Санкт-Петербургский государственный университет

Ю. К. Демьянович, Т. О. Евдокимова

ТЕОРИЯ РАСПАРАЛЛЕЛИВАНИЯ И СИНХРОНИЗАЦИЯ

Курс лекций

Санкт-Петербург
2004

Введение

Высокопроизводительные вычисления в настоящее время не мыслятся без распараллеливания, ибо наиболее мощные вычислительные системы имеют сотни и тысячи параллельных процессоров. Достаточно обратиться к регулярно обновляемому в Internet списку TOP500, содержащему перечень наиболее мощных компьютеров в мире, чтобы убедиться в том, что все компьютеры в этом списке — параллельные системы. Благодаря распараллеливанию удается достичь производительности в десятки терафлопс (10^{12} операций в секунду с плавающей точкой), но для наиболее сложных современных задач и этого недостаточно: требуются вычислительные мощности с быстродействием в десятки пентафлопс. Достижение таких скоростей трудно представить без распараллеливания.

Распараллеливание алгоритмов и написание параллельных программ — весьма сложное дело. Причины возникающих трудностей различны: с одной стороны, идеология распараллеливания трудно воспринимается после приобретения навыков последовательного программирования, а с другой стороны, методы распараллеливания задач недостаточно разработаны. Эффективность распараллеливания зачастую оказывается значительно ниже ожидаемой: в частности, производительноть параллельной системы на реальной задаче, как правило, в несколько раз ниже пиковой производительности. Ввиду высокой стоимости параллельных вычислительных систем их в первую очередь используют при решении тех задач, решение которых невозможно или затруднительно на однопроцессорных системах. К таким задачам относятся задачи прогнозирования климата на сотни лет вперед, задачи прогноза погоды на несколько дней или месяцев (и, в особенности, катастрофических изменений погоды: возникновения ураганов, тайфунов, резких из-

менений температуры и т.п.), сюда же относятся задачи геологии и геофизики (предсказания землетрясений, вулканических извержений и т.д.), задачи астрономии и астрофизики (предсказания поведения Солнца, столкновений метеоритов и болидов с Землей, обоснование космогонических гипотез), задачи, связанные с биологией и с чрезвычайно значимой для человека областью — с медициной (последнее достижение – расшифровка генома человека – один из ярчайших примеров применения высокопроизводительных вычислительных систем). Конечно, имеется много других примеров сложных задач, решение которых без высокопроиводительных параллельных вычислительных систем невозможно.

Решение многих задач, возникающих в перечисленных областях, можно проводить прямым численным моделированием соответствующих физических или физико-химических явлений, которые по-существу представляют собой огромное количество процессов, взаимодействующих друг с другом. Наиболее естественный путь моделирования — разбиение на группы подобных между собой процессов, выделение представителя каждой группы и параллельное моделирование этих представителей во взаимодействии их друг с другом. Без связи с распараллеливанием укрупнение процессов в том или ином смысле рассматривалось ранее: метод частиц в ячейках, метод конечных элементов, метод сеток и другие методы, фактически, приводят к похожему результату. Конечно, распараллеливание несет в себе большие трудности, но и вселяет значительные надежды. Исследования в этой области, практическое освоение теоретических результатов, создание соответствующего программного обеспечения, отладка программ на параллельных вычислительных системах (ВС) и проведение эффективных вычислений представляются чрезвычайно важными.

Численное моделирование параллельных процессов существенно усложняет программирование: скорость обработки информации значительно увеличивается с увеличением независимости процессов, но из-за этого приходится ослаблять контроль за этой обработкой. При обработке совокупности арифметических действий ускорение вычислений происходит за счет автоматизированного использования ассоциативности и коммутативности сложения и умножения, а также дистрибутивности (умножения относительно сложения), что отсутствует в машинных аналогах упомянутых действий.

Процессы находятся в постоянной конкурентной борьбе за ресурсы, но поскольку в совокупности они решают одну задачу, то необходимы моменты синхронизации и обмена полученной информацией. Для этих целей служат барьеры и критические секции, о которых следует позаботиться программисту. При написании параллельных программ используются специальные средства, которые могут предоставляться в виде специальных библиотек или расширений известных языков (например, библиотеки MPI, Open MP, PVM, язык Linda для Fortran, C, C^{++}), однако, важны не сами эти средства, а принципы их использования.

Предлагаемый курс лекций посвящен параллельному программированию на (ВС) с разделяемой памятью. Он содержит три главы, первая из которых посвящена операторам распараллеливания, вопросам неделимости операций, устранению взаимного вмешательства процессов, стратегиям планирования и критическим переменным. Во второй главе рассматривается задача о критической секции, активные блокировки, алгоритм разрыва узла, построение барьеров. В третьей главе излагаются вопросы синхронизации с помощью семафоров, рассматриваются решения задач "об обедающих философах", "о читателях и писателях", методы "передачи эстафеты" и "кратчайшее задание".

Данный курс лекций читается для студентов математико-механического факультета на отделении информатики. Он основан на известных идеях, принципах и моделях параллельного программирования; в нем широко использованы книги [1-6] и, в особенности, книга [7], из которой почерпнуты стиль изложения и многочисленные примеры. Ввиду ограниченности курса (курс рассчитан на 36 лекционных часов) основное внимание обращено на принципы параллельного программирования. Для более углубленного изучения предмета любознательный читатель может воспользоваться упомянутыми выше книгами, т.к. содержание последних лишь частично отражено в предлагаемом курсе лекций.

Авторы курса надеются, что читатели быстро усвоят излагаемые первоначальные сведения по методам параллельного программирования и смогут легко перейти к чтению более солидных книг, перечисленных в списке рекомендуемой литературы.

Данная работа частично поддержана грантами РФФИ 04-01-00692, 04-01-00026 и НШ-2268.2003.1.

Глава 1 Вычислительные системы, алгоритмы и программы

§1 Вычислительная система

Вычислительная система (ВС) характеризуется своей *архитектурой.* Поскольку в процессе работы архитектура системы может меняться (ввиду отказа оборудования, отключения или подключения новых устройств), то архитектуру можно рассматривать в зависимости от времени: с течением времени арихитектура меняется, таким образом, архитектура является *функцией времени* (время удобно считать дискретным). Фактически, архитектура является *абстрактной функцией*: ее значениями служат не числа, а возможные архитектурные решения вычислительной системы.

Для конкретной ВС область значений архитектуры содержит конечное множество элементов, ибо конечно число деталей, из которых состоит конструкция, конечно число аппаратных средств, которые могут подключаться или отключаться в процессе работы.

