**Курс*:\_\_\_****2\_\_\_\_\_\_\_\_,* **группа(ы)\_\_\_\_\_\_\_\_\_\_\_\_\_***ПКС-189* **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Дисциплина (МДК)** *\_\_\_Операционные системы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

**ФИО преподавателя***\_\_\_\_\_Гарифуллин Альберт Робертович\_\_\_*

**Тема: Алгоритм банкира.**

**Введение**

В данной лекции продолжается изложение темы, начатой в "Тупики (deadlocks), методы предотвращения и обнаружения тупиков" , - методы и алгоритмы борьбы с тупиками при распределении ресурсов операционной системой. Рассмотрены следующие вопросы:

* Определение безопасного *состояния системы*
* Алгоритм построения графа *распределения ресурсов*
* *Алгоритм банкира* для безопасного *распределения ресурсов* (с избежанием *тупиков*)
* Принципы обнаружения *тупиков*
* Граф wait-for
* Алгоритм обнаружения *тупиков* и его использование
* Восстановление после *тупика*
* Комбинированный подход к обработке *тупиков*.

**Безопасное состояние системы**

**Безопасным состоянием**назовем такое состояние, перевод системы в которое не приведет к появлению *тупиков*.

Общий принцип избежания *тупиков* состоит в следующем. Когда процесс запрашивает доступный *ресурс*, система должна определить, приведет ли немедленное выделение данного ресурса к *безопасному состоянию* системы.

Система находится в **безопасном состоянии**, если существует безопасная последовательность, состоящая из всех процессов в системе.

**Безопасной последовательностью процессов**называется последовательность процессов <P1, … Pn>, такая, что для каждого процесса Pi ресурсы, которые он может еще запросить, могут быть выделены из текущих доступных ресурсов и ресурсов, удерживаемых процессами Pj , где j < i.

Если последовательность процессов безопасна, то система может придерживаться следующей безопасной стратегии, с точки зрения *распределения ресурсов* и исполнения процессов:

* Если потребности процесса Pi в ресурсах не могут быть немедленно удовлетворены, то процесс может подождать, пока завершатся процессы Pj (где j < i ), удерживающие требуемые ресурсы;
* Когда процессы Pj завершены, процесс Pi может получить требуемые ресурсы, выполниться, вернуть удерживаемые ресурсы и завершиться;
* После *завершения процесса* Pi , процесс Pi+1 может получить требуемые им ресурсы, и т.д.

Таким образом, справедливы следующие утверждения:

* Если система в *безопасном состоянии*, *тупиков* нет;
* Если системы в небезопасном состоянии, тупики возможны;
* Для того, чтобы избежать *тупиков*, необходимо проверять перед выделением ресурсов, что система никогда не придет в небезопасное состояние.

**Модифицированный вариант графа распределения ресурсов для стратегии избежания тупиков**

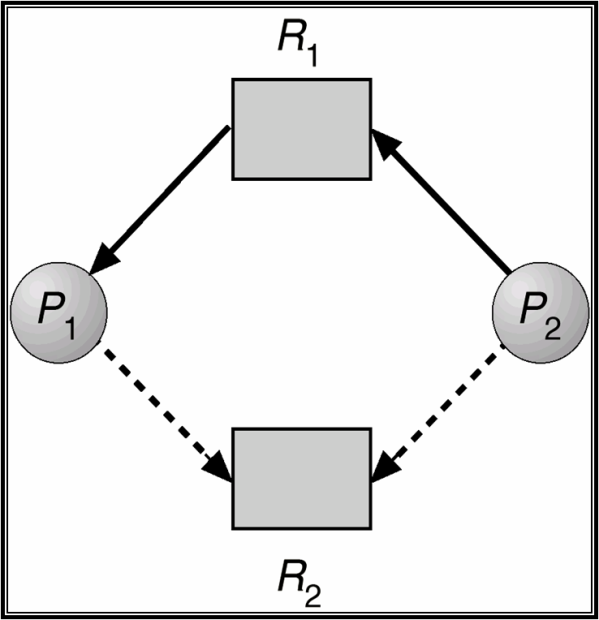
*Граф* *распределения ресурсов* рассмотрен в "Тупики (deadlocks), методы предотвращения и обнаружения тупиков" . Для реализации стратегии избежания *тупиков* к данному графу необходимо добавить информацию не только о фактических, но и о **возможных**в будущем запросах ресурсов со стороны процессов. Для этого, в *дополнение* к дугам запросов и присваиваний, введем в рассмотрение **дугу потребности (claim edge)**, которая ведет из вершины-процесса Pi в вершину-*ресурс* Rj, обозначается пунктирной линией и означает, что процесс Pi может потребовать *ресурс* Rj.

