Распределение ресурсов. Взаимоблокировки

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2010

Дата последней модификации: 23.02.2010 18:21:00

История изменений

21.02.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

Содержание

История изменений 2
Содержание 3
Лекция 4. Распределение ресурсов. Взаимоблокировки 4
Вопросы 4
Взаимоблокировки 4
Моделирование взаимоблокировок 4
Страусовый алгоритм 6
Обнаружение и устранение взаимоблокировок 7
Обнаружение взаимоблокировки при наличии одного ресурса каждого типа 7
Обнаружение взаимоблокировок при наличии нескольких ресурсов каждого типа 8
Выход из взаимоблокировки 10
Избежание взаимоблокировок 11
Траектории ресурсов 11
Безопасные и небезопасные состояния 12
Алгоритм банкира для одного вида ресурсов 13
Алгоритм банкира для нескольких видов ресурсов 14
Предотвращение взаимоблокировок 15
Атака условия взаимного исключения 15
Атака условия удержания и ожидания 16
Атака условия отсутствия принудительной выгрузки ресурса 16
Атака условия циклического ожидания 16
Литература 17

Лекция 4. Распределение ресурсов. Взаимоблокировки

Вопросы

1. Взаимоблокировки.

2. Обнаружение и устранение взаимоблокировок.

3. Избежание взаимоблокировок.

4. Предотвращение взаимоблокировок.

Взаимоблокировки

Ресурсы – объекты предоставления доступа.

Ресурсы можно представить как:

- **Выгружаемые** – можно безболезненно забирать у владеющего ими процесса.
- **Не выгружаемые** – ресурсы, которые нельзя безболезненно забрать у владеющего ими процесса, не уничтожив результаты работы с ними.

Потенциально проблемы (взаимоблокировки) с использованием возникают именно с не выгружаемыми ресурсами. В дальнейшем основное внимание будет уделено именно им.

Последовательность событий, необходимых для использования ресурса, представлена ниже в абстрактной форме:

1. Запрос ресурса.

2. Использование ресурса.

3. Возврат ресурса.

Группа процессов находится в **тупиковой ситуации** (**взаимоблокировка**), если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.

Для возникновения ситуации взаимоблокировки должны выполняться **четыре условия**:

1. **Условие взаимного исключения.** Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.

2. **Условие удержания и ожидания.** Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.

3. **Условие отсутствия принудительной выгрузки ресурса.** У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.

4. **Условие циклического ожидания.** Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Моделирование взаимоблокировок

Смоделировать четыре условия возникновения тупиков, используя **направленные графы**. Графы имеют два вида узлов:

- процессы, показанные кружочками,
- ресурсы, нарисованные квадратиками.

Ребро, направленное от узла ресурса (квадрат) к узлу процесса (круг), означает, что ресурс ранее был **запрошен процессом**, получен и в данный момент используется этим процессом. На рис. 1(а) ресурс B в настоящее время отдан процессу A.

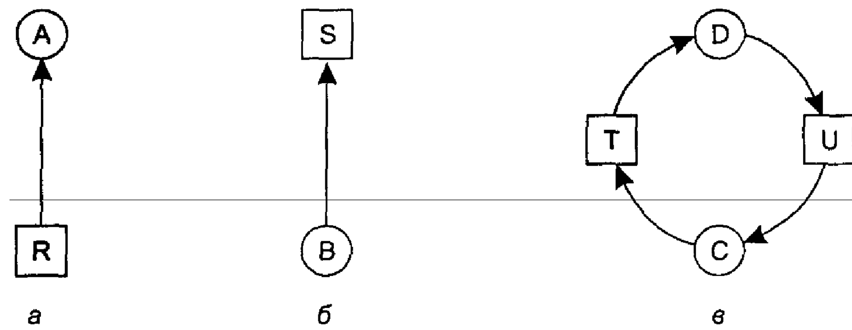
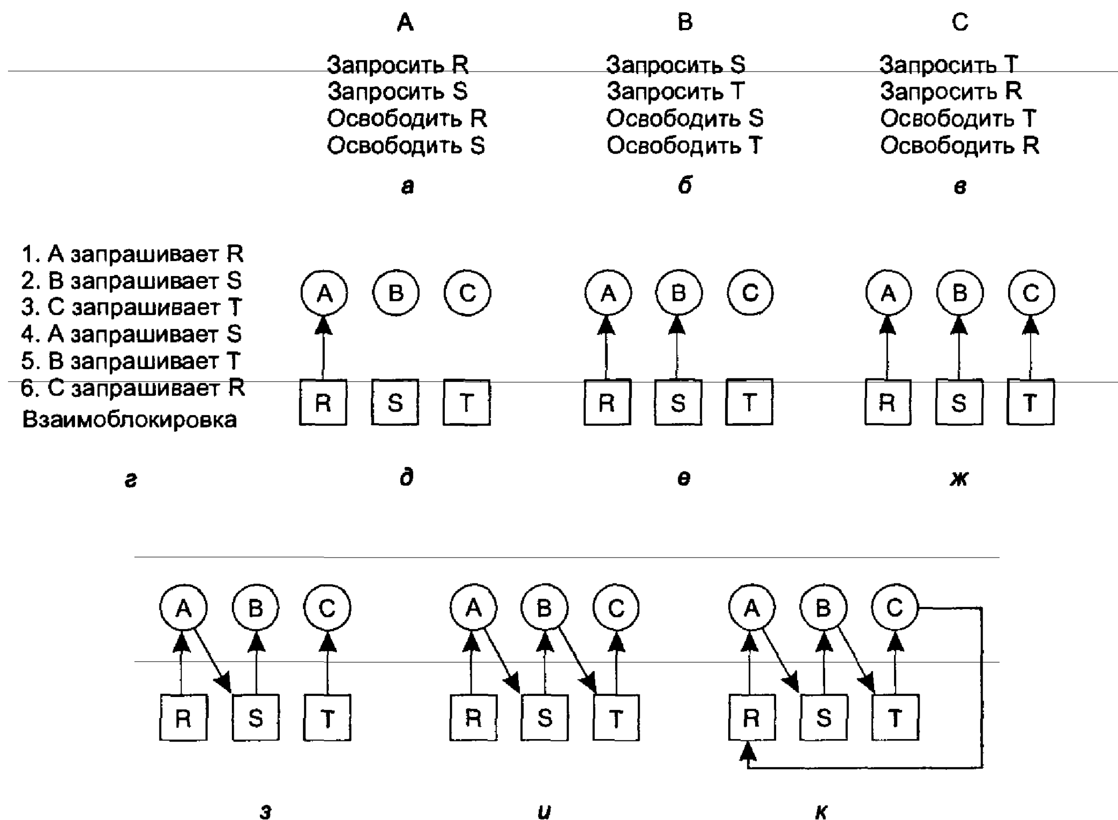


Рис. 1. Графы распределения ресурсов: ресурс занят (а); запрос ресурса (б); взаимоблокировка (в)

Ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу. На рис. 1(б) процесс В ждет ресурс S. На рис. 1(в) мы видим взаимоблокировку: процесс С ожидает ресурс Т, удерживаемый в настоящее время процессом D. Процесс D вовсе не намеревается освобождать ресурс Т, потому что он ждет ресурс U, используемый процессом С. Оба процесса будут ждать до бесконечности.

