УДК 004.415.53

Проверка корректности использования POSIX-сокетов при неблокирующем вводе-выводе

В.В. Казаков [1], В.А. Крищенко [1]

[1] МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Реализация неблокирующего ввода-вывода, необходимого для работы с несколькими сетевыми соединениям в одном потоке исполнения, достаточно трудоемка: ошибки могут привести к блокированию потока выполнения на неопределенное время. Поставлена задача разработки и создания программной реализации метода поиска таких ошибок при тестировании системы. Описан метод, с помощью которого осуществляется анализ системных вызовов процесса и определение ошибок при мультиплексировании сетевых соединений на этапе тестирования. Метод реализован как программный комплекс для ОС на базе ядра Linux.

E-mail: vk.mrvk@gmail.com, kva@bmstu.ru

Ключевые слова: *сетевые службы, тестирование ПО, мультиплексирование сокетов, ввод-вывод.*

Число одновременно поддерживаемых сетевой службой соединений может достигать нескольких тысяч, что приводит к проявлению недостатков стратегии, направленной на создание отдельного потока для каждого соединения. Для организации потока требуется как выделение ресурсов для дескриптора потока в ядре операционной системы (ОС), так и использование некоторого количества памяти в пространстве пользователя, в частности для хранения стека потока. Эффективность работы планировщика задач ОС достаточно резко снижается, если сетевая служба создает сотни однотипных потоков и более, из которых одновременно выполняются лишь несколько. При частом отключении и подключении клиентов играет роль и время, затрачиваемое на создание и уничтожение потоков. В итоге подход «одно соединение — один поток» уже при нескольких сотнях сетевых соединений приводит к существенно нелинейному снижению производительности из-за накладных расходов на формирование рабочих потоков [1].

Наиболее распространенным способом преодоления этой проблемы является стратегия работы с соединениями, предполагающая создание нескольких потоков, каждый из которых опрашивает множество сокетов, считывает из них данные и затем обрабатывает их [2]. Число таких рабочих потоков обычно фиксировано и может быть выбрано исходя из числа логических процессоров в системе. В корректной реализации данной стратегии, называемой мультиплексированием сокетов, следует использовать гарантированно неблокирующий ввод-вывод как минимум при чтении и записи данных из

сетевых сокетов. Блокировка рабочего потока приведет к прекращению обработки всех соединений, за которые был ответственен заблокированный поток, на неопределенное время. В худшем случае это время зависит от поведения клиента сетевой службы.

Таким образом, проверка корректности является актуальной проблемой, но известные существующие средства анализа программного обеспечения (ПО) не имеют специализированных функций для поиска таких ошибок. В работе приведено описание разработки метода проверки корректности реализации мультиплексированного сетевого ввода-вывода в сетевых службах. Программная реализация созданного метода состоит из загружаемого модуля для модифицированного ядра ОС, анализирующего системные вызовы ввода-вывода и выполняющего разработанные алгоритмы поиска ошибок, и прикладной программы, взаимодействующей с пользователем.

Ошибки при реализации мультиплексирования соединений. Стандарт POSIX.1-2008 определяет три системных вызова мультиплексирования для опроса сокетов на предмет возможности неблокирующих операций чтения или записи: `select( )`, `poll( )`, `pselect( )` [3]. При описании метода для краткости под `select( )` будем понимать любой возможный системный вызов мультиплексирования, под `read( )` — любую возможную функцию чтения из сокета, а под `write( )` — любую возможную функцию записи в сокет из ограниченных стандартом POSIX.1-2008.

Известные ошибки мультиплексирования можно подразделить на две группы (рис. 1): критические и некритические. Критические ошибки могут заблокировать поток на сколь угодно длительное время, зависящее от действий клиентской стороны. Последствиями некритических ошибок являются задержки в обслуживании клиентов, но сам рабочий поток не блокируется на неопределенное время.

Под ошибкой блокированием потока или процесса до чтения из готового сокета (см. рис. 1) понимается наличие операции, проводимой на сервере, которая вызывает кратковременное «засыпание» процесса. Примером такой операции может служить необходимость чтения из локального файла.

Чтение из одного и того же сокета в различных рабочих потоках приводит к тому, что нельзя гарантировать неблокирующий характер чтения: когда данные в сокете появятся, несколько потоков приступят к их чтению, но успешно выполнить эту операцию в общем случае может только один поток. Остальные потоки в состоянии сна будут ожидать прихода новых данных от клиента, при этом обслуживаемые ими соединения обрабатываться не будут.