Когда процесс фактически запрашивает данный *ресурс*, *дуга* потребности преобразуется в дугу запроса (пунктирная линия заменяется сплошной).

Когда процесс освобождает *ресурс*, *дуга* присваивания преобразуется обратно в дугу потребности.

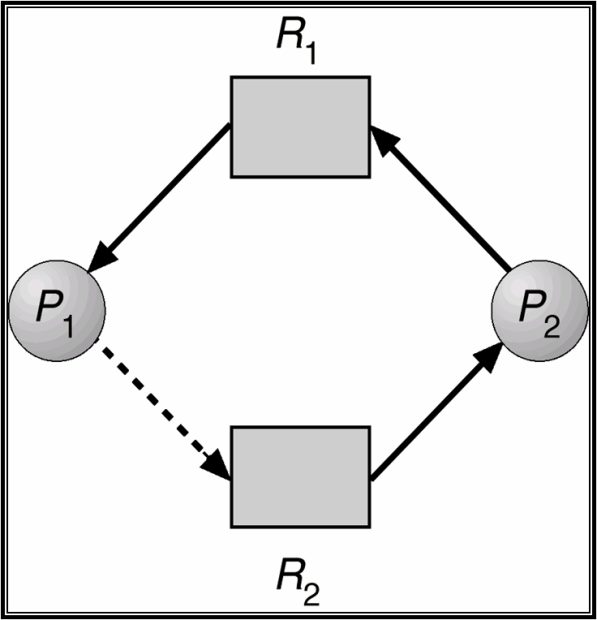
Цель данной модификации графа – обеспечить, чтобы потребность в ресурсах была априорно известна системе.

Пример графа *распределения ресурсов* для стратегии избежания *тупиков* приведен на рис. 14.1.



**Рис. 14.1.**Пример графа распределения ресурсов для стратегии избежания тупиков.

Легко видеть, что небезопасные *состояния системы* отображаются циклами в модифицированном графе *распределения ресурсов*. Пример небезопасного состояния на графе *распределения ресурсов* приведен на рис. 14.2.



**Рис. 14.2.**Пример небезопасного состояния на графе распределения ресурсов.

**Принципы алгоритма банкира**

*Алгоритм банкира* для безопасного *распределения ресурсов* операционной системой (с избежанием *тупиков*) был предложен Э. Дейкстрой и впервые реализован в операционной системе THE в конце 1960-х гг. Происхождение названия связано с тем, что поведение алгоритма напоминает осторожную стратегию банкира при проведении банковских операций. Принципы *алгоритма банкира* следующие.

* Каждый процесс должен априорно обозначить свои потребности в ресурсах по максимуму.
* Когда процесс запрашивает ресурс, ему, возможно придется подождать (выделение ресурсов по запросу не всегда может произойти немедленно).
* Когда процесс получает требуемые ресурсы, он должен их вернуть системе за ограниченный период времени.

**Структуры данных для алгоритма банкира**

Пусть в системе имеется n процессов и m типов ресурсов.

*Вектор* Available длины m содержит информацию о **доступных**ресурсах. Если Avaliable[j] = k,то в системе в данный момент доступно k единиц ресурса j.

*Матрица* Max ( n \* m ) отображает **максимальные**потребности процессов в ресурсах. Если Max [i, j] = k,то процесс i может запросить, самое большее, k единиц ресурса j.

*Матрица* *Allocation* ( n \* m ) отображает **фактическое**выделение системой ресурсов. Если *Allocation* [i, j] = k,то процессу i в данный момент выделено системой k единиц ресурса j.