В процессе работы ВС существенно состояние ее памяти; последнее меняется с течением времени и в каждый момент времени принимает определенное значение, представляющее собой двоичный код с большим (можно даже сказать, огромным) числом разрядов. Таким образом, в каждый фиксированный момент времени вычислительная система находится в *определенном состоянии,* под которым подразумевается совокупность значений ее архитектуры и памяти.

В следующий момент времени ВС переходит в *новое состояние,* ибо появляется (возможно новая) совокупность значений архитектуры и памяти.

Конструкция ВС обычно предусматривает *постоянные* и *переменные области;* это деление достаточно условно:

— к постоянным областям конструкции можно отнести те области, без которых нормальная работа ВС невозможна (например, для однопроцессорных систем невозможна работа без центрального процессора);

— к переменной области конструкции относят *подключаемые* или *отключаемые* устройства (Plug and Play, принтеры и т.п.).

Память также (условно) подразделяется на *постоянную* и *переменную* области, например

— постоянное запоминающее устройство (ПЗУ), обозначаемое также ROM (Read-Only Memory) – оно часто бывает полупроводниковым или оптическим – *постоянная область*;

— оперативное запоминающее устройство (ОЗУ), обозначаемое также RAM (Random Access Memory), – запоминающее устройство с произвольной выборкой – *переременная область*.

Переход ВС в очередное состояние определяется

1) предыдущим состоянием,

2) изменением переменной области конструкции (например, подключением устройств типа P & P),

3) изменением переменной области памяти (например, внесением новых данных с терминала).

Нетрудно видеть, что в ряде случаев (например, при работе в мультипрограммном режиме) рассматриваемую ВС можно разбить условно на ряд динамически меняющихся подсистем (ПВС), каждая из которых обслуживает конкретную программу. При этом можно предполагать, что между работающими программами обменов не происходит; в этом случае каждую подсистему можно рассматривать как отдельную ВС. Тут же можно рассмотреть и процессы, относящиеся к одной программе, или ряд взаимодействующих программ.

Во всех этих случаях можно считать, что как и исходная ВС рассматриваемые ПВС также имеют динамически меняющуюся архитектуру и динамически меняющуюся память, причем первая принимает значения из множества различных конструкций, а вторая — из множества числовых значений (т.е из множества двоичных слов большой длины); каждая из последних делится на переменную и постоянную области.

Совокупность состояний ВС в последовательные моменты времени представляет *историю ВС* в течение рассматриваемого промежутка времени.

§2 Алгоритмы и псевдоалгоритмы

Понятие алгоритма, используемого в теории алгоритмов, существенно отличается от понятия *"алгоритм",* которое будем использовать в дальнейшем.

Математическая энциклопедия (см. [7], с. 202) так определяет, что такое алгоритм.

"*Алгоритм* — точное предписание, которое задает вычислительный процесс (называемый в этом случае алгоритмическим), начинающийся с произвольного исходного данного (из некоторой предписанной совокупности данных) и направленный на получение полностью определяемого этим исходным данным результата".

На этом статья об алгоритме не кончается, каждый может ознакомиться с ее содержанием самостоятельно. Из дальнейшего текста этой статьи ясно, что слова "вычислительный процесс" *не означают* обязательно работу с числами — это может быть работа с *любыми* четко определенными объектами по *четко определенным правилам.*

На стр. 206 этой же энциклопеции дается понятие "Алгоритм в алфавите А": это — "точное общепринятое предписание, определяющее *потенциально осуществимый процесс* последовательного преобразования слов в алфавите А, процесс, допускающий любое слово в алфавите А в качестве исходного".

Очевидно, понятие "алгоритм в алфавите А" — более узкое понятие, чем понятие алгоритма, упомянутое ранее (т.е. это — частный случай более общего понятия алгоритма).

Используемое у нас понимание алгоритма отличается от того и от другого: оно содержит много элементов произвола (причем указанные элементы произвола существенны и не могут быть устранены, как это видно ниже). Ввиду этого можно было бы ввести понятие "схема алгоритма" или "псевдоалгоритма".

Будем называть *псевдоалгоритмом* (с указанной областью определения) однозначно понимаемую последовательность указаний о необходимых действиях, где сами действия первоначально

не имеют четкого определения, и их конкретизация производится последовательно в дальнейшем (и это — принципиальный момент).

Пример. Рассмотрим задачу об отыскании квадратного корня из числа $a > 0$. Можно действовать по правилу

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right), \quad x_0 > 0,$$

причем условие остановки итерационного процесса имеет вид

$$|x_{n+1} - x_n| < \varepsilon;$$

здесь a, x_0, ε — заданные числа.

Является ли это описание описанием алгоритма? На первый взгляд — да. Его можно реализовать в виде программы вычислений на компьютере.

Однако, не трудно видеть, что x_0 произвольным положительным числом выбирать нельзя (например, нельзя брать иррациональные числа, ибо нет возможности вести вычисления с бесконечным количеством знаков); при реальных вычислениях фактически x_0 окажется числом из множества чисел, представимых с помощью разрядной сетки данной ВС. Далее, как известно, арифметические действия в системе с плавающей точкой точно реализовать нельзя: они реализуются с ошибками округления "машинной арифметикой". Эти и другие факторы определяются характеристиками ВС, которые в исходном правиле вычислений не описаны. Для однозначности получаемых результатов следует уточнить *разрядность* ВС, *правила арифметических действий* (т.е. правила округления), *правило выбора* числа a (оно не должно быть слишком близким к нулю, ибо в некоторых ситуациях оно может восприниматься как отрицательное), *правило выбора* начального приближения x_0 (ибо при практических вычислениях сходимость может оказаться слишком медленная, и процесс на данной ВС не сможет завершиться за отведенное время).

Итак, в данном случае указанные действия не имеют четкого конструктивного определения и, значит, мы имеем здесь пример псевдоалгоритма. В случае изучения вопросов *распараллеливания* подобная *неопределенность* значительно возрастает. С этим приходится мириться, но при гигантской сложности современных задач

эта неопределенность требует повышенной *устойчивости параллельных* версий псевдоалгоритмов.

На *определенном этапе* исследований может появиться возможность некоторые (или все) упомянутые действия *заменить,* в свою очередь, некоторыми *псевдоалгоритмами.* Полученный в результате псевдоалгоритм называем *уточнением предыдущего* псевдоалгоритма.

Подобные уточнения могут происходить и дальше: в результате получается последовательность *уточняющих псевдоалгоритмов*; конечным результатом таких уточнений является алгоритм, согласно которому будут проведены интересующие нас вычисления. Трудность состоит в том, что первоначально этот алгоритм неизвестен и зависит от ряда обстоятельств, в том числе и от свойств используемой аппаратуры.