Примером ошибки, приводящей к блокирующему чтению, может служить последовательность вызовов `select( ) – read(fd) – . . . – read(fd)`, где `fd` — дескриптор сокета, о котором вызов `select( )` сообщил как о готовом для чтения. Первое чтение будет неблокирующим, но про второе этого сказать в общем случае нельзя.

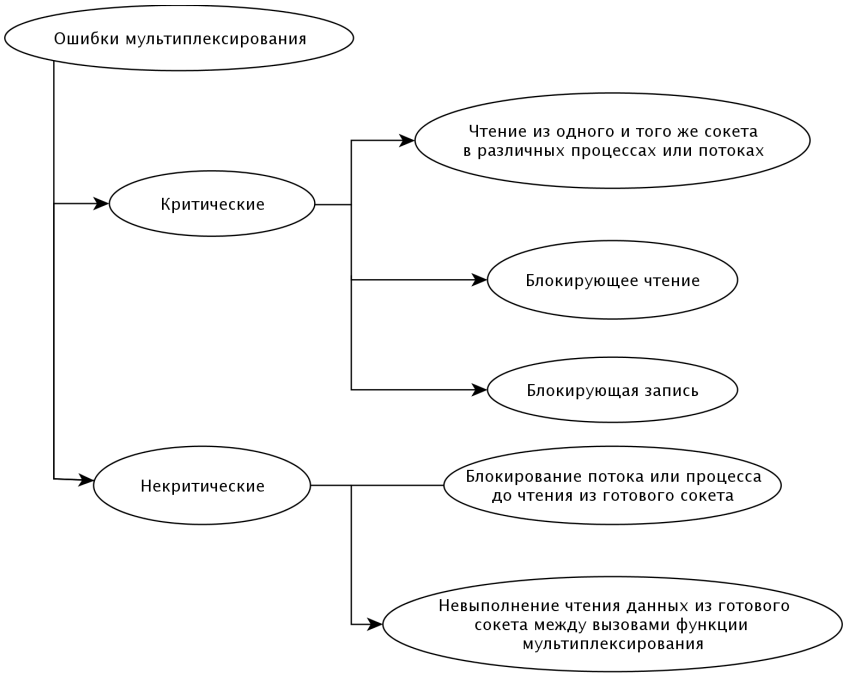

Рис. 1. Ошибки при реализации мультиплексирования сокетов

Блокирующая запись также является критической ошибкой в случае возможного злонамеренного поведения клиента, который, например, не будет допускать передачу ему сетевой службой ответа с результатами обработки запроса. В случае гарантии корректности клиента, т. е. если сетевая служба не доступна через Интернет, блокирующую запись можно рассматривать, скорее, как некритическую ошибку.

Метод поиска ошибок мультиплексирования соединений. При разработке метода поиска ошибок мультиплексирования был выбран динамический подход, предполагающий анализ работы программы в ходе выполнения уже созданных тестов. Хотя необходимость наличия уже созданных человеком тестов является явным недостатком такого подхода, основанные на нем методы позволяют проверять любые приложения, в то время как статический анализ может быть применен для исходного кода сетевых служб с ограничениями.

Неприменимость к сетевым службам статического анализа связана с тем, что типичным языком их разработки является язык C, а их исходный код использует потоки, указатели на функции и нетипизированные указатели. Именно поэтому применение статического анализа весьма затруднено. Например, до момента выполнения программы нельзя однозначно сказать, какая именно функция будет вызвана по указателю, причем в случае использования нетипизированных указателей как параметров эмпирический поиск кандидатов на роль вызываемой функции также затруднен.

Поскольку для разработки метода был выбран динамический подход, входными данными для метода служат анализируемое приложение и набор тестовых данных. Результатом работы метода являются сведения об обнаруженных ошибках (рис. 2). Для работы метода важны следующие моменты, которые можно отследить только внутри ядра операционной системы: изменения состояния сокетов, изменения состояния потока и возникающие в системе блокировки. На основании этих данных можно обнаруживать возникающие ошибки.