*Матрица* Need ( n \* m ) отображает **оставшиеся**потребности процессов в ресурсах. Если Need [i, j] = k,то процессу i могут потребоваться еще k единиц ресурса j для завершения работы.

Имеет *место* следующее соотношение между элементами матриц:

Need [i, j] = Max [i, j] – *Allocation* [i, j].

**Алгоритм проверки состояния системы на безопасность**

В обозначениях раздела **Структуры данных для алгоритма банкира**, рассмотрим *алгоритм* проверки *состояния системы* на то, является ли оно безопасным.

Введем целочисленный *вектор* Work (длины m ) и булевский *вектор* Finish (длины n ). *Вектор* Work отображает пробные выделения ресурсов. *Вектор* Finish представляет информацию о завершаемости процессов при данном *состоянии системы*.

**Алгоритм безопасности**.

Шаг 1. Инициализация.

Work = Available

Finish [i] = false для i = 1, …, n.

Здесь и в дальнейшем все присваивания и сравнения, в которых участвуют векторы или матрицы, выполняются **поэлементно**.

Шаг 2. Находим i, такое, что:

Finish [i] = false

Need [i] <= Work

Если такого i нет, переходим к шагу 4.

Шаг 3.

Work = Work + Allocation [i]

Finish [i] = true

Переход к шагу 2.

Шаг 4. Если Finish[i] = true для всех i = 1, …, n, то система в безопасном состоянии.

Необходимые пояснения к алгоритму:

* Алгоритм строит безопасную последовательность номеров процессов i, если она существует. На каждом шаге, после обнаружения очередного элемента последовательности, алгоритм моделирует освобождение i - м процессом ресурсов после его завершения.
* На шаге 1 присваивание векторов выполняется поэлементно.
* На шаге 2, Need – матрица потребностей ( n \* m ); Need[i] - строка матрицы, представляющая вектор потребностей (длины m ) процесса i. Сравнение выполняется поэлементно, и его результат считается истинным, если соотношение выполнено для всех элементов векторов. Проверяемое условие означает, что потребности процесса i с найденным номером могут быть удовлетворены немедленно, и процесс может получить их и завершиться.
* На шаге 3, *Allocation* [i] – строка матрицы *Allocation*, обозначающая текущие ресурсы, выделенные процессу i. С помощью вектора Work моделируется *освобождение ресурсов* i – м процессом, после чего процессу присваивается признак завершаемости.

Формальное *доказательство корректности* алгоритма и оценку его сложности предоставляем студенту.

**Алгоритм запроса ресурсов для процесса Pi – основная часть алгоритма банкира**

Для основного алгоритма введем *вектор* Requesti (длины m ) – *вектор* запросов для процесса Pi . Если Requesti [j] = k,то процесс Pi запрашивает k экземпляров ресурса Rj.

Шаг 1. Если Requesti <= Need[i], перейти к шагу 2.

Иначе – сгенерировать исключительную ситуацию

(процесс превысил свои максимальные потребности).

Шаг 2. Если Requesti <= Available, перейти к шагу 3.

Иначе процесс должен ждать, так как ресурс недоступен.

Шаг 3. Спланировать выделение ресурса процессу Pi , модифицируя состояние системы следующим образом:

Available = Available - Requesti

Allocation = Allocation + Requesti

Need [i] = Need [i] - Requesti

Вызвать алгоритм проверки безопасности полученного состояния.

Если состояние безопасно, выделить ресурс процессу Pi . Выход.

Если состояние небезопасно, восстановить предыдущее состояние;

процесс должен ждать.

**Пример использования алгоритма банкира**

Пусть имеется 5 процессов – P0 , …, P4 , и 3 типа ресурсов – *ресурс* **A**(10 экземпляров), *ресурс* **B**(5 экземпляров) и *ресурс* **C**(7 экземпляров). Пусть *состояние системы* в момент **T0**следующее:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ***Allocation*** | | | **Max** | | | **Available** | | |
|  | **A** | **B** | **C** | **A** | **B** | **C** | **A** | **B** | **C** |
| **P0** | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| **P1** | 2 | 0 | 0 | 3 | 2 | 2 |  |  |  |
| **P2** | 3 | 0 | 2 | 9 | 0 | 2 |  |  |  |
| **P3** | 2 | 1 | 1 | 2 | 2 | 2 |  |  |  |
| **P4** | 0 | 0 | 2 | 4 | 3 | 3 |  |  |  |