Вообще говоря, исходный псевдоалгоритм может уточняться по-разному: в результате можно рассмотреть *дерево* уточняющих псевдоалгоритмов, листьями которого служат (вообще говоря, различные) алгоритмы вычислений для исходной задачи. Заметим, что реализация вычислений может сопровождаться определением пути — последовательности уточняющих псевдоалгоритмов в этом дереве.

§3 Программа и ее история

Программа представляет собой определенный *этап уточнения* исходного псевдоалгоритма.

В различных условиях программой называют

1) *текст* на том или ином *языке высокого уровня;*

2) *текст* на *ассемблере* (который, в частности, может появиться после трансляции упомянутого текста с языка высокого уровня);

3) *машинный код* (который появляется при трансляции с автокода на машинный язык), называемый exe-файлом.

Могут быть и другие употребления слова “программа”.

Если в первом из упомянутых случаев *программа,* вообще говоря, иногда может выполняться на различных ВС (в том числе — на различных платформах), во втором и в третьем случаях, *как правило,* выполнение программы жестко связано с выбранной ВС.

Отметим также зависимость результатов от программного окружения (от используемых трансляторов, библиотек и т.п.)

Однако, во всех этих случаях можно считать, что в фиксированный момент времени программа характеризуется *значениями своих переменных* (в том числе — и неявных переменных — различных счетчиков, семафоров и т.п.).

Совокупность упомянутых значений в определенный момент времени называется *состоянием программы* в этот момент.

Поскольку совокупность этих значений *весьма велика,* обычно они скрыты и задают состояние программы неявно.

Программа начинает работу с некоторого *исходного* состояния, возникающего в момент обращения к программе, а ее *выполнение* состоит в *последовательной смене* состояний.

Эти понятия, конечно, относятся и к параллельной программе.

Программа запускает ряд *процессов,* которые имеют *значительную самостоятельность,* так что в определенной мере они выполняются независимо. Каждый процесс *проверяет* и *изменяет состояние программы.*

Процесс можно представить как последовательное *выполнение операторов. Операторы* можно рассматривать как *последовательность неделимых* действий. Неделимое действие характеризуется тем, что оно не может совершиться частично: оно может совершиться или не совершиться. Каждое из последних проверяет и изменяет состояние программы *неделимым образом:* это означает, что программа может находиться лишь в тех состояниях, при которых любое из упомянутых действий либо закончилось, либо не начиналось.

Примером таких действий могут служить *запись* и *чтение* слов в память, элементарные *логико-арифметические* действия.

Выполнение программы сводится к исполнению *групп неделимых* действий. *История выполнения программы* складывается из последовательного выполнения таких групп; нетрудно видеть, что *историю* можно рассматривать и как последовательность *состояний программы.*

Схема выполнения программы — один из допустимых вариантов ее выполнения на ВС; в ней должны учитываться моменты *синхронизации.*

Рис. 1. Схема выполнения программы (как последовательности групп неделимых действий).

В такой схеме можно считать, что группа неделимых действий выполняется одновременно. Аналогом схемы выполнения является временна́я развертка алгоритма; таким образом, имеется много схем выполнения программы.

Схема выполнения программы накладывается на вычислительную систему; при этом можно *маневрировать составом групп* неделимых действий с учетом необходимых синхронизаций.

В результате определяется отображение операций на модули вычислительной системы и моменты их включения.

Почему можно *маневрировать* составом групп неделимых действий?

Каждый процесс требует *определенной последовательности* неделимых действий, но *разнесенность* во времени этих действий *не влияет на* работу процесса, если такая разнесенность *согласована* с точками синхронизации, в которых происходит взаимодействие процессов. Поэтому состав групп неделимых действий в принципе

может меняться в весьма широких пределах (в другой терминологии и, может быть, с несколько иной точки зрения, объединения в группы неделимых действий соответствует объединению операций в ярусы при распараллеливании "алгоритма"). Благодаря этому, существует огромное количество вариантов реализации параллельной программы (каждый вариант реализации соответствует временно́й развертке алгоритма).

Каждый из таких вариантов определяет *историю выполнения* параллельной программы. Историй выполнения огромное количество. Все истории делятся на *желательные* и *нежелательные*. Условно говоря, желательные истории — это истории, приводящие к правильному результату в рассматриваемом классе данных.

Для отделения желательных историй от нежелательных служит *механизм синхронизации.* Синхронизация бывает условная и безусловная. Обе формы синхронизации могут *приостанавливать* те или иные процессы и *ограничивать* набор последующих неделимых действий.

Основное средство синхронизации — последовательность *критический секций.* Эти секции (которые появляются в процессе реализации и отражены в программе) должны быть неделимыми и их нельзя прервать действиями других процессов.

Имеется два фундаментальных свойства программы:

— *безопасность;*

— *живучесть.*

Эти свойства в той или мной мере должны сопровождать любую *правильную программу,* т.е. программу, которая имеет только желательные истории при *любом допустимом* наборе данных.

Свойство безопасности состоит в том, что программа никогда не попадает в плохое состояние, т.е. в состояние, при котором некоторые переменные имеют нежелательные (неправильные) значения.

Свойство живучести состоит в том, что программа в конце концов всегда попадает в хорошее состояние, т.е. в состояние, при котором все переменные имеют желаемые (правильные) значения.

К безопасности относится *частичная корректность* программы. Программа называется *частично корректной*, если она

— либо завершается и при завершении всегда выдает желаемый (правильный) результат;

— либо не завершается.

К свойству живучести можно отнести *завершаемость* программы при любом допустимом наборе данных; длина каждой ее истории конечна.

Имеется три подхода к исследованию свойств программы.

I. Первый подход сотоит в *отладке* и в *тестировании* программы. Таким образом удается перебрать некоторые истории программы и проверить степень их приемлемости.

Недостатком такого подхода является *ограниченность* числа историй, которые удается перебрать: их перебор может способствовать лишь возникновению уверенности в правильности программы, но строгим доказательством такой прием, конечно, не является.

II. Второй подход — *использование операторных рассуждений*, ведущий к исчерпывающему анализу случаев. Для этого анализируются способы чередования неделимых действий.

