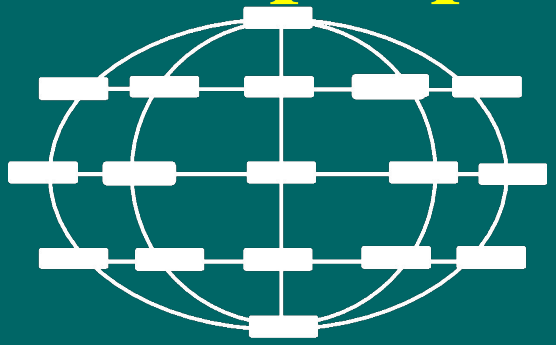
Основы Параллельного программирования



ЛЕКЦИЯ 3

Взаимодействующие процессы

В.Э.Малышкин
ИВМиМГ СО РАН
Новосибирский Национальный исследовательский университет
Новосибирск 2010
htpp://ssd.sscc.ruЛ

ВЗАИМОДЕЙСТВУЮЩИЕ ПРОЦЕССЫ

- Программа *P* для мультикомпьютера описывает систему взаимодействующих процессов и именно эта система процессов здесь рассматривается. Понятно, что вначале программа *P* загружается для исполнения в процессорные элементы (ПЭ) мультикомпьютера. При исполнении программа *P* порождает описанные в ней процессы, они в свою очередь назначаются на исполнение на процессорные элементы мультикомпьютера и тоже могут порождать процессы. Процессы должны иметь возможность обмениваться данными и синхронизовать свое исполнение. Для уточнения всех этих понятий рассмотрим вначале общую модель параллельной программы для мультикомпьютера.

Последовательные процессы

Программа *P* - объект статический и она именно описывает множество процессов и их взаимодействия. Процесс - объект динамический, он возникает только при исполнении программы. *Процессом (последовательным процессом)* называется исполняющаяся последовательная программа (*программа процесса*) со своими входными данными.

Первоначально программа *P* запускается на счет как один процесс, который обычно исполняется на управляющем компьютере (host computer) мультикомпьютера. Программа *P* фрагментирована, в ней выделяются фрагменты кода, способные выполняться независимо. При исполнении фрагменты *P* запускаются на счет как отдельные процессы (процессы *порождаются*) со своими входными данными. Для этого программный код и необходимые данные процесса пересылаются в выделенный для исполнения процесса процессорный элемент (ПЭ), процесс требует и получает необходимые для своего исполнения ресурсы и наконец стартует. Фрагменты *P* составляют программы процессов.

Порождение процесса делается специальным оператором языка программирования, исполнение которого может реализоваться системным вызовом, т.е. обращением к операционной системе (ОС) за выполнением этой работы. Программа процесса - это всем известная последовательная программа, со всеми необходимым для счета переменными и управлением. Переменные процесса, как правило, недоступны другим процессам. Все ее операторы исполняются последовательно в порядке, определенном языком программирования. Программа процесса сейчас, как правило, разрабатывается на языке С.

Одна и та же исполняющаяся программа с разными входными данными определяет разные процессы. Разные исполняющиеся программы также определяют разные процессы. Каждый процесс в ходе исполнения может быть полноправным клиентом ОС, потреблять для своего исполнения ресурсы (ПЭ, память, устройства ввода/вывода и т. п.), обмениваться информацией с другими процессами, конкурировать за ресурсы, порождать процессы, завершаться.

- Ясно, что в каждом ПЭ должна размещаться некоторая более или менее развитая ОС (это обычно одна из версий UNIX) для организации исполнения в ПЭ одного или более процессов. В совокупности эти ОС процессорных элементов составляют распределенную ОС мультикомпьютера. Для управления процессами ОС создает для каждого процесса *блок управления процессом,* который содержит описание текущего состояния процесса (имя, ресурсы, отношение к другим процессам и т.д.).
- Процессы могут порождаться и внутри программы, без участия ОС, так как порождение процесса в ОС - довольно времяемкая процедура, что не позволяет создавать небольшие процессы.

- Состояние процесса в общем случае определяется значениеми всех переменных, определенных или связанных с процессом, в частности, значениями переменных, определенных ОС для организации выполнения процесса (например, переменные блока управления процессом). Далее будут рассматриваться только взаимодействия между процессами, поэтому понятие состояния процесса для простоты ограничивается и определяется как значение особой переменной *состояние*, принимающей одно из следующих допустимых значений:
- **исполняется** - команды программы процесса исполняются, процесс активен,
- **ожидает** - процесс ждет совершения некоторого события,
- **готов** - процесс имеет все необходимые ресурсы и готов продолжить исполнение.

- В ходе исполнения параллельной программы процессы могут порождаться и завершаться (для этого в языках параллельного программирования есть специальные операторы). Процесс, породивший новый процесс, называется *родительским* процессом по отношению к порожденному новому процессу. Порожденные процессы называется *детьми* родительского процесса.
- Порожденные процессы-дети в свою очередь могут порождать новые процессы, по отношению к которым они выступают в качестве родительских процессов и таким образом программа *P* распространяется по ПЭ мультикомпьютера. Доступные ПЭ выделяются программе *P* в момент начала вычислений либо несколько программ динамически конкурируют за ПЭ. Каждый ПЭ обычно отдается в монопольное использование одной из программ .

Допускаются разные способы организации выполнения родительских процессов и их детей.

1. Порядок выполнения процессов.

- Родительский процесс порождает новые процессы и:
  - продолжает свое исполнение параллельно с процессами-детьми,
  - задерживает свое исполнение и ожидает того момента, когда завершатся все порожденные им процессы-дети.

2. Использование ресурсов.

- Родительский процесс и процессы-дети:
  - разделяют общие ресурсы,
  - все процессы имеют свои собственные не разделяемые ресурсы.

3. Завершение.

Своей командой родительский процесс может завершить выполнение своих процессов-детей. Для такого завершения может быть несколько причин: работа процессов-детей более не нужна, либо они потребляют слишком много ресурсов и не дают работать другим процессам, либо в них случились события, не позволяющие продолжить вычисления (например, нет затребованного ресурса).

Процессы-дети не могут завершить свой родительский процесс, хотя, конечно, могут служить причиной, по которой родительский процесс примет решение о своем завершении.

Для принятия решения о завершении процессов-детей родительский процесс должен знать *состояние* своих детей, таким образом, состояние процессов-детей доступно родительскому процессу.

Если родительский процесс завершается, то завершаются и все его дети. ОС должна отследить эту ситуацию, что удобно при программировании и отладке (не надо беспокоиться о «зависших» процессах). Иногда допускается и полная автономность порожденных процессов-детей.

В зависимости от возможностей влиять на ход исполнения других процессов все множество процессов разбивается на *независимые* и *взаимодействующие* процессы. Полномочия процессов влиять друг на друга определяются при их порождении.

Независимые процессы не могут влиять на выполнение других процессов (передавать данные, изменять значения переменных, останавливать и т.п.) и их исполнение не может управляться из других процессов. А потому:

- их состояние недоступно другим процессам,
- их исполнение и результат зависят только от входных данных,
- их исполнение может быть задержано ОС без оказания влияния на работу остальных процессов.
- они не разделяют ресурсы и данные с другими процессами.

Таким образом, независимые процессы полностью автономны. В частности, процессы, порожденные разными программами *P1* и *P2*, выполняются независимо друг от друга.

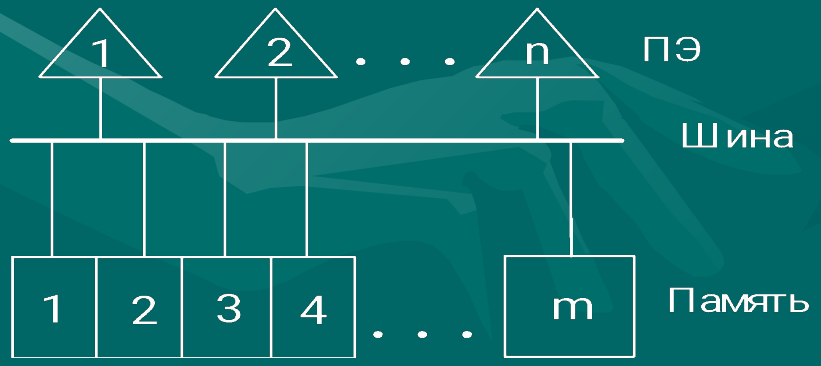
Процессы, которые совместно выполняют общую работу, должны и могут влиять друг на друга. Такие процессы:

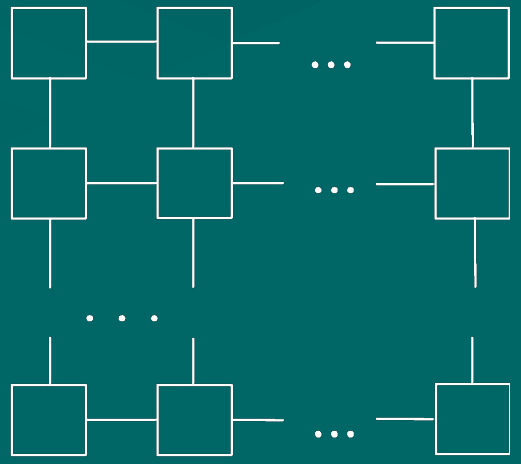
- имеют взаимный доступ к переменной *состояние*.
- их исполнение недетерминировано. При каждом новом исполнении такие процессы могут по разному влиять друг на друга (хотя бы только по времени) и, следовательно, вырабатывать в случае ошибки разные результаты.
- их исполнение в общем случае невозможно повторить. Каждое исполнение системы процессов происходит по-разному, что крайне затрудняет отладку.
- Параллельно протекающие процессы программы *P* могут выполняться в одном и том же ПЭ, разделяя ПЭ в режиме мультипрограммирования, либо на разных ПЭ мультикомпьютера, общаясь друг с другом посредством обмена данными.