Вычислим матрицу потребностей Need = Max – *Allocation*:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Need** | | |
|  | **A** | **B** | **C** |
| **P0** | 7 | 4 | 3 |
| **P1** | 1 | 2 | 2 |
| **P2** | 6 | 0 | 0 |
| **P3** | 0 | 1 | 1 |
| **P4** | 4 | 3 | 1 |

Нетрудно видеть, что система – в *безопасном состоянии*. Последовательность процессов <P1, P3, P4, P2, P0> удовлетворяет критерию безопасности. Проверку предоставляем студенту.

В продолжение примера, предположим, что процесс P1 сделал *запрос* (1 0 2). Проверяем, что Request <= Available: <(1 0 2) <= (3 3 2) = true.

Удовлетворяем *запрос*.

*Состояние системы* принимает вид:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ***Allocation*** | | | **Max** | | | **Available** | | |
|  | **A** | **B** | **C** | **A** | **B** | **C** | **A** | **B** | **C** |
| **P0** | 0 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 0 |
| **P1** | 3 | 0 | 2 | 0 | 2 | 0 |  |  |  |
| **P2** | 3 | 0 | 1 | 6 | 0 | 0 |  |  |  |
| **P3** | 2 | 1 | 1 | 0 | 1 | 1 |  |  |  |
| **P4** | 0 | 0 | 2 | 4 | 3 | 1 |  |  |  |

*Исполнение* алгоритма безопасности показывает, что последовательность процессов <P1, P3, P4, P0, P2> удовлетворяет критерию безопасности. Предоставляем студенту проверку корректности данных преобразований и предлагаем ответить на следующие дополнительные вопросы:

* может ли быть удовлетворен запрос (3 3 0) для процесса P4 ?
* может ли быть удовлетворен запрос (0 2 0) для процесса P0 ?

### Методы обнаружения тупиков

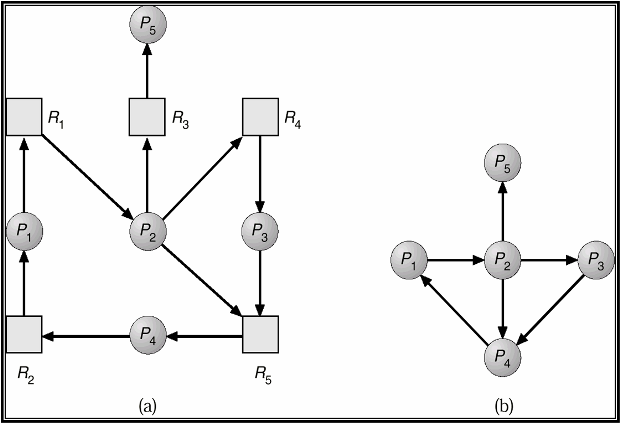
Как уже отмечалось в "Тупики (deadlocks), методы предотвращения и обнаружения тупиков" , альтернативным подходом к решению проблемы *тупиков* является обнаружение *тупиков*. При таком подходе система может позволить себе войти в состояние *тупика*. После этого применяется *алгоритм* обнаружения *тупиков*. После обнаружения *тупика* применяется схема восстановления после *тупика*.

### 

### Граф wait-for

В *дополнение* к графу *распределения ресурсов*, введем более простой по струтуре **граф wait-for**: вершины в нем соответствуют процессам, и *дуга* проводится из вершины Pi в вершину Pj, если процесс Pi ожидает процесса Pj . Если каждый тип ресурса в системе существует в единственном экземпляре, то очевидно, что цикл в данном графе означает *тупик*. Система для обнаружения *тупиков* должна периодически проверять отсутствие циклов в графе wait-for. Как известно, *алгоритм* обнаружения *цикла* в графе требует O(n2) операций, где n – число вершин в графе.

На рис. 14.3 приведен пример графа *распределения ресурсов* и соответствующего ему графа wait-for для системы с тупиком.