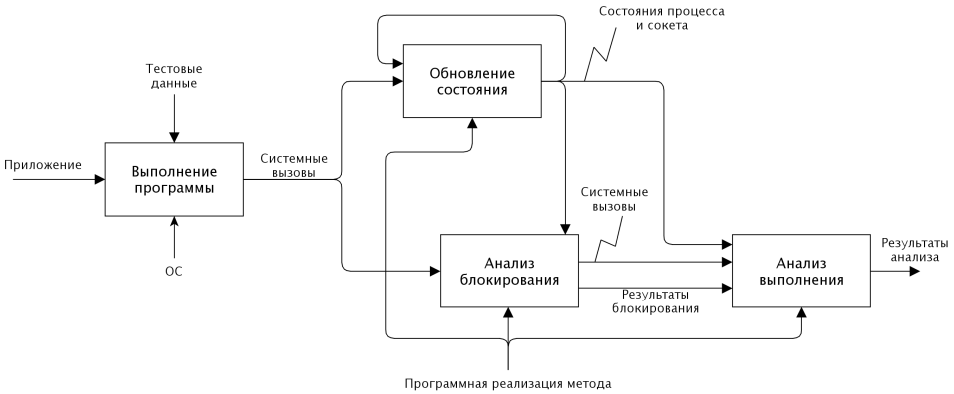

Рис. 2. Схема работы рассматриваемого метода

При выполнении функции чтения сокет может находиться в трех состояниях: «Нет данных», «Ожидание чтения», «Заблокирован до чтения» (рис. 3). В момент создания сокет находится в состоянии «Нет данных». Переходы между состояниями и условия для их осуществления показаны в табл. 1.

Рис. 3. Состояния сокета с точки зрения созданного метода

Таблица 1

Переходы между состояниями сокета

Исходное состояние	Конечное состояние	Событие	Условие
Нет данных	Нет данных	`select( )`	Сокет не готов для неблокирующего чтения
Нет данных	Ожидание чтения	`select( )`	Сокет готов для неблокирующего чтения
Ожидание чтения	Нет данных	`read( )`	—
Ожидание чтения	Заблокирован до чтения	`select( )`	В списке отслеживаемых сокетов для чтения нет данного сокета
Заблокирован до чтения	Заблокирован до чтения	`select( )`	—
Заблокирован до чтения	Нет данных	`read( )`	—

При выполнении функции записи аналогичные состояния выделить нельзя, поскольку если сокет готов для записи, то делать ее не обязательно, а единственная ошибка, определяемая при записи (см. рис. 1), не требует информации о возникавших ранее событиях.

Другим объектом ядра ОС, важным для разрабатываемого метода, является поток. Для разрабатываемого метода он имеет следующие состояния: «Без блокирования», «Потенциальное блокирование», «Происходило блокирование».

Поток начинает работу в состоянии «Без блокирования». При выполнении операции, которая могла заблокировать поток, но не сделала этого, он изменяет свое состояние на «Потенциальное блокирование». Если поток будет переведен в состояние сна планировщиком задач, то его состояние изменится на «Происходило блокирование».

Для системного вызова мультиплексирования блокирование процесса является нормальным действием, поэтому при вызове `select( )` процесс перейдет в состояние «Без блокирования».

Диаграмма переходов между состояниями изображена на рис. 4. Для хранения требуемой методу информации можно добавить нужные поля в структуры ядра Linux [4]. На рис. 5 показаны необходимые изменения структур данных системы.