Цикл в графе означает **наличие взаимоблокировки**, циклично включающей процессы и ресурсы (предполагается, что в системе есть по одному ресурсу каждого вида). В этом примере циклом является последовательность С-T-D-U-С.

Теперь рассмотрим пример того, как можно использовать графы ресурсов. Представим, что у нас есть три процесса: А, В и С, и три ресурса: R, 5 и Т. Последовательность запросов и возвратов ресурсов для трех процессов показаны на рис. 2(а)—(в). ОС может запустить любой незаблокированный процесс в любой момент времени, значит, она может решить запустить сначала процесс А. Процесс А будет выполняться до тех пор, пока не закончит всю свою работу, затем будет запушен процесс В до его завершения и. наконец, процесс С.



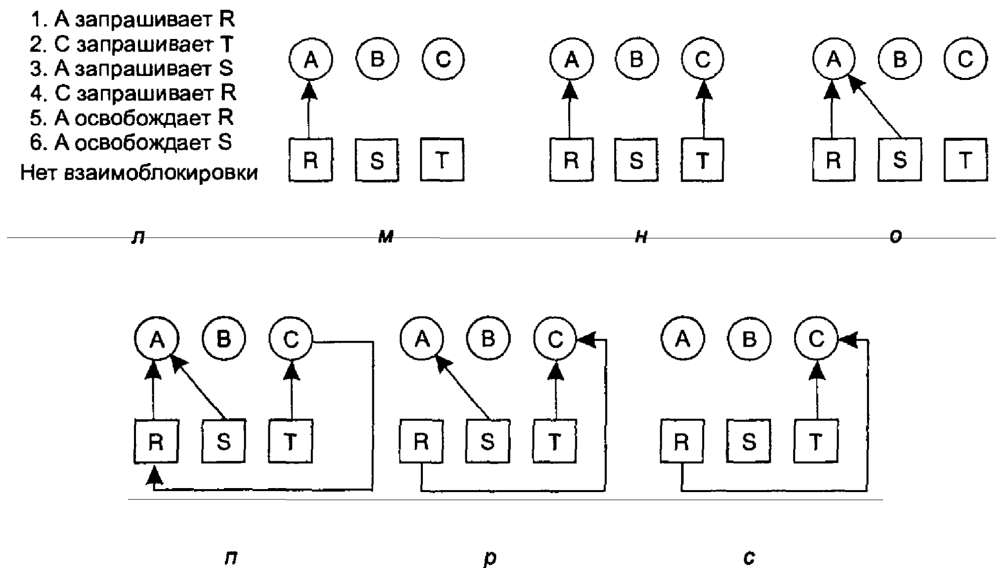


Рис. 2. Пример возникновения блокировки и способов ее избежать

При столкновении с взаимоблокировками используются четыре стратегии:

1. Пренебрежение проблемой в целом. Если вы проигнорируете проблему, возможно, затем она проигнорирует вас.

2. Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.

3. Динамическое избежание тупиковых ситуаций с помощью аккуратного распределения ресурсов.

4. Предотвращение с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки.

Далее рассмотрим каждый из этих методов.

Страусовый алгоритм

Самым простым подходом является «страусовый алгоритм»: воткните голову в песок и притворитесь, что проблема вообще не существует. Различные люди отзываются об этой стратегии по-разному. Математики считают ее полностью неприемлемой и говорят, что взаимоблокировки нужно предотвращать любой ценой. Инженеры спрашивают, как часто встает подобная проблема, как часто система попадает в аварийные ситуации по другим причинам и насколько серьезны последствия взаимоблокировок. Если взаимоблокировки случаются в среднем один раз в пять лет, а сбои ОС, ошибки компилятора и поломки компьютера из-за неисправности аппаратуры происходят раз в неделю, то большинство инженеров не захотят добровольно уступать в производительности и удобстве для того, чтобы ликвидировать возможность взаимоблокировок.

Для усиления контраста между этими подходами добавим, что большинство ОС потенциально страдают от взаимоблокировок, которые даже не обнаруживаются, не говоря уже об автоматическом выходе из тупика. Суммарное количество процессов в системе определяется количеством записей в таблице процесса. Таким образом, ячейки таблицы процесса являются ограниченным ресурсом. Если системный вызов `fork` получает отказ, потому что таблица целиком заполнена, разумно будет, что программа, вызывающая `fork`, подождет какое-то время и повторит попытку.

Теперь предположим, что система UNIX имеет 100 ячеек процессов. Работают десять программ, каждой необходимо создать 12 (под) процессов. После образования каждым процессом девяти процессов 10 исходных и 90 новых процессов заполнят таблицу целиком. Теперь каждый из десяти исходных процессов попадает в бесконечный цикл, состоящий из попыток разветвления и отказов, то есть возникает взаимоблокировка. Вероятность того, что произойдет подобное, минимальна, но это могло бы случиться. Должны ли мы отказаться от процессов и вызова `fork`, чтобы устранить данную проблему?

Максимальное количество открытых файлов также ограниченно размером таблицы i-узлов, следовательно, когда таблица заполняется целиком, возникает та же самая проблема. Пространство для подкачки файлов на диск является еще одним ограниченным ресурсом. Фактически почти каждая таблица в ОС представляет собой ресурс, имеющий пределы. Должны ли мы упразднить их все из-за того, что может произойти ситуация, когда в группе из n процессов каждый может потребовать 1/n от целого, а затем попытаться получить еще часть?

Большая часть ОС, включая UNIX и Windows, игнорируют эту проблему. Они исходят из предположения, что большинство пользователей скорее предпочтут иметь дело со случающимися время от времени взаимоблокировками, чем с правилом, по которому всем пользователям разрешается только один процесс, один открытый файл и т. д. Если бы можно было легко устранить взаимоблокировки, не возникло бы столько разговоров на эту тему. Сложность заключается в том, что цена достаточно высока, и в основном она, как мы вскоре увидим, исчисляется в наложении неудобных ограничений на процессы. Таким образом, мы столкнулись с неприятным выбором между удобством и корректностью и множеством дискуссий о том, что более важно и для кого. При всех этих условиях трудно найти верное решение.