**Рис. 14.3.**Граф распределения ресурсов и граф wait-for.

### Обнаружение тупиков для случая ресурсов с множественными экземплярами

В общем случае для построения алгоритма обнаружения *тупиков* будем использовать те же структуры, что и для *алгоритма банкира*:

*Вектор* Available длины m содержит информацию о **доступных**ресурсах. Если Avaliable[j] = k,то в системе в данный момент доступно k единиц ресурса j.

*Матрица* *Allocation* ( n \* m ) отображает **фактическое**выделение системой ресурсов. Если *Allocation* [i, j] = k,то процессу i в данный момент выделено системой k единиц ресурса j.

*Вектор* Requesti (длины m ) – *вектор* запросов для процесса Pi . Если Requesti [j] = k,то процесс Pi запрашивает k экземпляров ресурса Rj.

### Алгоритм обнаружения тупиков

Аналогично алгоритму безопасности (раздел 14.6), введем целочисленный *вектор* Work (длины m ) и булевский *вектор* Finish (длины n ). *Вектор* Work отображает пробные выделения ресурсов. *Вектор* Finish представляет информацию о завершаемости процессов при данном *состоянии системы*.

Шаг 1. Инициализация.

Work = Available

Для i = 1, …, n, если Allocation [i] != 0 то finish [i] = false иначе finish [i] = true.

Шаг 2. Находим i, такое, что:

Finish [i] = false

Request [i] <= Work

Если такого i нет, переходим к шагу 4.

Шаг 3.

Work = Work + Allocation [i]

Finish [i] = true

Переход к шагу 2.

Шаг 4. Если Finish[i] = false для некоторого i от 1 до n, то система в состоянии тупика.

Более того, если Finish[i] = false,

то процесс Pi – в состоянии тупика.

Обоснование и *доказательство корректности* алгоритма предоставляем студенту. *Алгоритм* требует O(m \* n2) операций для определения того, находится ли система в состоянии *тупика*.

### Пример применения алгоритма обнаружения тупиков

Пусть имеется 5 процессов – P0 , …, P4 , и 3 типа ресурсов – *ресурс* **A**(7 экземпляров), *ресурс* **B**(2 экземпляра) и *ресурс* **C**(6 экземпляров). Пусть *состояние системы* в момент **T0**следующее:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ***Allocation*** | | | **Request** | | |
|  | **A** | **B** | **C** | **A** | **B** | **C** |
| **P0** | 0 | 1 | 0 | 0 | 0 | 0 |
| **P1** | 2 | 0 | 0 | 2 | 0 | 2 |
| **P2** | 3 | 0 | 3 | 0 | 0 | 0 |
| **P3** | 2 | 1 | 1 | 1 | 0 | 0 |
| **P4** | 0 | 0 | 2 | 0 | 0 | 2 |

В данном *состоянии системы* последовательность процессов <P0, P2, P3, P1, P4> безопасна (проверьте это!).

В продолжение примера, пусть процесс P2 запрашивает дополнительный *ресурс* типа **C**:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Request** | | |
|  | **A** | **B** | **C** |
| **P0** | 0 | 0 | 0 |
| **P1** | 2 | 0 | 1 |
| **P2** | 0 | 0 | 1 |
| **P3** | 1 | 0 | 0 |
| **P4** | 0 | 0 | 2 |

В данном случае имеет *место* *тупик*, в котором находятся процессы P1, P2, P3, P4 .

Проверьте это.

### Использование алгоритма обнаружения тупиков

Как часто и в каких ситуациях системе следует использовать рассмотренный *алгоритм* обнаружения *тупиков*, зависит от того, как часто, по всей вероятности, будет иметь *место* *тупик* и сколько процессов будет необходимо откатить назад, чтобы выйти из *тупика*. Ответ на последний вопрос: **по одному процессу для каждого из не пересекающихся циклов**.

Если *алгоритм* обнаружения *тупиков* будет вызываться произвольным образом, то в графе *распределения ресурсов* будет много циклов, и не будет возможности с достоверностью утверждать, какой из многих заблокированных процесов вызвал данный *тупик*.