Рис. 4. Состояния потока с точки зрения созданного метода

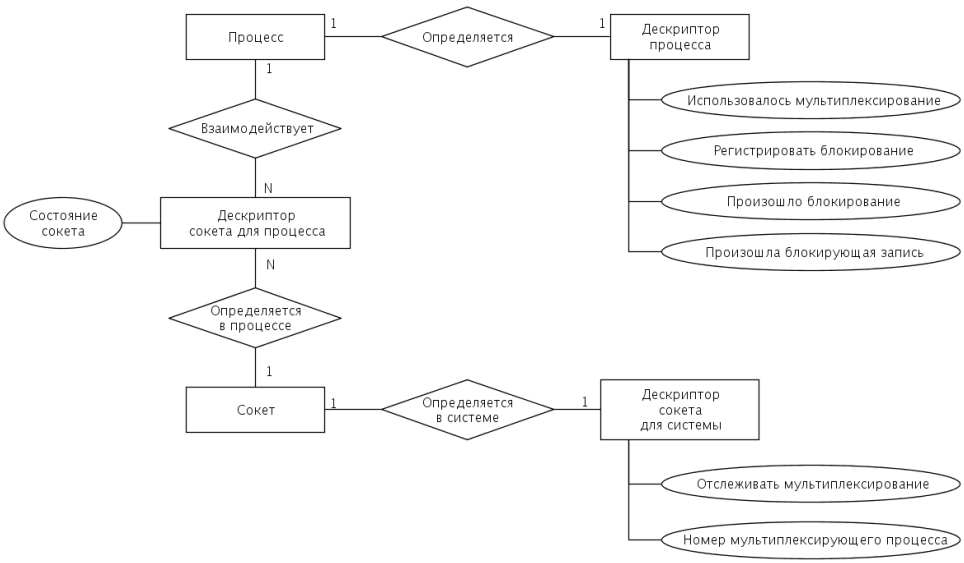

Рис. 5. Атрибуты, добавленные в структуры данных ядра ОС

Каждый процесс и поток в ядре Linux описывается своим дескриптором задачи. Его можно расширить полями, используемыми реализуемым методом. Дополнительно в структуру дескриптора задачи введем флаги:

– использовалось мультиплексирование;
– регистрировать блокирование;
– произошло блокирование;
– произошла блокирующая запись.

Первый флаг позволяет выделить потоки, потенциально анализируемые созданным методом. Третий флаг устанавливается в том случае, если задача была переведена системой в режим сна. Он может быть

установлен, только если уже стоит второй флаг. Такая защита предназначена для использования внутри `select( )`, блокирование которого не является ошибкой. Четвертый флаг показывает, что в процессе была выполнена блокирующая запись. Поскольку состояния процесса рассматриваются между двумя вызовами мультиплексирования, то вызов `select( )` сбрасывает все флаги и устанавливает первый флаг.

В ядре ОС сокет связан с процессом некоторым дескриптором. Его структуру можно также расширить, добавив поле состояния, в котором будет храниться текущее состояние сокета. Кроме того, в ядре ОС имеется дескриптор сокета для системы, не связанный с процессом. Его следует расширить флагом «Отслеживать мультиплексирование» и полем «Номер текущего процесса». Первый флаг необходим, поскольку серверный сокет, созданный для приема входящих соединений, может использоваться разными потоками, которым соответствуют различные дескрипторы задач.

Обнаружение ошибки происходит в системном вызове. В табл. 2 показано, какие из них отвечают за тот или иной тип ошибки. Вызов `select( )` должен определять два вида ошибок. В табл. 3 показаны виды ошибок, признаки, позволяющие сделать вывод о возникновении ошибки, а также данные, используемые для ее выявления.

Таблица 2

Системные вызовы для обнаружения ошибок

Ошибка	Системные вызовы
Чтение из одного и того же сокета в разных процессах или потоках	`select( ), read( )`
Блокирующее чтение	`read( )`
Блокирующая запись	`write( )`
Блокирование потока или процесса до чтения из готового сокета	–
Невыполнение чтения данных между вызовами функции мультиплексирования	`select( )`

Таблица 3

Обнаружение ошибок при вызове select( )

Вид ошибки	Признак ошибки	Используемые данные
Чтение из одного и того же сокета в разных процессах или потоках	Сокет является клиентским. Он обслуживается другим процессом	Уникальный дескриптор сокета. Данные системного вызова
Отсутствие чтения из готового сокета между вызовами `select( )`	Сокет готов для чтения. Находится не в состоянии «Нет данных»	Дескриптор сокета. Данные системного вызова

Также системный вызов мультиплексирования должен изменять состояния сокетов. Те сокеты, которые обслуживаются текущим процессом, готовы для чтения, но не были переданы в функцию для отслеживания чтения, переходят в состояние «Заблокирован до чтения». Готовые для чтения сокеты должны сменить состояние на «Ожидание чтения».

Алгоритм поиска ошибок, выполняемый в начале системного вызова мультиплексирования, показан на рис. 6, выполняемый в его конце — на рис. 7. В системном вызове записи данных в сокет этот метод может выявить ошибку выполнения блокирующей записи, как показано на рис. 8.