Недостаток этого подхода в том, что количество возможных чередований весьма велико: для программы из n процессов, каждый из которых содержит k неделимых действий, число возможных чередований равно $\dfrac{(kn)!}{(k!)^n}$. В числителе стоит количество всех возможных перестановок из имеющихся kn неделимых действий, но поскольку для данного процесса возможен только один порядок выполнения, то следует разделить на $k!$ для каждого из n процессов, поэтому в знаменателе появляется $(k!)^n$ В частности, уже при $k = 2$, $n = 3$ получаем $\dfrac{(2 \cdot 3)!}{(2!)^3} = 90$ возможных вариантов.

III. Третий подход — *применение абстрактного анализа* (доказательных утверждений), что сродни обычному логическому доказательству теоремы. В этом случае используются формулы логики, причем неделимые действия рассматриваются как *предикатные преобразователи*; они меняют состояние программы, удовлетворяющее одному предикату, на состояние, удовлетворяющее другому.

Преимущества данного подхода:

— компактное представление состояний и их преобразований;

— работа при построении и анализе программы пропорциональна числу неделимых действий в этой программе.

Замечание. Параллельные программы очень сложны при тестировании и отладке из-за того, что

1) трудно сразу остановить все процессы и проверить их состояния;

2) каждое выполнение программы приводит к новой истории.

§4 Замечания о способе представления программ в данном курсе

Для пояснения логики параллельного программирования и представления программных конструкций необходим некоторый общепонятный способ изображения программ. В качестве такого способа часто используют изображение программ, близкое к языку C, с добавлением параллельных операторов (см. [1], [6]). Для удобства читателя дадим здесь описание используемого далее способа изображения программ.

1. Декларации

Декларация (объявление) *переменной* задает тип данных и перечисляет имена одной или нескольких переменных этого типа.

Декларация массива характеризуется добавлением размера по каждому измерению.

В обоих случаях возможна инициализация при объявлении, как видно из следующего примера:

`int i, j = 5;`

`double k = 0.0;`

`int a[n];` # совпадает с `int a[0 : n − 1]`

`int c[1 : n] = ([n]0);` # вектор нулей

`double c[n, n] = ([n]([n]1.0));` # матрица единиц

2. Последовательные операторы

В дальнейшем используются некоторые последовательные операторы:

– оператор присваивания ::=< `переменная` >=< `выражение` >,

– оператор *инкремента* (увеличения) $\mathtt{a[n]}^{++} \Longleftrightarrow \mathtt{a[n]} = \mathtt{a[n]} + 1$ (здесь и далее двойная стрелка $\Longleftrightarrow$ означает эквивалентность двух записей),

– оператор *декремента* (уменьшения): $\mathtt{b}^{--} \Longleftrightarrow \mathtt{b} = \mathtt{b} - 1$.

Обычным образом изображается *вызов функции:* `f(x)` и *условные операторы:*

```
if ([условие])
  [оператор];
```

или

```
if ([условие]) {
  [оператор 1];
  [оператор 2];
  ...
  [оператор N];
}
```

В дальнейшем часто список операторов обозначается через S, так что если S= [оператор 1]; [оператор 1];...; [оператор N], то последний пример может быть записан короче:

```
if ([условие]) {
  S;
}
```

Допускается также полная форма условного оператора:

```
if ([условие])
  [оператор 1];
else
  [оператор 2];
```

Операторы цикла могут использовать `while`

```
while ([условие]) {
  [оператор 1];
  ...
  [оператор N];
}
```

или использовать `for` в более компактной записи, чем обычно:

```
for [[квантификатор 1], ..., [квантификатор M]] {
  [оператор 1];
  ...
  [оператор N];
}
```

Каждый квантификатор вводит новую индексную переменную (параметр цикла), инициализирует ее и указывает диапазон ее значений. Эта переменная локальна в цикле: область ее видимости — тело данного цикла; ее не нужно декларировать.

Примеры.

1). Пример цикла с одним квантификатором:

```
for [i = 0 to n − 1]
  a[i] = i;
```

2). Пример цикла с двумя квантификаторами:

```
for [i = 0 to n − 1,  j = 0 to k − 1]
  w[i, j] = 0;
```

3). Предыдущий пример эквивалентен вложенному циклу вида:

```
for [i = 0 to n − 1]
  for [j = 0 to k − 1]
    w[i, j] = 0;
```

4). Пример перечисления с шагом 2:

```
[i = 1 to n by 2]
```

5). Если в перечислении для переменной `i` нужно исключить значение `x`, то пишут `st! = x` (*st* означает *suchthat* – "такое, что"); например:

```
[i = 0 to n − 1 st i| = x]
```

3. Параллельные операторы и процессы

По умолчанию операторы выполняются последовательно; однако, для параллельных вычислений необходимы операторы, которые можно выполнять параллельно. Параллелизм можно вводить двумя способами. Первый способ — использование оператора `co` (`co` от английского слова *concurrent* — параллельный), а второй — применение *декларации процесса*.

1) Оператор `co` указывает, что несколько операторов могут выполняться параллельно; он начинается ключевым словом `co` и заканчивается ключевым словом `oc`.

Имеется *две формы* оператора `co`.

В *первой форме* оператор `co` имеет несколько ветвей:

```
co [оператор 1];
 // ...
 // [оператор N];
oc
```

Ветви отделяются символом параллелизма //.

Во *второй форме* используется один или более квантификаторов, указывающих, что набор операторов должен выполняться параллельно для каждой комбинации значений параметров цикла.

Пример.

```
co [i = 0 to n − 1] {
 a[i] = 0; b[i] = 0;
} # в i-й ветви a[i], b[i] присваивается 0.
```

2) *Декларация процесса* начинается ключевым словом `process` с именем процесса, а далее следует тело процесса, заключаемое в фигурные скобки.

Пример.

```
process foo {
 int sum = 0;
 for [i = 1 to 10]
  sum+ = i;
```

```
  x = sum
}
```

Декларация процесса записывается *на уровне декларации* процедур (поскольку декларация не является оператором), и к тому же объявляемые процессы выполняются в фоновом режиме, в *противоположность* тому, что выполнение оператора, следующего за оператором `co`, начинается лишь после завершения работы `co`.

Массив процессов объявляется добавлением квантификатора к имени процесса:

```
process bar[i = 1 to n] {
  write(i);
}
```

Замечание. Поскольку процессы, вообще говоря, могут выполняться на различных ВМ, то порядок выполнения последней программы недетерминирован: существует много вариантов выполнения предыдущей программы ($\geq n!$).

4. Процедуры и функции

Процедуры и функции объявляются так же, как в языке C. Например,

```
int Number(int v) { # функция возвращает целое число
  return(v + 1);
}
main() { # "void"-процедура
  int n; sum;
  read(n); # прочитать целое из stdin
  for [i = 1 to n]
    sum = sum + Number(i);
  write("результат", sum);
}
```

Если исходное значение 5, то `sum` $= 0 + 2 + 3 + 4 + 5 + 6 = 20$.