Процессы взаимодействуют, передавая друг другу данные и синхронизируя ход вычислений. Предполагается, что передача данных между процессами осуществляется через особые *разделяемые* переменные, доступные всем процессам программы *P*. Одни процессы присваивают значение разделяемым переменным, а другие в нужный момент (после некоторого ожидания, если потребуется) могут считывать эти значения.

Разделяемые переменные реализуются по-разному для разных мультикомпьютеров. В мультикомпьютерах с разделяемой памятью (*shared memory*), которые часто называются также *мультипроцессорами* , разделяемые переменные могут быть реализованы как участки памяти, доступные нескольким процессам, где они могут оставлять и забирать значения переменных. Такая реализация легко осуществима, так как всем процессорам доступна вся память мультипроцессора.

В мультикомпьютерах с распределенной памятью (*distributed memory)* разделяемые переменные обычно реализуются в виде логического канала для передачи данных между процессами. Здесь ПЭ соединены коммутационной сетью, все взаимодействия между процессами сильно зависят и по-разному реализуются в зависимости от ее структуры. Большое разнообразие структур коммутационной сети значительно затрудняет программирование мультикомпьютеров с распределенной памятью. На рисунке показан мультикомпьютер со структурой решетка





Решетка ПЭ

- Понятно, что реализация доступа к разделяемой переменной существенно зависит от того, где размещаются взаимодействующие процессы. Если взаимодействующие процессы назначены на исполнение на один ПЭ, то доступ к разделяемой переменной реализуется в программе процесса обычным образом - упоминанием имени переменной в программе, например, z:=x+1; . Если же взаимодействующие процессы назначены на исполнение в разные ПЭ, то доступ к разделяемой переменной реализуется в программе процесса через логический канал передачи данных. Так что даже текстуально программы процессов могут различаются в зависимости от способа исполнения программы.

Выполнение системы процессов

- Проблема организации выполнения множества взаимодействующих процессов обсуждается в форме двух задач: *взаимное исключение* (*mutual exclusion*) и *производитель/потребитель* (*producer/consumer*).
- Задача взаимного исключения формулируется следующим образом. Пусть выполняются два или более процессов, которым необходим доступ к одному и тому же неразделяемому ресурсу. Ресурс называется *разделяемым*, если несколько процессов одновременно могут его использовать, и *неразделяемым* в противном случае. Примером разделяемого ресурса служит переменная (вещественная переменная или файл), значение которой может одновременно считываться несколькими процессами. Процессы при этом не влияют друг на друга.

Эта же переменная является *неразделяемым* ресурсом при выполнении операции записи, поскольку запись новых данных влияет на выполнение других процессов, считывающих и использующих текущее значение переменной. Поэтому доступ к переменной «по считыванию» могут одновременно иметь несколько процессов, но только один процесс может записывать новое значение переменной, запрещая при этом другим процессам и запись и считывание значения переменной.

Если считывать значение переменной в ходе выполнения записи нового значения, тогда результат считывания не будет однозначно определен (будет недетерминирован).

Взаимное исключение заключается в обеспечении доступа только одного процесса к неразделяемому ресурсу. Эта задача в действительности состоит из двух задач. Прежде всего, если несколько процессов *p1*, *p2*, ... , *pn* требуют доступа к неразделяемому ресурсу, то только один из них должен быть допущен к ресурсу.

Выбор процесса, который допускается к ресурсу, составляет вторую задачу. Здесь возможна ситуация, когда один из процессов *pi*, *i=1,2,...,n*, постоянно не выбирается и попадает в *вечное ожидание* затребованного ресурса (*starvation*). Алгоритм выбора процесса должен обеспечивать прогресс и не допускать вечного ожидания.

Задача производитель/потребитель заключается в следующем. Пусть есть пара процессов *p1* и *p2*. Один из них - *p1* (производитель) - вырабатывает некоторый результат и помещает его в буфер *v*, а другой процесс - *p2* (потребитель) - должен результат считать. Для определенности будем предполагать, что *p1* вырабатывает значение переменной и помещает его в буфер *v*, а *p2* считывает значение переменной из буфера *v*.

Если буфер *v* пуст, то потребитель должен ждать момента, когда новый результат будет произведен и помещен в буфер *v*.

- Буфер может быть *ограниченным* или *неограниченным*. Ограниченный буфер может хранить конечное число значений (не более n, n - натуральное число) переменной. Такой буфер можно представить себе как очередь значений ограниченного размера. В неограниченном буфере всегда есть место для размещения следующего значения.
- Если ограниченный буфер уже содержит n значений, то следующее значение не может быть в него помещено и процесс-производитель должен ждать момента, когда процесс-потребитель заберет очередное значение и освободит место для размещения нового значения.
- Конечно, следует рассмотреть и случай нескольких процессов-производителей и нескольких процессов-потребителей.
- Для анализа и решения этих задач необходимо прежде всего сформулировать и описать их в некотором формализованном виде. Для этой цели будут использованы сети Петри.

Сети Петри

- Сеть Петри - это поведенческая математическая модель, которая имеет широкое применение для описания поведения параллельных устройств и систем процессов. В настоящее время определены и изучены разнообразные классы сетей Петри и разработаны алгоритмы для анализа их свойств. Мы рассмотрим лишь самые общие понятия и возможности использования сетей Петри как для анализа названных задач, так и для задания прямого управления в параллельных программах. Наиболее интересны сети Петри тем, что они позволяют представлять и изучать в динамике поведение системы параллельно протекающих и взаимодействующих процессов программы или любого другого дискретного устройства.

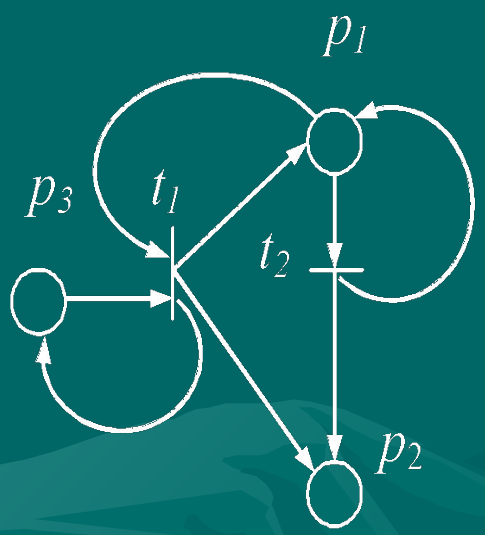
Определение сети Петри

- Определение сети Петри дадим в три приема. *Сеть* есть двудольный ориентированный граф. Напомним, что двудольный граф - это такой граф, множество вершин которого разбивается на два подмножества и не существует дуги, соединяющей две вершины из одного подмножества.

I. Итак, сеть - это набор

- $G = (T,P,A)$, $T \cap P = \varnothing$,
- где
- $T=\{t_1,t_2,...,t_n\}$-подмножество вершин, называющихся *переходы*,
- $P=\{p_1,p_2,...,p_m\}$-подмножество вершин, называющихся *местами*,
- $A \subseteq (T \times P) \cup (P \times T)$ - множество ориентированных дуг.
- По определению, ориентированная дуга соединяет либо место с переходом либо переход с местом.

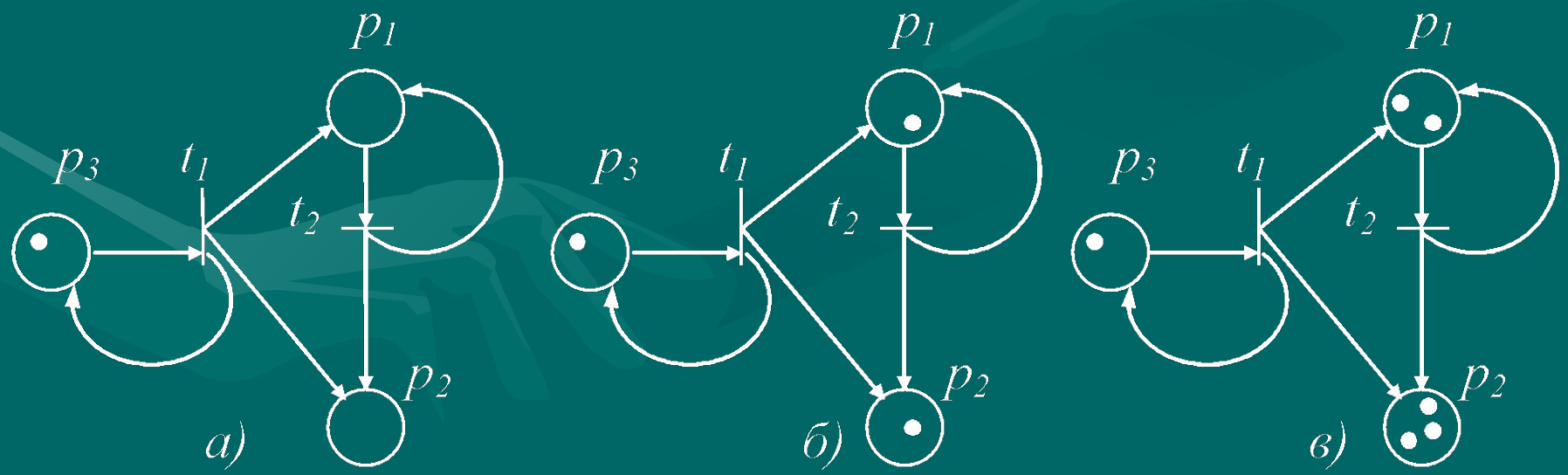
На рисунке приведен пример сети в графическом представлении. Переходы обозначены черточками, а места - окружностями. Каждый переход t имеет набор входных $in\{t\}$ и набор выходных $out\{t\}$ дуг.



