О. В. КАРАВАЕВА

ОПЕРАЦИОННЫЕ СИСТЕМЫ. МЕТОДЫ БОРЬБЫ С ТУПИКАМИ

Учебно-методическое пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

О. В. КАРАВАЕВА

ОПЕРАЦИОННЫЕ СИСТЕМЫ. МЕТОДЫ БОРЬБЫ С ТУПИКАМИ

Учебно-методическое пособие

Киров

2017

УДК 004.451(07)
К21

Допущено к изданию методическим советом факультета автоматики и вычислительной техники ВятГУ в качестве учебно-методического пособия для студентов направлений 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия» всех профилей подготовки

Рецензент:
кандидат технических наук, доцент кафедры
автоматики и телемеханики ВятГУ
В. В. Куклин

Караваева, О. В.

К21 Операционные системы. Методы борьбы с тупиками: учебно-методическое пособие / О. В. Караваева. – Киров: ВятГУ, 2017. – 50 с.

УДК 004.451(07)

Издание предназначено для выполнения лабораторных работ по дисциплине «Операционные системы».

Авторская редакция
Тех. редактор Д. В. Дедюхина

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Тупик (тупиковая ситуация, deadlock, clinch) – это ситуация в системе, когда два и более процесса находятся заблокированном состоянии; в состоянии ожидания события, которое никогда не произойдет.

Тупики возникают из-за конкуренции несвязанных параллельных процессов за ресурсы, а также из-за ошибок в программировании.

В данном пособии будут рассмотрены вопросы обнаружения, предотвращения, обхода тупиков и восстановления системы после обнаружения тупиков.

Как правило, борьба с тупиками – очень дорогостоящее мероприятие. Но, тем не менее, для ряда систем, например, для систем реального времени, это необходимо.

1. ПОНЯТИЕ ТУПИКОВОЙ СИТУАЦИИ

1.1. Теоретические сведения

При организации параллельного выполнения нескольких процессов одной из главных функций операционной системы является решение сложной задачи корректного распределения ресурсов между выполняющимися процессами и обеспечение последних средствами взаимной синхронизации и обмена данными.

При параллельном исполнении процессов могут возникать ситуации, при которых два или более процесса все время находятся в заблокированном состоянии. Самым простым является случай, когда каждый из двух процессов ожидает ресурс, занятый другим процессом. Из-за такого ожидания ни один из процессов не может продолжить исполнение и освободить в конечном итоге ресурс, необходимый другому процессу. Эта тупиковая ситуация называется дедлоком, тупиком или клинчем. Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ждет события, которое никогда не произойдет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса (рис. 1.1).

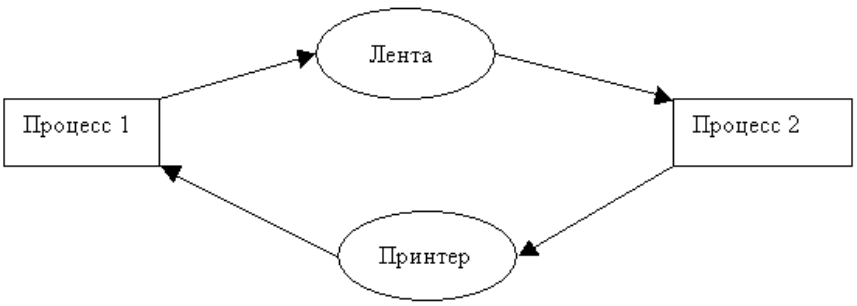

Рис. 1.1. Пример тупиковой ситуации

Тупики чаще всего возникают из-за конкуренции несвязанных параллельных процессов за ресурсы вычислительной системы, но иногда к тупикам приводят и ошибки программирования.

С точки зрения изучения тупиков и способов их предотвращения все ресурсы систему целесообразно разделить на два класса – повторно используемые (или системные) ресурсы (типа RR – reusable resource или SR – system resource) и потребляемые (или расходуемые) ресурсы (типа CR – consumable resource).

Повторно используемый ресурс (SR) есть конечное множество идентичных единиц со следующими свойствами:

– число единиц ресурса постоянно;

– каждая единица ресурса или доступна, или распределена одному и только одному процессу (разделение либо отсутствует, либо не принимается во внимание, так как не оказывает влияния на распределение ресурсов, а значит, и на возникновение тупиковой ситуации);

– процесс может освободить единицу ресурса (сделать ее доступной), только если он ранее получил эту единицу, то есть никакой процесс не может оказывать какое-либо влияние ни на один ресурс, если он ему не принадлежит.

Данное определение выделяет существенные для изучения проблемы тупика свойства обычных системных ресурсов, к которым относятся такие компоненты аппаратуры, как основная память, вспомогательная (внешняя) память, периферийные устройства и, возможно, процессоры, а также программное и информационное обеспечение, такое как файлы данных, таблицы и «разрешение войти в критическую секцию».

Расходуемые ресурсы (CR) обладают отличными от ресурса типа SR свойствами:

– число доступных единиц некоторого ресурса типа CR изменяется по мере того, как приобретаются (расходуются) и освобождаются (производятся) отдельные их элементы выполняющимися процессами, и такое число

единиц ресурса является потенциально неограниченным; процесс «производитель» увеличивает число единиц ресурса, освобождая одну или более единиц, которые он «создал»;

– процесс «потребитель» уменьшает число единиц ресурса, сначала запрашивая и затем приобретая (потребляя) одну или более единиц. Единицы ресурса, которые приобретены, в общем случае не возвращаются ресурсу, а потребляются (расходуются).

Эти свойства потребляемых ресурсов присущи многим синхронизирующим сигналам, сообщениям и данным, порождаемым аппаратурой или программным обеспечением, и могут рассматриваться как ресурсы типа CR при изучении тупиков. В их число входят: прерывания от таймера и устройств ввода/вывода, сигналы синхронизации процессов, сообщения, содержащие запросы на различные виды обслуживания или данные, а также соответствующие ответы.

Для исследования параллельных процессов и, в частности, проблемы тупиков было разработано несколько моделей. Одной из них является модель повторно используемых ресурсов Холта. Здесь система представляется как набор процессов и набор ресурсов, причем каждый из ресурсов состоит из фиксированного числа единиц. Любой процесс может изменять состояние системы с помощью запроса получения или освобождения единицы ресурса.

В графической форме процессы и ресурсы представляются квадратами и кружками соответственно. Каждый кружок содержит некоторое количество маркеров (фишек) в соответствии с существующим количеством единиц этого ресурса. Дуга, указывающая из «процесса» на «ресурс», означает запрос одной единицы этого ресурса. Дуга, указывающая из «ресурса» на «процесс», представляет выделение ресурса процессу. Поскольку каждая единица любого ресурса типа SR может быть выделена одновременно не более чем одному процессу, то число дуг, исходящих из ресурса к различным процессам, не может превышать общего числа единиц этого ресурса. Такая модель называется графом повторно используемых ресурсов.

Одно из состояний примера системы из 2 процессов с ресурсами SR представлено на рис. 1.2.

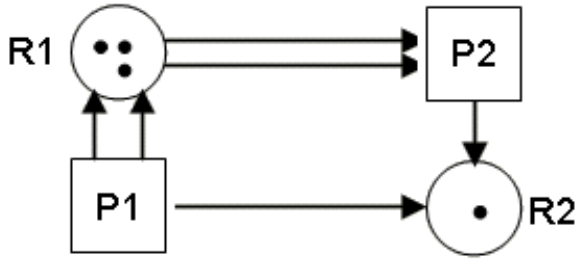

Рис. 1.2. Пример модели Холта для системы из двух процессов

Пусть процесс P1 запрашивает две единицы ресурса R1 и одну единицу ресурса R2. Процессу P2 принадлежат две единицы ресурса R1 и ему нужна одна единица R2. Предположим, что процесс P1 получил бы теперь запрошенную им единицу R2. Если принято правило, по которому процесс должен получить все запрошенные им ресурсы, прежде чем освободить хотя бы один из них, то удовлетворение запроса P1 приведет к тупиковой ситуации: P1 не сможет продолжиться до тех пор, пока P2 не освободит единицу ресурса R1, а процесс P2 не сможет продолжиться до тех пор, пока P1 не освободит единицу R2. Причиной этого дедлока являются неупорядоченные попытки процессов войти в критический интервал, связанный с выделением соответствующей единицы ресурса.

1.2. Построение графа модели в лабораторной установке

В работе рассматривается модель повторно используемых ресурсов Холта – система представляется как множество процессов и ресурсов, причем каждый из ресурсов состоит из фиксированного числа единиц. Любой процесс может изменять состояние системы с помощью запроса, получения или освобождения единицы ресурса.

В лабораторной работе процессы и ресурсы представляются квадратами и кружками соответственно. Каждый круг содержит число в соответствии с существующим количеством единиц этого ресурса. Дуга, указывающая

из «процесса на «ресурс», означает запрос единиц этого ресурса. Дуга, указывающая из «ресурса» на «процесс», представляет выделение ресурса процессу. Для добавления процесса, ресурса или связи используются кнопки на верхней панели окна «Граф модели». Для удаления используется кнопка на нижней панели «Удалить объект».

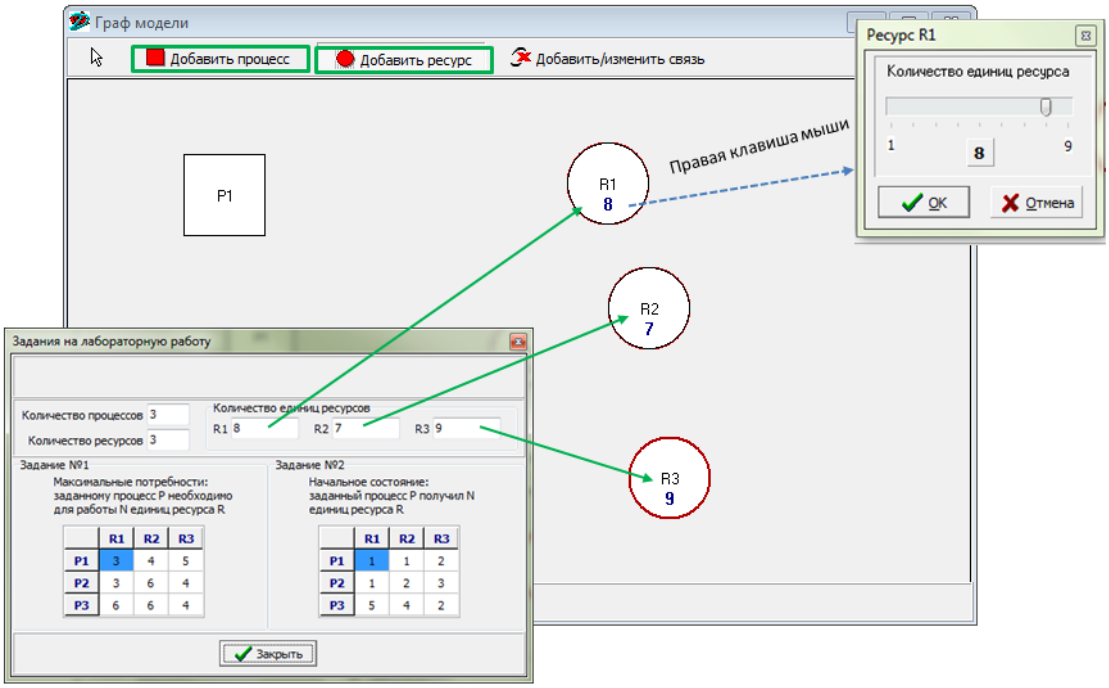

Рис 1.3. Создание графа

Для того, чтобы создать граф модели согласно варианту, задания поместите на форму заданное количество процессов и ресурсов при помощи кнопок на верхней панели. Затем, кликнув в рабочем окне правой кнопкой мыши (или двойным щелчком левой кнопки) на ресурсе, установите заданное количество единиц ресурса (рис. 1.3). После добавления данного количества ресурса для всех дуг граф будет готов (рис. 1.4).

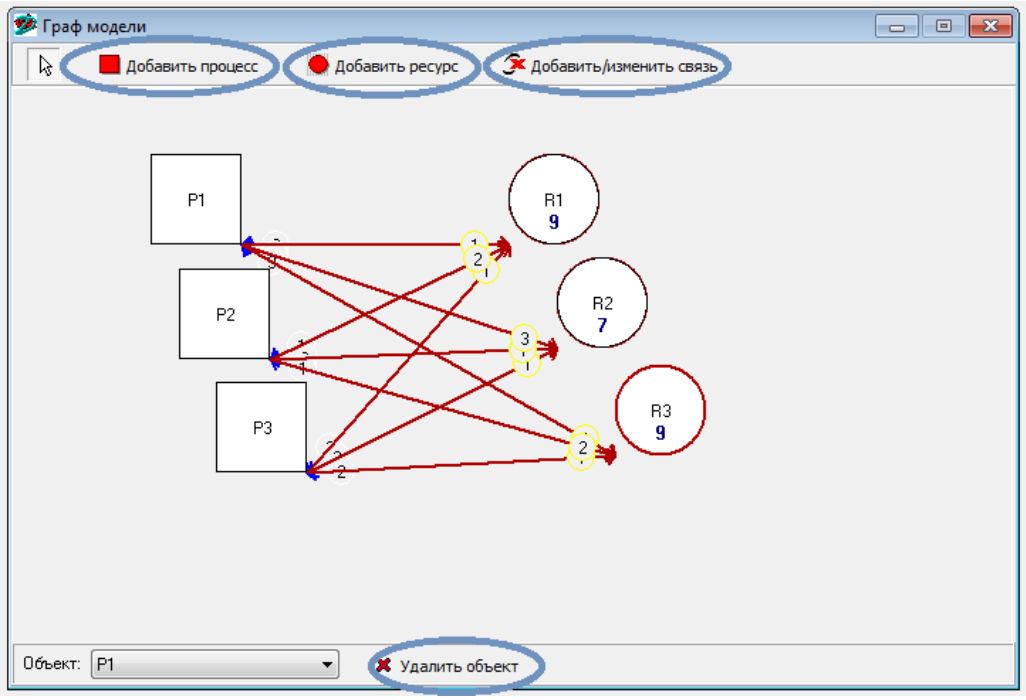

Рис. 1.4. Граф модели

При построении графа необходимо помнить, что каждая единица любого ресурса типа SR (Повторно используемый ресурс) может быть выделена одновременно не более чем одному процессу, соответственно число дуг, исходящих из ресурса к различным процессам, не может превышать общего числа единиц этого ресурса.

