096}{39 \cdot 10^6} \approx 100 \text{ мкс},$$

и блока данных в 128 килобайт:

$$\frac{131072}{39 \cdot 10^6} \approx 3300 \text{ мкс}.$$

При распараллеливании процесса обработки на 2 или 4 вычислительных ядра время обработки одного подблока будет соответственно составлять 50 и 25 мкс.

При желании распараллелить обработку по нескольким вычислительным ядрам столь малые времена обработки одного запроса накладывают жесткие требования на процесс самой обработки:

- время инициации обработки должно быть заметно меньше времени обработки одного запроса;
- время, затраченное на нахождение вычислительного потока внутри примитивов синхронизации, должно быть заметно меньше времени обработки одного потока.

Между тем испытания показали, что характерное время создания вычислительного потока в вычислительных системах составляет порядка 100 мкс на 1 Гигагерц одного ядра Core2. Характерное время, затрачиваемое на исполнение примитива синхронизации на той же вычислительной системе, примерно равно 1 мкс, то есть меньше на два порядка. Таким образом, механизм с порождением вычислительного потока для каждого запроса здесь неприемлем по соображениям эффективности и должен быть заменён на метод, использующий заранее созданные вычислительные потоки и использующий механизмы синхронизации. Для решения данной проблемы разумно использовать концепцию пула вычислительных потоков.

Пул вычислительных потоков (ПВП)

Пул вычислительных потоков (ПВП) – абстракция, представляющая собой множество вычислительных потоков и предоставляющая методы для их использования. Для наиболее эффективной совместной работы нескольких вычислительных потоков требуется их достаточно тщательное планирование. Поскольку обычно операционная система не даёт доступа к механизмам планирования вычислительных потоков (максимум, что обычно доступно – это указание приоритета вычислительного потока), то для достижения максимальной производительности в критических местах требуется использовать некий аналог статического планирования. Вычислительные потоки должны запускаться, возобновлять работу, останавливать работу и завершаться в удобные моменты времени. Для минимизации накладных расходов будут использованы доступные в операционной системе примитивы синхронизации.

Создание ПВП происходит однократно. Дальнейшая работа с пулом осуществляется следующим образом: в случае появления запроса, допускающего параллельное исполнение на нескольких вычислительных ядрах, выдаётся запрос к ПВП для формирования

начала пакета заданий. Каждый фрагмент кода, допускающий параллельное исполнение, добавляется к пакету заданий. Запрос запуска пакета приводит к назначению на свободные процессоры заданий из очереди. После завершения исполнения задания из очереди новое задание назначается для исполнения на первый освободившийся вычислительный поток. После исполнения всего пакета заданий очередь становится пуста, что и служит признаком конца исполнения. Данный подход позволяет минимизировать как количество переключений контекста ВП, так и совокупное время исполнения запроса в контексте МЗС.

Требования, накладываемые на реализацию ПВП

Существует достаточно большое количество реализаций ПВП. Некоторые (Microsoft) служат, скорее, для обработки запросов на достаточно продолжительные операции, типа управления вводом/выводом, к тому же они нереализуемы в Unix-подобных ОС (для которых тоже имеется соответствующая реализация ЗВН).

Другие (QT) содержат лицензионные ограничения, реализованы только на C++, требуют значительную систему поддержки времени исполнения и не могут быть реализованы внутри ядра ОС.

Было рассмотрено несколько реализаций как Open Source, так и проприетарных, но ни одна из них не удовлетворяла следующим требованиям:

- должны быть поддержаны языки программирования C и C++;
- время, затрачиваемое на накладные расходы, должно быть как можно более малым;
- реализация должна быть максимально переносимой, и поэтому все синхронизирующие примитивы должны быть стандартными и реализуемыми на разных операционных системах;
- не должно быть значительных периодов активного ожидания;
- должна быть возможность использования механизма в ядре операционной системы.

Как следствие, из второго пункта выводится, что количество примитивов синхронизации, исполняемых в одном потоке, должно быть минимальным.

Организация ПВП и методы работы с пулом

Возможны два принципиально разных метода организации ПВП – со статическим планированием и с динамическим планированием.