2. Разметка сети

- Сеть можно понимать (интерпретировать) по-разному. Можно представить себе, что места представляют условия (буфер пуст, значение переменной вычислено и записано в буфер и т.п.), а переходы - события (посылка сообщения в буфер, считывание значения переменной из буфера и т.п).
- Состояние сети в каждый текущий момент определяется системой условий. Для того, чтобы стало возможным и удобным задавать условия типа “в буфере находится три значения” в определение сети Петри добавляются *фишки (размеченные сети)*. Фишки изображаются точками внутри места. В применении к программированию можно представлять себе переходы как процедуры, а места - как переменные, которые в качестве значения хранят признак того, назначено ли значение некоторой переменной программы или нет, и сколько значений назначено.

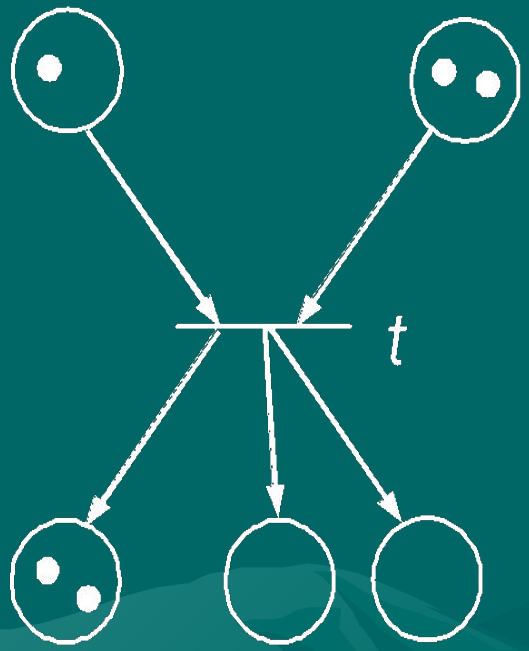
Фишка свидетельствует о том, что в переменной/буфере имеется значение, а если место имеет, к примеру, 3 фишки, то это может интерпретироваться как наличие трех значений в буфере. Если место содержит фишку, то место *маркировано* и сеть называется *маркированной*. Начальное распределение фишек задает начальную маркировку *M0* сети. Маркировка сети определяет ее текущее состояние. Начальная маркировка сети представляется вектором *M0*={0,0,1}. M1= {1,0,1}. M2={2,3,1}.



3.Срабатывания перехода

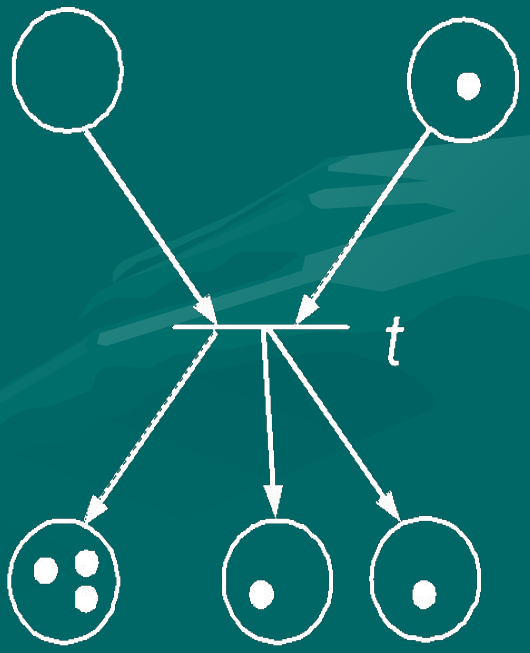
- Срабатывание перехода состоит из того, что из всех входных мест перехода забирается по одной фишке и во все выходные места добавляется по одной фишке. Если представить себе переход как процедуру, то она корректно выполняется и вырабатывает значения своих выходных переменных, если есть значения всех аргументов. Таким образом, переход может сработать, если хотя бы одна фишка находится в каждом из его входных мест. Сеть переходит из одного состояния в другое (от одной маркировки к другой) когда происходит событие - срабатывание перехода.
- В другой интерпретации переход может представлять некоторое устройство. Устройство может - но не обязано! - сработать, если выполнились все входные условия. Если одновременно несколько переходов готовы сработать, то срабатывает один из них (любой), или некоторые из них, или все