2. МЕТОДЫ БОРЬБЫ С ТУПИКАМИ

Проблема тупиков является чрезвычайно серьезной и сложной. В настоящее время разработано несколько подходов и методов разрешения этой проблемы, однако ни один из них нельзя считать панацеей. В некоторых случаях цена, которую приходится платить за то, чтобы сделать систему свободной от тупиков, слишком высока. В других случаях, например, в системах управления процессами реального времени, просто нет иного выбора, поскольку возникновение тупика может привести к катастрофическим последствиям.

В ОС тупики в большинстве случаев возникают при конкуренции процессов за выделение ресурсов последовательного доступа, которые в каждый момент времени отводятся только одному пользователю.

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

– условие взаимоисключения (Mutual exclusion) когда одновременно использовать ресурс может только один процесс;

– условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы;

– условие отсутствия перераспределения (No preemtion) – ресурс, выделенный ранее, не может быть принудительно забран у процесса, освобождены они могут быть только процессом, который их удерживает;

– условие кругового ожидания (Circular wait) – существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение всех четырех условий.

Проблема борьбы с тупиками становится все более актуальной и сложной, по мере развития и внедрения параллельных вычислительных систем. При проектировании таких систем разработчики стараются проанализиро-

вать возможные неприятные ситуации, используя специальные модели и методы. Борьба с тупиковыми ситуациями основывается на одной из трех стратегий:

– предотвращения тупика – основывается на предположении о его чрезвычайно высокой стоимости, поэтому лучше потратить дополнительные ресурсы системы, чтобы исключить вероятность его возникновения при любых обстоятельствах. Предотвращение можно рассматривать как запрет существования хотя бы одного из четырех опасных состояний;

– обход тупика – можно интерпретировать как запрет входа в опасное состояние. Если ни одно из четырех условий не исключено, то вход в опасное состояние можно предотвратить, при наличии у системы информации о последовательности запросов, связанных с каждым параллельным процессом. Доказано, что если вычисления находятся в любом неопасном состоянии, то существует по крайней мере одна последовательность состояний, которая обходит опасное. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, то запрос отклоняется. Если нет, его можно выполнить;

– обнаружение тупика с последующим восстановлением работоспособности системы – распознавание тупика основано на анализе модели распределения ресурсов. Один из алгоритмов использует информацию о состоянии системы, содержащуюся в двух таблицах: таблице текущего распределения (назначения) ресурсов RATBL и таблице заблокированных процессов PWTBL (для каждого вида ресурса может быть свой список заблокированных процессов). При каждом запросе на получение или освобождении ресурсов содержимое этих таблиц модифицируется, а при запросе на ранее выделенный ресурс анализируется в соответствии с алгоритмом. Распознавание тупика требует дальнейшего восстановления:

- принудительный перезапуск системы, характеризующийся потерей информации о всех процессах, существовавших до перезапуска;
- принудительное завершение процессов, находящихся в тупике;

- принудительное последовательное завершение процессов, находящееся в тупике, с последующим вызовом алгоритма распознавания после каждого завершения до исчезновения тупика;
- перезапуск процессов, находящихся в тупике, с некоторой контрольной точки, т. е. из состояния, предшествовавшего запросу на ресурс;
- перераспределение ресурсов с последующим последовательным перезапуском процессов, находящихся в тупике.

Основные издержки восстановления составляют потери времени на повторные вычисления, которые могут оказаться весьма существенными. К сожалению, в ряде случаев восстановление может стать невозможным: например, исходные данные, поступившие с каких-либо датчиков, могут уже измениться, а предыдущие значения будут безвозвратно потеряны.

В следующих главах более подробно рассмотрены эти методы, реализованные в лабораторных установках.

3. ОБХОД ТУПИКА

3.1. Теоретические сведения

Обход тупика можно интерпретировать как запрет входа в опасное состояние. Если ни одно из упомянутых четырех условий не исключено, то вход в опасное состояние можно предотвратить при наличии у системы информации о последовательности запросов, связанных с каждым параллельным процессом. Доказано, что если вычисления находятся в любом неопасном состоянии, то существует, по крайней мере, одна последовательность состояний, которая обходит опасное. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, то запрос отклоняется. Если нет, его можно выполнить. Определение того, является ли состояние опасным или нет, требует анализа последующих запросов процессов.

Рассмотрим следующий пример. Пусть имеется система из трех вычислительных процессов, которые потребляют некоторый ресурс типа SR, который выделяется дискретными взаимозаменяемыми единицами, причем существует всего десять единиц этого ресурса. В табл. 1 приведены сведения о текущем распределении процессами этого ресурса R, об их текущих запросах на этот ресурс и о максимальных потребностях процессов в ресурсе R.

Последний столбец в табл. 1 показывает, сколько еще единиц ресурса может затребовать каждый из процессов, если получит ресурс на свой текущий запрос.

Если запрос процесса A будет удовлетворен первым, то он в принципе может запросить еще одну единицу ресурса R, и уже в этом случае получается тупиковая ситуация, поскольку ни один из процессов не сможет продолжить свои вычисления. Следовательно, при выполнении запроса процесса A мы попадаем в ненадежное состояние (термин «ненадежное состояние» не предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае

некоторой неблагоприятной последовательности событий система может зайти в тупик).

Таблица 1

Пример распределения ресурсов

Имя процесса	Выделено	Запрос	Максимальная потребность	«Остаток» потребностей
A	2	3	6	1
B	3	2	7	2
C	2	3	5	0

Если первым будет выполнен запрос процесса B, то останется свободной еще одна единица ресурса R. Однако если процесс B запросит еще две, а не одну единицу ресурса R, а он может это сделать, то опять получается тупиковая ситуация.

Если же сначала выполнить запрос процесса C и выделить ему не две (как у процесса B), а все три единицы ресурса R и при этом даже не останется никакого резерва, то, поскольку на этом его потребности в ресурсах заканчиваются, процесс C сможет благополучно завершиться и вернуть системе все свои ресурсы. Это приведет к тому, что свободное количество ресурса R станет равно пяти. Теперь уже можно будет выполнить запрос либо процесса B, либо процесса A, но не обоих сразу.

Часто бывает так, что последовательность запросов, связанных с каждым процессом, неизвестна заранее. Но если заранее известен общий запрос на ресурсы каждого типа, то выделение ресурсов можно контролировать. В этом случае необходимо для каждого требования, предполагая, что оно удовлетворено, определить, существует ли среди общих запросов от всех процессов некоторая последовательность требований, которая может привести к опасному состоянию. Данный подход является примером контролируемого выделения ресурса.

3.1.1. Алгоритм Дейкстры

Классическим решением задачи контролируемого выделения ресурса является алгоритм банкира Дейкстры, или алгоритм банкира, рис. 3.1. Этот алгоритм напоминает процедуру принятия решения, может ли банк безопасно для себя дать деньги взаймы. Принятие решения основывается на информации о потребностях клиента (текущих и максимально возможных) и учете текущего баланса банка. Несмотря на то, что этот алгоритм нигде практически не используется, его рассмотрение интересно с методической и академической точек зрения. Текст программы алгоритма банкира приведен в приложении 1.

Пусть существует N процессов, для каждого из которых известно максимальное количество потребностей в некотором ресурсе R. Обозначим эти потребности через Max(i). Ресурсы выделяются не сразу все, а в соответствии с текущим запросом. Считается, что все ресурсы i-го процесса будут освобождены по его завершении. Количество полученных ресурсов для i-го процесса обозначается как Получ(i). Остаток в потребностях i-го процесса на ресурс R обозначим через Остаток(i). Признак того, что процесс может не завершиться – это значение false для переменной Заверш(i). Наконец, переменная Своб_рес будет означать количество свободных единиц ресурса R, а максимальное количество ресурсов в системе определено значением Всего_рес.

Каждый раз, когда какой-то ресурс из числа оставшихся незанятыми ресурсов может быть выделен, предполагается, что соответствующий процесс работает, пока не окончится, а затем его ресурсы освобождаются. Если, в конце концов, все ресурсы освобождаются, значит, все процессы могут завершиться, и система находится в безопасном состоянии. Другими словами, согласно алгоритму банкира, система удовлетворяет только те запросы, при которых ее состояние остается надежным. Новое состояние безопасно тогда и только тогда, когда каждый процесс все же может окончиться. Именно это условие и проверяется в алгоритме банкира. Запросы процессов, приводящие к переходу системы в ненадежное состояние, не выполняются и откладываются до момента, когда его нее же можно будет выполнить.

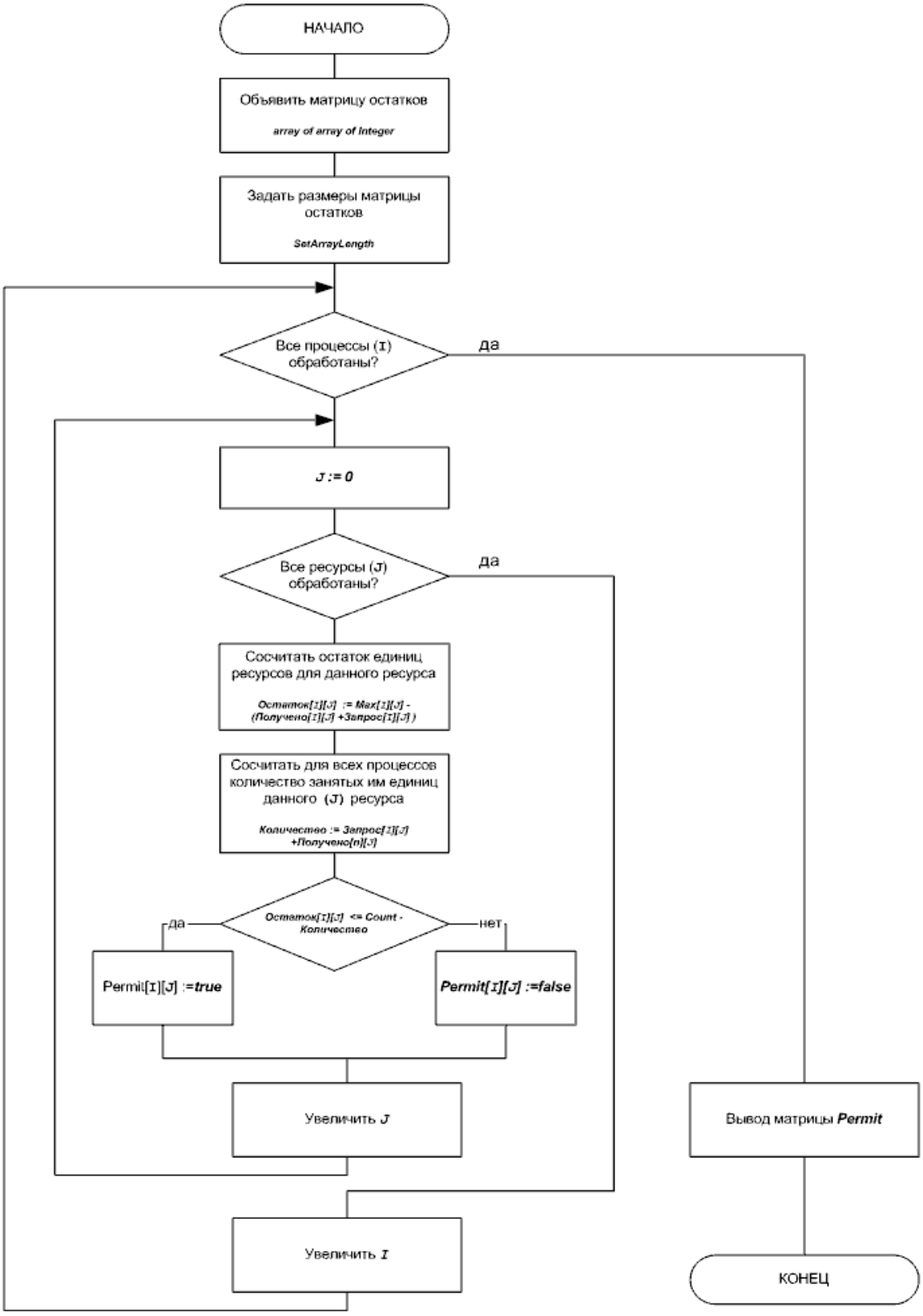

Рис. 3.1. Алгоритм банкира Дейкстры

Алгоритм банкира позволяет продолжать выполнение таких процессов, которым в случае системы с предотвращением тупиков пришлось бы ждать. Хотя алгоритм банкира относительно прост, его реализация может обойтись довольно дорого. Основным накладным расходом стратегии обхода тупика с помощью контролируемого выделения ресурса является время выполнения алгоритма, так как он выполняется при каждом запросе. Причем алгоритм работает медленнее всего, когда система близка к тупику. Необходимо отметить, что обход туника неприменим при отсутствии информации о требованиях процессов на ресурсы.

Рассмотренный алгоритм примитивен, в нем учитывается только один вид ресурса, тогда как в реальных системах количество различных типов ресурсов бывает очень большим. Были опубликованы варианты этого алгоритма для большого числа различных типов системных ресурсов. Однако все равно алгоритм не получил распространения. Причин тому несколько:

– алгоритм исходит из того, что количество распределяемых ресурсов в системе фиксировано, постоянно. Иногда это не так, например, вследствие неисправности отдельных устройств;

– алгоритм требует, чтобы пользователи заранее указывали свои максимальные потребности в ресурсах. Это чрезвычайно трудно реализовать. Часть таких сведений, конечно, могла бы подготавливать система программирования, но все равно часть информации о потребностях в ресурсах должны давать пользователи. Однако, поскольку компьютеры становятся все более дружественными по отношению к пользователям, все чаще встречаются пользователи, которые не имеют ни малейшего представления о том, какие ресурсы им потребуются;

– алгоритм требует, чтобы число работающих процессов оставалось постоянным. Это возможно только для очень редких случаев. Очевидно, что выполнение этого требования в общем случае не реально, особенно в мультитерминальных системах, либо, если пользователь может запускать по несколько процессов параллельно.