### Восстановление после тупика

Для выхода из *тупика*, очевидно, система должна прекратить все заблокированные процессы и освободить занимаемые ими ресурсы. Для более оптимального выполнения данного действия, система может прекращать на каждом шаге по одному процессу и после этого анализировать, ликвидирован ли *тупик*.

Важный вопрос – **в каком порядке**необходимо прекращать процессы? Существуют различные подходы:

* В порядке *приоритетов процессов*;
* В зависимости от того, насколько долго процесс уже выполняется и сколько времени осталось до его завершения;
* В зависимости от объема ресурсов, которые удерживал процесс;
* В зависимости от объема ресурсов, требуемого для *завершения процесса*;
* В зависимости от того, сколько всего процессов требуется прекратить;
* В зависимости от того, является ли процесс интерактивным или пакетным.

После выбора процесса-"жертвы" с минимальной стоимостью (по одному из приведенных критериев), система прекращает выбранный процесс (процессы), освобождает их ресурсы и выполняет перераспределение ресурсов. Система выполняет "*откат*" к какому-либо предыдущему *безопасному состоянию*.

В результате многократного выполнения подобных действий системы, возможно "голодание", так как в качестве жертвы может многократно выбираться один и тот же процесс.

### Комбинированный подход к обработке тупиков

Скомбинируем три рассмотренных базовых подхода – предотвращение, избежание и обнаружение *тупиков*. Это позволит использовать оптимальный подход для каждого из *системных ресурсов*. Ресурсы могут быть разделены на иерархически упорядоченные классы, и система сможет использовать наиболее подходящий метод для обработки *тупиков* внутри каждого класса.

### Ключевые термины

**Алгоритм банкира (banker’s algorithm)**- *алгоритм* Э. Дейкстры для избежания *тупиков* при распределении ресурсов операционной системой.

**Безопасная последовательность процессов –**такая последовательность процессов <P1, … Pn>, что для каждого процесса Pi ресурсы, которые он может запросить, могут быть выделены из текущих доступных ресурсов и ресурсов, удерживаемых процессами Pj, где j < i.

**Безопасное состояние**– состояние, перевод системы в которое не приведет к появлению *тупиков*.

**Граф wait-for**- *ориентированный граф*, вершины в которой соответствуют процессам, а *дуга* проводится из вершины Pi в вершину Pj, если процесс Pi ожидает процесса Pj .

**Дуга потребности (claim edge)**– *дуга* в графе *распределения ресурсов*, которая ведет из вершины-процесса в вершину-*ресурс*, обозначается пунктирной линией и означает, что данный процесс **может потребовать**данный *ресурс*.

### Краткие итоги

*Безопасное состояние* системы – такое состояние, перевод системы в которое не приведет к созданию *тупиков*. Система в *безопасном состоянии*, если существует безопасная последовательность их всех процессов в системе. Безопасная последовательность – последовательность исполнения процессов, при которой каждый процесс использует только свободные ресурсы, либо ресурсы, освобождаемые процессом с меньшим номером после его завершения.

Таким образом, если система в *безопасном состоянии*, *тупиков* нет. Если система в небезопасном состоянии, тупики возможны. Избежание *тупиков* – стратегия, обеспечивающая, чтобы система никогда не могла прийти в небезопасное состояние.

Добавим к графу *распределения ресурсов* (лекция 13) новый вид дуг – дуги потребности. Она ведет из вершины-процесса в вершину-*ресурс*, обозначается пунктиром и означает, что процесс может потребовать данный *ресурс*. *Дуга* потребности преобразуется в дугу присваивания, когда система фактически выделяет данный *ресурс* процессу. При освобождении ресурса *дуга* присваивания преобразуется обратно в дугу потребности.

*Алгоритм банкира* (Э. Дейкстра) – *алгоритм* *распределения ресурсов* операционной системой, обеспечивающий избежание *тупиков*. Его условия и принципы: возможны множественные экземпляры ресурсов; каждый процесс должен априорно обозначить свои максимальные потребности в ресурсах; при запросе ресурса возможно ожидание процесса; после получения ресурсов процесс должен вернуть их за ограниченный период времени. Для работы алгоритма используются *вектор* доступности ресурсов каждого вида, *матрица* максимальных потребностей процессов, *матрица* фактического выделения системой ресурсов процессам, *матрица* оставшихся потребностей процессов в ресурсах.