Обнаружение и устранение взаимоблокировок

Вторая техника представляет собой обнаружение и восстановление. При использовании этого метода система не пытается предотвратить попадание в тупиковые ситуации. Вместо этого она позволяет взаимоблокировке произойти, старается определить, когда это случилось, и затем совершает некие действия к возврату системы к состоянию, имевшему место до того, как система попала в тупик. Ь этом разделе мы рассмотрим некоторые из способов обнаружения тупиковых ситуаций и выхода из них.

Обнаружение взаимоблокировки при наличии одного ресурса каждого типа

Начнем с самого простого варианта: в системе существует только один ресурс каждого типа. Подобная система могла бы иметь один сканер, одно устройство для записи компакт-дисков, один плоттер и один накопитель на магнитной ленте, то есть не более чем по одному представителю каждого класса. Другими словами, мы исключаем из рассмотрения системы с двумя одновременно подключенными принтерами. Мы обратимся к ним позже и будем использовать другой метод.

Для такой системы можно сконструировать граф ресурсов вида, продемонстрированного на рис. 3. Если этот граф содержит один или больше циклов, значит, произошла взаимоблокировка и блокирован любой процесс, являющийся частью цикла. Если в графе нет циклов, система не попала в тупик.

В качестве примера более сложной системы, чем те, которые мы рассматривали до сих пор, обсудим систему с семью процессами, обозначенными буквами от A до G, и шестью ресурсами, обозначенными буквами от R до W. Состояние системы, то есть то, какой процесс владеет каким ресурсом и какой ресурс запрашивается процессом в данный момент, соответствует следующему списку:

1. Процесс Л занимает ресурс R и хочет получить ресурс S.

2. Процесс B ничего не использует, но хочет получить ресурс T.

3. Процесс C ничего не использует, но хочет получить ресурс S.

4. Процесс D занимает ресурс U и хочет получить ресурсы S и T.

5. Процесс E занимает ресурс T и хочет получить ресурс V.

6. Процесс F занимает ресурс W и хочет получить ресурс S.

7. Процесс G занимает ресурс V и хочет получить ресурс U.

Вопрос: «Заблокирована ли эта система и если да, то какие процессы в этом участвуют?»

Чтобы ответить на этот вопрос, мы можем составить граф ресурсов (рис. 3.3, а). Этот граф содержит один цикл, который виден при визуальном обследовании. Цикл показан на рис. 3(б). Изучая его, можно заметить, что процессы D, E и G заблокированы. Процессы A, C и F не попали в тупик, потому что любому из них можно предоставить ресурс 5, после чего процесс, получивший ресурс, закончит свою работу и вернет ресурс. Затем два других процесса по очереди могут получить ресурс и также успешно выполнить свою работу.

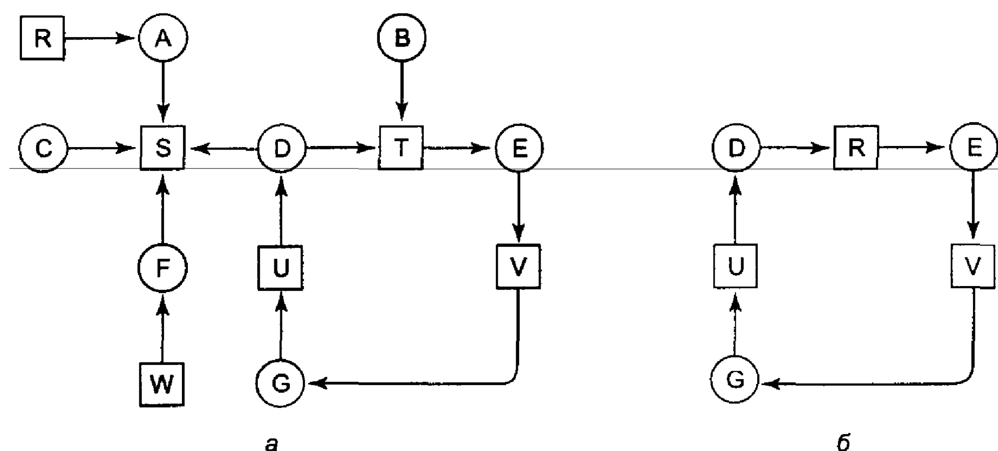


Рис. 3. Граф ресурсов (а); цикл, извлеченный из а (б)

Рассмотрим простой алгоритм, который изучает граф и завершается или когда находит цикл, или когда показывает, что циклов в этом графе не существует. Он использует одну структуру данных — список узлов L. Во время работы алгоритма на ребрах графа будет ставиться метка, говорящая о том, что их уже проверили, это делается во избежание повторной проверки.

Алгоритм работает, осуществляя пять перечисленных ниже шагов.

1. Для каждого узла N в графе выполняются следующие пять шагов, где N является начальным узлом.

2. Задаем начальные условия: L — пустой список, все ребра не маркированы.

3. Текущий узел добавляем в конец списка L и проверяем количество появлений узла в списке. Если узел присутствует в двух местах, граф содержит цикл (записанный в список L) и работа алгоритма завершается.

4. Для заданного узла смотрим, выходит ли из него хотя бы одно немаркированное ребро. Если да, то переходим к шагу 5, если нет, то переходим к шагу 6.

5. Случайным образом выбираем любое немаркированное исходящее ребро и отмечаем его. Затем по нему переходим к новому текущему узлу и возвращаемся к шагу 3.

6. Теперь мы зашли в тупик. Удаляем последний узел из списка и возвращаемся к предыдущему узлу, то есть тому, который был текущим перед тупиковым узлом. Обозначаем его текущим узлом и возвращаемся к шагу 3. Если это первоначальный узел, граф не содержит циклов и алгоритм завершается.

Этот алгоритм по очереди берет каждый узел в качестве корня того, что, как он надеется, окажется деревом, и выполняет в дереве поиск в глубину. Если в процессе обхода алгоритм возвращается к уже встречавшемуся узлу, то он нашел цикл. Если алгоритм обходит все ребра из какого-нибудь заданного узла, то он возвращается к предыдущему узлу. Если он возвращается к корню и не может идти дальше, то подграф текущего узла не содержит циклов. Если данное свойство сохраняется для всех узлов, значит, полный граф не содержит циклов, а система не заблокирована.

Алгоритм далеко не оптимален, однако он приведен лишь для демонстрации подхода. На практике применяются более сложные алгоритмы.