3.2. Выполнение лабораторной работы

3.2.1. Выполнение задания № 1

После создания графа модели нужно открыть форму «Таблица распределений» и заполнить «Таблицу потребностей процессов» (максимальные потребности процессов в единицах ресурсов), согласно варианту (рис. 3.2).

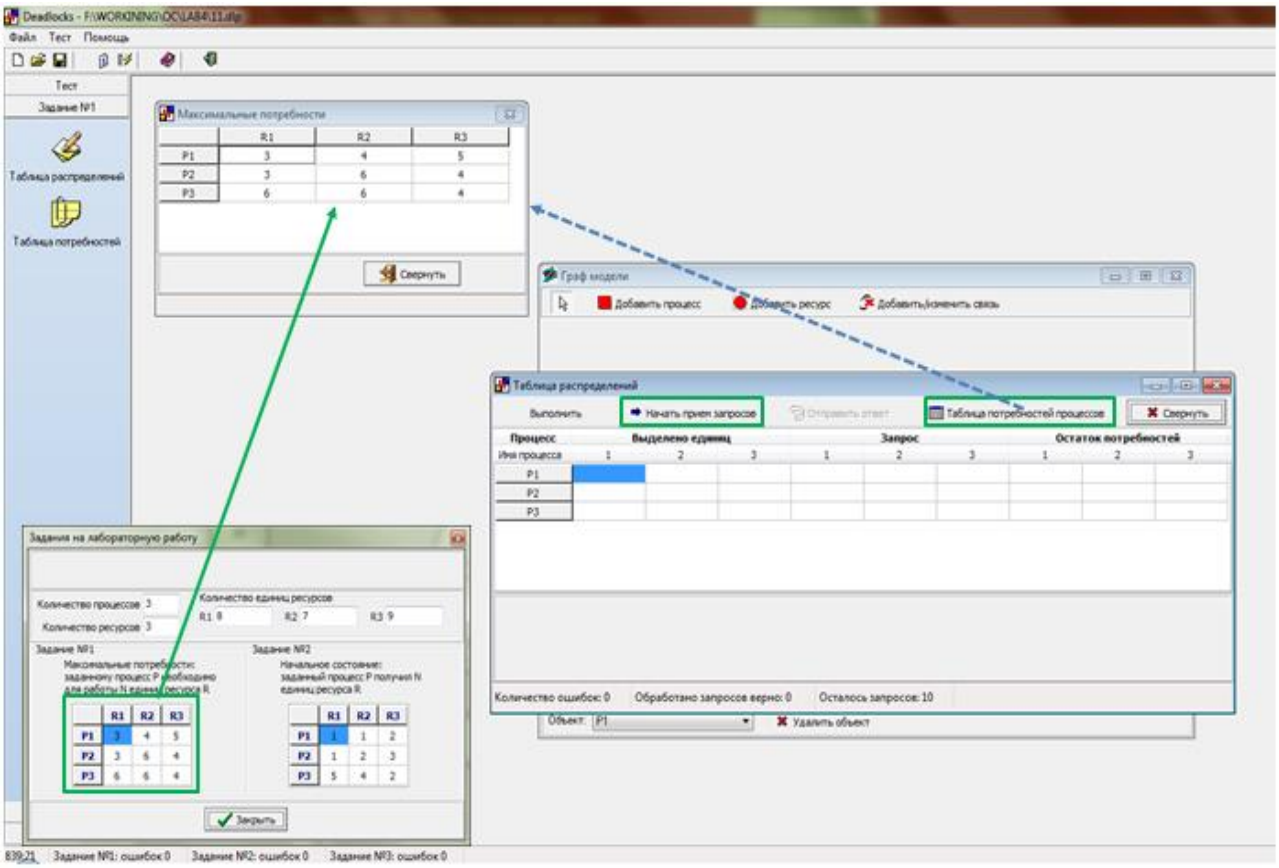

Рис. 3.2. Заполнение максимальных потребностей

Выполнение задания начинается с нажатия на кнопку «Начать прием запросов». Далее необходимо заполнить матрицу «Остатки потребностей», рис. 3.3, согласно формуле:

Остаток[i][j]:= Макс_потребность[i][j] – (Выделено_процессу[i][j] + Запрос[i][j]).

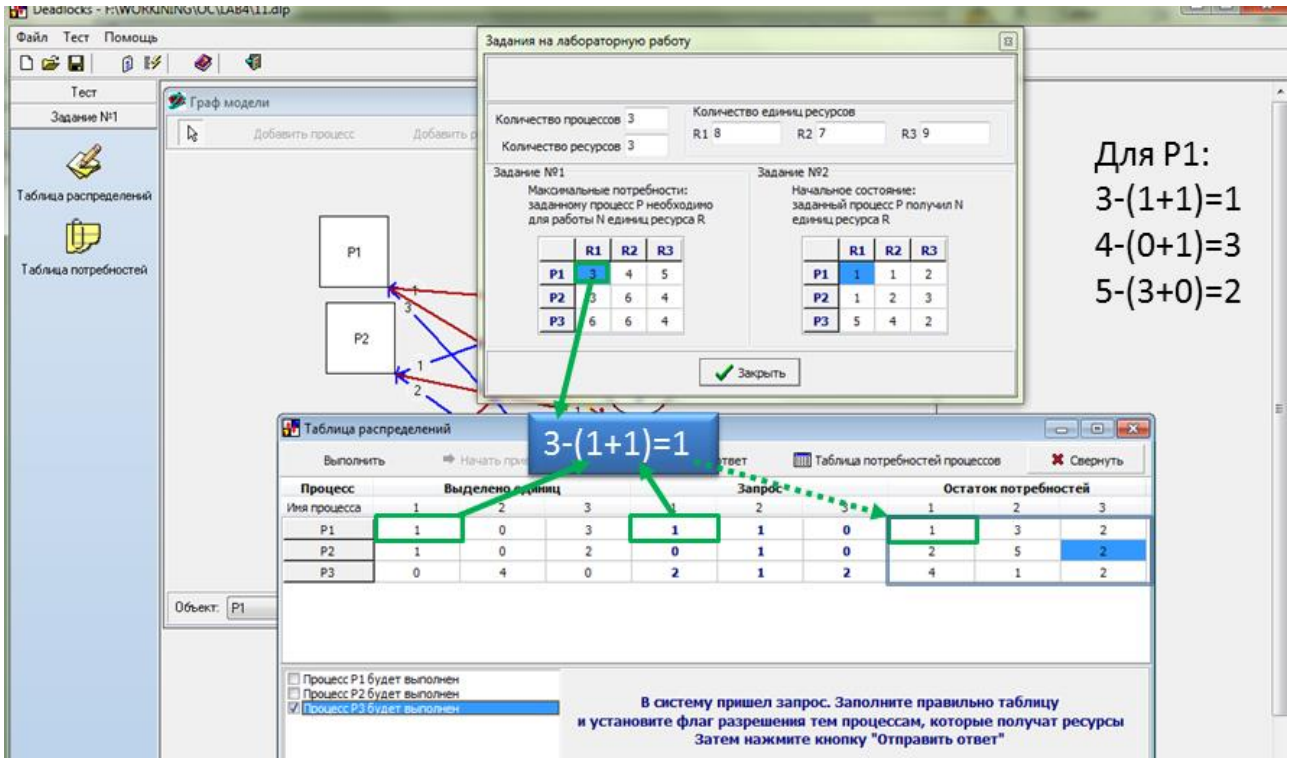

Рис. 3.3. Заполнение остатков потребностей

Затем нужно определить, какие запросы будут удовлетворены и поставить флажок напротив того процесса, чей запрос будет выполнен (рис. 3.4). Для этого необходимо по графу или в окне заданий на лабораторную работу проверить, хватит ли в системе оставшихся ресурсов для завершения процесса, т. е. проверить на истинность выражение: не больше ли остаток потребностей процесса i в ресурсе j разности общего количества единиц ресурса j и занятых (выделено + запрос[i][j]) единиц. Для каждого процесса необходимо вычислить логическое произведение («И») значения этого выражения для всех ресурсов. Если полученное произведение истинно, то запрос процесса может быть удовлетворен.

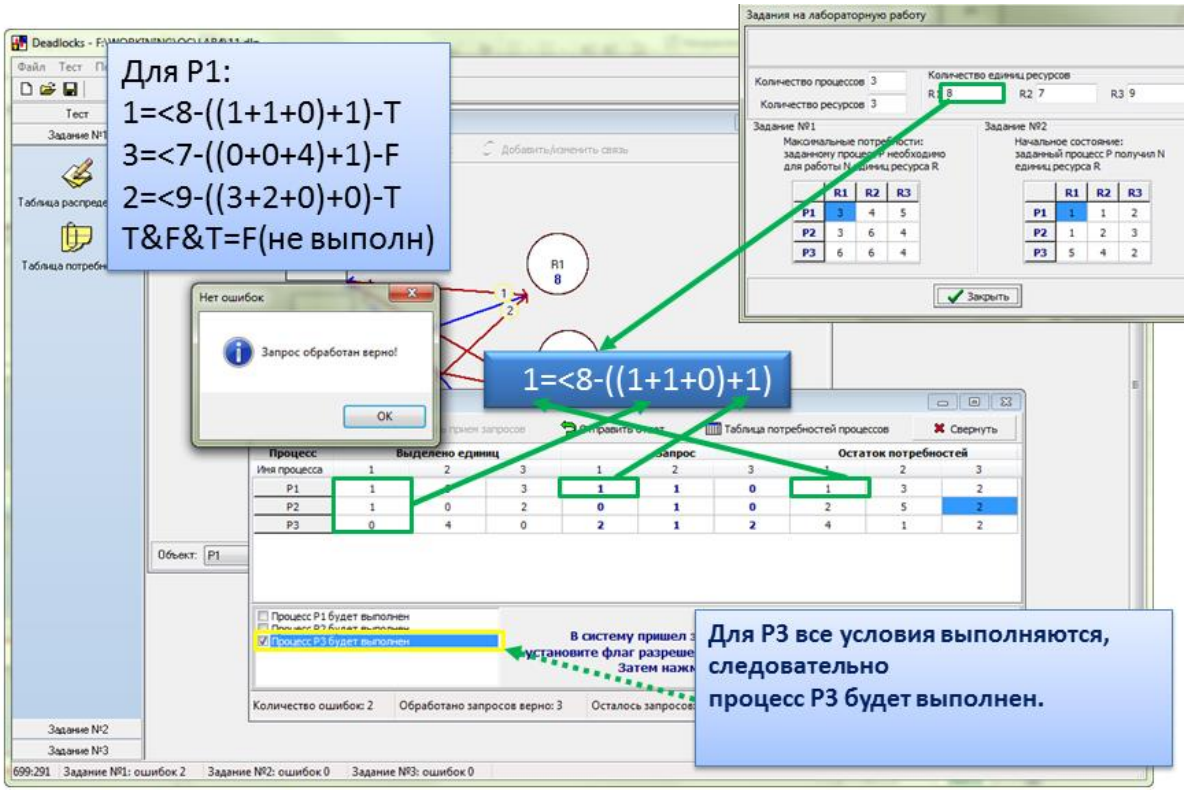

Рис. 3.4. Определение процесса на выполнение

Когда решение о выдаче (или невыдаче) ресурсов процессам будет принято, необходимо «Отправить ответ», нажав на соответствующую кнопку. Автоматически произойдет проверка правильности принятого решения с выдачей сообщения. Данную последовательность действий повторить для некоторого числа запросов.

3.2.2. Выполнение задания № 2

Сначала необходимо подготовить граф. Задать его начальное состояние можно, установив принадлежность некоторых единиц ресурса, согласно варианту. Для этого выбирается кнопка «Добавить/изменить связь», затем выбирается необходимый процесс, а затем ресурс. При этом необходимо установить число занятых единиц ресурса (рис. 3.5).

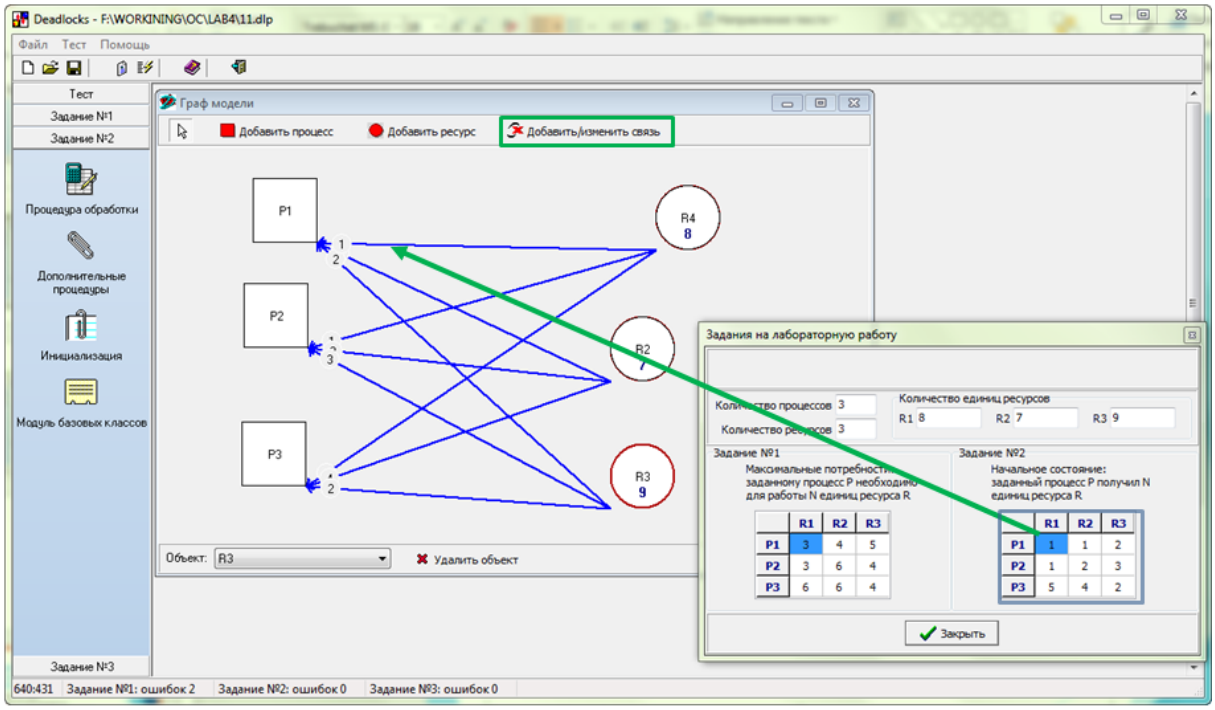

Рис. 3.5. Подготовка графа

Далее надо написать программу в соответствии с алгоритмом банкира Дейкстры (схему алгоритма можно увидеть, нажав на кнопку Алгоритм программы), отладить и откомпилировать написанную программу, нажав «Проверить программу». Затем нажимается кнопка «Выполнить программу: шаг 1» (рис. 3.6).