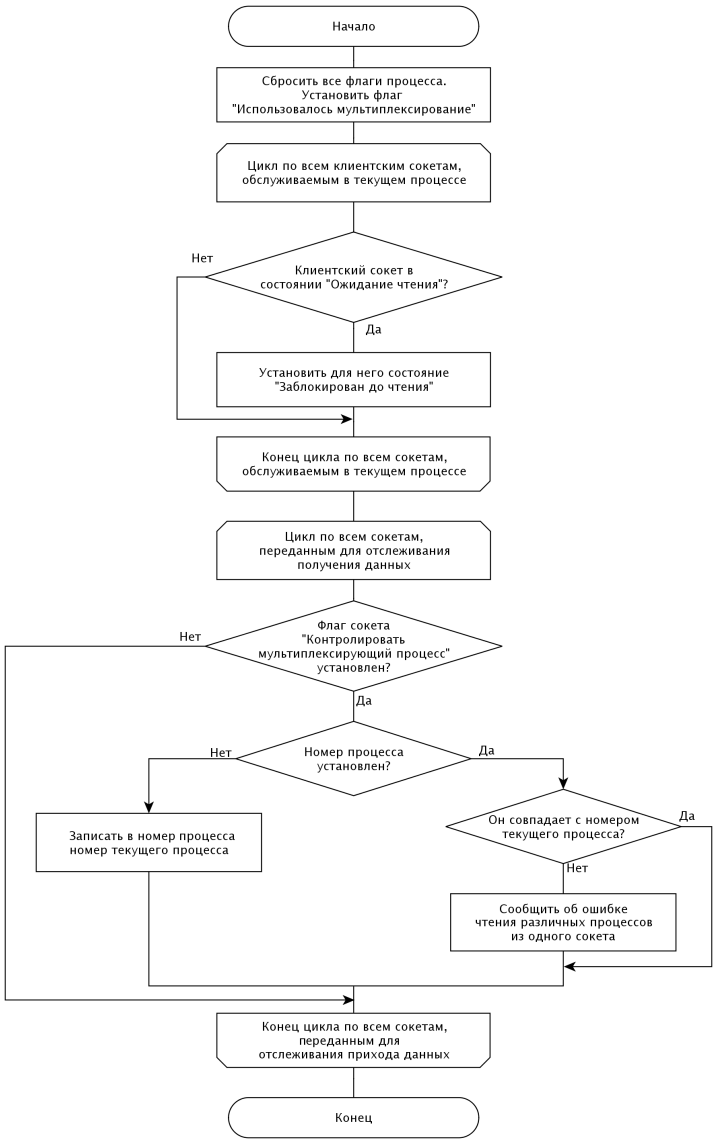

Рис. 6. Алгоритм проверки в начале вызова мультиплексирования

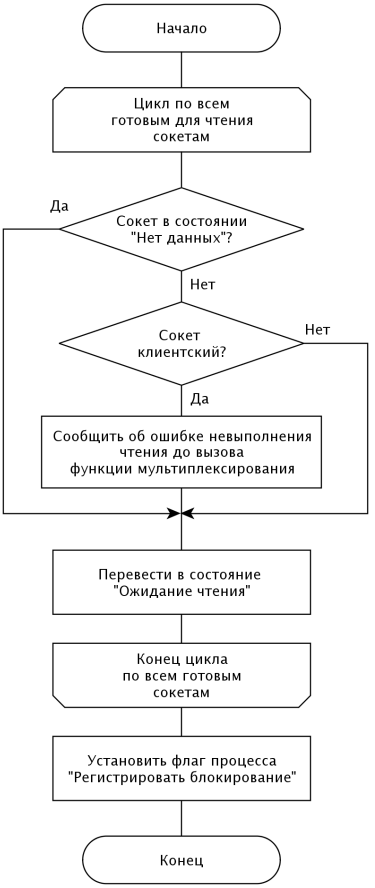

Рис. 7. Алгоритм проверки в конце вызова мультиплексирования

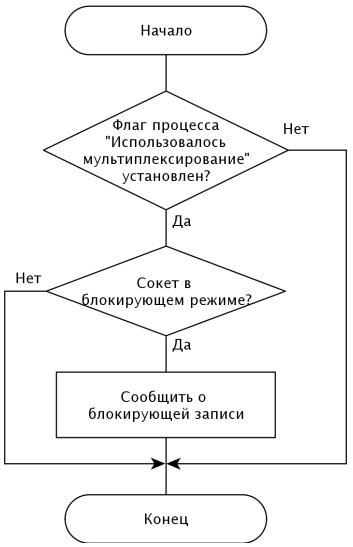

Рис. 8. Алгоритм проверки при системном вызове записи

Перехват системного вызова чтения данных позволяет определять три вида ошибок (табл. 4). Алгоритм, выполняемый реализацией метода в начале метода `read( )`, показан на рис. 9.