Обнаружение взаимоблокировок при наличии нескольких ресурсов каждого типа

Когда в системе существует несколько экземпляров некоторых из ресурсов, для обнаружения взаимоблокировок необходим другой метод, который основан на матрицах, и обнаруживающем тупики среди n n процессов, от P_1 до P_n. Пусть m — это число классов ресурсов, причем в системе E_1 ресурсов класса 1, E_2 ресурсов класса 2 и E_i, ресурсов класса i (где $1 \le i \le m$). E — это **вектор существующих ресурсов**. Он передает общее количество имеющихся в наличии экземпляров каждого ресурса. Например, если класс 1 представляет собой накопители на магнитных лентах, то $E_1 = 2$ означает, что в системе есть два магнитофона.

В любой момент времени некоторые из ресурсов могут оказаться занятыми и, соответственно, недоступны. Пусть A будет **вектором доступных ресурсов**, где A_i равно количеству экземпляров ресурса i, доступных в текущий момент (то есть не использующихся). Если оба накопителя на магнитной ленте заняты, $A_1 = 0$.

Теперь нам нужны два массива: C — **матрица текущего распределения** и R — **матрица запросов**, i-я строка в матрице C говорит о том, сколько представителей каждого класса ресурсов в данный момент использует процесс P_i. Таким образом, C_{ij} - это **количество экземпляров ресурса** j, которое занимает процесс i. Аналогично, R_{ij} — это количество экземпляров ресурса j, которые хочет получить процесс P_i. Эти четыре структуры показаны на рис. 4.

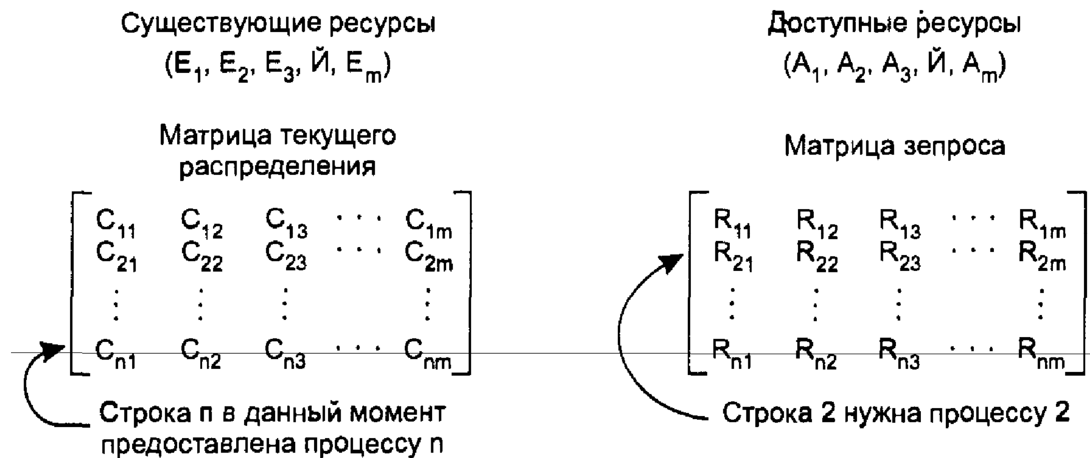


Рис. 4. Четыре структуры данных, необходимые для алгоритма обнаружения тупиков

Для этих четырех структур данных существует важный инвариант. В принципе каждый ресурс или занят, или свободен. Это наблюдение означает, что

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j .$$

Другими словами, если сложить все экземпляры ресурса, предоставленные процессам и доступные в данный момент, то в результате мы получим существующее в системе количество экземпляров этого класса ресурсов.

Алгоритм обнаружения взаимоблокировок основан на сравнении векторов. Определим, что для двух векторов A и B отношение $A \le B$ означает, что каждый элемент вектора A меньше или равен соответствующему элементу вектора B: $A \le B$ тогда и только тогда, когда $A_i \le B_i$, для $1 \le i \le m$.

Пусть в исходном положении все процессы не маркированы. По мере продвижения алгоритма на процессы будет ставиться отметка, служащая признаком того, что они могут закончить свою работу и, следовательно, не находятся в тупике. После завершения алгоритма известно, что любой немаркированный процесс находится в тупиковой ситуации.

Алгоритм обнаружения тупиков состоит из следующих шагов:

1. Ищем немаркированный процесс P_i для которого i-я строка матрицы R меньше вектора A или равна ему.

2. Если такой процесс найден, прибавляем i-ю строку матрицы C к вектору A, маркируем процесс и возвращаемся к шагу 1.

3. Если таких процессов не существует, работа алгоритма заканчивается.

Завершение алгоритма означает, что все немаркированные процессы, если такие есть, попали в тупик.

На первом шаге алгоритм ищет процесс, который может доработать до конца. Такой процесс характеризуется тем, что все требуемые для него ресурсы должны находиться среди доступных в данный момент ресурсов. Тогда выбранный процесс проработает до своего завершения и после этого вернет ресурсы, которые он занимал, в общий фонд доступных ресурсов. Затем процесс маркируется как законченный. Если окажется, что все процессы могут работать, тогда ни один из них в данный момент не заблокирован. Если некоторые из них никогда не смогут запуститься, значит, они попали в тупик. Несмотря на то, что алгоритм не является детерминированным (поскольку он может просматривать процессы в любом допустимом порядке), результат всегда одинаков.

Выход из взаимоблокировки

Предположим, что наш алгоритм обнаружения взаимоблокировок закончился успешно и нашел тупик. Далее необходимы методы для восстановления и получения в итоге снова работающей системы.

Восстановление при помощи принудительной выгрузки ресурса

Иногда можно временно отобрать ресурс у его текущего владельца и отдать его другому процессу. Во многих случаях требуется ручное вмешательство, особенно в ОС пакетной обработки, работающих на мэйнфреймах.

Например, чтобы забрать лазерный принтер у использующего его процесса, оператор может взять все уже напечатанные листы и сложить их в стопку, соблюдая последовательность их появления из принтера. Затем процесс можно приостановить (пометить как неработоспособный). В этот момент принтер можно предоставить другому процессу. Когда он закончит работу, можно сложить стопку напечатанных листов обратно на выходной поднос принтера и возобновить первоначальный процесс.

Способность забирать ресурс у процесса, отдавать его другому процессу и затем возвращать назад так, что исходный процесс этого не замечает, в значительной мере зависит от свойств ресурса. Выйти из тупика таким образом зачастую трудно или невозможно. Выбор приостанавливаемого процесса главным образом зависит от того, какой процесс владеет ресурсами, которые легко могут быть у него отняты.