Если вычислительная работа организуется в очередь, которая обрабатывается пулом, – налицо динамическое планирование (то есть вычислительная работа может исполняться любым свободным потоком из пула). Если перед началом вычислительной работы мы распределяем потоки заранее – налицо статическое планирование.

Динамическое планирование очень удобно в том случае, если у нас заранее неизвестно ни количество вычислительной работы, ни моменты начала отдельных её этапов. Если же и количество, и точный момент начала всех этапов работы нам известны – имеет смысл воспользоваться статическим планированием.

Статическое планирование с точки зрения накладных расходов имеет явные преимущества перед динамическим – не требуются добавочные действия по синхронизации входной/выходной очереди (хотя эти накладные расходы можно свести к минимуму, тщательно используя механизмы активного ожидания).

Все перечисленные выше примеры реализаций ПВП используют динамическое планирование.

В данной работе принят механизм статического планирования, и для работы с ним используются следующие примитивы:

- *PoolId ThreadPoolCreate(numThreads, flags)* – создание ПВП с *numThreads* ожидающими потоками. Возвращает идентификатор пула.

- *JobId ThreadPoolPrepareJob(PoolId)* – начало формирования задания. Возвращается идентификатор задания, который затем используется в дальнейших операциях с ПВП.

- *ThreadPoolAddToJob(JobId, Executor)* – добавить к заданию *JobId* нового исполнителя.

- *ThreadPoolStartJob(JobId)* – запустить задание. Возможны реализации, в которых задачи будут запускаться сразу же при добавлении их к заданию, в этом случае данный примитив будет иметь пустое тело.

- *ThreadPoolWaitForJobDone(JobId)* – дожидается исполнения всех запущенных заданий. Он похож на примитив **join** в мультипоточном программировании, в отличие от которого вычислительные потоки не завершаются, а переходят в свободное состояние (состояние ожидания).

- *ThreadPoolDestroy(PoolId)* – дожидается завершения всех активных вычислительных потоков и затем останавливает их, уменьшая количество вычислительных потоков в системе.

Существует возможность создания нескольких пулов вычислительных потоков. Это полезно при наличии большого количества физических вычислительных ядер и при неуниморфности групп процессоров. Например, данные, располагающиеся в памяти группы процессоров, быстрее обрабатывать именно на этой группе.

Оптимизация использования ПВП

Многопоточная обработка информации при всей своей крайней привлекательности в качестве средства требует интенсивного использования примитивов синхронизации для достижения оптимальной производительности на многопроцессорных/многоядерных системах и соответственно требует тщательного анализа потоков управления во избежание двух основных проблем, свойственных всем системам с параллельным доступом к общим данным – проблеме тупиков и проблеме состояния гонок.

Существует значительное количество техник, применяемых во избежание этих проблем, – методы иерархии ресурсов, метод мониторов и много других. Следование данным методам часто позволяет создать надёжное многопоточное приложение с затратами сравнительно небольшого времени на создание алгоритма. К недостаткам данных методов можно отнести избыточную общность методов – алгоритмы, созданные с использованием этих методов, часто не являются наиболее оптимальными алгоритмами, хотя и можно доказать отсутствие у них проблем синхронизации.

Можно сказать, что создание многопоточных алгоритмов с использованием основных техник недопущения проблем синхронизации имеет сходство с программированием на языках высокого уровня – алгоритмы пишутся легко, они легко верифицируются, но результирующий машинный код обычно требует больше ресурсов для исполнения и работает не самым оптимальным образом.

В задачах, подобных использованию ПВП для реализации ЗВН, где со скоростью алгоритмов параллельной обработки, а тем самым и с качеством реализации этих алгоритмов напрямую связано быстродействие всей вычислительной системы в целом, данный подход не оправдывает себя. Недостаточно тщательное (хотя и правильное по своей структуре) построение алгоритмов параллельной обработки приводит как к замедлению скорости обработки конкретных запросов ввода/вывода, так и к деградации производительности вычислительной системы в целом из-за увеличения количества контекстных переключений потоков для обработки излишних примитивов синхронизации.

Таким образом, для достижения максимальной производительности требуется «ручная» реализация алгоритмов параллельной обработки, а это значит, что после реализации очередной модели обработки она должна пройти верификацию для того, чтобы убедиться,

что эта модель не содержит тупиков и состояния гонок. Проблема обнаружения тупиков обычно имеет более высокий порядок сложности, чем проблема обнаружения состояния гонок, к тому же появление тупика при реализации обработки запроса ввода/вывода внутри ядра операционной системы обычно приводит к отказу части или даже всего механизма ввода/вывода ядра операционной системы.