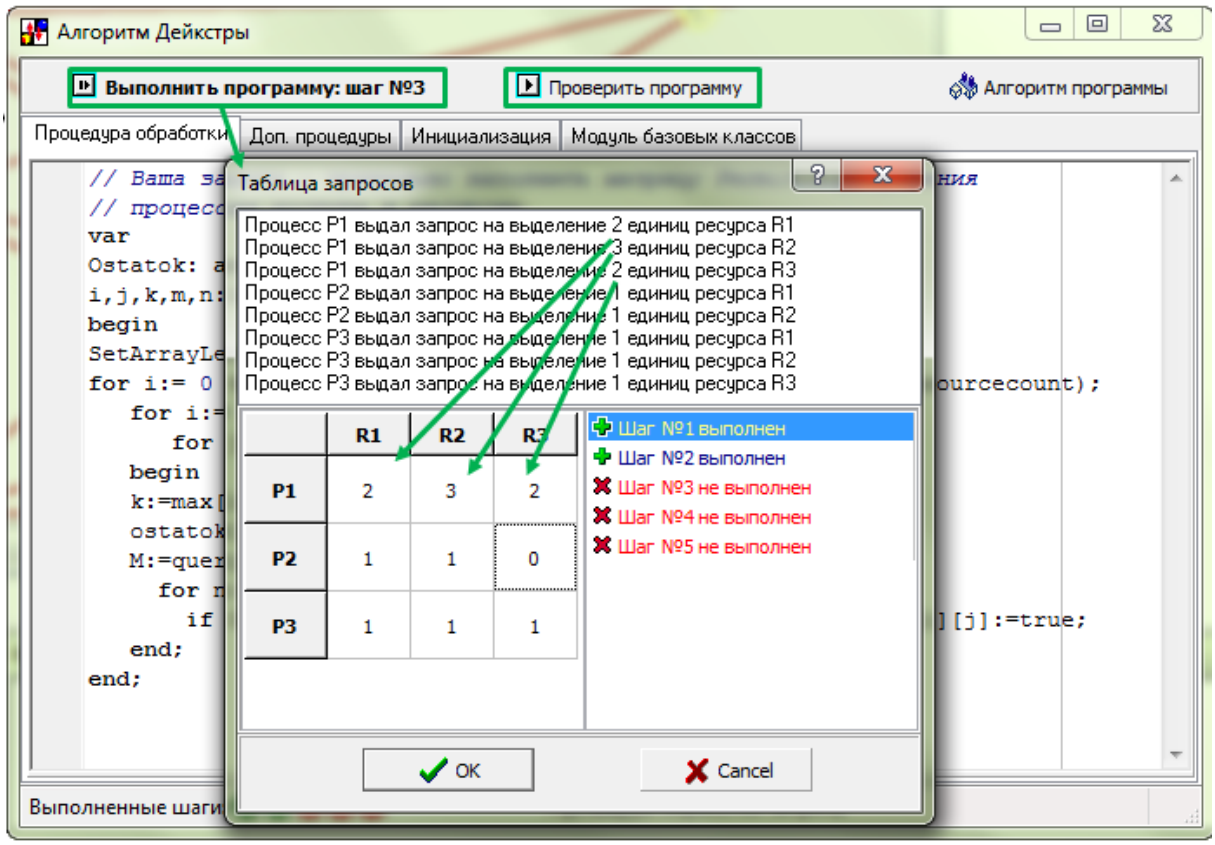

Рис. 3.6. Проверка корректности программы

В появившейся форме заполняется таблица согласно пришедшему запросу. Далее, если программа написана корректно, нужно проверить правильность работы программы еще несколько раз для других запросов.

4. ПРЕДОТВРАЩЕНИЕ ТУПИКА

4.1. Теоретические сведения

Предотвращение тупика основывается на предположении о чрезвычайно высокой его стоимости, поэтому лучше потратить дополнительные ресурсы системы, чтобы исключить вероятность возникновения тупика при любых обстоятельствах. Этот подход используется в наиболее ответственных системах, часто это системы реального времени.

При предотвращении тупиков целью является обеспечение условий, исключающих возможность возникновения тупиковых ситуаций. Часто такой подход ведет к нерациональному использованию ресурсов, но, тем не менее, достаточно часто используется разработчиками ОС. Хавендер в своей работе показал, что возникновение тупика невозможно, если нарушено хотя бы одно из указанных выше четырех необходимых условий, и предложил следующую стратегию предотвращения тупиков.

– каждый процесс должен запрашивать все требуемые ему ресурсы сразу, причем не может начать выполнение до тех пор, пока все они не будут ему предоставлены; заметим, что такой стратегический принцип ведет в ряде случаев к снижению эффективности системы;

– если процесс, удерживающий определенные ресурсы, получает отказ в удовлетворении запроса на дополнительные ресурсы, этот процесс должен освободить свои первоначальные ресурсы и при необходимости запросить их снова вместе с дополнительными ресурсами. Этот принцип предотвращает возникновение условия неперераспределяемости; Одним из серьезных недостатков этой стратегии является возможность бесконечного откладывания;

– введение линейной упорядоченности по типам ресурсов для всех процессов – другими словами, если процессу выделены ресурсы данного типа, в дальнейшем он может запросить только ресурсы более далеких по порядку

типов; этот принцип Хавендера исключает круговое ожидание, однако, отрицательно сказывается на возможности пользователя свободно и легко писать прикладные программы, т. е. приводит к нарушению дружественности ОС.

Предотвращение можно рассматривать как запрет существования опасных состояний. Поэтому дисциплина, предотвращающая тупик, должна гарантировать, что какое-либо из четырех условий, необходимых для его наступления, не может возникнуть.

Условие взаимного исключения можно подавить путем разрешения неограниченного разделения ресурсов. Это удобно для повторно вводимых программ и ряда драйверов, но совершенно неприемлемо к совместно используемым переменным в критических интервалах.

Условие ожидания можно подавить, предварительно выделяя ресурсы. При этом процесс может начать исполнение, только получив все необходимые ресурсы заранее. Следовательно, общее число затребованных параллельными процессами ресурсов должно быть не больше возможностей системы. Поэтому предварительное выделение может привести к снижению эффективности работы вычислительной системы в целом. Необходимо также отметить, что предварительное выделение зачастую невозможно, так как необходимые ресурсы становятся известны процессу только после начала исполнения.

Условие отсутствия перераспределения можно исключить, позволяя операционной системе отнимать у процесса ресурсы. Для этого в операционной системе должен быть предусмотрен механизм запоминания состояния процесса с целью последующего восстановления. Перераспределение процессора реализуется достаточно легко, в то время как перераспределение устройств ввода/вывода крайне нежелательно.

Условие кругового ожидания можно исключить, предотвращая образование цепи запросов. Это можно обеспечить с помощью принципа иерархического выделения ресурсов. Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать

ресурсы только на более высоком уровне. Он может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях. После того как процесс получил, а потом освободил ресурсы данного уровня, он может запросить ресурсы на том же самом уровне. Пусть имеются процессы ПР1 и ПР2, которые могут иметь доступ к ресурсам R1 и R2, причем R2 находится на более высоком уровне иерархии. Если ПР1 захватил R1, то ПР2 не может захватить R2, так как доступ к нему проходит через доступ к R1, который уже захвачен ПР1. Таким образом, создание замкнутой цепи исключается. Иерархическое выделение ресурсов часто не дает никакого выигрыша, если порядок использования ресурсов, определенный в описании процессов, отличается от порядка уровней иерархии. Тогда ресурсы будут использоваться крайне неэффективно.

Другой способ – присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке возрастания номеров. Тогда круговое ожидание возникнуть не может.

Небольшой вариацией этого алгоритма будет нумерация в возрастающем порядке не ресурсов, а запросов процесса. После последнего запроса и освобождения всех ресурсов можно разрешить процессу опять осуществит первый запрос.

Очевидно, что невозможно найти порядок, который удовлетворит всех.

В целом стратегия предотвращения тупиков – это очень дорогое решение проблемы, и она используется нечасто.

4.2. Выполнение задания № 3

В третьем задании необходимо предотвратить появление тупиков. Поэтому дисциплина, предотвращающая тупик, должна гарантировать, что какое-либо из четырех условий, необходимых для его наступления, не может возникнуть:

1. Условие взаимного исключения

В соответствии с требованиями и приоритетами процессов необходимо расставить галочки в окне разрешения процессов, при этом необходимо учи-

тывать, что такие ресурсы как сканер, принтер, магнитная лента и память на запись могут быть распределены только одному процессу.

При заполнении таблицы нужно нажать на кнопку «Проверить» Произойдет проверка правильности принятого решения с выдачей сообщения, при удачном решении переходом к следующему условию (рис. 4.1).

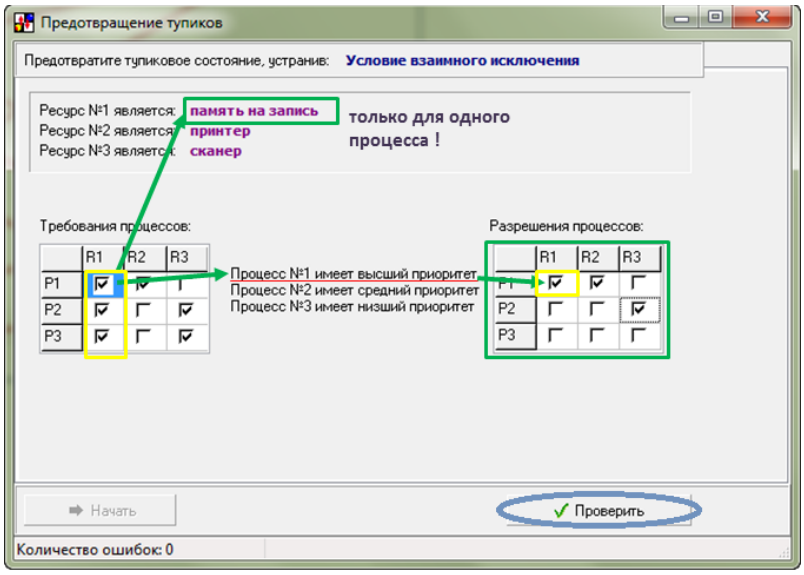

Рис. 4.1. Условие взаимного исключения

2. Условие ожидания

В соответствии с требованиями и приоритетами процессов необходимо выбрать нужное состояние (выделение ресурсов или ожидание). Для этого необходимо проверить условие: Количество (оставшихся) ресурсов >= Требования процессов[i][j], если условие истинно, то необходимо выделение ресурсов процессу. Количество оставшихся ресурсов нужно вычислить по формуле Количество (оставшихся) ресурсов[i][j]:= Количество ресурсов[i][j]– Требования процессов[i][j].

После принятия решения о выдаче (невыдаче) ресурсов процессам нужно нажать на кнопку «Проверить», произойдет проверка правильности принятого решения с выдачей сообщения, с переходом при удачном решении к следующему условию (рис. 4.2).

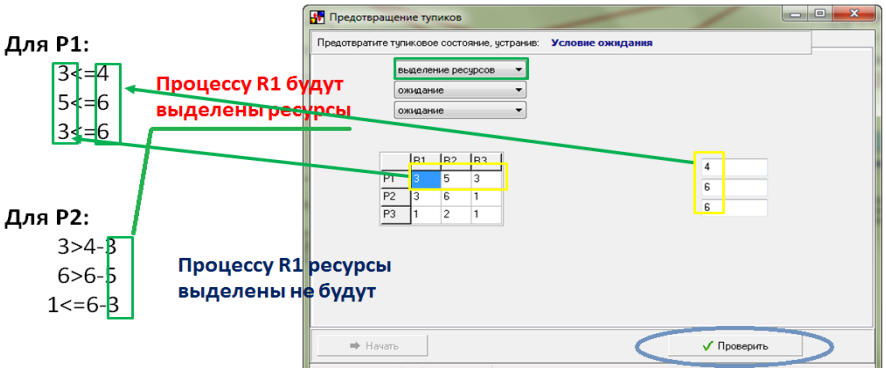

Рис. 4.2. Условия ожидания

3. <u>Условие отсутствия перераспределения</u>

В соответствии с требованиями и приоритетами процессов необходимо выбрать нужное состояние (ввод в состояние ожидания, завершение или выполнение процесса). Сначала необходимо проверить нельзя ли завершить i-й процесс. Для этого необходимо проверить условие: Остаток ресурсов + Выделено единиц ресурсов процессам < Требования процессов[i][j] и, если условие истинно, то необходимо завершить процесс. После того, как i-й процесс завершился необходимо выделенные ему единицы ресурсов вернуть в систему, то есть прибавить к остаткам.