Восстановление через откат

Если разработчики системы и машинные операторы знают о том, что есть вероятность появления взаимоблокировок, они могут организовать работу таким образом, чтобы процессы периодически создавали контрольные точки. Создание процессом контрольной точки означает, что состояние процесса записывается в файл, в результате чего впоследствии процесс может быть возобновлен из этого файла. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, то есть информацию о том, какие ресурсы в данный момент предоставлены процессу. Для большей эффективности новая контрольная точка должна записываться не поверх старой, а в новый файл, так что во время выполнения процесса образуется целая последовательность контрольных точек.

Когда взаимоблокировка обнаружена, достаточно просто понять, какие ресурсы нужны процессам. Чтобы выйти из тупика, процесс, занимающий необходимый ресурс, откатывается к тому моменту времени, перед которым он получил данный ресурс, для чего запускается одна из его контрольных точек. Вся работа, выполненная после этой контрольной копии, теряется (например, выходные данные, напечатанные позднее контрольной копии, отбрасываются и позже печатаются заново). В результате процесс вновь запускается с более раннего момента, когда он не занимал тот ресурс, который теперь предоставляется одному из процессов, попавших в тупик. Если возобновленный процесс снова пытается получить данный ресурс, ему придется ждать того момента, когда ресурс опять станет доступен.

Восстановление путем уничтожения процессов

Грубейший, но одновременно и простейший способ выхода из ситуации взаимоблокировки заключается в уничтожении одного или нескольких процессов. Можно уничтожить процесс, находящийся в цикле взаимоблокировки. При небольшом везении другие процессы смогут продолжить работу. Если первое удаление не помогает, процедуру можно повторять до тех пор, пока цикл наконец не будет разорван.

Можно, наоборот, в качестве жертвы выбрать процесс, не находящийся в цикле, чтобы он освободил свои ресурсы. При этом подходе уничтожаемый процесс выбирается с особой тщательностью, потому что он должен занимать ресурсы, которые нужны некоторым процессам в цикле. Например, один процесс может использовать принтер и требовать плоттер, другой, наоборот, получил плоттер и запрашивает принтер. Оба попали в тупик. Третий процесс может удерживать другие принтер и плоттер и успешно работать. Уничтожение третьего процесса приведет к освобождению этих ресурсов и разрушит взаимоблокировку первых двух процессов.

Там, где это возможно, лучше всего уничтожать те процессы, которые можно запустить с самого начала без всяких болезненных эффектов. Например, процедуру компиляции всегда можно повторить заново, поскольку она всего лишь читает исходный файл и создает объектный файл. Если процедуру компиляции уничтожить в процессе работы, первый ее запуск не повлияет на второй

С другой стороны, процесс, который обновляет базу данных, не всегда можно успешно выполнить во второй раз. Если процесс прибавляет 1 к какой-нибудь записи в базе данных, то его запуск, потом уничтожение, затем повторный запуск приведут к прибавлению к записи 2, что неверно.

Избежание взаимоблокировок

Рассматривая обнаружение взаимоблокировок, предполагали, что когда процесс запрашивает ресурсы, он требует их все сразу (матрица R на рис. 4). Однако в большинстве систем ресурсы запрашиваются поочередно (по одному). Система должна уметь решать, является ли предоставление ресурса безопасным или нет, и предоставлять его процессу только в первом случае. Таким образом, возникает новый вопрос: существует ли алгоритм, который всегда может избежать ситуации взаимоблокировки, все время делая правильный выбор? Ответом является условное «да» — мы можем избежать тупиков, но только если заранее будет доступна определенная информация. Далее изучим способы уклонения от взаимоблокировок с помощью **аккуратного** предоставления ресурсов.

Траектории ресурсов

Основные алгоритмы, позволяющие предотвращать взаимоблокировки, базируются на **концепции безопасных состояний**. Перед тем как начать описывать алгоритм, сделаем небольшое отступление, чтобы взглянуть на идею безопасности с точки зрения простого для понимания графического метода. Несмотря на то, что графический подход не переносится напрямую в пригодный к употреблению алгоритм, он дает прекрасное интуитивное понимание существа вопроса.

На рис. 5 представлена модель для системы с двумя процессами и двумя ресурсами, например принтером и плоттером. Горизонтальная ось отображает номера команд, выполняемых процессом A. По вертикальной оси показаны номера команд, выполняемых процессом B. В команде I_1 процесс A запрашивает принтер, в команде I_2 ему требуется плоттер. Принтер и плоттер освобождаются командами I_3 и I_4 соответственно. Процессу B необходим плоттер с команды I_5 по команду I_7 и принтер с команды I_6 по команду I_8.

Каждая точка на диаграмме представляет совместное состояние двух процессов. Изначально система находится в точке p, когда ни один процесс еще не выполнил ни одну инструкцию. Если планировщик запустит процесс A первым, мы попадем в точку q, в которой процесс A выполнил какое-то количество команд, а процесс B еще ничего не сделал. В точке q траектория становится вертикальной, показывая, что планировщик решил запустить в работу процесс B. При наличии одного процессора все отрезки траектории могут быть только вертикальными или горизонтальными, но не наклонными. Кроме того, движение всегда происходит на север или восток (вверх и вправо), и никогда на юг или запад (вниз и влево), так как процессы не могут работать в обратном направлении.

Когда процесс A пересекает линию I_1 на отрезке от точки r до точки s, он запрашивает и получает принтер. Когда процесс B достигает точки t, он запрашивает плоттер.