*Алгоритм* безопасности определяет, является ли *состояние системы* безопасным, путем построения безопасной последовательности процессов. *Алгоритм* моделирует последовательное *освобождение ресурсов* процессами после их завершения.

Основной *алгоритм банкира* – *алгоритм* запроса ресурсов для процесса. Он проверяет, не превысил ли процесс свои максимальные потребности (иначе – *исключительная ситуация*). Далее проверяется, не превышает ли *запрос* имеющихся объемов доступных ресурсов, иначе – ожидание процесом их освобождения. Если все необходимые ресурсы для удовлетворения запроса имеются, *алгоритм* вычисляет новое состояние после удовлетворения запроса и проверяет его *безопасность*. Если новое состояние безопасно, *запрос* удовлетворяется. Если нет, происходит *откат*, и процесс ожидает освобождения необходимых ресурсов.

Другая возможная стратегия – обнаружение *тупиков*: позволить системе войти в состояние *тупика*, применить *алгоритм* обнаружения *тупиков* и выполнить схему восстановления после *тупика*. Если каждый *ресурс* существует в единственном экземпляре, для обнаружения *тупиков* используется *граф* wait-for, в котором вершины соответствуют процессам, а *дуга* ведет из вершины A в вершину B, если процесс A ожидает процесса B. Сложность алгоритма обнаружения *цикла* в графе wait-for – O(n\*\*2), где n – число процессов.

Если имеются множественные экземпляры ресурсов, то для обнаружения *тупиков* используется *алгоритм*, аналогичный алгоритму построения безопасной последовательности процессов. Его сложность – O (m \* n\*\*2), где m – число типов ресурсов.

После обнаружения *тупиков* система определяет, какое минимальное число процессов необходимо прекратить для ликвидации *тупика*. Для выбора процесса-жертвы существует ряд критериев, например, *приоритет процесса* или оставшееся время его выполнения. При прекращении процесса его ресурсы освобождаются. При применении данной стратегии возможно голодание, если в качестве жертвы система все время выбирает один и тот же процесс.

Комбинированный подход к обработке *тупиков* сочетает в себе все три рассмотренных стратегии – предотвращение, избежание и обнаружение. Система подразделяет ресурсы на иерархически упорядоченные классы. Для каждого класса используется наиболее *оптимальная стратегия*.

***Контрольные вопросы:***

1. Что такое *безопасное состояние* системы?
2. Что такое безопасная последовательность процессов?
3. Есть ли в системе тупики, если система находится в *безопасном состоянии*?
4. Возможны ли в системе тупики, если она находится в небезопасном состоянии?
5. В чем суть стратегии избежания *тупиков*?
6. Что такое дуга потребности в графе *распределения ресурсов*?
7. В какую дугу преобразуется дуга потребности при фактическом выделении ресурса?
8. В какую дугу преобразуется дуга присваивания при освобождении ресурса?
9. Каковы основные принципы *алгоритма банкира*?
10. Какие структуры данных используются для *алгоритма банкира*?
11. В чем идея и каковы основные шаги алгоритма определения того, является ли *состояние системы* безопасным?
12. В чем идея и каковы основные шаги алгоритма удовлетворения запроса процесса?
13. В каких случаях в алгоритме банкира процесс должени ждать освобождения ресурсов?
14. В какой момент проверяется безопасность следующего состояния в алгоритме банкира?
15. В чем основные принципы стратегии обнаружения *тупиков*?
16. Что такое граф wait-for и как он используется для обнаружения *тупиков*?
17. В чем идея и каковы основные шаги алгоритма обнаружения *тупиков* для ресурсов с множественными экземплярами?
18. Как происходит восстановление системы после *тупика*?
19. По каким принципам выбирается процесс-жертва, который необходимо прекратить для ликвидации *тупика*?
20. Почему при многократном выборе процессов-жертв для выхода из *тупиков* возможно голодание процессов?

***Примечание****:*

В понедельник 23 марта, жду от вас законченные работы по данной теме на электронную почту [tiger-r-r-r@mail.ru](mailto:tiger-r-r-r@mail.ru) Спасибо