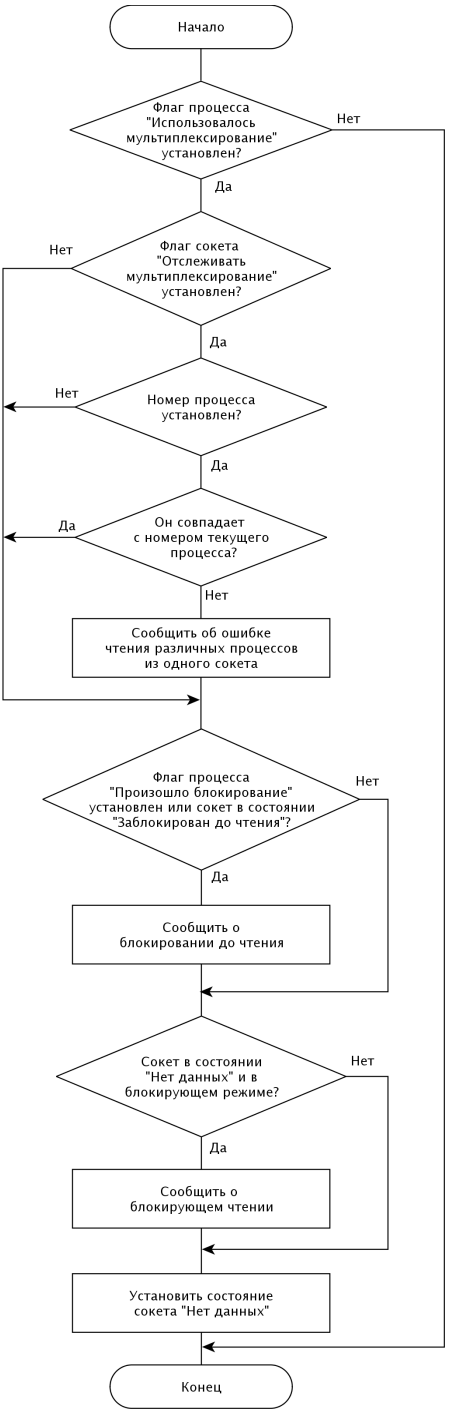

Рис. 9. Алгоритм проверки при системном вызове чтения

Таблица 4

Обнаружение ошибок при вызове read( )

Вид ошибки	Признак ошибки	Используемые данные
Чтение из одного и того же сокета в разных процессах или потоках	Сокет является клиентским. Он обслуживается другим процессом	Уникальный дескриптор сокета. Данные системного вызова
Блокирующее чтение	Сокет находится в состоянии «Нет данных». Сокет в блокирующем режиме	Дескриптор сокета
Блокирование потока или процесса до чтения из готового сокета	Процесс находится в состоянии блокировки	Дескриптор процесса

Необходим способ, позволяющий определить, является сокет клиентским или серверным, т. е. созданным сервером для получения соединений. Последний может использоваться в вызовах `select( )` для ожидания чтения в нескольких процессах или потоках. Клиентские сокеты могут обслуживаться только в одном потоке, иное является критической ошибкой.

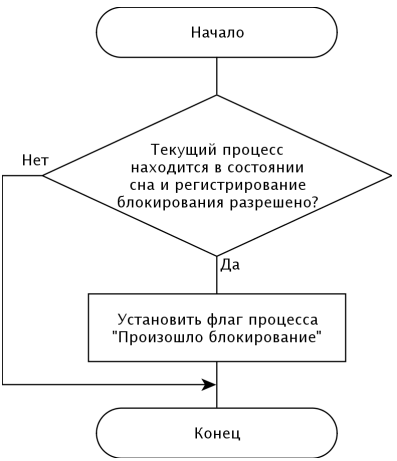

Рис. 10. Действия метода в контексте планировщика задач

Клиентские сокеты должны иметь установленный флаг «Отслеживать мультиплексирование» в своем уникальном дескрипторе. Системным вызовом, который мог бы устанавливать его, может служить вызов `accept( )`.

В диаграмме состояний процесса (см. рис. 4) имеется событие «Блокирование». Чтобы определить факт его наступления, можно

воспользоваться планировщиком задач ядра ОС. Действия, которые следует выполнить созданному методу в контексте планировщика, показаны на рис. 10.