Начальное состояние сети



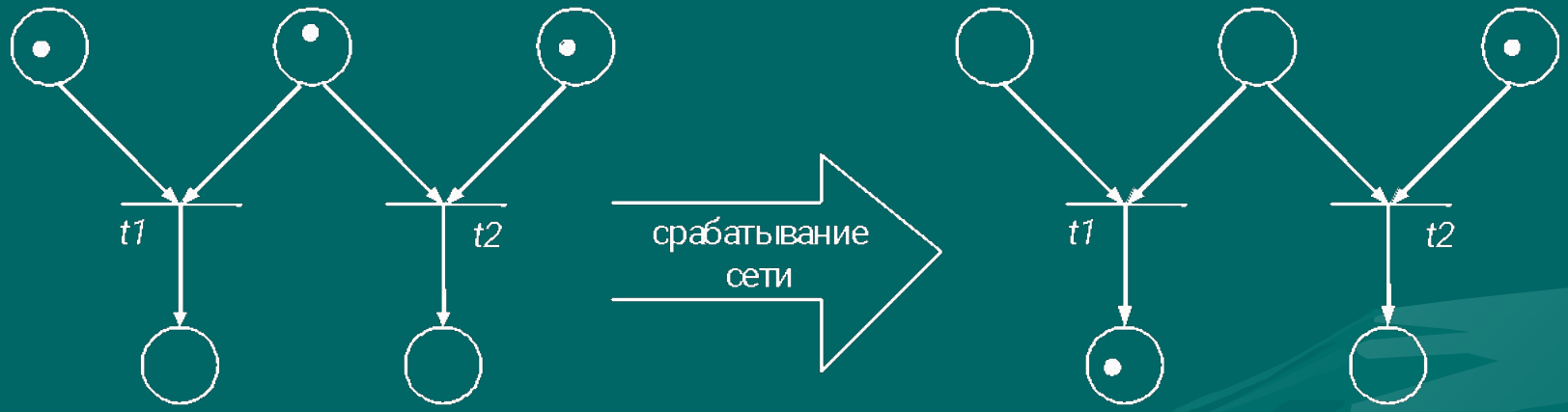
Состояние сети до срабатывания перехода

Состояние сети после срабатывания перехода *t*

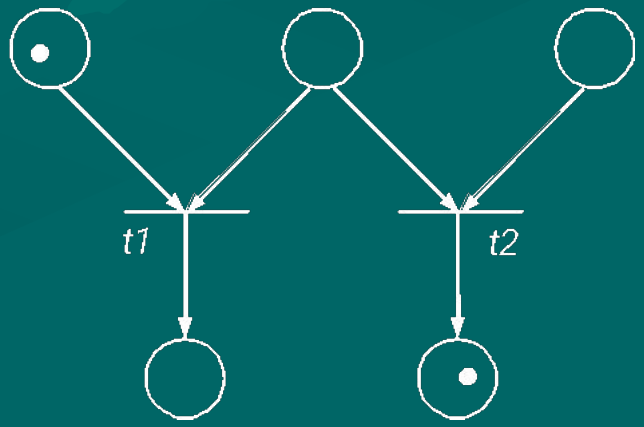


Состояние сети после срабатывания перехода

1-й вариант



2-й вариант

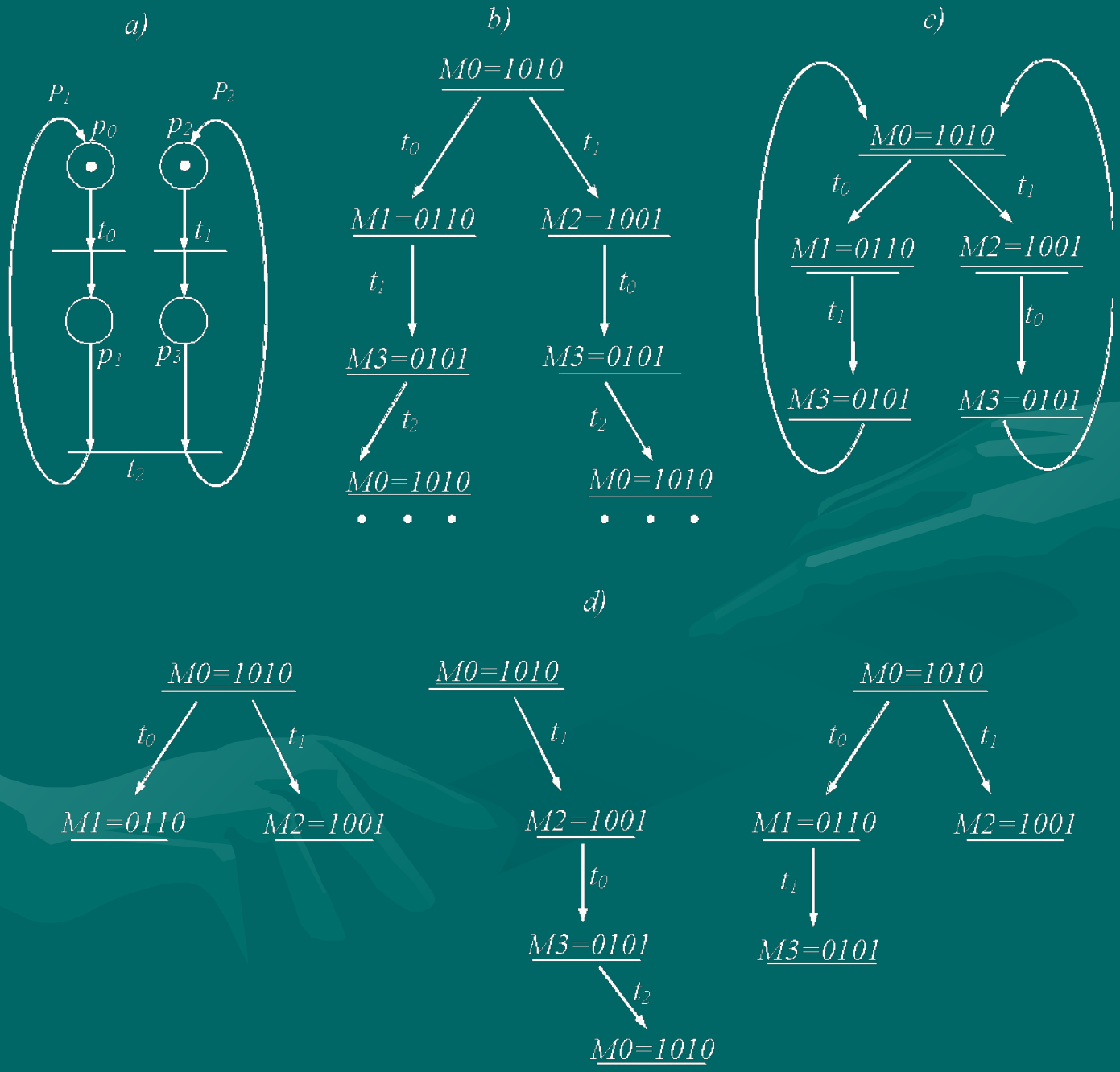


- Итак, сетью Петри называется набор G=(T,P,A,M), $T \cap P = \varnothing$, где
- T=$\{t_1, t_2, \ldots, t_n\}$-подмножество вершин, называющихся *переходы*,
- P=$\{p_1, p_2, \ldots, p_m\}$-подмножество вершин, называющихся *местами*,
- $A \subseteq (T \times P) \cup (P \times T)$ - множество ориентированных дуг.
- M – функция, $M: P \rightarrow I$, $I=\{0,1,2,\ldots\}$.
- Функционирование сети происходит от одного состояния к другому в результате срабатывания переходов.

Граф достижимости

- Для использования сетей Петри необходимо знать их свойства, такие, например, как безопасность, а для этого следует изучить множество всех возможных разметок сети с заданной начальной разметкой.
- Разметка *M* называется *достижимой*, если при некоторой конечной последовательности срабатываний переходов, начиная с начальной разметки *M0*, сеть переходит к разметке *M*.
- *Граф достижимости* определяет все достижимые разметки и последовательности срабатываний переходов, приводящих к ним. Его вершинами являются разметки, а дуга, помеченная символом перехода *t*, соединяет разметки *M1* и *M2* такие, что сеть переходит от разметки *M1* к разметке *M2* при срабатывании перехода *t*.
- Любой конечный фрагмент графа достижимости, начинающийся с начальной разметки и до некоторых достижимых разметок называется *разверткой* сети. Множество всех разверток определяет *поведение* сети Петри. Каждая разметка *M* определяет *состояние* сети. *Состояние* сети характеризуется множеством переходов, которые могут сработать в состоянии *M*.

- **П2**.Сеть на рисунке *а*) определяет управление двумя параллельно протекающими процессами с синхронизацией - оба процесса поставляют фишки, необходимые для срабатывания перехода t_2. Граф достижимости показан на рисунке *б*). Он бесконечен, однако после разметки *М3* в каждой ветви повторяется один и тот же фрагмент и потому возможно конечное представление графа достижимости (рисунок *в*). Множество разверток сети (ее поведение) бесконечно, примеры разверток приведены на рисунок *d*).

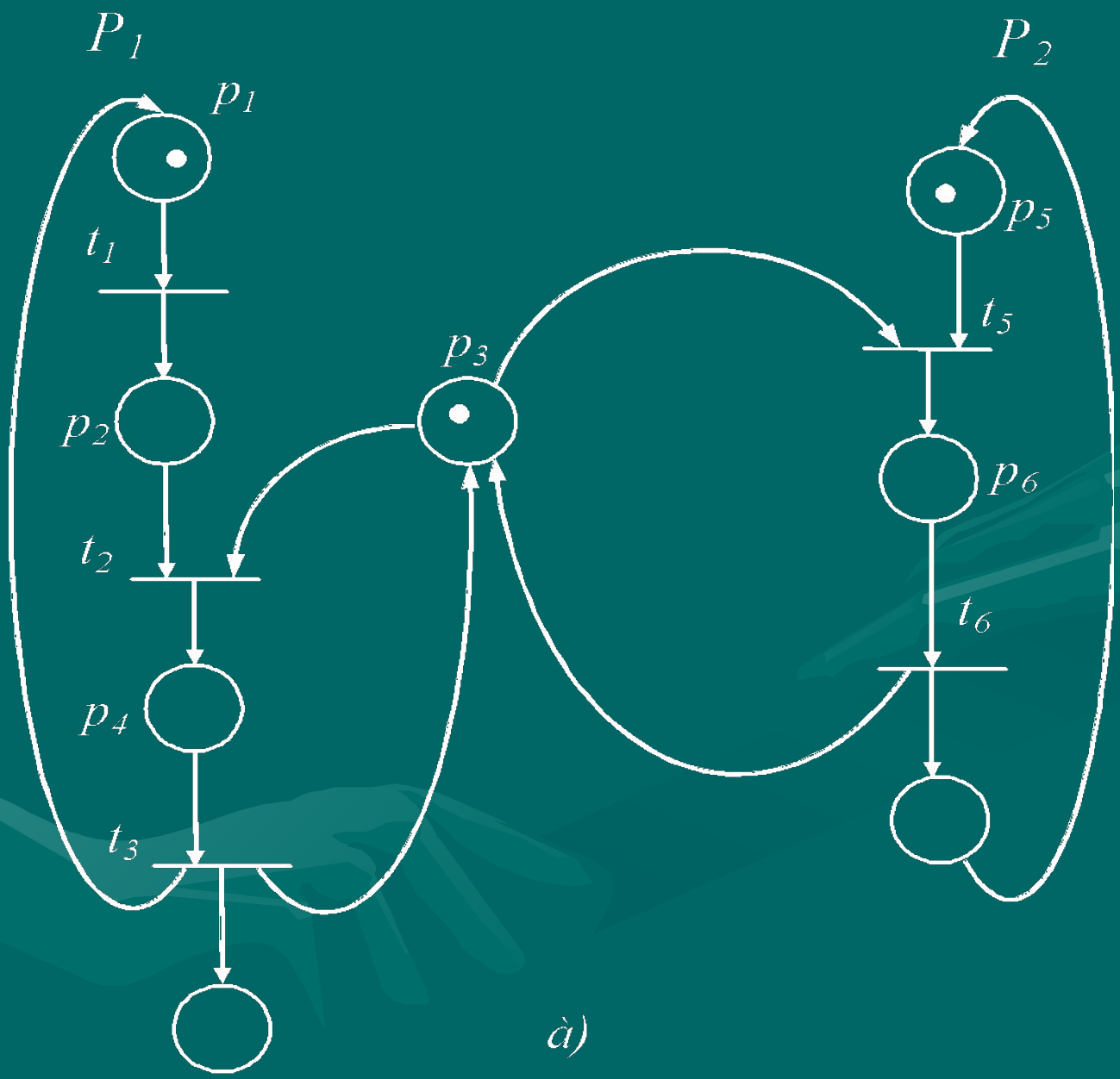

a)
P1
P2
p0
p2
t0
t1
p1
p3
t2
b)
M0=1010
t0
t1
M1=0110
M2=1001
t1
t0
M3=0101
M3=0101
t2
t2
M0=1010
M0=1010
c)
M0=1010
t0
t1
M1=0110
M2=1001
t1
t0
M3=0101
M3=0101
d)
M0=1010
t0
t1
M1=0110
M2=1001
M0=1010
t1
M2=1001
t0
M3=0101
t2
M0=1010
M0=1010
t0
t1
M1=0110
M2=1001
t1
M3=0101

Вообще, всякое конечное дискретное устройство, вырабатывающее бесконечный результат, обнаруживает некоторого рода регулярность в поведении, что и обеспечивает конечную его представимость. Это демонстрируют и примеры графов достижимости.

В теории формальных языков, например, этот факт выражается в *pumping* теоремах.

Задача взаимного исключения

- Теперь сетью Петри можно строго описать поведение процессов в задаче взаимного исключения. Такая сеть должна описывать поведение системы процессов с взаимным исключением доступа к неразделяемому ресурсу.
- Пусть заданы два процесса P_1 и P_2, конкурирующие за доступ к общему неразделяемому ресурсу . На следующем рисунке общий ресурс изображается местом p_3. Переходы обозначают какие-то действия с использованием ресурсов. Например, если процессу P_1 выделен затребованный блок памяти p_3, то процесс P_1 сможет выполнить свою процедуру t_2. Количество экземпляров ресурса p_3 (количество ресурса $p3_)$ обозначается фишками в месте p_3 - один экземпляр. Итак, два процесса (переходы t_2 и t_5) запрашивают единственный экземпляр ресурса p_3, но только одному процессу ресурс может быть выделен. Во множестве *поведение* сети нет такой развертки сети, которая бы привела к одновременному срабатыванию переходов t_2 и t_5.