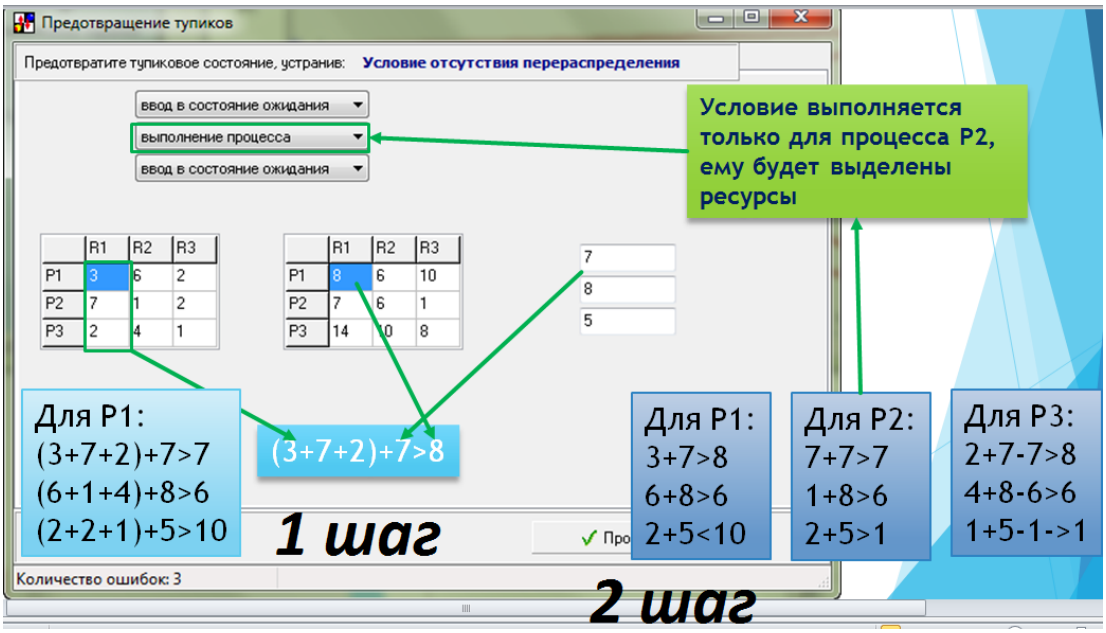

Рис. 4.3. Условие отсутствие перераспределения

Сразу после завершения i-го процесса, необходимо выбрать нужное состояние (ввод в состояние ожидания или выполнение процесса). Для этого нужно проверить условие: Требования процессов [i][j] – Выделено ресурсов [i][j]<= Остатка ресурса. Если условие истинно, то необходимо выделить ресурсы процессу.

Когда решение о выдаче (невыдаче или завершении) ресурсов процессам будет принято, необходимо проверить правильность решения, нажав на соответствующую кнопку. Автоматически произойдет проверка правильности принятого решения с выдачей сообщения об ошибке или, при удачном решении, переходом к следующему условию (рис. 4.3).

4. Условие кругового ожидания

В соответствии с требованиями процессов необходимо присвоить всем ресурсам уникальные номера(приоритеты) (рис. 4.4). Для этого необходимо вычислить дефицит ресурсов (по столбцам) по формуле: дефицит ресурсов:= Количество ресурсов[i][j]– сумма единиц ресурсов[i][j], наименьший дефицит имеет наименьший приоритет(1).

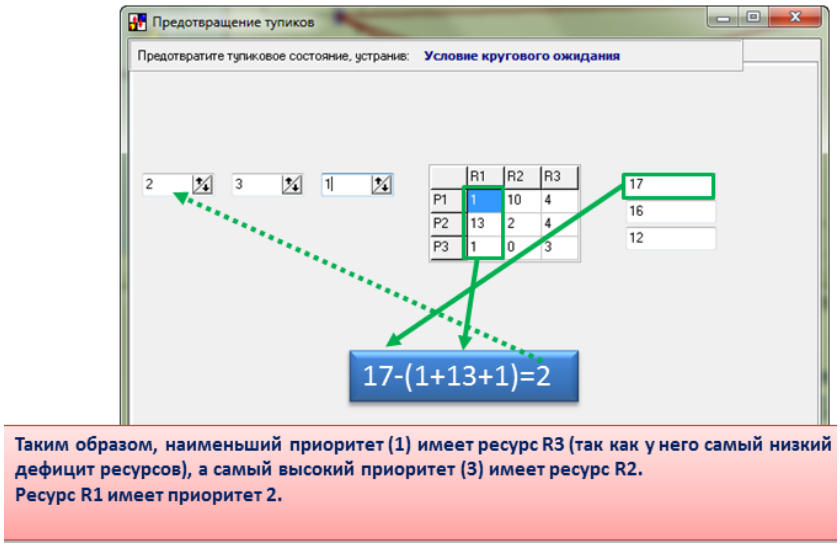

Рис. 4.4. Условие кругового ожидания

Когда решение о присвоении уникальных номеров ресурсам будет принято, необходимо проверить правильность решения, нажав на соответствующую кнопку. Автоматически произойдет проверка правильности принятого решения с выдачей сообщения.

5. ОБНАРУЖЕНИЕ ТУПИКА

5.1. Теоретические сведения

5.1.1. Обнаружение тупиков

Чтобы распознать тупиковое состояние, необходимо для каждого процесса определить, сможет ли он когда-либо снова развиваться, то есть изменять свои состояния. Так как нас интересует возможность развития процесса, а не сам процесс смены состояния, то достаточно рассмотреть только самые благоприятные изменения состояния.

Очевидно, что незаблокированный процесс (он только что получил ресурс и поэтому не заблокирован) через некоторое время освобождает все свои ресурсы и затем благополучно завершается. Освобождение ранее занятых ресурсов может «разбудить» некоторые ранее заблокированные процессы, которые, в свою очередь, развиваясь, смогут освободить другие ранее занятые ресурсы. Это может продолжаться до тех пор, пока либо не останется незаблокированных процессов, либо какое-то количество процессов все же останется заблокированными. В последнем случае, если существуют заблокированные процессы при завершении указанной последовательности действий, начальное состояние S является состоянием тупика, а оставшиеся процессы находятся в тупике в состоянии S. В противном случае S не есть состояние тупика.

5.1.2. Обнаружение тупиков посредством редукции графа повторно используемых ресурсов

Наиболее благоприятные условия для незаблокированного процесса P, могут быть представлены редукцией (сокращением) графа повторно используемых ресурсов. Редукция графа может быть описана следующим образом:

– граф повторно используемых ресурсов сокращается процессом P, который не является ни заблокированной, ни изолированной вершиной,

с помощью удаления всех ребер, входящих в вершину P; и выходящих из p|. Эта процедура является эквивалентной приобретению процессом P| неких ресурсов, на которые он ранее выдавал запросы, а затем освобождению всех его ресурсов. Тогда P, становится изолированной вершиной;

– граф повторно используемых ресурсов не редуцируется, если он не может быть сокращен ни одним процессом;

– граф ресурсов типа SR является полностью сокращаемым, если существует последовательность сокращений, которая удаляет все дуги графа.

Приведем лемму, которая позволяет предложить алгоритмы обнаружения тупика.

Лемма. Для ресурсов типа SR порядок сокращений несуществен; все последовательности ведут к одному и тому же несокращаемому графу.

Доказательство. Допустим, что лемма неверна. Тогда должно существовать некоторое состояние S, которое сокращается до некоторого несокращаемого состояния S1 с помощью последовательности seq1 и до несокращаемого состояния S2 – с помощью последовательности seq2 так, что S1 ≠ S2 (то есть все процессы в состояниях S1и S2 или заблокированы, или изолированы).

Если сделать такое предположение, то мы приходим к противоречию, которое устраняется только при условии, что S1 = S2. Действительно, предположим, что последовательность seq1 состоит из упорядоченного списка процессов (P1, P2, ..., Pk). Тогда последовательность seq1 должна содержать процесс P, который не содержится в последовательности seq2. В противном случае S1=S2, потому что редукция графа только удаляет дуги, уже существующие в состоянии S, а если последовательности seq1 и seq2 содержат одно и то же множество процессов (пусть и в различном порядке), то должно быть удалено одно и то же множество дуг. И доказательство по индукции покажет, что P ≠ Pi (i = 1, 2,.... k), что приводит к указанному нами противоречию.

– P ≠ P1, так как вершина S может быть редуцирована процессом P1, а состояние S2 должно, следовательно, также содержать процесс P1.

– пусть P ≠ Pi, (i = 1, 2,..., j). Однако, поскольку после редукции процессами Pi, (i= 1, 2, ..., j) возможно еще сокращение графа процессом Pj + 1, это же самое должно быть справедливо и для последовательности seq2 независимо от порядка следования процессов. То же самое множество ребер графа удаляется с помощью процесса Pi. Таким образом, P ≠ Pj+1.

Следовательно, P ≠ Pi для i = 1, 2,..., k и P не может существовать, а это противоречит нашему предположению, что S1 ≠ S2. Следовательно, S1=S2.

Теорема о тупике. Состояние S есть состояние тупика тогда и только тогда, когда граф повторно используемых ресурсов в состоянии S не является полностью сокращаемым.

Доказательство.

а) Предположим, что состояние S есть состояние тупика и процесс Pi находится в тупике в S. Тогда для всех Sj, таких что S —*—> Sj процесс Pi заблокирован в Sj (по определению). Так как сокращения графа идентичны для серии операций процессов, то конечное несокращаемое состояние в последовательности сокращений должно оставить процесс Pi блокированным. Следовательно, граф не является полностью сокращаемым.

б) Предположим, что S не является полностью сокращаемым. Тогда существует процесс Pi, который остается заблокированным при всех возможных последовательностях операций редукции в соответствие с леммой. Так как любая последовательность операций редукции графа повторно используемых ресурсов, оканчивающаяся несокращаемым состоянием, гарантирует, что все ресурсы типа SR, которые могут когда-либо стать доступными, в действительности освобождены, то процесс Pi навсегда заблокирован и, следовательно, находится в тупике.

Следствие 1. Процесс Pi не находится в тупике тогда и только тогда, когда серия сокращений приводит к состоянию, в котором Pi не заблокирован.

Следствие 2. Если S есть состояние туника (по ресурсам типа SR), то, по крайней мере, два процесса находятся в тупике в S.

Из теоремы о тупике непосредственно следует и алгоритм обнаружения тупиков. Нужно просто попытаться сократить граф по возможности эффективным способом; если граф полностью не сокращается, то начальное состояние было состоянием тупика для тех процессов, вершины которых остались в несокращенном графе. Рассмотренная нами лемма позволяет удобным образом упорядочивать сокращения.

Граф повторно используемых ресурсов может быть представлен или матрицами, или списками. В обоих случаях экономия памяти может быть достигнута использованием взвешенных ориентированных мультиграфов (слиянием определенных дуг получения или дуг запроса между конкретным ресурсом и данным процессом в одну дугу с соответствующим весом, определяющим количество единиц ресурса).

Рассмотрим вариант с матричным представлением. Поскольку граф двудольный, он представляется двумя матрицами n*m. Одна матрица – матрица распределения D = ||di j||, в которой элемент dij отражает количество единиц Rj ресурса, распределенного процессу Pi, то есть dij = |(Rj,Pi)|, где (Rj, Pi) – дуга между вершинами Rj и Pi, ведущая из Rj в Pi. Вторая матрица – матрица запросов N = ||nij||, где nij = |(Pi, Rj)|.

В случае использования связанных списков для отображения той же структуры можно построить две группы списков. Ресурсы, распределенные некоторому процессу Pi, связаны с Pi указателями: Pi®(Rx, dx)®(Ry, dy)®...®(Rz, dz), где Rj – вершина, представляющая ресурс, а dj – вес дуги dj=|(Rj,Pi)|. Подобным образом и ресурсы, запрошенные процессом Pi, связаны вместе.

Аналогичные списки создаются и для ресурсов (списки распределенных и запрошенных ресурсов) Ri®(Pu, du)®(Pv, nv)®...®(Pw, nw), где (Pj,Ri)|.

Для обоих представлений удобно также иметь одномерный массив доступных единиц ресурсов (r1, r2, ...rm), где ri указывает число доступных (нераспределенных) единиц ресурса Ri , т. е. Ri = |Ri| – S |(Ri,Pk)|

Простой метод прямого обнаружения тупика заключается в просмотре по порядку списка (или матрицы) запросов, причем, где возможно, производятся сокращения дуг графа до тех пор, пока нельзя будет сделать более ни одного сокращения. При этом самая плохая ситуация возникает, когда процессы упорядочены в некоторой последовательности P1,P2, ... , Pn, а единственно возможным порядком сокращений является обратная последовательность, то есть Pn, Pn – 1, ... , P2, P1, а также в случае, когда процесс запрашивает все m ресурсов. Тогда число проверок процессов равно:

$$n + (n – 1) + ... + 1 = n * (n + l) / 2,$$

причем каждая проверка требует испытания m ресурсов. Так время выполнения такого алгоритма в наихудшем случае пропорционально *m * n2*.

Более эффективный алгоритм может быть получен за счет хранения некоторой дополнительной информации о запросах. Для каждой вершины процесса P, определяется так называемый счетчик ожиданий wi отображающий количество ресурсов (не число единиц ресурса), которые в какое-то время вызывают блокировку процесса. Кроме этого, можно сохранять для каждого ресурса запросы, упорядоченные по размеру (числу единиц ресурса). Тогда следующий алгоритм сокращений, записанный на псевдокоде, имеет максимальное время выполнения, пропорциональное m * n.

```
For all PÎL do Begin
For all RjÎ|(Rj,P)|>0 do Begin
rj:=rj+|(Rj,P)|;
For all PiÎ 0<|(Pi,Rj)| rj do Beginwi:=wi -1;
If wi=0 then L:=LÈ{Pi}
End
```

```
End
End
DeadLock:=Not(L={P1,P2,...,Pn});
For all
```

Здесь L – это текущий список процессов, которые могут выполнять редукцию графа. Можно сказать, что L := {Pj | wi=0}. Программа выбирает процесс P из списка L, процесс P сокращает граф, увеличивая число доступных единиц rj всех ресурсов Rj, распределенных процессу P, обновляет счетчик ожидания wj каждого процесса Pj, который сможет удовлетворить свой запрос на частный ресурс Rj, и добавляет Pj к L, если счетчик ожидания становится нулевым.

5.1.3. Обнаружение тупика по наличию замкнутой цепочки запросов

Структура графа обеспечивает простое необходимое, но не достаточное условие для тупика. Для любого графа G = <X, E> и вершины xÎ X пусть P(x) обозначает множество вершин, достижимых из вершины x, то есть:

$$P(x) = \{ y \mid (x. y) \hat{I} E \} U \{ z \mid (y, z) \hat{I} E \& y \hat{I} P(x) \}.$$

Можно сказать, что в ориентированном графе потомством вершины x, которое мы обозначаем как P(x), называется множество всех вершин, в которые ведут пути из x.

Тогда если существует некоторая вершина x Î X : x Î P(x), то в графе G имеется цикл.

Теорема 1. Цикл в графе повторно используемых ресурсов является необходимым условием тупика.

Для доказательства этой теоремы можно воспользоваться следующим свойством ориентированных графов: если ориентированный граф не содержит цикла, то существует линейное упорядочение вершин, такое, что если

существует путь от вершины i к вершине j, то i появляется перед j в этом упорядочении.