Для решения задачи верификации такой системы была применена теория сетей Петри. Краткая теория сетей Петри изложена в [1] и [2].

Описание алгоритма функционирования

Всё множество активных вычислительных потоков в данном алгоритме может быть разбито на два типа: первичные, которые занимаются обработкой запросов ввода/вывода, и вторичные, работающие в составе пула потоков. При поступлении нового запроса ввода/вывода поток первого типа анализирует наличие целесообразности и возможности для распараллеливания криптографической обработки. Запросы ввода/вывода небольшого размера обрабатываются в один вычислительный поток, для пакетов больших размеров запрос на криптографическое преобразование разбивается на несколько подзапросов меньшего размера (при наличии свободных вычислительных потоков из пула). Вызовом метода *ThreadPoolPrepareJob* резервируются необходимые ресурсы и подготавливаются структуры данных для дальнейших запросов. По количеству добавочных вычислительных потоков вызываются методы *ThreadPoolAddToJob* для подготовленных запросов. Задание запускается методом *ThreadPoolStartJob*. В это же время работает и основной поток, обрабатывающий часть запроса. После завершения основного потока вызывается метод *ThreadPoolWaitForJobDone*, синхронизирующий основной и дополнительные вычислительные потоки. Результаты параллельной работы остаются в структурах данных, подготовленных ранее. Основной вычислительный поток должен проанализировать коды возвратов обрабатывающих модулей и консолидировать их в единый код возврата.

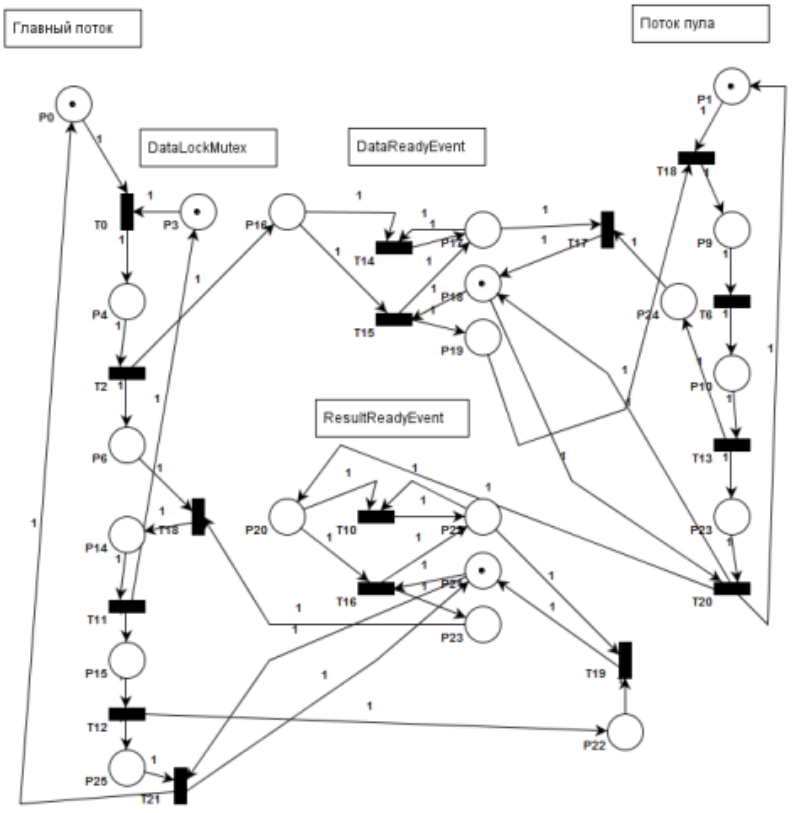

Рис. 5. Модель сети Петри пула вычислительных потоков

Дополнительные вычислительные потоки создаются заранее, однократно, методом *ThreadPoolCreate*. Метод *ThreadPoolDestroy* возвращает ресурсы обратно в операционную систему, завершая все добавочные вычислительные потоки.