5. Комментарии и предикаты

Комментарии, как обычно, бывают однострочные — они начинаются с # — и многострочные — они начинаются /∗ и кончаются ∗/.

Предикаты — утверждения, которые должны выполняться в некоторых точках программы. Их введение позволяет контролировать правильность вычислений. Они начинаются с ##. Например, ##`x` > `0` означает, что к этому моменту выполнения программы переменная `x` должна быть положительна.

§5 Поиск образца в файле и распараллеливание

Рассмотрим задачу о поиске образца `pattern` в файле.

Как нам известно, Unix выполняет такой поиск с помощью одной команды:

`grep pattern filename`

Это соответствует такой последовательной программе:

```
string line;
[прочитать строку ввода из stdin в переменную line];
while (!EOF) { # пока нет конца файла
  [искать pattern в line];
  if ([pattern есть в line])
    [вывести line];
  [прочитать следующую строку ввода];
}
```

Зададимся вопросами:

— можно ли распараллелить эту программу?

— если можно, то как это сделать?

Основное требование для возможности распараллеливания состоит в том, что программа должна содержать *независимые части.*

Части программы называются *зависимыми*, если хотя бы одна из частей влияет на работу другой части. Если части программы не являются зависимыми, то они называются *независимыми.*

В *простейшем случае* такое влияние может непосредственно передаваться через область памяти, общую для обеих частей программы. Однако, часто ситуация может быть более сложной, при которой передача влияния происходит с помощью какой-либо еще части программы. Ясно, что необходимым условием зависимости частей программы является наличие "канала" передачи информации от одной части к другой (в число составляющих такого "канала" могут входить некоторые участки памяти и программные единицы). Заметим, однако, что наличе такого "канал" не является достаточным условием зависимости частей программы; лишь передача по "каналу" информации от одной части программы, реально влияющей на результаты работы второй части является достаточным условием их зависимости.

Для иллюстрации дальше рассматривается упомянутый выше простейший случай.

Множество переменных, которые программа (процесс) читает, назовем *множеством чтения,* а множество переменных, которые программа записывает, назовем *множеством записи* программы (процесса).

Для того, чтобы две части программы (две программы, два процесса) были *зависимы,* необходимо, чтобы перечение множества чтения одной из них с множеством записи другой было *не пусто.*

Обратно, для независимости двух процессов достаточно, чтобы множество чтения любого из них и множество записи второго не пересекались.

Замечание. Если *два процесса* работают в рамках одной программы, то результаты их работы, возможно, приводят к *общему результату,* смысл которого меняется в зависимости от *порядка* поступления результатов упомянутых процессов. Поэтому даже в случае независимости двух процессов порядок их выполнения может быть *важен.*

Возвращаясь к вопросу о распараллеливании поиска образца в файле, запишем программу

```
string line;
[прочитать входную строку из stdin в переменную line];
while (!EOF) {
 co
```

```
   [искать pattern в line];
   if ([pattern есть в line])
     [вывести line];
   // [прочитать следующую строку вводу и записать ее в line];
  oc;
}
```

Независимы ли эти параллельные процессы? Очевидно, нет, поскольку первый читает из переменной `line`, а другой записывает в нее; если второй процесс выполняется быстрее первого, то перезапись произойдет раньше, чем строка будет проверена.

Предположим, что второй процесс записывает не в ту переменную, которую проверяет первый:

```
string line1, line2;
[прочитать строку ввода из stdin в переменную line1];
while (!EOF) {
  co
    [искать pattern в line1];
    if ([pattern есть в line1])
      [вывести line1];
    // [прочитать следующую строку в line2];
  oc;
}
```

Теперь процессы работают с разными строками, но в этом случае *первый процесс* все время ищет в *одной и той же строке,* а *второй* считывает в `line2`, которая *никогда не рассматривается.*

Для исправления ситуации можно поступить очень просто: в конце каждого цикла поменять роли строк так, что получается программа:

```
string line1, line2;
[прочитать строку воода из stdin в line1];
while (!EOF) {
  co
```

```
   [искать pattern в line1];
   if ([pattern есть в line1])
    [вывести line1];
   // [прочитать следующую строку в line2];
  oc;
  line1 = line2;
}
```

Итак, процессы внутри оператора `co` независимы, но их действия связаны последним присваиванием. Теперь программа работает правильно, но совершенно неэффективно: 1) копирование `line2` в `line1` ведет к перезаписи большого числа символов; 2) оператор распараллеливания `co` находится в теле цикла, и поэтому при каждом повторении цикла будут *возобновляться и уничтожаться два процесса,* что ведет к дополнительным накладным расходам (ибо создание и уничтожение процесса требует значительно больше времени, чем вызов процедуры).

Есть более эффективная параллельная программа: 1) следует поместить цикл `while` в каждый процесс оператора `co`; 2) взаимодействие процессов проводить через общую буферную переменную `buffer` (а переменные `line1` и `line2` — локальные для процессов).

```
string buffer; # содержит одну строку ввода
bool done = false; # используется для сигнала о завершении
co # процесс 1 - поиск шаблона в строках и печатание строк
  string line1;
  while (true) {
    [ожидание заполнения буфера или true в переменной done];
    if (done) break; # выход из цикла
      line1 = buffer;
    [сигнализировать, что буфер пуст];
    [искать pattern в line1];
    if ([pattern есть в line1])
      [напечатать line1];
```

```
  }
  // # процесс 2: прочитать новые строки и запомнить
  string line2;
  while (true) {
    [прочитать следующую строку ввода line2];
    if (EOF) {done = true; break; }
      [ожидать опустошения буфера];
    buffer = line2;
    [сигнализировать о заполнении буфера];
oc;
```

Программа работает следующим образом: пока первый процесс ищет шаблон в строке, второй (параллельный) процесс считывает новую строку и ее запоминает в буфере (буфер считается быстродействующим устройством: запись и чтение с использованием буфера — быстрые операции).

§6 Мелкомодульная неделимость

Напомним, что *неделимым действием* называется действие, в котором ни одно его промежуточное состояние *неразличимо* в процессами данной программы.

Действия, реализуемые аппаратно, заведомо являются неделимыми; они называются *мелкомодульными неделимыми действиями.*

В последовательной программе неделимым действием является, например, *операция присваивания.* Заметим, однако, что в *параллельных* программах оператор присваивания может *не быть неделимым*, т.к. часто он реализуется с помощью последовательности инструкций.