Теорема 2. Если S>не является состоянием тупика и S— pi—> ST, где ST есть состояние тупика в том и только в том случае, когда операция процесса Pi есть запрос и Pi находится в тупике в S T.

Это следует понимать таким образом, что тупик может быть вызван только при запросе, который не удовлетворен немедленно. Учитывая эту теорему, можно сделать вывод, что проверка на тупиковое состояние может быть выполнена более эффективно, если она проводится непрерывно, то есть по мере развития процессов. Тогда надо применять редукцию графа только после запроса от некоторого процесса Pi и на любой стадии необходимо сначала попытаться сократить с помощью процесса Pi. Если процесс Pi смог провести сокращение графа, то никакие дальнейшие сокращения не являются необходимыми.

Ограничения, накладываемые на распределители, па число ресурсов, запрошенных одновременно, и количество единиц ресурсов, приводят к более простым условиям для тупика.

Пучок (или узел) в ориентированном графе G = <X, E> – это подмножество вершин ZÎ X, таких что V x Î Z, P(x) = Z, то есть потомством каждой вершины из Z является само множество Z. Каждая вершина в узле достижима из каждой другой вершины этого узла, и узел есть максимальное подмножество с этим свойством. Пример узла на модели Холта изображен на (рис. 5.1).

Следует заметить, что наличие цикла – это необходимое, но не достаточное условие для узла. Пусть даны два подграфа (рис. 5.2). Подграф а представляет собой пучок (узел), тогда как подграф представляет собой цикл, но узлом не является. В узел должны входить дуги, но они не должны из него выходить.

Если состояние системы таково, что удовлетворены все запросы, которые могут быть удовлетворены, то существует простое достаточное условие существования тупика. Эта ситуация возникает, если распределители ресур-

сов не откладывают запросы, которые могут быть удовлетворены, а выполняют их по возможности немедленно (большинство распределителей следует этой дисциплине).

Состояние называется фиксированным, если каждый процесс, выдавший запрос, заблокирован.

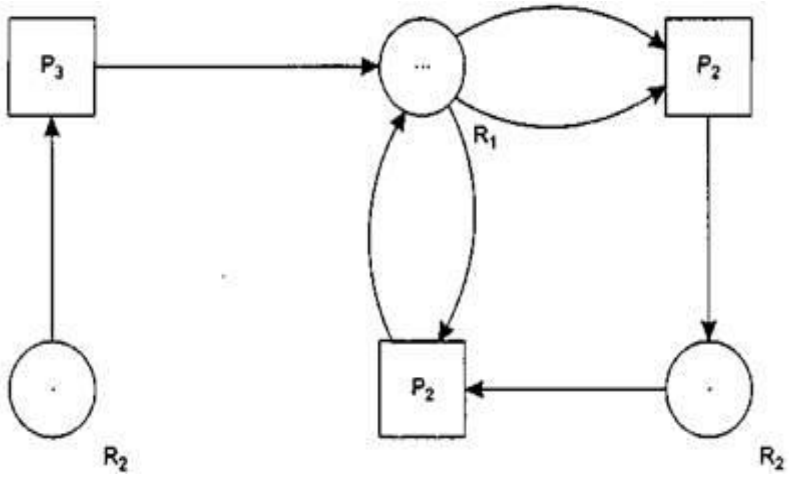

Рис. 5.1. Пример узла на модели Холта

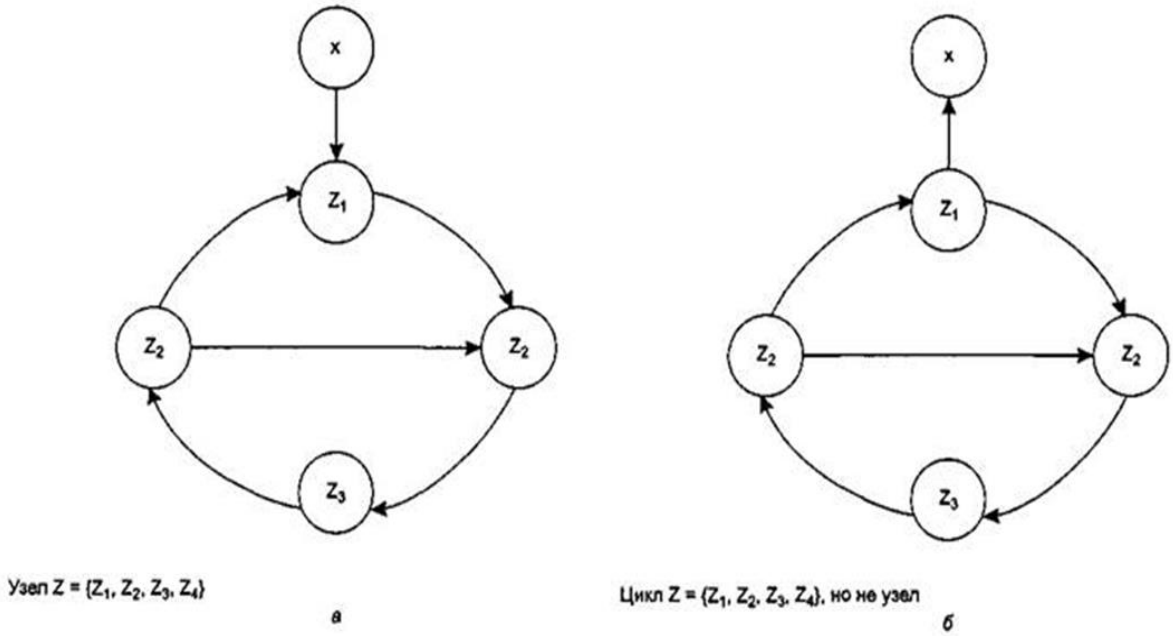

Рис. 5.2. Узел и цикл в ориентированном графе

Теорема 3. Если состояние системы фиксированное (все процессы, имеющие запросы, удовлетворены), то наличие узла в соответствующем графе повторно используемых ресурсов является достаточным условием тупика.

Теорема 4. Граф повторно используемых ресурсов с единичной емкостью указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

5.2. Выполнение лабораторной работы

5.2.1. Описание интерфейса

При запуске программы появляется окно входа в программу (рис. 5.3), в которое необходимо ввести свою фамилию, имя и группу. В дальнейшем эта информация будет отображена в отчете о выполнении лабораторной работы. После нажатия кнопки «Вход» появится окно с подтверждением успешного входа в программу.

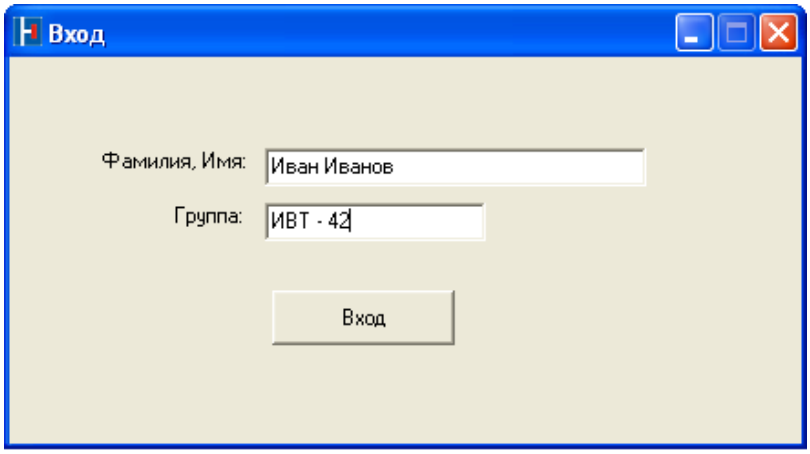

Рис. 5.3. Окно входа в программу

Далее отображается главное окно программы (рис. 5.4), которое содержит главное меню и рабочую зону.

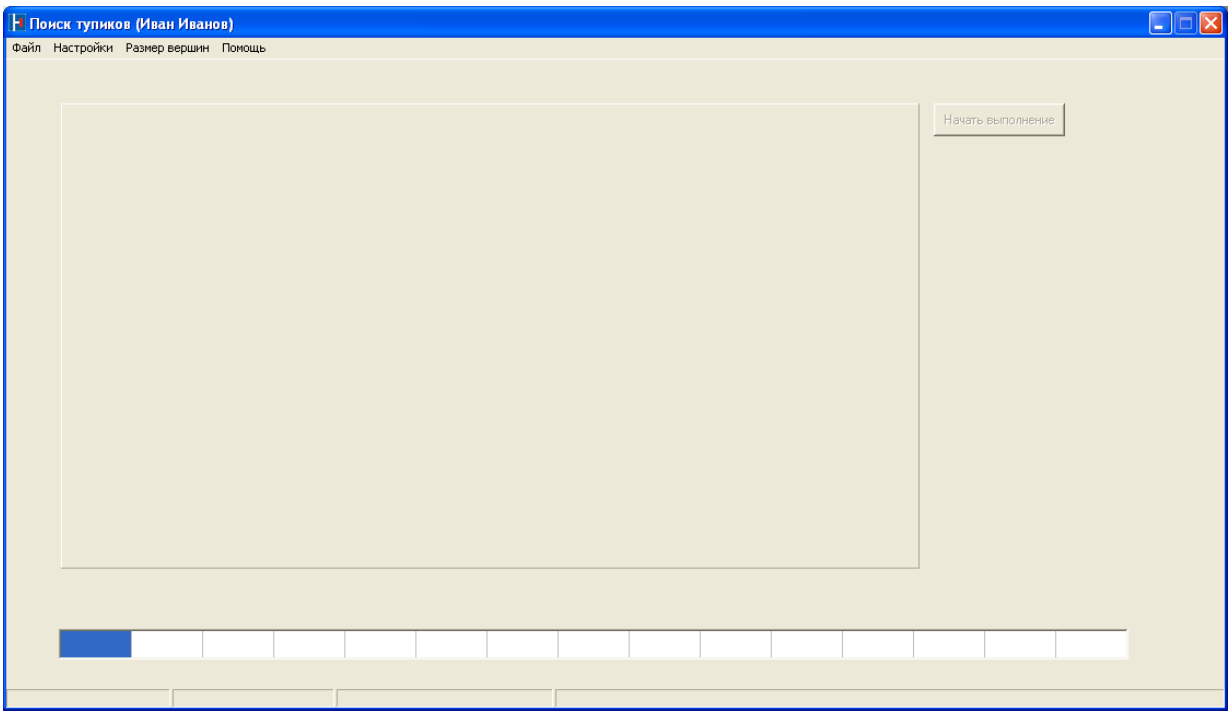

Рис. 5.4. Главное окно программы

Главное меню программы содержит следующие пункты:

1) Файл
    a. Новое задание
        i. Открыть задание
        ii. Сгенерировать задание
    b. Открыть решение
    c. Сохранить решение
    d. Открыть отчет
    e. Выход
2) Настройки
3) Размер вершин
    a. Мелко
    b. Средне
    c. Крупно
4) Помощь
    a. Справка
    b. О программе

Обратите внимание, что при выполнении лабораторной работе ведется подсчет совершенных ошибок.

5.2.2. Порядок выполнения лабораторной работы

Для начала выполнения лабораторной работы необходимо сгенерировать новое задание (рис. 5.5) или открыть уже существующее задание или решение. Для этого необходимо выбрать соответствующее действие в пункте «Файл» главного меню. Вершины графа можно перемещать для удобного просмотра.

После выбора задания для выполнения необходимо нажать кнопку «Начать выполнение», тогда произойдет изменение главного окна программы (рис. 5.6).

В правой части окна появляются:

1) Область для выбора ответа о существовании тупика на текущей итерации.
2) Кнопка «Выбрать/изменить начальную вершину».
3) Кнопка «Шаг» для утверждения текущего ответа.
4) Кнопка «Отмена» для отката всех действий на текущей итерации, если на текущей итерации не было произведено ни одного действия, то будет полностью отменена предыдущая итерация.