Программная реализация метода. Программный комплекс, реализующий разработанный метод, состоит из модулей (рис. 11): обновления состояний сокетов; отслеживания состояния процесса; анализа; пользовательского интерфейса; связи пользовательского интерфейса с модулем анализа.

Первые два модуля отслеживают изменения системных объектов и предоставляют информацию о них модулю анализа. Задачей модуля анализа является выявление ошибок мультиплексирования. Первые три модуля работают с объектами ядра Linux и поэтому должны находиться в пространстве ядра ОС. Для обмена данными с пользовательским приложением будет использоваться файл в файловой системе /proc. Такие файлы служат для того, чтобы сделать сведения об определенных системных компонентах доступными пользователю [4].

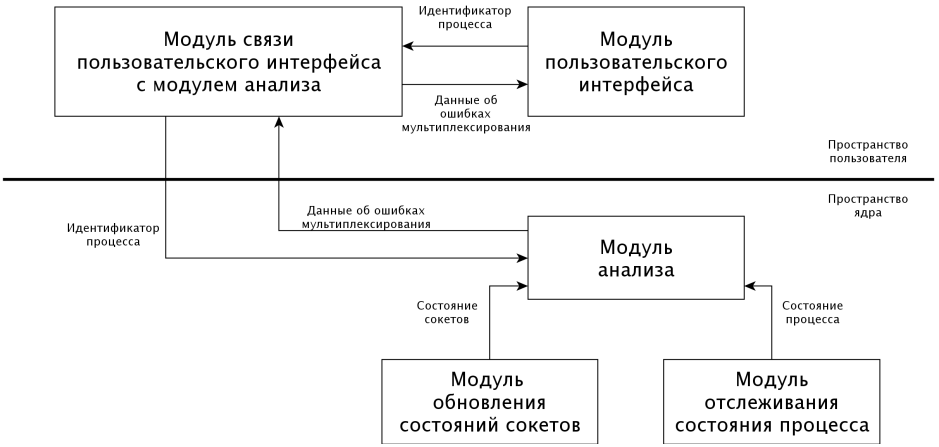

Рис. 11. Структура программной реализации

Остановимся подробнее на реализации части комплекса, работающего в режиме ядра ОС. Метод использует ядро Linux версии 3.2.0.

Процесс и потоки в ядре Linux описываются дескриптором задачи — экземпляром структуры `task_struct`, определенной в заголовочном файле `sched.h` [4]. На рис. 12 показано расширение этой структуры необходимыми для работы метода полями. Каждый флаг, отслеживающий состояние процесса, представлен одним битом в поле `stest_flags`. Для работы с ним используются атомарные битовые операции, определенные в заголовочном файле `bitops.h`.

Файлы, открытые процессом, представляются объектом структуры `file`, которая определена в заголовочном файле `fs.h`. Сокет явля-

ется специальным видом файлового дескриптора [3] и поэтому также представляется объектом этой структуры.

Рис. 12. Расширение структуры `task_struct`

Для одного и того же сокета объект типа `file` в разных процессах может быть различным. Поэтому его нельзя использовать для выявления совместного чтения несколькими процессами из клиентского сокета: у каждого из них этот дескриптор может быть своим. Уникальным объектом, соответствующим сокету, является структура индексного дескриптора `inode` (файл `fs.h`). Ее также следует расширить необходимыми полями, указанными на рис. 5. Поскольку эти данные могут использоваться различными процессами, то их следует защитить с помощью средства синхронизации. Для этого в структуру inode добавляется спин-блокировка, которая позволит обеспечивать монопольный доступ к полям объекта.

В разработанном методе используются системные вызовы для определения ошибок, поэтому требуется найти, какие функции реализуют интересующие системные вызовы. Такие системные вызовы мультиплексирования, как `select( )`, `pselect( )`, `poll( )`, реализуются в файле `select.c`. Целесообразно встроить соответствующий код, реализующий показанные на рис. 6 и 7 алгоритмы, в то место, где системная функция выполняет обход по переданным сокетам. Это делают в функции `do_select( )`. Кроме того, ее же используют в системном вызове `pselect( )`. В случае `poll( )` необходимая для встраивания кода функция носит название `do_poll( )`.