Допустим, что мелкомодульными являются чтение и запись переменных, и рассмотрим следующую программу:

```
int y = 0, z = 0;
co x = y + z // y = 1; z = 2;
oc;
```

Если выражение `x = y + z` реализовано загрузкой значения `y` в регистр с последующим прибавлением значения `z`, то окончательным значением `x` могут быть числа `0, 1, 2, 3`.

Этот эффект происходит из-за того, что в качестве `y` могут фигурировать числа `0, 1`, а в качестве `z` — числа `0, 2`; все зависит от взаимной скорости выполнения первого и второго процессов.

Предположим, что ВС имеет следующие свойства:

1) Значения базовых типов (`int` и др.) хранятся в элементах памяти (например, в словах), которые считываются и записываются неделимыми операциями.

2) Значения обрабатываются по схеме:

— помещаются *в регистры;*

— к ним *применяются операции;*

— результаты помещаются обратно в память.

3) Каждый процесс имеет *собственный набор регистров;* это реализуется или путем предоставления каждому процессу *отдельного набора* физических регистров или виртуально: сохранением и восстановлением значений регистров при выполнении различных процессов (это называется *переключением контекста,* ибо регистры образуют контекст выполнения процесса).

4) Любые промежуточные *результаты,* появляющиеся в вычислениях, *сохраняются в областях, принадлежащих исполняемому процессу* (в регистрах или в стеке процесса).

Благодаря этому, те операции процесса, промежуточные значения которых не видны другим процессам, также могут рассматриваться как неделимые.

Ссылка на переменную, изменяемую другим процессом, называется *критической ссылкой* данного процесса.

Если оператор присваивания содержит не более *одной критической ссылки,* то говорят, что он *удовлетворяет условию "не больше одного".*

С точки зрения других процессов, оператор присваивания, удовлетворяющий условию "не больше одного", может рассматриваться как *неделимый оператор,* ибо упомянутые процессы не могут детализировать картину вычисления этого оператора.

Рассмотрим программу

```
int x = 0,  y = 0;
```

```
co x = x + 1; // y = y + 1; oc
```

Здесь нет критических ссылок; окончательными значениями x и y всегда будут единицы. Эта программа заведомо удовлетворяет условию “не больше одного”.

Следующая программа также удовлетворяет этому условию:

```
int x = 0, y = 0;
co x = y + 1; // y = y + 1; oc;
```

Первый процесс ссылается на y; в результате получается *одна критическая* ссылка, а во *втором* процессе *нет критических ссылок.*

Конечным значением переменной x будет или 1, или 2, а конечным значением y будет 1. Первый процесс увидит переменную y либо до увеличения, либо после увеличения, однако, он не знает, в какой момент (до увеличения или после) он ее видит.

Приведем еще одну программу; здесь ни одно присваивание не удовлетворяет условию “не больше одного”:

```
int x = 0, y = 0;
co x = y + 1; // y = x + 1; oc
```

Окончательными значениями x и y могут быть пары (x, y) = (1, 1), (1, 2), (2, 1).

§7 Оператор ожидания и синхронизация

В ряде случаев некоторую последовательность операторов необходимо выполнить, как неделимое действие.

Для этого необходим механизм, позволяющий задать *крупномодульное неделимое действие.*

Примеры, когда необходимо задать крупномодульное неделимое действие легко себе представить:

1) если одна область памяти обязана быть копией другой (например, при кэшировании), но в динамике в какие-то моменты тождественность этих областей нарушается; в этом случае изменение/перезапись должны быть неделимой операцией с тем, чтобы для внешних процессов вре́менная нетождественность областей не наблюдалась.

2) Если один процесс вставляет элементы в очередь, представленную связанным списком, а другой удаляет элементы из такой очереди. Вставка и удаление связаны с изменением некоторых ссылок и конца очереди. Необходимо, чтобы вставка проходила как единая операция; то же касается и удаления.

Неделимые действия задаются с помощью угловых скобок $<$, $>$, так что, например, выражение $<$ `S` $>$ должно быть выполнено неделимым образом.

Для синхронизации используется оператор `await`; в частности строка

$<$ `await (B) S;` $>$

означает, что при выполнении условия `B` последовательность операторов `S` должна быть выполнена неделимым образом. При этом ни одно промежуточное действие при выполнении последовательности операторов `S` не будет видно другим процессам. Например, выполнение выражения

$<$ `await (j > 0) j = j - 1;` $>$

откладывается до того момента, когда `j` станет положительным: как только `j` становится положительным производится неделимое действие — уменьшение `j` на 1.

Общая форма оператора `await` определяет как *исключение*, так и *синхронизацию по условию.*

Для определения *лишь исключения* можно использовать сокращенную форму оператора `await`, а именно

$<$ `S;` $>$

Для задания *только* условной синхронизации пишут

$<$ `await (B);` $>$

Например, выполнение процесса

$<$ `await (count > 0);` $>$

откладывается до момента, когда `count` станет положительным.

Если выполнено условие “не больше одного”, то $<$ `await (B);` $>$ может быть реализовано в виде

```
while (not B);
```

это так называемый *цикл ожидания* (spin loop). Тело этого оператора пусто; поэтому он просто зацикливается до момента, когда будет выполнено присваивание `B = False`.

Безусловное неделимое действие — это действие, не содержащее условия: оно выполняется немедленно в соответствии с требованием неделимости его компонент.

Условное неделимое действие — это оператор `await` с условием `B`. Если `B` ложно, то процесс приостанавливается до тех пор, пока `B` станет истинным с помощью других процессов.

§8 Синхронизация "производитель – потребитель"

Рассмотрим задачу копирования всех элементов массива `a[n]` от производителя (producer) в массив `b[n]` к потребителю (consumer).

Условие синхронизации процессов имеет вид

$$\texttt{PC}: \ \texttt{c} <= \texttt{p} <= \texttt{c} + 1,$$

где `p` — число перемещенных элементов, `c` — число извлеченных элементов.

Здесь имеется два процесса: 1) `Producer` — производитель, 2) `Consumer` — потребитель. Производитель имеет *локальный* массив `a[n]`, который считается инициализированным, а потребитель имеет *локальный* массив `b[n]`.

Цель: скопировать `a[n]` в `b[n]`.