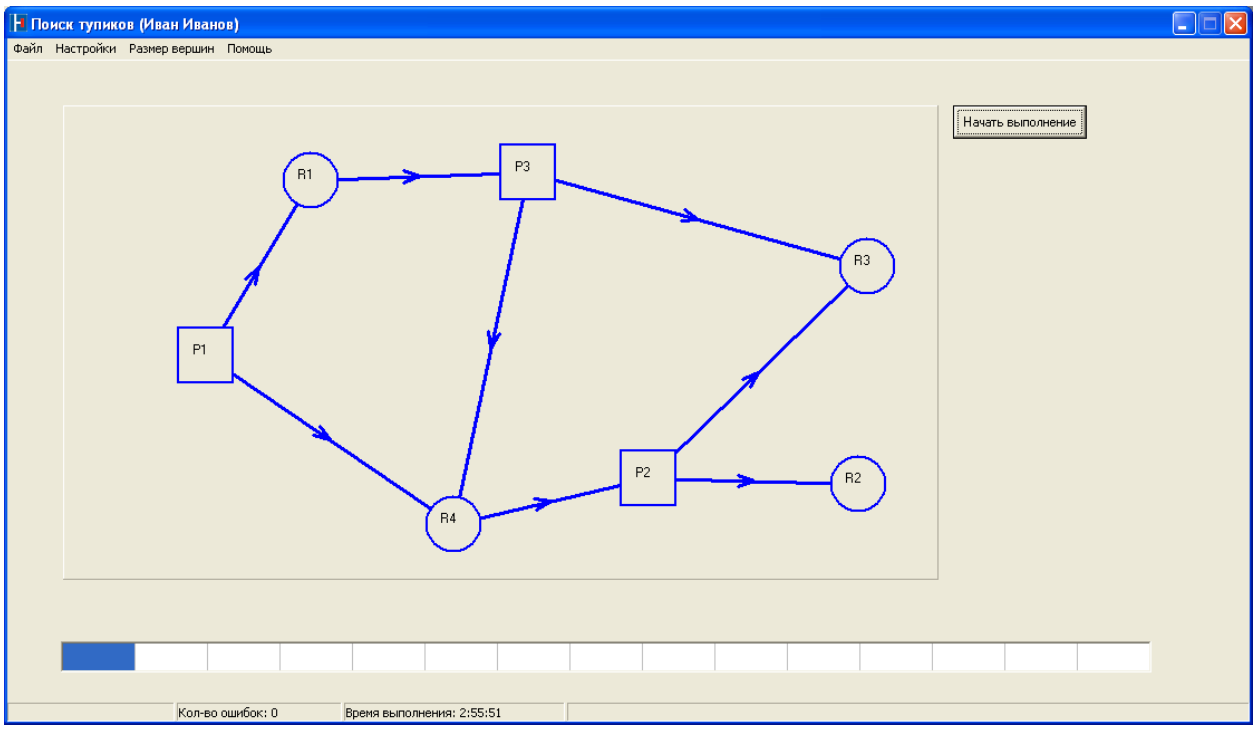

Рис. 5.5. Сгенерированный вариант задания

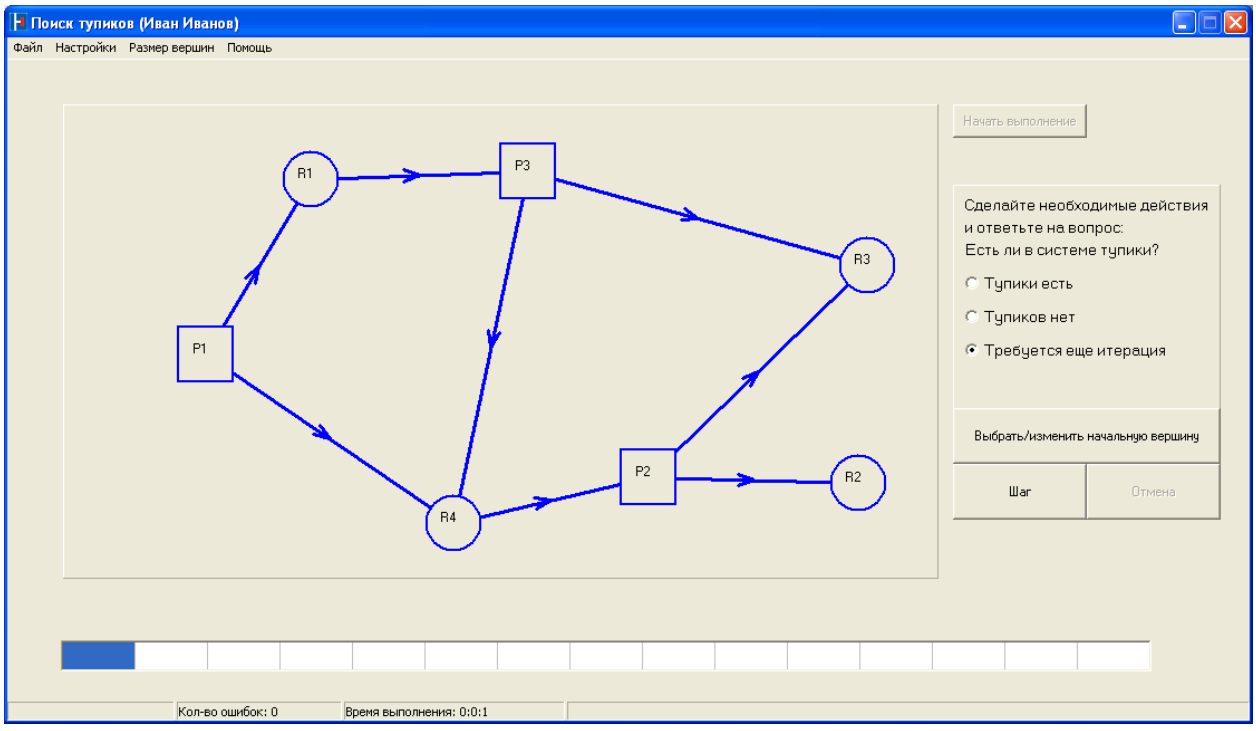

Рис. 5.6. Изменение главного окна

На каждой итерации есть возможность выполнить только одно из трех действий, при этом они состоят из нескольких операций:

1) Выбор или изменение начальной вершины:
    a. Выбор или изменение начальной вершины.
    b. Добавление вершины в список.
2) Проход по ребру:
    a. Пройти по данному ребру.
    b. Добавить вершину в список.
    c. Поставить пометку ребра.
3) Откат по ребру:
    a. Откатиться по данному ребру.
    b. Удалить вершину из списка.

Для завершения действий на текущей итерации и перехода к следующей нужно нажать кнопку «Шаг». Для того чтобы добавить или удалить вершину из списка, нужно кликнуть по вершине правой кнопкой мыши и выбрать соответствующий пункт из выпавшего меню. Для того чтобы пройти по ребру, откатиться по ребру, поставить или снять отметку с ребра, нужно кликнуть по ребру правой кнопкой мыши и выбрать соответствующий пункт из выпавшего меню. При этом необходимо отметить, что при клике по ребру меню выпадает не всегда, поэтому нужно пробовать разный участок ребра для клика.

Добавленные вершины отображаются в списке, расположенном в нижней части окна.

На первой итерации нужно выбрать начальную вершину (рис. 5.7) и добавить ее в список (рис. 5.8). По умолчанию это P1. Изменить начальную вершину можно только в начале выполнения задания или в случае отката до начальной вершины.

Далее необходимо анализировать наличие выходящих дуг из текущей вершины. Если она есть, то нужно пройти по ней. В случае если выходящих дуг несколько, то сначала нужно совершить проходы по дугам, ведущим к вершинам с меньшим индексом (рис 5.9).

В случае, когда выходящих дуг из текущей вершины нет, необходимо откатиться к предыдущей вершине (рис 5.10).

Если после откатов в списке останется только одна вершина – начальная, то необходимо изменить начальную вершину, если на графе есть не помеченные дуги.

В данном примере необходимо совершить обходы, где в качестве начальной вершины последовательно будет каждая из вершин графа. Когда последняя из вершин будет начальной и после обхода будет возврат в текущую начальную вершину, можно будет сказать, что тупика нет.