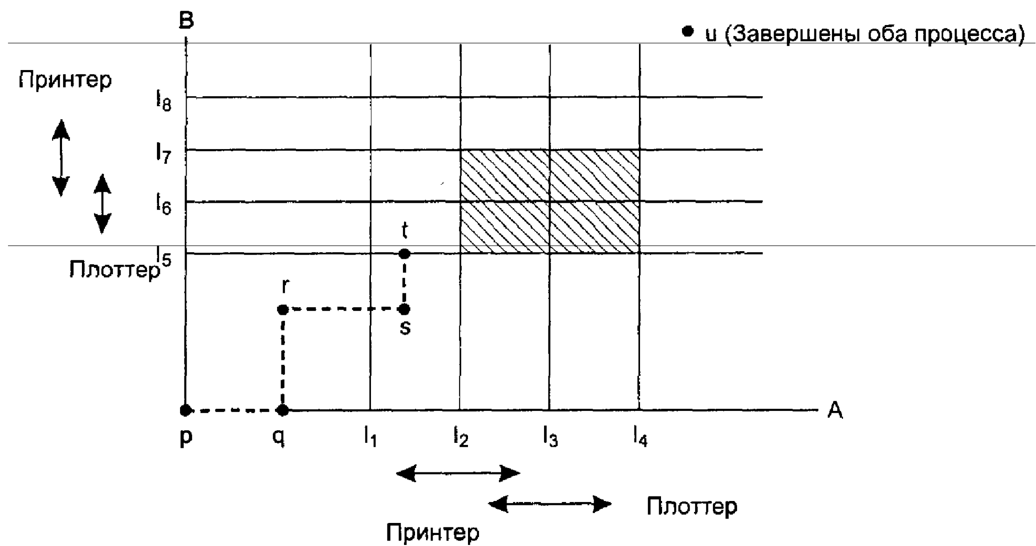


Рис. 5. Две траектории ресурсов процессов

Особенно интересны заштрихованные области. Область со штриховкой из верхнего левого угла в правый нижний представляет промежуток времени, когда оба процесса занимают принтер. Правило взаимного исключения делает попадание в эту область невозможным. Вторая заштрихованная область соответствует тому, что оба процесса используют плоттер, и это также невозможно.

Если система войдет в прямоугольник, ограниченный линиями I_1 и I_2 по сторонам и линиями I_5 и I_6 сверху и снизу, она в конце концов доберется до пересечения линий I_2 и I_6, попадет в тупик. В этот момент процесс A запросит плоттер, а процесс B потребует принтер, но оба ресурса будут к тому времени заняты. Получается, что небезопасным является целый прямоугольник, и в него нельзя входить. В точке t единственно безопасный вариант состоит в том, чтобы оставить процесс A работать до тех пир, пока он не достигнет команды I_4. После нее любая траектория дойдет до точки u.

Важный для понимания момент заключается в том, что в точке t процесс B запрашивает ресурс. Система должна принять решение: предоставлять его или нет. Если выдается разрешение, система попадает в небезопасную область и в итоге блокируется. Чтобы избежать тупика, нужно приостановить процесс B до тех пор, пока процесс A не запросит и не освободит плоттер.

Безопасные и небезопасные состояния

Алгоритмы предотвращения взаимоблокировок, которые мы будем изучать дальше, используют информацию рис. 4. В любой момент времени существует текущее состояние, составленное из величин E, A, C и R. Говорят, что состояние безопасно, если оно не находится в тупике и существует некоторый порядок планирования, при котором каждый процесс может работать до завершения, даже если все процессы вдруг захотят немедленно получить свое максимальное количество ресурсов. Проще всего проиллюстрировать эту идею на примере с одним ресурсом. На рис. 6, а у нас есть состояние, в котором процесс A занимает 3 экземпляра ресурса, но ему в итоге могут потребоваться 9 экземпляров. Процесс B в настоящий момент занял 2 экземпляра, но позже ему могут понадобиться всего 4. Процесс C владеет двумя, но может потребовать еще 5 штук. В системе есть всего 10 экземпляров данного ресурса, 7 из них уже распределены, три пока свободны.

	Имеет	Max
A	3	9
B	2	4
C	2	7

Свободно: 3

а

	Имеет	Max
A	3	9
B	4	4
C	2	7

Свободно: 1

б

	Имеет	Max
A	3	9
B	0	-
C	2	7

Свободно: 5

в

	Имеет	Max
A	3	9
B	0	-
C	7	7

Свободно: 0

г

	Имеет	Max
A	3	9
B	0	-
C	0	-

Свободно: 7

д

Рис. 6. Демонстрация того, что состояние а – безопасно

Состояние на рис. 6(а) безопасно, потому что существует такая последовательность предоставления ресурсов, которая позволяет завершиться всем процессам. А именно, планировщик может просто запустить в работу только процесс В на то время, пока он запросит и получит два дополнительных экземпляра ресурса, что приведет к состоянию, изображенному на рис. 6(б). Когда процесс В закончится, мы получим состояние рис. 7(в). Затем планировщик может запустить процесс С, что со временем приведет нас к ситуации рис. 6(г). По завершении процесса С, получим рис. 6(д) Теперь процесс А наконец может занять необходимые ему шесть экземпляров ресурса и также успешно завершиться. Таким образом, состояние на рис. 6(а) является безопасным, потому что система может избежать тупика с помощью аккуратного планирования процессов.

Разница между безопасным и небезопасным состоянием заключается в следующем: в безопасном состоянии система может **гарантировать**, что все процессы закончат свою работу, а в небезопасном состоянии такой **гарантии дать нельзя**.

Алгоритм банкира для одного вида ресурсов

Алгоритм планирования, позволяющий избегать взаимоблокировок, был разработан Дейкстрой (Dijkstra) и носит название **алгоритма банкира**. Он представляет собой расширение алгоритма обнаружения тупиков, о котором было рассказано ранее. Модель алгоритма основана на примере банкира в маленьком городке, имеющего дело с группой клиентов, которым он выдал ряд кредитов. Алгоритм проверяет, ведет ли выполнение каждого запроса к небезопасному состоянию. Если да, то запрос отклоняется. Если удовлетворение запроса к ресурсу приводит к безопасному состоянию, ресурс предоставляется процессу. На рис. 7(а) видим четырех клиентов: А, В, С и D, каждый из которых получил определенное количество единиц кредита (например, 1 единица равна 1 К долларов). Банкир знает, что не всем клиентам понадобится их максимальный кредит немедленно, поэтому он зарезервировал только 10 единиц, а не все 22, которые требуются клиентам. (Чтобы провести аналогию с компьютерной системой, считаем, что клиенты — процессы, единицами, скажем, являются накопители на магнитной ленте, а банкир — ОС.)

	Имеет	Max
A	0	6
B	0	5
C	0	4
D	0	7

Свободно: 10

а

	Имеет	Max
A	1	6
B	1	5
C	2	4
D	4	7

Свободно: 2

б

	Имеет	Max
A	1	6
B	2	5
C	2	4
D	4	7

Свободно: 1

в

Рис. 7. Три состояния распределения ресурсов: безопасное (а), безопасное (б), небезопасное (в)

Клиенты вращаются в соответствующем бизнесе, время от времени прося у банка ссуды (то есть запрашивая ресурсы). В некоторый момент возникает ситуация, показанная на рис. 7(б). Это состояние безопасно, потому что остались две единицы и банкир может задержать все обращения, кроме запросов клиента или процесса С, таким образом, позволяя процессу С завершиться и вернуть все четыре отданных ему ресурса. Имея на руках четыре единицы, банкир может отдать их или клиенту D, или B, обеспечивая их необходимыми единицами и т. д.