Пусть буфером взаимодействия служит *одиночная* разделяемая переменная `buf`. Поставленная цель реализуется программой

```
int buf, p = 0, c = 0;
process Producer {
  int a[n] [инициализация]
  while (p < n) {
    < await (p == c); >
    buf = a[p];
```

```
    p = p + 1;
  }
}
process Consumer {
  int b[n];
  while (c < n) {
    < await (p > c); >
    b[c] = buf;
    c = c + 1;
  }
}
```

Процессы `Producer` и `Consumer` получают доступ к переменной `buf` *по очереди,* а разделяемые переменные `p` и `c` служат счетчиками помещенных и извлеченных элементов (соответственно). Начальные значения этих переменных `0`.

Приведенная выше формула $\texttt{PC} : c <= p <= c+1$ означает, что Producer может поместить в буфер столько же, сколько забирает Consumer или Producer может поместить на единицу больше, чем забирает Consumer.

Процесс находится в *состоянии активного ожидания:* он занят проверкой условия в операторе `await`, что можно рассматривать как повторение цикла до тех пор, пока упомянутое условие не будет выполнено.

§9 Обзор логики программирования (ЛП)

Логика программирования является формальной логической системой, позволяющей устанавливать и обосновывать свойства программ.

В *логике программированиея* имеются:

— символы,

— формулы,

— аксиомы,

— правила вывода.

К символам относятся:

— предикаты,
— фигурные скобки,
— операторы языка программирования.

Формулы в ЛП называются тройками. Они имеют вид $\{P\}\,S\,\{Q\}$, где P и Q — предикаты, определяющие отношения между значениями переменных программы, а S — оператор или список операторов.

(*) Интерпретация тройки. Тройка $\{P\}S\{Q\}$ *истинна* при условии, что если выполнение S начинается в состоянии, удовлетворяющем P, и если в конце концов выполнение S завершается, то результирующее состояние удовлетворяет Q.

Эта интерпретация называется частичной корректностью (правильностью); характерная черта частичной корректности в том, что выполнение S может и не завершиться. Частичная корректность относится к *свойствам безопасности.*

Тотальная (полная) корректность получается, если известно, что S всегда завершается; это свойство относится к *свойствам живучести.*

Предикат P называется *предусловием* S, а предикат Q — *постусловием* S.

Примером тройки, не обладающей свойством (*) служит такая тройка

$$\{\texttt{x} == 0\}\ \underbrace{\texttt{x} = \texttt{x} + 1}_{\texttt{S}}\ \{\texttt{y} == 1\}$$

Эта тройка не является истинной, ибо операция `S` никак с `y` не связана.

Аксиома присваивания: $\{P_{x\leftarrow e}\}\ x = e\ \{P\}$.

Запись $P_{x\leftarrow e}$ означает текстуальную подстановку: заменить все свободные вхождения x в предикат P выражением e (переменная называется *свободной* в предикате, если она *не входит* в область действия квантора существования или квантора всеобщности, имеющих переменную с тем же именем).

Эта аксиома означает: если нужно, чтобы присваивание приводило к предикату P, то предшествующее состояние должно удовлетворять предикату P, в котором вместо переменной x записано выражение e.

Перечислим наиболее важные правила вывода.

Правило композиции

$$\frac{\{P\}\, S_1\, \{Q\},\ \{Q\}\, S_2\, \{R\}}{\{P\}\, S_1;\, S_2\, \{R\}}.$$

Правило оператора `if`

$$\frac{\{P \wedge B\}\, S\, \{Q\},\ \{P \wedge \neg B\} \Rightarrow Q}{\{P\}\, \texttt{if}\, (B) S;\, \{Q\}}.$$

Правило оператора `while`

$$\frac{\{I \wedge B\}\, S\, \{I\}}{\{I\}\, \texttt{while}(B) S;\, \{I \wedge \neg B\}}.$$

Правило следования

$$\frac{\{P' \Rightarrow P\},\ \{P\}\, S\, \{Q\},\ Q \Rightarrow Q'}{\{P'\}\, S\, \{Q'\}}.$$

Для правила оператора `while` используется инвариант цикла I; этот предикат, значение которого истинно перед *каждым* повторением цикла и после него. Если инвариант I и условие B истинны до S, то I истинно и после S; на этом обстоятельстве и зиждется выполнение цикла. Как только B становится ложным, цикл прекращается (а I остается истинным), состояние принимает вид $I \wedge \neg B$.

Пример. (программа просмотра массива и поиска первого вхождения значения `x`).

```
i = 1;
{i == 1 ∧ (∀j : 1 <= j < i : a[j]! = x)}
while (a[i]! = x)
  i = i + 1;
{(∀j : 1 <= j < i : a[j]! = x) ∧ a[i] == x}
```

§10 Логика программирования при параллельном выполнении

Оператор параллельного выполнения `co` является *управляющим* оператором. *Процессы* состоят из *последовательных* операторов и операторов *синхронизации* (например, `await`): < `await` (`B`) `S`; >.

1) *Правило оператора* `await`. Поскольку действие оператора `await` похоже на действие оператора `if`, то правило вывода аналогично:

$$\frac{\{P \wedge B\}\, S\, \{Q\}}{\{P\} < \text{await}(B) S; > \{Q\}}.$$

Предпосылка: "если $P \wedge B$ истинны, то если S завершается, то Q будет истинным"; вывод: "из состояния P оператор `await` приводит к Q" (если `await` завершается).

Замечание. Правила вывода ничего не говорят о задержках выполнения (задержки выполнения влияют на свойство живучести и на свойство безопасности).

2) *Правило оператора со.* Рассмотрим выражение

$$\texttt{co}\ S_1;\ //S_2;\ // \ldots //S_n;\ \texttt{oc}.$$

Говорят, что процессы S_i *не влияют друг на друга*, если ни один процесс не совершает операции, нарушающие условия выполнения другого процесса.

Предположим, что выражения

$$\{P_i\}\, S_i\, \{Q_i\}$$

свободны от взаимного вмешательства (т.е. процессы S_i не влияют друг на друга) и истинны. Тогда истинно выражение

$$\{P_1 \wedge \ldots \wedge P_n\}\, \texttt{co}\, S_1; // \ldots //S_n;\, \texttt{oc}\, \{Q_1 \wedge \ldots \wedge Q_n\}.$$

Это правило можно записать следующим образом

$$\frac{\{P_i\}\, S_i\, \{Q_i\}\ [\text{свободны от взаимного вмешательства}]}{\{P_1 \wedge \ldots \wedge P_n\}\, \texttt{co}\, S_1; // \ldots //S_n;\, \texttt{oc}\, \{Q_1 \wedge \ldots \wedge Q_n\}}.$$

Рассмотрим программу

$\{\texttt{x} == 0\}$

$\texttt{co}\ \ < \texttt{x} = \texttt{x} + 1; > // < \texttt{x} = \texttt{x} + 2; > \texttt{oc}$