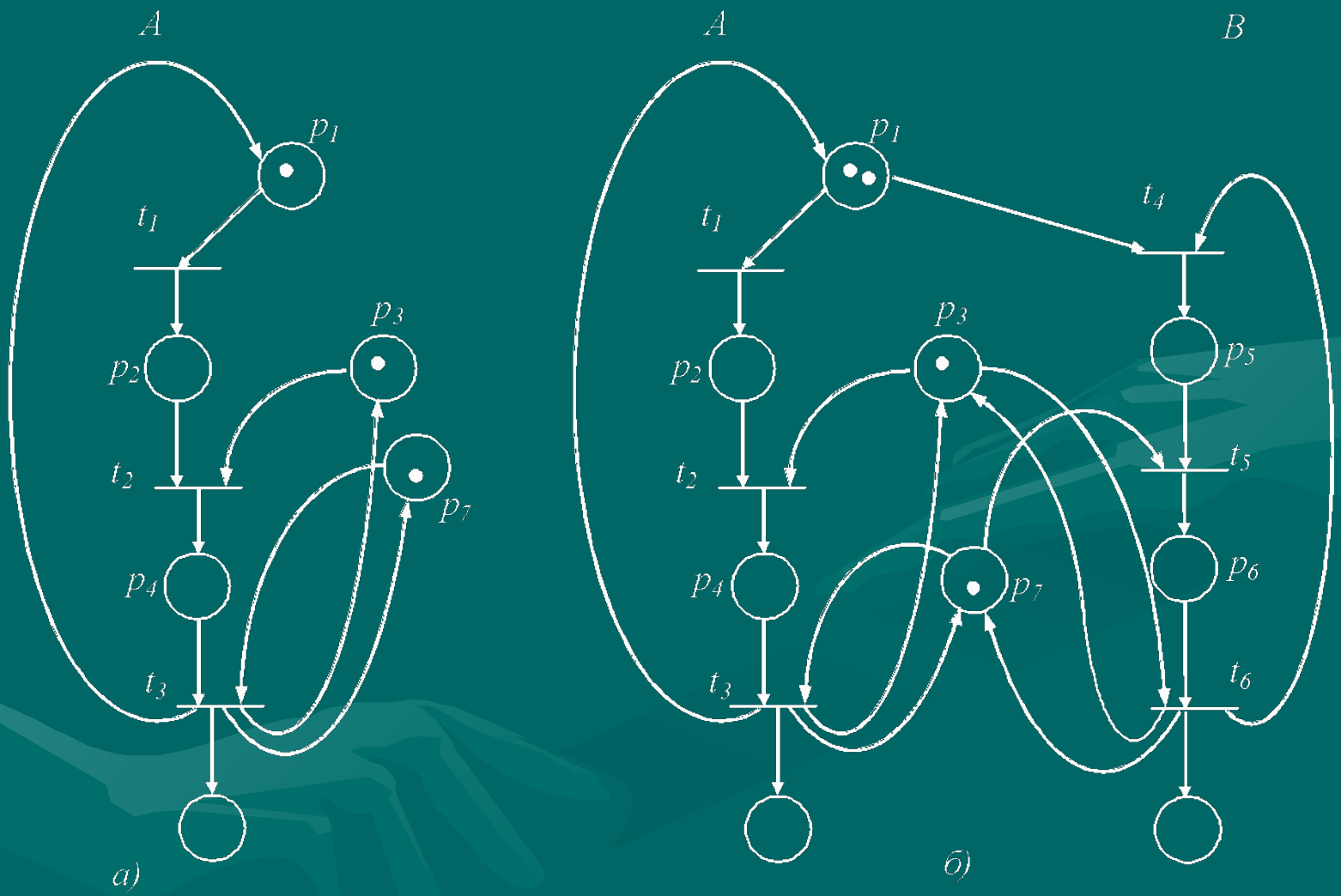


à)

- Сеть Петри не определяет, какой именно из двух конкурирующих процессов получит доступ к ресурсу, она лишь описывает ограничение на доступ к ресурсу - только один процесс (все равно какой) получает доступ к ресурсу p_3. Как следствие, при таком задании управления возможна ситуация, когда доступ к ресурсу будет постоянно получать один и тот же процесс, например P_1, а процесс P_2 останется “навечно” ожидать выделения ему ресурса p_3 (состояние *starvation*).
- При организации выполнения системы процессов эта ситуация разрешается с использованием дополнительных средств. Например, в операционных системах в описание состояния процессов вводится дополнительная характеристика - *приоритет*. Запрошенный ресурс выделяется процессу с наибольшим приоритетом. Начальный приоритет любого процесса растет с ростом времени ожидания ресурса. Таким образом, всякий процесс со временем получит запрошенный ресурс.
- Другой (наиболее распространенный) способ выделения ресурса - устройство очереди. Все запросы ресурса выстраиваются в очередь к ресурсу и процесс, раньше всех запросивший ресурс, получит его первым (дисциплина обслуживания FIFO - First_In - First_Out).

Дедлоки. Определение дедлока

- Если запрос ресурсов в системе не может быть удовлетворен, то система останавливается (ни один переход не может сработать). Это может быть нормальный останов сети, а может быть следствие конкуренции за ресурсы.
- На рисунке *а*) сеть *A* определяет циклическое неограниченное срабатывание переходов t_1, t_2, t_3 (процесс *A*). При срабатывании переходы t_2 и t_3 потребляют единицу ресурса из мест p_3 и p_7 каждый. Можно представить себе для определенности, что места p_3 и p_7 обозначают какие-то внешние устройства. Пока процесс, управляемый сетью *A*, выполняется один, ситуации дедлока не возникает. Но если появляется другой процесс (рисунок *б*), выполняющийся параллельно с *A* и управляемый сетью *B* (процесс *B*), тогда появляется конкуренция за ресурс в местах p_3 и p_7.


A
p_1
t_1
p_3
p_2
t_2
p_7
p_4
t_3
а)
A
B
p_1
t_4
t_1
p_3
p_5
p_2
t_5
t_2
p_6
p_4
p_7
t_6
t_3
б)

- Состояние дедлока возникает при следующей последовательности срабатываний переходов сети: t_1, t_4, t_2, t_5. Теперь имеем дедлок: все ресурсы потреблены, новый запрос не может быть удовлетворен и ни один переход не может сработать. Однако сеть будет нормально функционировать, если в месте p_1 оставить одну фишку, т.е. разрешить выполняться либо процессу *A* либо процессу *B*, но не обоим одновременно.
- Основную опасность при таком динамическом (в ходе вычислений) захвате ресурсов вызывает *дозахват* ресурса (запрос дополнительной порции того же или другого ресурса без освобождения уже захваченных ресурсов). Система процессов в состоянии дедлока “зависает” в ожидании и не может никогда завершиться. Следовательно, необходимо разработать стратегии для борьбы с дедлоками.
- ДЕДЛОК – состояние системы процессов, при котором ресурсов для их исполнения достаточно, но в силу ошибки в управлении возникают взаимные блокировки

Необходимые условия возникновения дедлока

- **1.Взаимное исключение.** Для правильного исполнения процессов A и B необходимо было организовать их взаимное исключение, так как ресурсы p_3 и p_7 являются неразделяемыми ресурсами. Если процесс A получил все экземпляры ресурса p_3, то другой процесс, затребовавший этот же ресурс должен быть задержан и ждать освобождения необходимого количества ресурса p_3.
- **2.Дозахват ресурса.** Каждый процесс получил часть ресурса p_3, держит его (не освобождает) и еще затребовал дополнительный ресурс p_7.

3.Не допускается временное освобождение ресурса.

Дедлок не возник бы, если бы была возможность:

1.задержать один из процессов (например *A*) до его завершения,

2.временно освободить принадлежащий *A* ресурс p_3 (*preemption*).

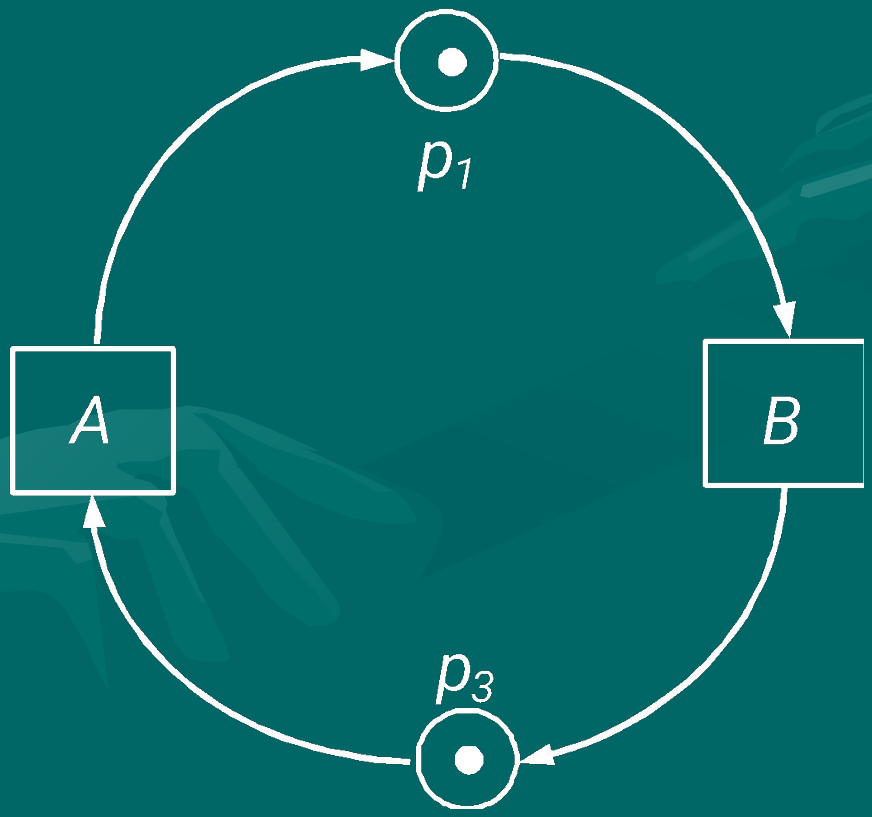
3.отдать ресурс p_3 для завершения процесса *B*, после завершения процесса *B* продолжить исполнение *A* уже со всеми необходимыми ресурсами.

Некоторые ресурсы допускают такое освобождение, но не все. Например, если p_3 и p_7 - участки памяти, то для завершения процесса B операционная система может эвакуировать процесс A во вторичную память (*swapping*), отдать освободившийся участок памяти p_7 процессу B и после его завершения передать освободившиеся ресурсы для завершения A.