Рассмотрим, что могло бы произойти, если бы в ситуации на рис. 7(б) был бы удовлетворен запрос еще одной единицы для клиента B. Мы попали бы в состояние рис. 7(в), не являющееся безопасным. Если бы все клиенты вдруг запросили максимальные ссуды, то банкир не смог бы их обеспечить и мы попали бы в тупик. Небезопасное состояние не обязано приводить к взаимоблокировке, так как клиентам не обязательно потребуется весь доступный кредит, но банкир не может рассчитывать на такую ситуацию.

Алгоритм банкира рассматривает каждый запрос по мере поступления и проверяет, приведет ли его удовлетворение к безопасному состоянию. Если да, то процесс получает ресурс, иначе запрос откладывается на более позднее время. Чтобы понять, является ли состояние безопасным, банкир проверяет, может ли он предоставить достаточно ресурсов для завершения работы какого-либо клиента. Если да, то эти ссуды считаются погашенными, после чего проверяется следующий ближайший к пределу займа клиент и т. д. Если, в конце концов, все ссуды могут быть погашены, состояние является безопасным и исходный запрос можно удовлетворить.

Алгоритм банкира для нескольких видов ресурсов

Алгоритм банкира можно обобщить для управления системой с несколькими видами ресурсов. На рис. 8 показано, как он работает.

Процесс	Накопители на магнитных дисках	Плоттеры	Сканеры	Устройства для чтения компакт-дисков
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Распределенные ресурсы

Процесс	Накопители на магнитных дисках	Плоттеры	Сканеры	Устройства для чтения компакт-дисков
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Ресурсы, которые еще нужны

E = (6342)
P = (5322)
A = (1020)

Рис. 8. Алгоритм банкира в системе с несколькими ресурсами

На рис. 8 изображены:

- матрица слева показывает, сколько ресурсов каждого вида занимает в настоящее время каждый из пяти процессов;
- матрица справа показывает количество ресурсов, которое нужно добавить каждому процессу для успешного завершения.

Эти матрицы на рис. 4 назывались C и R. Как и в случае одного вида ресурсов, процессы должны точно определять необходимое суммарное количество ресурсов до начала работы для того, чтобы система могла рассчитать правую матрицу в каждый момент времени.

Три вектора, изображенные справа от матриц, показывают, соответственно, существующие ресурсы (вектор E), занятые ресурсы (вектор P) и доступные ресурсы (вектор A). Из вектора E видим, что система имеет шесть накопителей на магнитной ленте, три плоттера, четыре принтера и два устройства для чтения компакт-дисков. Из них заняты в данный момент пять накопителей, три плоттера, два принтера и два устройства для чтения компакт-дисков. Чтобы увидеть этот факт, нужно

просуммировать четыре столбца, соответствующие ресурсам, в левой матрице. Вектор доступных ресурсов является разницей между тем, что присутствует в системе, и тем, что используется в настоящее время.

Теперь можно изложить алгоритм для проверки безопасности состояния системы.

1. Ищем в матрице R строку, соответствующую процессу, чьи неудовлетворенные потребности ресурсов меньше или равны вектору А. Если такой строки не существует, то система в конце концов попадет в тупик, так как ни один процесс не может проработать до успешного завершения.

2. Допускаем, что процесс, строку которого выбрали в пункте 1, запрашивает все необходимые ресурсы (гарантируется, что это возможно) и заканчивает работу. Отмечаем этот процесс как завершенный и прибавляем все его ресурсы к вектору А.

3. Повторяем шаги 1 и 2 до тех пор, пока или все процессы будут помечены как завершенные — и состояние в этом случае является безопасным, или произойдет взаимоблокировка — тогда состояние небезопасно.

Если на первом шаге можно выбрать несколько процессов, не имеет значения, какой из них будет взят: общий резерв доступных ресурсов или увеличится или, в худшем случае, останется неизменным.

Теперь вернемся к примеру на рис. 8. Текущее состояние является безопасным. Предположим, что процесс В в данный момент запрашивает принтер. На этот запрос можно ответить положительно, потому что получающееся в результате состояние все еще будет безопасным (процесс D может доработать до конца, затем процесс А или Е, затем остальные).

Теперь представим, что после того, как процесс В получил один из двух оставшихся принтеров, процесс £ потребует последний принтер. Удовлетворение этого запроса сократит вектор доступных ресурсов до (1 0 0 0), что приведет к взаимоблокировке процессов. Ясно, что следует отложить на время запрос процесса Е.

Алгоритм замечателен в теории, на практике он, по существу, бесполезен, потому что нечасто можно определить заранее, сколько ресурсов потребуется процессам в будущем. Кроме того, количество процессов не фиксировано, оно динамически изменяется.

Предотвращение взаимоблокировок

Как мы видели, уклонение от взаимоблокировок, в сущности, невозможно, потому что оно требует наличия никому не известной информации о будущих процессах. Тогда возникает справедливый вопрос: как же реальные системы избегают попадания в тупики? Для того чтобы ответить на этот вопрос, вернемся назад к четырем условиям, сформулированным в **Условия взаимоблокировки**, и посмотрим, смогут ли они дать нам ключ к разрешению проблемы. Если мы сможем гарантировать, что хотя бы одно из этих условий никогда не будет выполнено, тогда взаимоблокировки станут конструктивно невозможными.

Атака условия взаимного исключения

Сначала попробуем атаковать условие взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, то никогда не попадем в тупик. Но в равной степени понятно, что если позволить двум процессам одновременно печатать данные на принтере, воцарится хаос. Используя подкачку выходных данных для печати, несколько процессов могут одновременно генерировать свои выходные данные. В такой модели только один процесс, который фактически запрашивает физический принтер, является демоном (сервисом) принтера. Так как демон не запрашивает никакие другие ресурсы, для принтера мы можем исключить тупики.

К сожалению, не все устройства поддерживают подкачку данных (таблицу процессов невозможно подкачивать с диска). Кроме того, конкуренция за дисковое пространство для подкачки сама по себе может привести к тупику. Что получится, если два процесса заполнили своими выходными данными каждый по половине дискового пространства, отведенного под подкачку данных, и ни один из них не закончил вычисления? Демон может быть запрограммирован так, что начнет печать, не дожидаясь подкачки всех выходных данных, и принтер тогда простоит впустую в том случае, если вычисляющий процесс решил подождать несколько часов после первого пакета выходных данных. По этой причине обычно демоны программируют так, что они начинают печать только

после того, как файл выходных данных целиком станет доступен. В этом случае мы получаем два процесса, каждый из которых обработал часть выходных данных, но не все и не может продолжать вычисления дальше. Ни один из двух процессов никогда не завершится, так что произошла взаимоблокировка на диске.