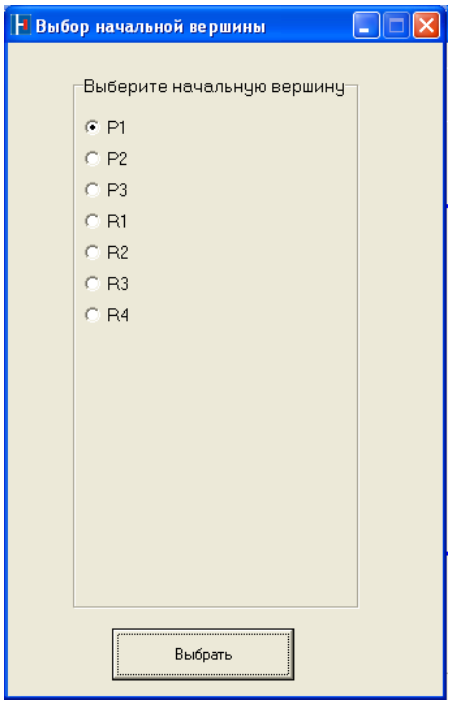

Рис. 5.7. Выбор начальной вершины

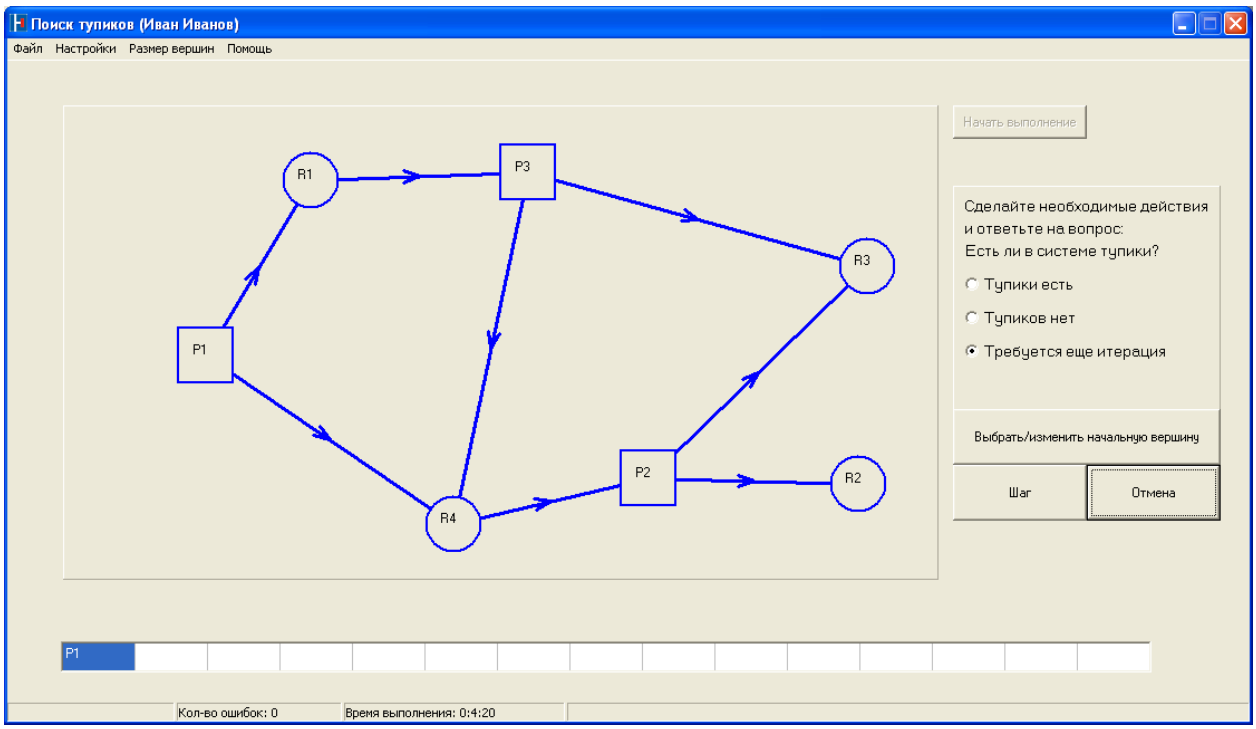

Рис. 5.8. Добавление начальной вершины в список

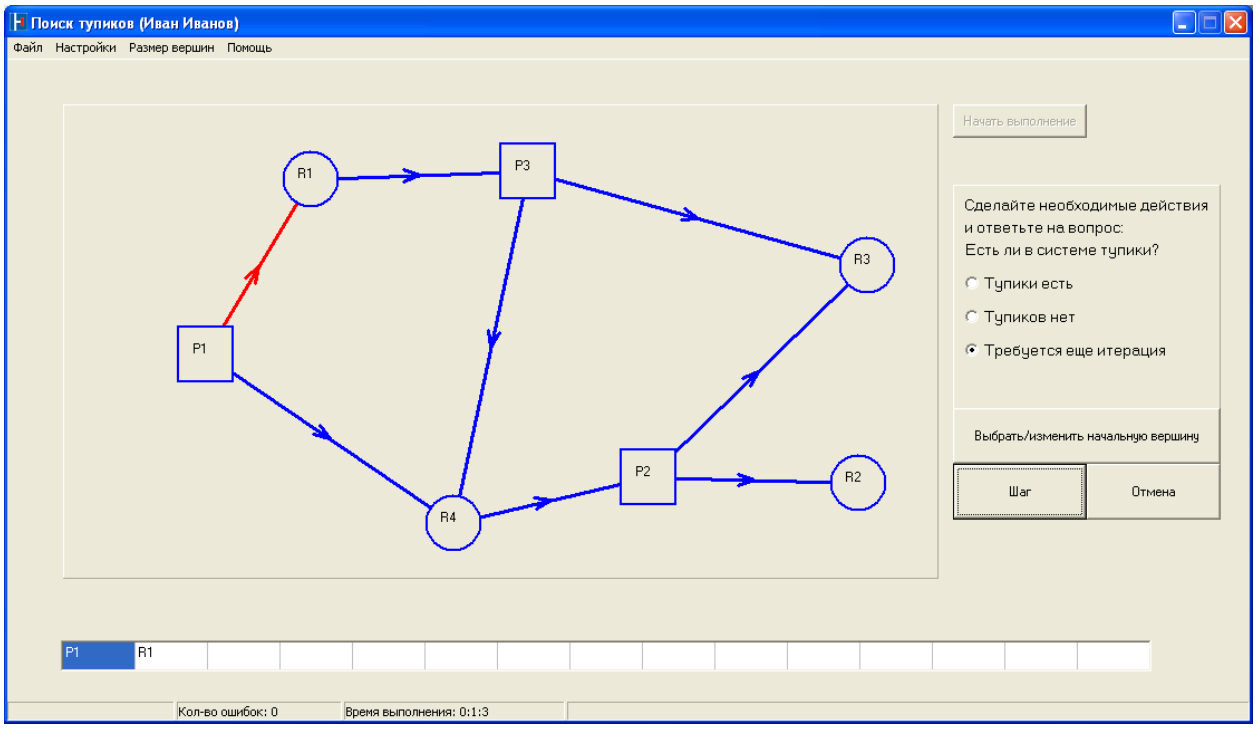

Рис. 5.9. Выбор вершины для прохода из нескольких

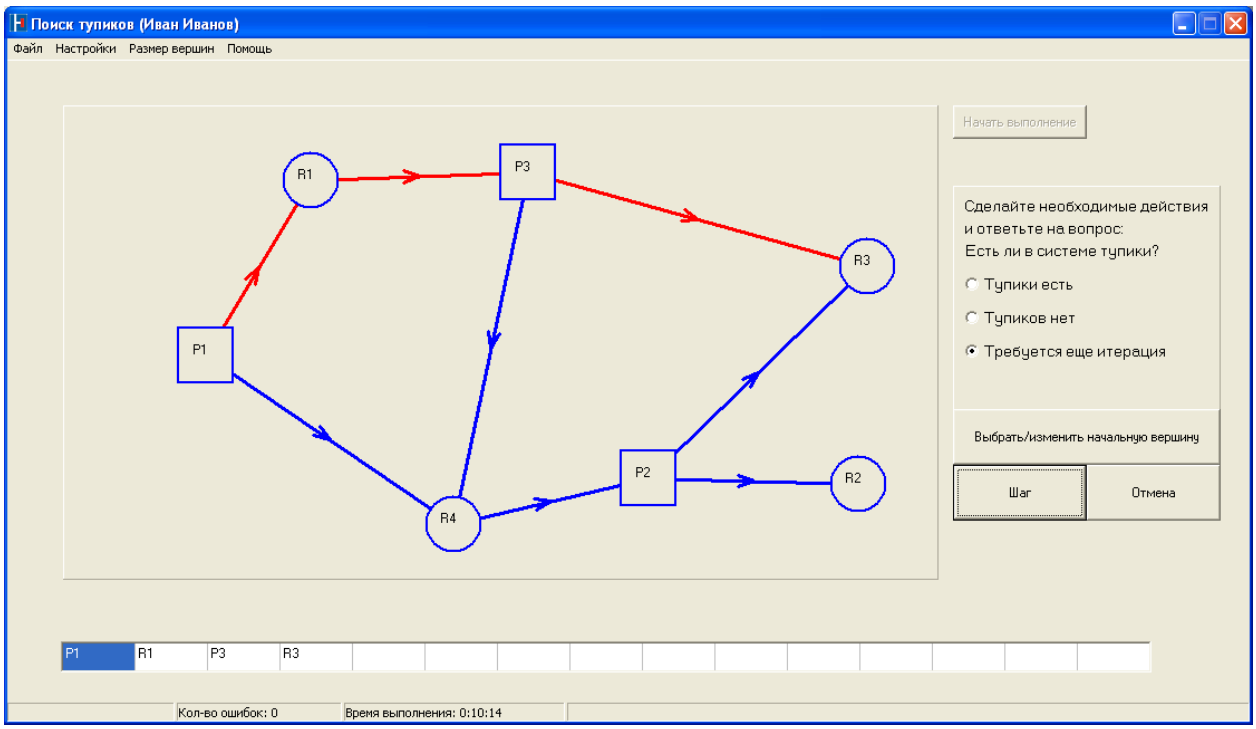

Рис. 5.10. Откат по ребру

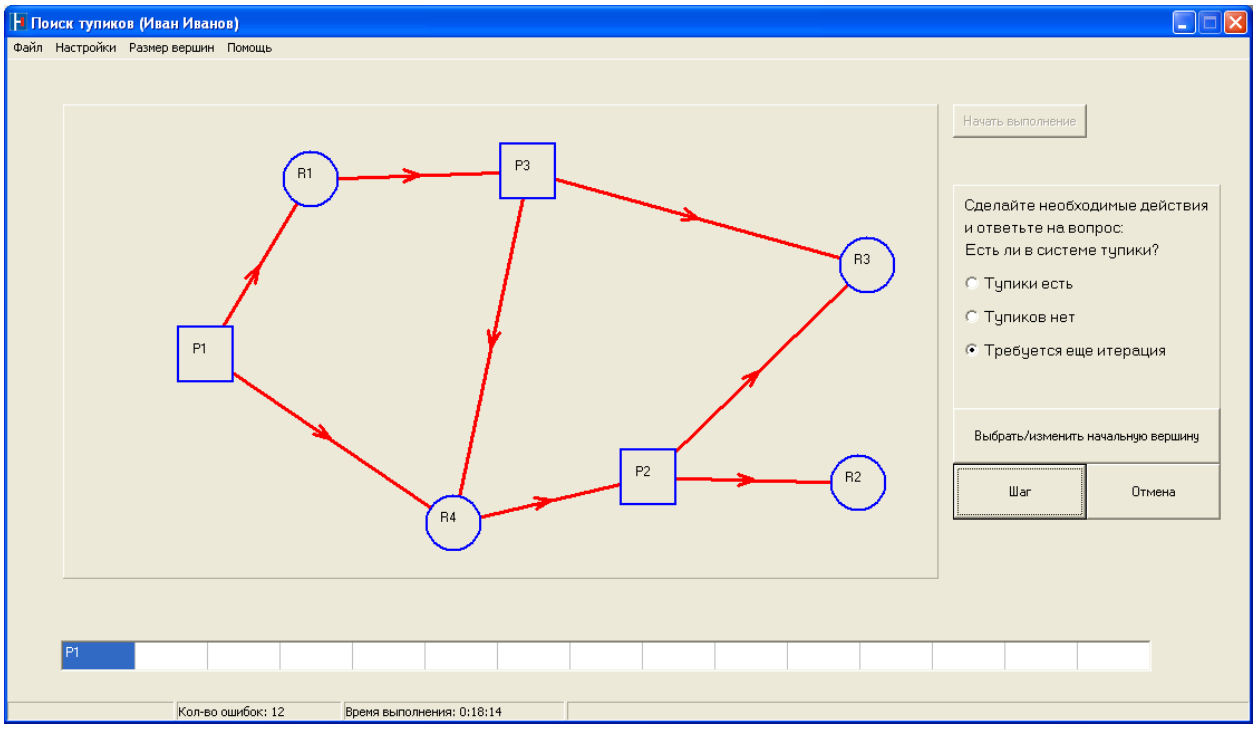

Рис. 5.11. Необходимость изменения начальной вершины

Необходимо также рассмотреть граф, в котором будет тупик. Можно утверждать, что тупик есть, в том случае, когда в список добавляется вершина, которая уже есть в списке (рис 5.12).

После успешного выполнения работы с графом необходимо выбрать вершины, которые входят в цикл (рис. 5.13).

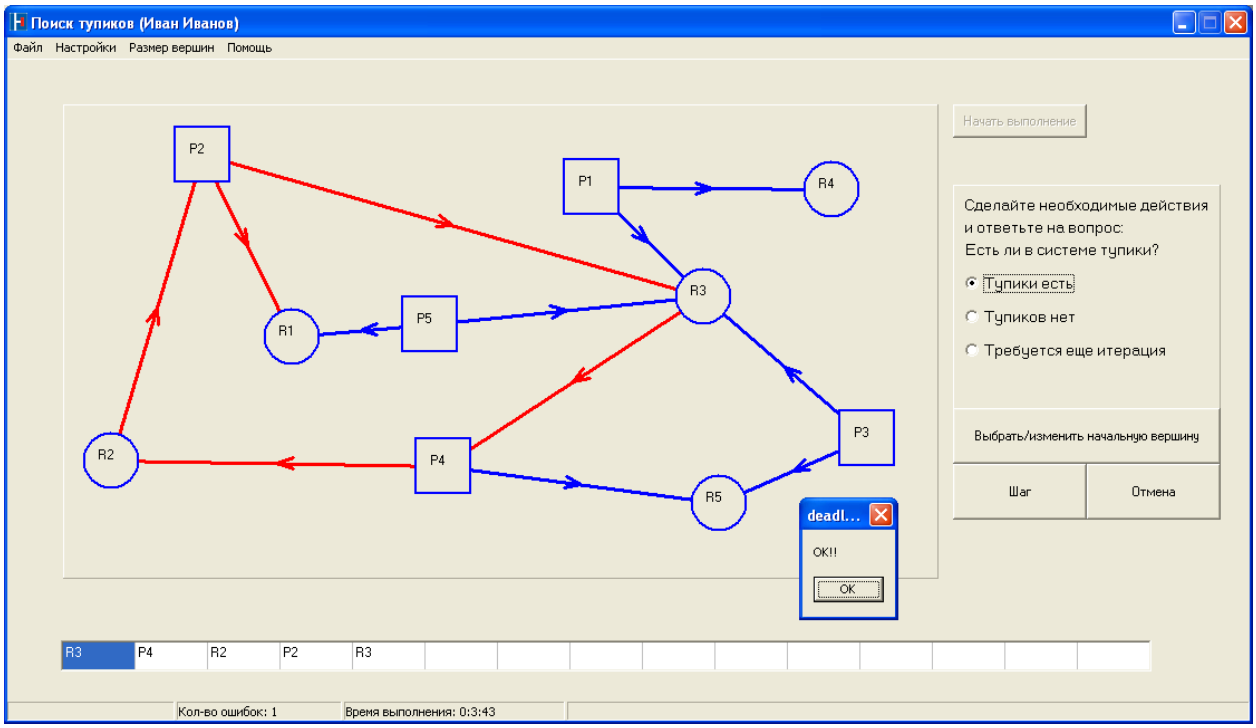

Рис 5.12. Обнаружение тупика

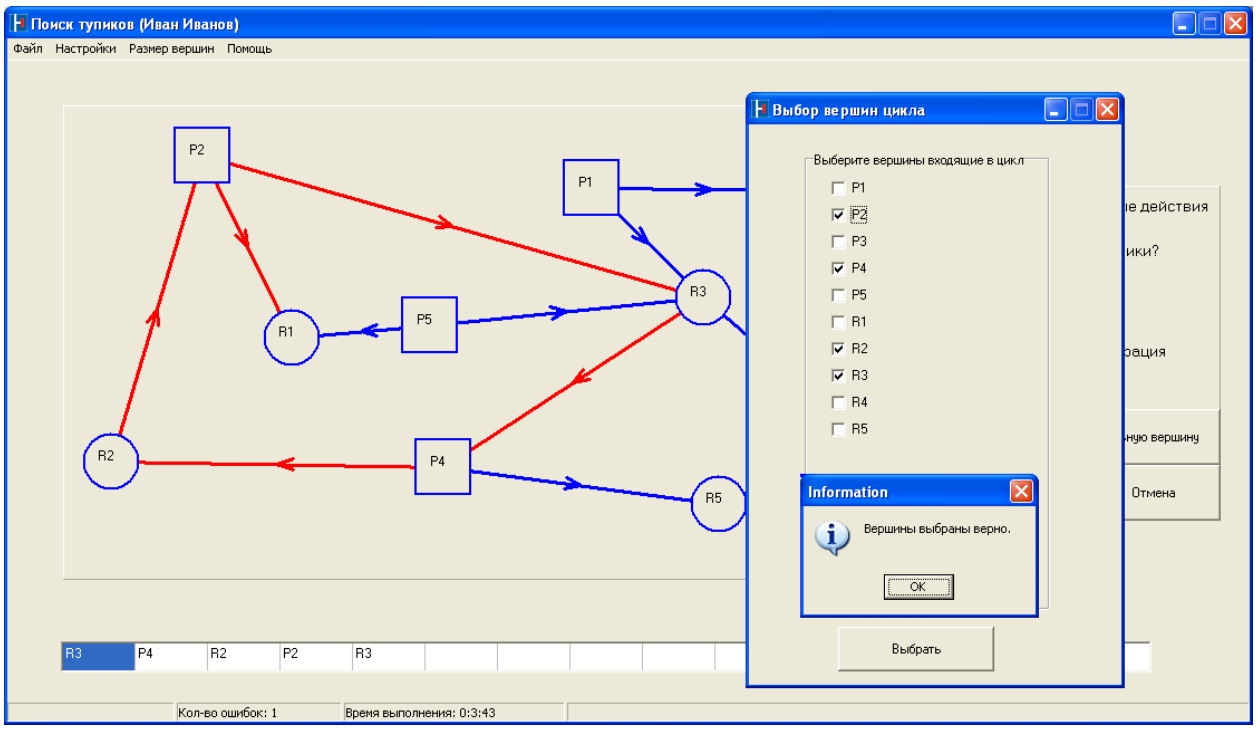

Рис 5.13. Выбор вершин цикла

ПОСЛЕ ВЫПОЛНЕНИЯ ОПИСАННЫХ В ЭТОЙ ГЛАВЕ ДЕЙСТВИЙ БУДЕТ СФОРМИРОВАН ОТЧЕТ О ВЫПОЛНЕННОЙ РАБОТЕ, КОТОРЫЙ НЕОБХОДИМО ПОКАЗАТЬ ПРЕПОДАВАТЕЛЮ.

ЗАКЛЮЧЕНИЕ

Возникновение тупиков является потенциальной проблемой любой операционной системы. Они возникают, когда имеется группа процессов, каждый из которых пытается получить исключительный доступ к некоторым ресурсам и претендуют на ресурсы, принадлежащие другому процессу. В итоге все они оказываются в состоянии ожидания события, которое никогда не произойдет.

С тупиками можно бороться, обнаруживать их, избегать и восстанавливать систему после тупиков. Однако цена подобных действий высока и соответствующие усилия должны предприниматься только в системах, где игнорирование тупиковых ситуаций приводит к катастрофическим последствиям.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Таненбаум, Э. Современные операционные системы [Текст] : пер. с англ. / Э. Танебаум, Х. Бос. – 4-е изд. – Санкт-Петербург : Питер, 2015. – 1119 с. – (Серия «Классика computer science»).

2. Дроздов, С. Н. Операционные системы [Текст] : учеб. пособие / С. Н. Дроздов. – Ростов-на-Дону : Феникс, 2016. – 362 с. – (Высшее образование).

3. Илюшечкин, В. М. Операционные системы [Текст] : учеб. пособие / В. М. Илюшечкин. – Москва : Бином. Лаборатория знаний, 2009. – 109 с.

4. ИНТУИТ: Основы операционных систем [Электронный ресурс] – Режим доступа: URL http://www.intuit.ru/studies/courses/913/31/lecture/980.

Приложение 1

Алгоритм банкира Дейкстры

```
Begin
     Своб_рес := Всего_рес:
     For i:= 1 to N do
     Begin
          Своб_рес := Своб_рес - Получ(i)
          Остаток(i) := Max(i) – Получ(i)
          Заверш(i):= false ;    { процесс может не завершиться }
     End
     flag := true ;     { признак продолжения анализа }
     while flag do
          begin
               flag := false;
               For i := 1 to N do
               Begin
                    if ( not (Заверш (i))) and ( Остаток (i) <=
               Своб_ рес) then
                    begin
                         Заверш(i) := true
                         Своб_рес := Своб_рес + Получ(i);
                         Flag := true
                    End
          End
     End:
     If Своб_рес= Всего_рестм then
          {Состояние системы безопасное и можно выдать ресурс}
     Else
          {Состояние не безопасное и выдавать ресурс нельзя}
     end.
```

Учебное издание

Караваева Ольга Владимировна

ОПЕРАЦИОННЫЕ СИСТЕМЫ. МЕТОДЫ БОРЬБЫ С ТУПИКАМИ

Учебно-методическое пособие

Подписано в печать 19.01.2017. Печать цифровая. Бумага для офисной техники. Усл. печ. л. 3,22. Тираж 5 экз. Заказ № 4101.

Федеральное государственное бюджетное образовательное учреждение высшего образования «Вятский государственный университет».

610000, г. Киров, ул. Московская, 36, тел.: (8332) 74-25-63, http://vyatsu.ru