Если p_3 - принтер, на который процесс A уже начал вывод информации, то освободить его на время и передать процессу B невозможно, если, конечно, не планируется специально напечатать смесь сообщений из обоих процессов.

4.Циклическое ожидание

- Дедлок на рисунке демонстрирует циклическое ожидание: процесс *A* получил ресурс p_3 и ожидает получения ресурса p_7.
- А процесс *B* имеет ресурс p_7 и ожидает получения ресурса p_3.



- Для анализа распределения ресурсов рассматривается *граф распределения ресурсов* (ГРР). ГРР - это двудольный граф, его вершины - имена ресурсов и процессов. Дуга ведет из вершины-ресурса *r* в вершину-процесс *p*, если ресурс *r* выделен процессу *p*. Дуга ведет из вершины-процесса *p* в вершину-ресурс *r* , если процесс *p* запросил ресурс *r*, но еще не получил его и находится в состоянии ожидания. Система процессов в состоянии дедлока характеризуется ГРР на рисунке выше. Как обычно, фишки показывают количество ресурсов. На графе явно видно циклическое ожидание процессов. Говорят, что процессы *A* и *B* *входят в цикл ожидания*. Понятно, что такой цикл могут образовать и более чем два процесса и ресурса.

Борьба с дедлоками

Возможно два подхода к борьбе с дедлоками.

1. Первый основан на таком аккуратном планировании ресурсов, при котором дедлок заведомо не может возникнуть - *предотвращение дедлоков.*
2. Второй подход допускает дедлоки, но есть алгоритмы обнаружения и выхода из состояния дедлока - *преодоление* дедлока. Реализация обоих подходов оказалась весьма дорогостоящей.

Предотвращение дедлоков. Условия 1-4 составляют необходимыми, но не достаточнными условиями возникновения дедлока. Все эти условия могут удовлетворяться, но дедлок не обязательно возникает. Однако если нарушено хотя бы одно из условий 1-4, то дедлок вообще не возникнет. На этом основано предотвращение дедлоков. Посмотрим, каким образом можно избежать удовлетворения условий 1-4.

Первое условие - взаимное исключение - нарушить не удастся, если распределяется неразделяемый ресурс. Тут ничего поделать нельзя.

- Второе условие - дозахват - нарушить можно. Одна возможность используется операционными системами: процесс требует все необходимые ресурсы изначально и начинает исполняться лишь тогда, когда получит все, что запросил. Часть ресурсов при этом может долго не использоваться процессом. Неиспользуемые, но захваченные, ресурсы, а также начальное ожидание процессов, пока для них будут выделены все необходимые ресурсы, и есть цена безопасности системы взаимодействующих процессов.

- Например, если мультикомпьютер состоит из 500 процессоров и параллельная программа запросила для своего исполнения 200 процессоров, то такая программа при хорошей загрузке мультикомпьютера может стоять в очереди сутками, ожидая, когда освободится разом 200 процессоров. Либо оператор должен остановить продвижение очереди программ, не распределять освободившиеся процессы ожидающим программам и копить 200 свободных процессоров – тоже может быть очень накладно.

- Другой способ заключается в том, что процессу разрешается запрашивать новый ресурс лишь тогда, когда процесс не имеет никаких ресурсов вовсе. Следовательно, процесс должен сначала отказаться от всех своих ресурсов и лишь тогда снова сможет запросить все нужные ресурсы.

- Временное освобождение ресурса (если это возможно) должно специально обеспечиваться ОС и оборудованием вычислителя так, как в современных ЭВМ обеспечивается временная приостановка выполнения программы и её эвакуация во вторичную память (поддерживается системой прерывания, страничной организацией памяти и т.п.) для освобождения оперативной памяти, понадобившейся для завершения другой, более приоритетной программы.

- Если неразделяемый ресурс не может быть временно освобожден, то в ходе выполнения процесса он может иногда заменяться на другой, но уже разделяемый ресурс (а потому не нужно обеспечивать взаимное исключение), а реально необходимый неразделяемый ресурс может быть позже запрошен на короткое время в наиболее удобный момент. Примером может служить техника *спулинга* (spooling), при которой, к примеру, принтер (его временное освобождение невозможно) заменяется дисковым файлом печати. Диск не назначается ни одному процессу в монопольное использование. В файле печати накапливается весь материал для печати и затем реальный принтер используется кратковременно в удобное для системы распределения ресурсов время (когда процесс освободил, к примеру, все свои ресурсы) для “залповой” печати.

- Другие подходы к предотвращению дедлоков требуют использования дополнительной информации. Полезнее и проще всего изначально знать максимальное количество ресурсов, которое может запросить процесс в ходе выполнения. Эта информация позволяет всегда из множества готовых процессов выбирать на исполнение такое подмножество процессов, которое не попадает в состояние дедлока. Идея такого планирования формулируется в виде *алгоритма банкира*, суть которого следующая.

Алгоритма банкира

- Пусть в вычислительной системе есть n типов ресурсов, каждого типа ресурсов - r_i экземпляров. Одновременно исполняются m процессов $P_1, P_2, \ldots, P_m$. Каждый процесс P_j, $j=1,\ldots,m$, максимально может затребовать k^j_1 экземпляров ресурса первого типа, k^j_2 - экземпляров ресурса второго типа, k^j_n - экземпляров ресурса n-го типа. Для каждого j-го ресурса и процессов $P_1, P_2, \ldots, P_m$ должно выполняться условие банкира:
- $\sum_i k^i_j \le r_j$
- Другими словами, суммарный максимальный запрос всех процессов $P_1, P_2, \ldots, P_m$ любого j-го ресурса не превосходит его наличного количества r_j. Следовательно, все запросы процессов могут быть удовлетворены. Каждому новому процессу разрешается начать исполнение лишь в том случае, если он не нарушит условие банкира.

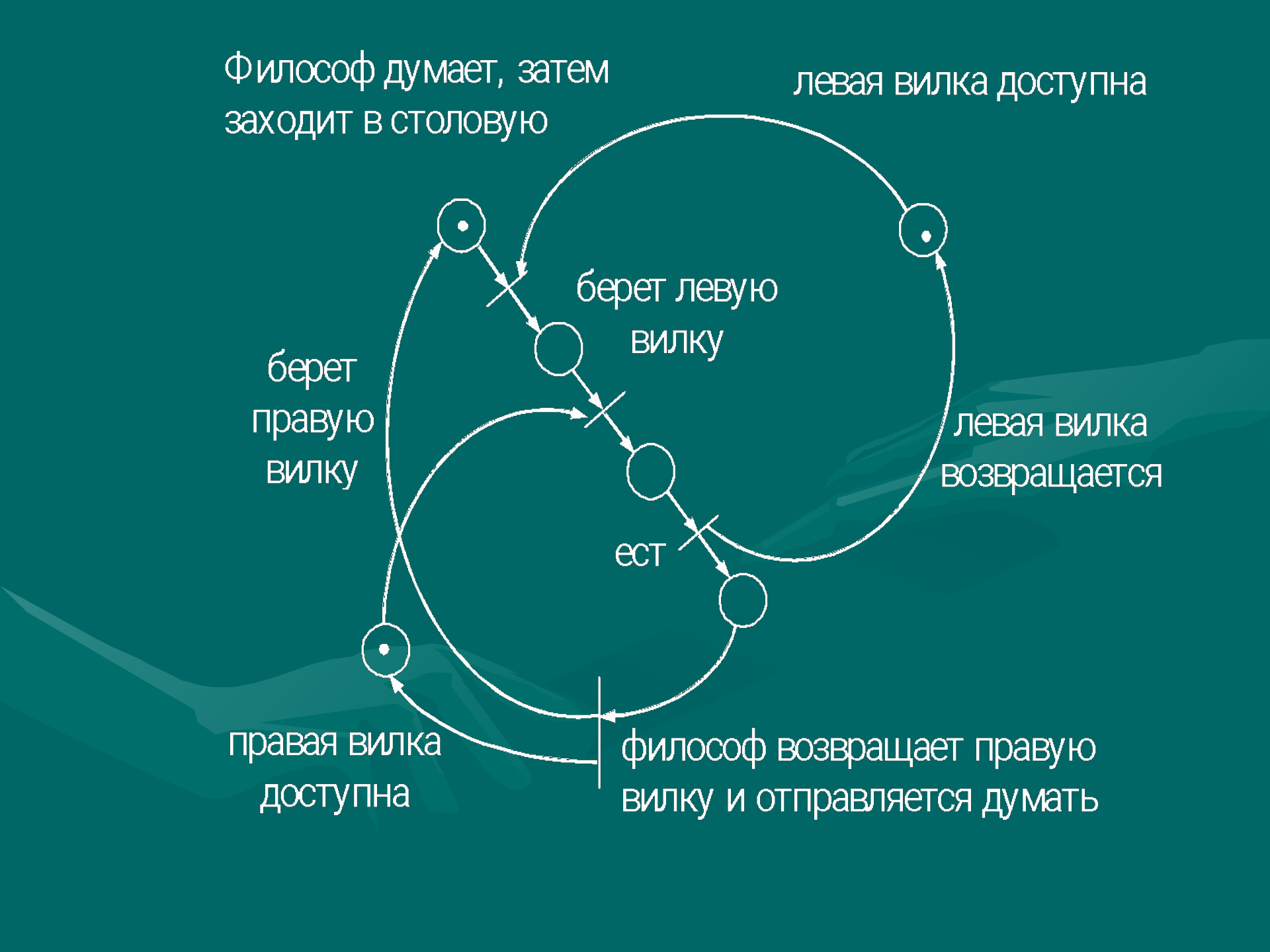
Преодоление дедлока

Если вычислительная система допускает состояние дедлока, то его необходимо уметь обнаружить и уметь выйти из этого состояния. Дедлок обнаруживается, если процессы находятся в циклическом ожидании. Такая проверка должна делаться систематически.