Процесс может использовать различные системные вызовы чтения (`read( )`, `recv( )` и др.) и записи в сокет (`write( )`, `send( )` и др.). Для их перехвата находят ту функцию ядра, которую вызовут все функции чтения или записи. Для чтения такой функцией является `sock_recvmsg( )`, а для записи — `sock_sendmsg( )`. Кроме того, необходимо перехватывать системный вызов `accept( )`. Он определен в файле `af_inet.c`, и функция носит название `inet_accept( )`.

Отслеживать состояния блокирования можно с помощью кода, добавленного в системный планировщик задач: когда задача блокируется для сна, вызывается планировщик, который выбирает новую задачу для выполнения.

Результаты применения метода. Программная реализация метода протестирована и отлажена на наборе тестов, включающем все приведенные на рис. 1 типы ошибок. После тестирования метода были проведены эксперименты, в которых в качестве анализируемых программ выступали различные сетевые службы для POSIX-систем, созданные студентами кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н. Э. Баумана в рамках курсовой работы по дисциплине «Протоколы вычислительных сетей». Поскольку данные серверы были написаны студентами, они обычно имели те или иные ошибки мультиплексирования.

Для каждого приложения выполнялось исследование исходного кода вручную и выявление ошибок мультиплексирования, а также применение разработанного программного комплекса в процессе тестирования приложения. По результатам проведенных экспериментов (табл. 5) можно сделать вывод о том, что метод применим для анализа серверных приложений, использующих мультиплексирование сокетов.

Таблица 5

Результаты экспериментов

Номер эксперимента	Описание сервера	Ожидаемые ошибки	Обнаруженные ошибки
1	SMTP-сервер, использующий в работе `poll( )` и рабочие потоки	Блокирующая запись	Блокирующая запись
2	SMTP-сервер, использующий в работе `poll( )` и рабочие процессы	Блокирующая запись	Блокирующая запись
3	SMTP-сервер, использующий в работе `select( )` и рабочие потоки	Нет ошибок	Нет ошибок
4	SMTP-сервер, использующий в работе `select( )` и рабочие потоки	Блокирующая запись	Блокирующая запись
5	SMTP-сервер, использующий в работе `pselect( )` и рабочие потоки	Блокирование до чтения из готового сокета; блокирующая запись	Блокирование до чтения из готового сокета; блокирующая запись
6	SMTP-сервер, использующий в работе `pselect( )` и рабочие потоки	Нет ошибок	Непрочитанные данные из готового сокета

Окончание табл. 5

Номер эксперимента	Описание сервера	Ожидаемые ошибки	Обнаруженные ошибки
7	SMTP-сервер, использующий в работе `pselect( )`	Блокирование до чтения из готового сокета	Блокирование до чтения из готового сокета
8	Эхо-сервер, чтение и запись в отдельных потоках	Непрочитанные данные из готового сокета	Непрочитанные данные из готового сокета

В результате одного эксперимента было выявлено предположительное ложное срабатывание реализации метода: неправильно установлена ошибка «Непрочитанные данные из готового сокета». В приложении, использованном в эксперименте, системный вызов мультиплексирования определял готовность как дескрипторов сокетов, так и файловых дескрипторов. На одной итерации цикла потока обслуживания вызов `pselect( )` производился дважды с одинаковым набором сокетов. После первого вызова обрабатывались файлы, после второго — сетевые соединения. Вследствие готовности некоторых сетевых соединений уже после первого вызова при втором вызове срабатывал созданный метод. Поскольку логика работы потока не предполагала обслуживания сетевых соединений до второго вызова, это срабатывание является ложным. Таким образом, созданный метод имеет ограничение: предполагается, что после вызова мультиплексирования данные из готового сокета будут прочитаны до следующего вызова мультиплексирования. Это соответствует типичному подходу к обработке сетевых соединений.

Работа выполнена при частичной поддержке Российского фонда фундаментальных исследований (грант № 13-07-00918).

СПИСОК ЛИТЕРАТУРЫ

1. Kegel D. The C10K problem. 1999. URL: http://kegel.com/c10k.html
2. Стивенс. У.Р., Феннер Б., Рудофф Э.М. UNIX: разработка сетевых приложений. СПб.: Питер, 2007. 1039 с.
3. POSIX.1-2008. The system interfaces volume. URL: http://pubs.opengroup.org/onlinepubs/9699919799/
4. Бовет Д., Чезати М. Ядро Linux. М.: BHV, 2007. 1104 с.

Статья поступила в редакцию 25.10.2012