Поскольку в данном механизме дополнительные вычислительные потоки не создаются динамически, единственными накладными расходами остаются расходы на синхронизирующие примитивы.

Построение совокупной сети Петри

Исследуемая вычислительная модель была преобразована в модель сети Петри, представленную на рис. 5.

Исследование данной сети Петри показало, что она не имеет тупиков. Таким образом, созданная модель может использоваться для реализации пула вычислительных потоков.

Результаты

Испытания реализованной модели показали следующее изменение производительности операций чтения с закодированного раздела жёсткого диска (Seagate Barracuda 7200.11 320 Gb, процессор Intel Q6600 4×2.400 GHz, 4GB RAM) блоков фиксированного размера:

Размер блока, байт	Некодированное чтение, МБ/с	Однопоточное чтение, МБ/с	Использование пула на 2 потока, МБ/с	Использование пула на 4 потока, МБ/с
4096	54,4	19,4	27,1	29,4
65536	110,1	60,7	88,7	98,4
131072	110,1	65,1	97,4	104,3

Рис. 6. Последовательное чтение

Размер блока, байт	Некодированное чтение, МБ/с	Однопоточное чтение, МБ/с	Использование пула на 2 потока, МБ/с	Использование пула на 4 потока, МБ/с
4096	0,56	0,49	0,55	0,56
65536	12,1	9,1	11,7	12
131072	22	17,7	20,9	21,8

Рис. 7. Случайное чтение

Таким образом, для типичного жёсткого диска в режиме последовательного чтения потери производительности при использовании подсистемы защищённых виртуальных носителей при наиболее распространённом размере записи ввода/вывода составляют не более 3–6%. Для операций случайного ввода/вывода потери производительности составляют около 1% для больших блоков и менее 0,5% для малых блоков.

Выводы

В данной работе рассмотрена модель пула вычислительных потоков со статическим планированием для оптимизации подсистемы защищённых виртуальных носителей модифицированной защищённой среды. Исследование математическими методами показало, что модель не содержит тупиков. В практическом применении, в ситуациях, требующих параллельной обработки запросов заранее неизвестной длины, данная модель позволяет эффективно сократить влияние криптографического преобразования на время обработки запроса.

Литература

1. *Питерсон Дж.* Теория сетей Петри и моделирование систем. — М.: Мир, 1984. – 264 с.
2. *Tadao Murata* Petri Nets: Properties, Analysis and Applications // Proceedings of the IEEE, V. 77 N 4, April 1989.
3. *Бабичев С.Л., Бобьяков А.С., Коньков А.К., Коньков К.А.* Математическая модель защищенной компьютерной системы под управлением Windows // Труды 51-й научной конференции МФТИ. Современные проблемы фундаментальных и прикладных наук. Москва–Долгопрудный, 2008 г. Т 2. С. 56–57.
4. *Бабичев С.Л., Бобьяков А.С., Коньков А.К., Коньков К.А.* Эффективное использование ресурсов вычислительных систем при решении задач информационной безопасности // Труды 52-й научной конференции МФТИ. Современные проблемы фундаментальных и прикладных наук. Часть VII Управление и прикладная математика. Москва–Долгопрудный, 2009. Т. 3. С. 7–8.
5. *Семененко В.Л., Бабичев С.Л., Бобьяков А.С., Коньков А.К., Коньков К.А., Телицын М.А.* Защита корпоративной информации от внутренних угроз на основе метода доверенной загрузки системы // Труды 50-й научной конференции МФТИ. Современные проблемы фундаментальных и прикладных наук. Современные проблемы фундаментальных и прикладных наук. Москва–Долгопрудный, 2007. С. 174–175.
6. *Бабичев С.Л., Коньков А.К., Коньков К.А.*, Об оптимальном использовании ресурсов вычислительной системы для реализации модифицированной защищенной среды // Труды 53-й научной конференции МФТИ. Современные проблемы фундаментальных и прикладных наук. Часть VII Управление и прикладная математика. Москва–Долгопрудный, 2010.
7. *Бабичев С.Л., Коньков А.К., Коньков К.А.*, Дополнительная защита ресурсов операционной системы методом криптографической защиты данных // Моделирование процессов обработки информации: Сб. науч. тр. – М.: МФТИ, 2007. С. 251–259.

Поступила в редакцию 24.11.2011.