Тем не менее, в этом проглядывает росток часто применяющегося решения. **Избегайте выделения ресурса, когда это не является абсолютно необходимым, и пытайтесь обеспечить ситуацию, в которой фактически претендовать на ресурс может минимальное количество процессов**.

Атака условия удержания и ожидания

Второе из условий, сформулированных Коффманом (Coffman) и другими, кажется, все же подает надежду. **Если сможем уберечь процессы, занимающие некоторые ресурсы, от ожидания остальных ресурсов, то устраним ситуацию взаимоблокировки**. Один из способов достижения этой цели состоит в требовании, следуя которому любой процесс должен запрашивать все необходимые ресурсы до начала работы. Если все ресурсы доступны, процесс получит все, что ему нужно, и сможет работать до успешного завершения. Если один или несколько ресурсов заняты, процессу ничего не предоставляется, и он непременно попадает в состояние ожидания.

Проблемы применения:

1. **Первая проблема** при этом подходе заключается в том, что многие процессы не знают, сколько ресурсов им понадобится, до тех пор, пока не начнут работу. На самом деле, если бы они обладали подобными сведениями, то мог бы использоваться и алгоритм банкира.

2. **Другая проблема** состоит в том, что при этом методе ресурсы не будут использоваться оптимально. Возьмем, например, процесс, который читает данные с входной ленты, анализирует их в течение часа и затем пишет выходную ленту, а заодно и чертит результаты на плоттере. Если все ресурсы нужно запрашивать заранее, то процесс в течение часа не позволит работать накопителю на магнитной лепте и принтеру.

И все-таки некоторые пакетные системы на мэйнфреймах требуют, чтобы пользователи объявляли список всех ресурсов в первой строке каждого задания. Затем система немедленно запрашивает все ресурсы и сохраняет их до окончания задачи. **Этот способ накладывает ограничения на деятельность программиста и занимается расточительством ресурсов, зато предотвращает безвыходные тупиковые ситуации.**

Немного отличный метод, позволяющий нарушить условие удержания и ожидания, заключается в наложении следующего требования на процесс, запрашивающий ресурс: **процесс сначала должен временно освободить все используемые им в данный момент ресурсы. Затем этот процесс пытается сразу получить все необходимое**.

Атака условия отсутствия принудительной выгрузки ресурса

Попытка **исключить третье условие** (нет принудительной выгрузки ресурса) подает еще меньше надежд, чем устранение второго условия. Если процесс получил принтер и в данный момент печатает выходные данные, насильственное изъятие принтера по причине недоступности требуемого плоттера в лучшем случае сложно, а в худшем — невозможно.

Атака условия циклического ожидания

Остается только одно условие. Циклическое ожидание можно устранить несколькими способами.

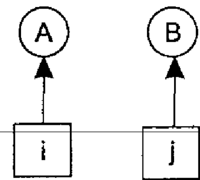
Один их них: просто следовать правилу, гласящему, что **процессу дано право только на один ресурс в конкретный момент времени**. Если нужен второй ресурс, процесс обязан освободить первый. Но подобное ограничение неприемлемо для процесса, копирующего огромный файл с магнитной ленты на принтер.

Другой способ уклонения от циклического ожидания заключается в **поддержке общей нумерации всех ресурсов**, как показано на рис. 9(а). Тогда действует следующее правило: **процессы могут запрашивать ресурс, когда хотят этого, но все запросы должны быть сделаны в соответствии с нумерацией ресурсов**.

Процесс может запросить сначала принтер, затем накопитель на магнитной ленте, но не может сначала потребовать плоттер, а затем принтер.

1. Фотонаборное устройство
2. Сканер
3. Плоттер
4. Накопитель на магнитной ленте
5. Устройство для чтения компакт-дисков

а



б

Рис. 9. Пронумерованные ресурсы (а); граф ресурсов (б)

При выполнении такого правила граф распределения ресурсов никогда не бу-дет иметь циклов. Покажем, что это так, в случае двух процессов, рис. 9(б). Мы можем попасть в тупик, только если процесс A запросит ресурсу, а процесс B обратится к ресурсу i. Предположим, что ресурсы i и j различны, значит, они имеют разные номера. Если г >j, тогда процессу A не позволяется запрашивать ресурсу, потому что его номер меньше, чем номер уже имеющегося у него ресурса. Если же i <j, тогда процесс B не может запрашивать ресурс i, потому что этот номер меньше номера уже занятого им ресурса. Так или иначе, взаимоблокировка невозможна.

При работе с несколькими процессами сохраняется та же самая логика. В каждый момент времени один из предоставленных ресурсов будет иметь наивысший номер. Процесс, использующий этот ресурс, уже никогда не запросит другие занятые ресурсы. Он или закончит свою работу или, в худшем случае, запросит ресурс с еще большим номером, а любой такой ресурс окажется доступен. В итоге процесс завершит работу и освободит свои ресурсы. На этот момент сложится ситуация, когда ресурс с высшим номером уже занят каким-то другим процессом, который так же сможет закончить свою работу. То есть существует алгоритм, по которому все процессы завершатся без попадания в тупик.

Вариантом этого алгоритма является схема, в которой отбрасывается требование приобретения ресурсов в строго возрастающем порядке, но сохраняется условие, что процесс не может запросить ресурсы с меньшим номером, чем уже у него имеющиеся. Если процесс на начальной стадии запрашивает ресурсы 9 и 10, затем освобождает их, то это равнозначно тому, как если бы он начал работу заново, поэтому нет причины теперь запрещать ему запрос ресурса 1.

Несмотря на то что систематизация ресурсов с помощью их нумерации устраняет проблему взаимоблокировки, бывают ситуации, когда невозможно найти порядок, удовлетворяющий всех. Когда ресурсы включают в себя области таблицы процессов, дисковое пространство для подкачки данных, закрытые записи базы данных и другие абстрактные ресурсы, число потенциальных ресурсов и вариантов их применений может быть настолько огромным, что никакая систематизация не сможет работать.

В табл. 1 подведены итоги различных методов для предотвращения тупиков.

Таблица 1. Методы предотвращения тупиков

Условие	Метод
Взаимное исключение	Организовывать подкачку данных
Удержание и ожидание	Запрашивать все ресурсы на начальной стадии
Нет принудительной выгрузки ресурса	Отобрать ресурсы
Циклическое ожидание	Пронумеровать ресурсы и упорядочить

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. −360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. −680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. −576 с.

5. Microsoft Development Network. URL: http://msdn.com