При обнаружении дедлока необходимо разорвать циклическое ожидание, которое находится из анализа ГРР. Для этого можно принести в жертву один из процессов, входящих в цикл ожидания, и “убить” его. Если циклическое ожидание еще не разорвано, тогда необходимо “убить” следующий процесс и так до тех пор, пока система процессов не выйдет из состояния дедлока. Конечно, выбрать очередную “жертву” невозможно без привлечения дополнительной информации о процессах и хорошего решения, как правило, найти не удается.

Задача о пяти обедающих философа

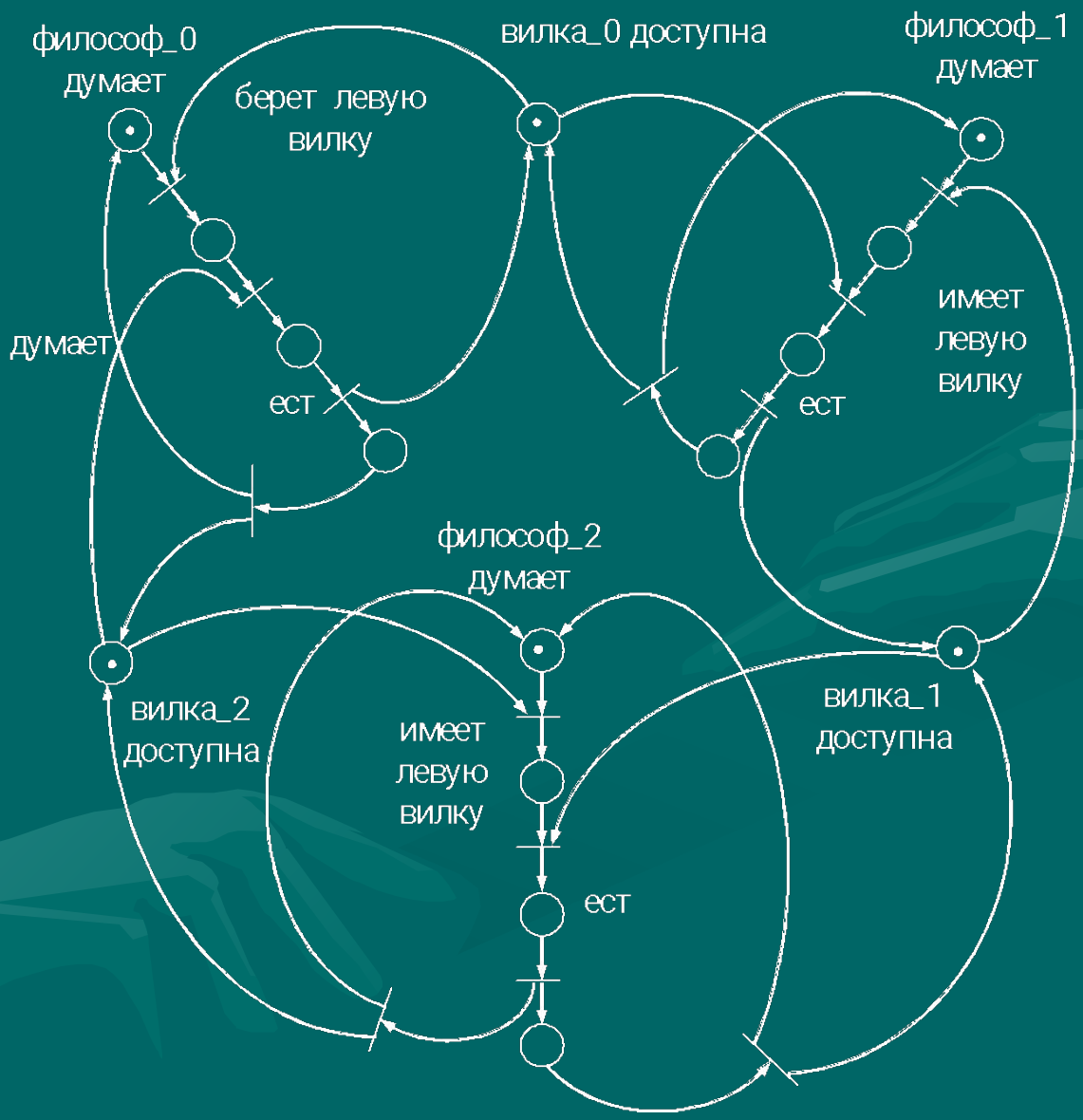
- Пять философов, прогуливаясь и размышляя, время от времени испытывают приступы голода. Тогда они заходят в столовую, где стоит круглый стол, на нем всегда приготовлены пять приборов. Между соседними блюдами лежит одна вилка (всего лежит ровно пять вилок). Голодный философ:
- а) входит в столовую, садится за стол и берет вилку слева,
- б) берет вилку справа,
- в) ест (обязательно двумя вилками),
- г) кладет обе вилки на стол, выходит из столовой и продолжает думать.
- При конструировании управления в этой задаче следует учитывать самые разнообразные варианты поведения философов.


Философ думает, затем заходит в столовую
левая вилка доступна
берет левую вилку
берет правую вилку
левая вилка возвращается
ест
правая вилка доступна
философ возвращает правую вилку и отправляется думать

- Рассмотрим некоторые из них.
- а).Необходимо организовать действия философов так, чтобы они **все** были накормлены и не случилось бы так, что пять философов одновременно войдут в столовую, возьмут левую вилку и застынут в ожидании освобождения правой вилки (дедлок!). Голодная смерть всех философов неминуема, если никто из них не пожелает расстаться на время со своей левой вилкой. Будет не лучше, если они одновременно положат левые вилки, а затем вновь одновременно попытаются завладеть необходимыми двумя вилками. Результат, понятно, тот же. Типичный дедлок в результате попытки дозахватить ресурс (вторую вилку)! Выход из него - “убить” один из процессов (выставить за дверь одного, все равно какого, голодного философа и никого более в столовую пока не пускать). Однако гарантировать, что все философы будут накормлены и ни один из них не попадет в состояние вечного ожидания в этой стратегии нельзя.

- Очевидно отсюда, что грубым (не эффективным) приемом, чтобы избежать этой ситуации, является введение ограничения на число философов, допущенных одновременно в столовую. Например, можно пускать в столовую не более четырех философов одновременно. Тогда, по крайней мере, один из них сможет захватить две вилки, поесть и освободить ресурсы (вилки) для других.
- Однако гарантировать, что все философы будут накормлены, нельзя

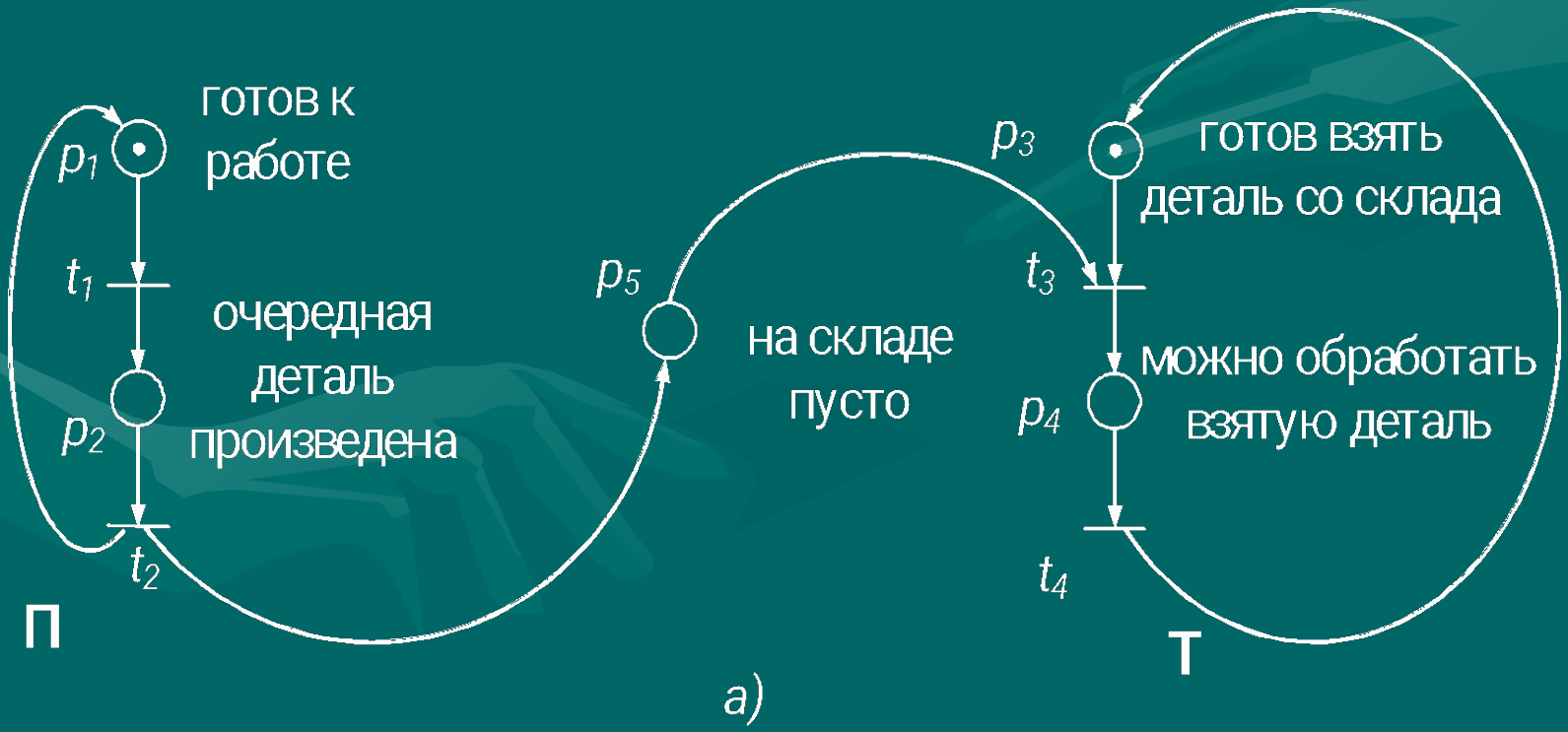
- б).Необходимо также предусмотреть, чтобы два философа одновременно не хватали одну и ту же вилку (обеспечить взаимное исключение). Зная вспыльчивый характер философов, нетрудно в этом случае предсказать результат.
- в).Стеснительный философ не должен умереть в столовой голодной смертью из-за того, что его вилки постоянно раньше него хватают более напористые соседи (не допустить состояния вечного ожидания).
- г).Легко представить себе ситуацию, когда банда сговорившихся философов завладеет всеми вилками и, передавая их только в своей среде, уморит голодом всех прочих.


философ_0
думает
берет левую
вилку
вилка_0 доступна
философ_1
думает
думает
ест
ест
имеет
левую
вилку
философ_2
думает
вилка_2
доступна
имеет
левую
вилку
ест
вилка_1
доступна

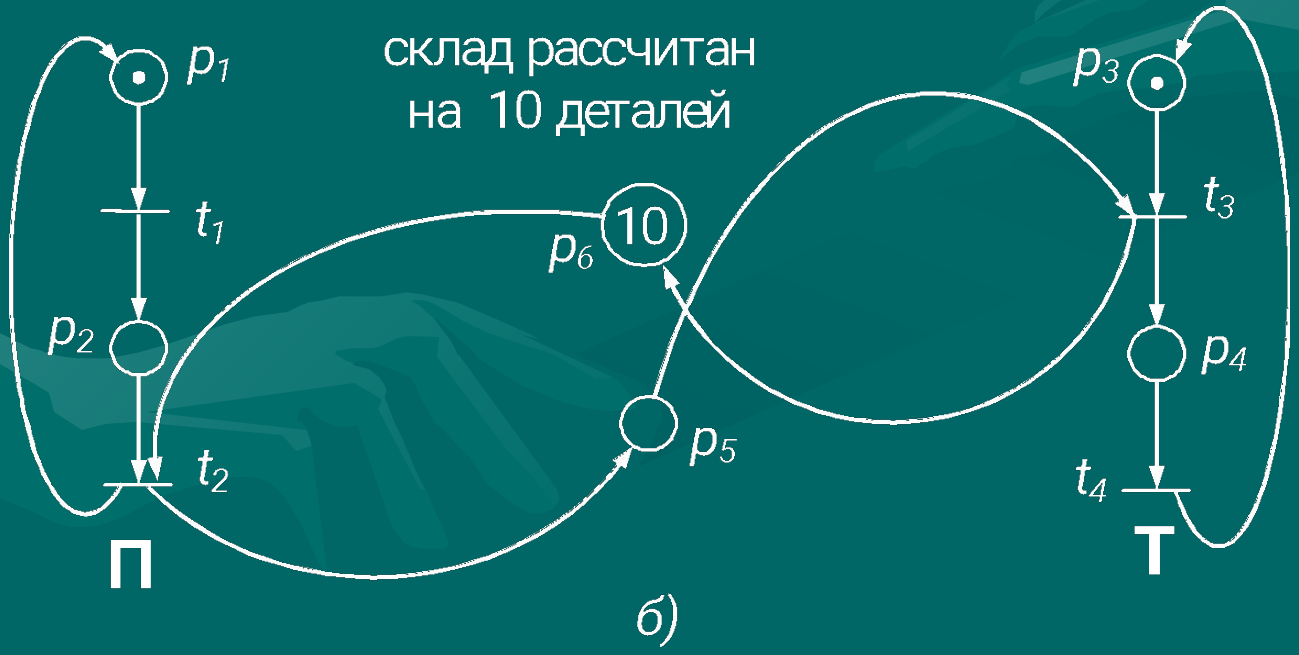
- Сеть Петри на рисунке задаёт управление распределением вилок трем обедающим философам, человеческий фактор в ней не учитывается. Эта сеть конструировалась пошагово.
- Вначале была сконструирована сеть, управляющая поведением одного (любого) философа. Затем три таких сети, соответствующие трем философам, объединяются в одну. Распределяемые ресурсы - левые и правые вилки соседних философов - совмещаются.
- В качестве упражнения можно попробовать сконструировать сеть Петри, которая бы управляла философами так, что ни один из них не умрет голодной смертью (всем будет обеспечен равный доступ в столовую). Необходимо принять во внимание, что сеть Петри – объект неинтерпретированный.

Задача производитель/потребитель

- производитель П производит детали и оставляет их на складе, а потребитель Т забирает их со склада, когда они там есть. Регулирует это взаимодействие склад (место p_5). Склад здесь описан неограниченно большой емкости. Процесс-производитель может сработать неограниченно большое число раз, даже если процесс потребитель не работает совсем.



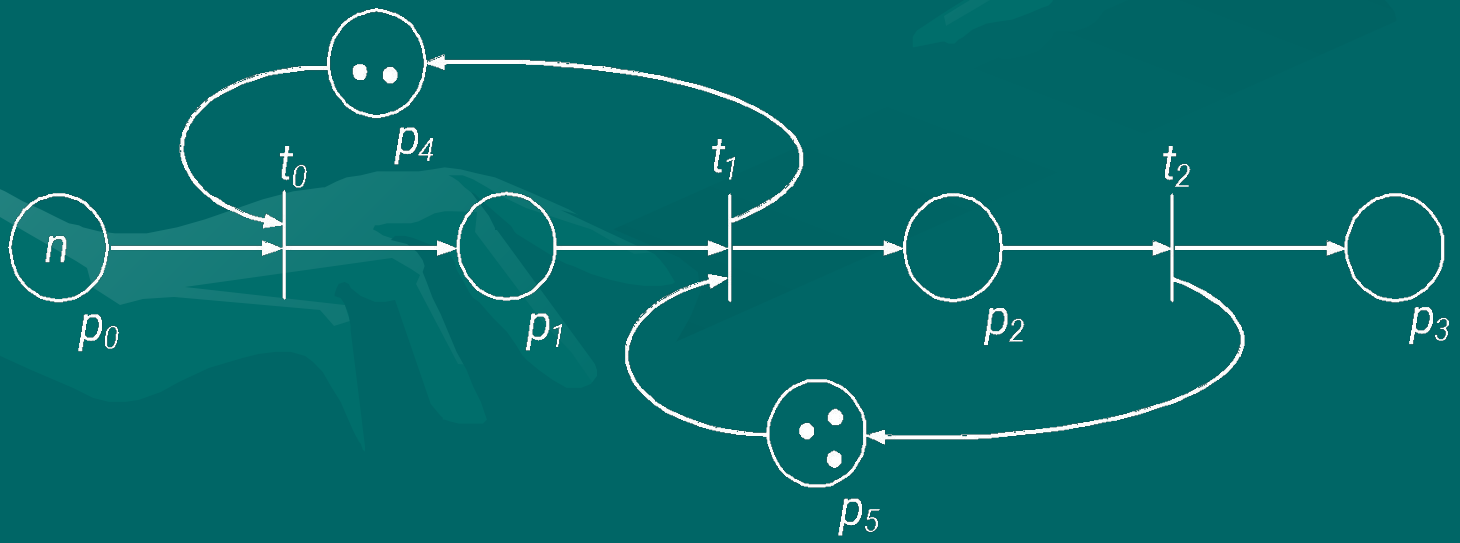
Если необходимо принять во внимание ограниченную вместимость склада, тогда в сеть добавляется место p_6. Оно определяет наличие мест для хранения не более чем 10 деталей. Взятие фишки из места p_6 может интерпретироваться как взятие разрешения поместить очередную деталь на склад (занять свободное место для хранения).



б)

- **П3**.Рассмотрим пример конвейера. Пусть есть три обрабатывающих устройства t_0, t_1, t_2, организованные в виде конвейера. Это могут быть, например, станки на заводе или функциональные устройства конвейерного процессора и вообще любой конвейер, в котором каждое обрабатывающее устройство выполняет лишь часть общей работы, а конечный результат будет выработан последним из них.

- Пусть требуется, чтобы места p_1 и p_2 могли содержать ограниченное число результатов: место p_1 может вместить лишь два результата (место p_1 сети *2-ограничено*) предшествующего этапа работы конвейера (вырабатывается переходом t_0), а место p_2 – 3-ограничено. Символ n в месте p_0 означает наличие n фишек в нем, n - целое положительной число.



- Сеть Петри, в которой все места 1-ограничены, называется *безопасной*. Такой сетью можно задавать прямое управления в программах. Безопасная сеть никогда не допустит, чтобы в переменную было положено новое значение, если старое еще не было использовано по назначению. Нарушения этого правила может быть причиной ошибок в параллельных программах.
- При использовании сетей Петри в языках программирования стандартные схемы управления (например, сеть управления конвейерным исполнением процедур) могут быть описаны как управляющие процедуры, например:
- **control procedure** *pipiline* (t_0, p_4, $t1$, p_5, t_2);
- <описание сети управления конвейером>;

- Теперь при необходимости задать в программе конвейерное вычисление некоторых процедур, их имена подставляются в обращение к управляющей процедуре *pipiline* вместо формальных параметров t_0, t_1, t_2.
- *call pipiline* (t_0= *proc1*, p_4 =2, t_1=*proc2*, p_5=3, t_2=*proc3*)
- Наличие библиотеки стандартных управляющих процедур способно значительно облегчить отладку взаимодействия процессов.