$\{\texttt{x} == 3\}$

Истинна ли эта тройка? Если выполнение программы начинается при `x` = 0, то нет возможности сказать, какое `x` поступит в

каждый из процессов ($\mathtt{0,2}$ в первый и $\mathtt{0,1}$ во второй). От этого будет зависеть результат (на первый взгляд кажется, что $\mathtt{x}$ может быть $\mathtt{1,2}$ или $\mathtt{3}$), ибо *порядок выполнения недетерминирован.*

Очевидно, достаточно проверить истинность программы с предусловием $\mathtt{x} == \mathtt{0}$ и с постусловием $\mathtt{x} == \mathtt{3}$, а именно:

$\{\mathtt{x} == \mathtt{0}\}$
$\mathtt{co}\{\mathtt{x} == \mathtt{0} \vee \mathtt{x} == \mathtt{2}\}$
$< \mathtt{x} = \mathtt{x} + \mathtt{1}; >$
$\{\mathtt{x} == \mathtt{1} \vee \mathtt{x} == \mathtt{3}\}$
$//\{\mathtt{x} == \mathtt{0} \vee \mathtt{x} == \mathtt{1}\}$
$< \mathtt{x} = \mathtt{x} + \mathtt{2}; >$
$\{\mathtt{x} == \mathtt{2} \vee \mathtt{x} == \mathtt{3}\}$
$\mathtt{oc}$
$\{\mathtt{x} == \mathtt{3}\}$

Утверждения в каждом процессе учитывают оба порядка выполнения, причем конъюнкция предусловий дает $\{x == 0\}$:

$$\{x == 0 \vee x == 2\} \wedge \{x == 0 \vee x == 1\} = \{x == 0\},$$

а конъюнкция постусловий дает $\{x == 3\}$:

$$\{x == 1 \vee x == 3\} \wedge \{x == 2 \vee x == 3\} = \{x == 3\}.$$

Вывод: Истинность рассматриваемой тройки доказана.
Замечание. Приведенный пример иллюстрирует *схему доказательства* программы (все три тройки оказались истинными).

Формальное определение невмешательства

Действие присваивания — это оператор присваивания или оператор `await`, содержащий одно или несколько присваиваний.

Критическое утверждение — это предусловие или постуловие *вне* оператора `await`.

Если a — действие присваивания в некотором процессе, $pre(a)$ — его предусловие, C — критическое утверждение в другом процессе, действие a *не вмешивается* в C, если *истинна тройка*

$\{C \wedge pre(a)\}\, a\, \{C\}$, т.е. если истинны $C \wedge pre(a)$ перед применением a, то истинно C после применения a.

Иначе говоря, критическое утверждение C инвариантно относительно действия a.

Рассмотрим пример, вытекающий из приведенной выше программы: предусловие первого процесса является *критическим утверждением* (поскольку оно находится вне оператора `await`). Однако на него *не влияет* оператор присваивания во втором процессе, ибо истинна такая тройка:

$\{(\mathtt{x} == 0 \vee \mathtt{x} == 2) \wedge (\mathtt{x} == 0 \vee \mathtt{x} == 1)\}$

$\mathtt{x} = \mathtt{x} + 2$

$\{\mathtt{x} == 0 \vee \mathtt{x} == 2\}$

В рассматриваемой программе имеется еще три критических утверждения: постусловие первого процесса, предусловие и постусловие второго процесса. Аналогично проверяется взаимные невмешательства и в этих сдучаях.

§11 Устранение взаимного вмешательства: непересекающиеся множества переменных

Имеется четыре основных метода устранения взаимного вмешательства:

1) использование непересекающихся множеств переменных;
2) применение ослабленных утверждений;
3) отслеживание глобальных инвариантов;
4) синхронизация.

Остановимся более подробно на первом методе.

Множество записи процесса — множество переменных, которым он присваивает значения.

Множество чтения — это множество переменных, которые он читает, но *не изменяет.*

Множество ссылок процесса — множество переменных, которые встречаются в *утверждениях доказательства правильности* процесса.

Часто множество ссылок процесса совпадает с объединением множества записи и множества чтения.

По отношению к взаимному вмешательству *критические переменные* — это *переменные в утверждениях.*

В простейшем случае (см. параграф 5) если *множество записи* одного процесса не пересекается с *множеством ссылок* другого процесса и наоборот, то эти процессы не могут влиять друг на друга.

Например, для программы

$\texttt{co}\ \texttt{x} = \texttt{x} + 1;\ //\ \texttt{y} = \texttt{y} + 1;\ \texttt{oc}$

имеем

$\{\texttt{x} == 0\}\ \texttt{x} = \texttt{x} + 1\ \{\texttt{x} == 1\}$

$\{\texttt{y} == 0\}\ \texttt{y} = \texttt{y} + 1\ \{\texttt{y} == 1\}$

Каждый процесс имеет один оператор присваивания и два утверждения: нужно доказать 4 теоремы взаимного невмешательства. Каждая из них тривиально истинна, ибо процессы ссылаются на разные переменные и подстановки в аксиоме присваивания будут пустыми.

Непересекающиеся множества записи и ссылок характерны для итерационных процессов (например, для процессов, связанных с перемножением матриц, с исследованием разных ветвей дерева, с транзакциями в различных частях базы данных и т.д.).

§12 Ослабленные утверждения

Ослабленное утверждение допускает *больше* состояний программы, чем утверждение, которое могло бы быть истинным для изолированного процесса.

В качестве примера рассмотрим приведенную ранее программу:

$\{\texttt{x} == 0\}$

$\texttt{co}\{\texttt{x} == 0 \vee \texttt{x} == 2\}$

$\quad < \texttt{x} = \texttt{x} + 1; >$

$\quad \{\texttt{x} == 1 \vee \texttt{x} == 3\}$

```
//{x == 0 ∨ x == 1}
   < x = x + 2; >
  {x == 2 ∨ x == 3}
oc
{x == 3}
```

В этой программе предусловия и постусловия для каждого процесса слабее, чем могли бы быть при изолированном их выполнении. Например, для процесса $x = x + 1$; можно было бы записать

```
{x == 0}
x = x + 1;
{x == 1}
```

Аналогично — для второго процесса.

Рассмотрим часто встречающуюся задачу.

Пусть процесс планирует операции диска с перемещаемыми головками, а другие процессы добавляют операции в очередь. Когда диск не занят, он выбирает (по некоторому критерию) наилучшую операцию и начинает ее выполнение. На самом деле здесь реализуется *ослабленная ситуация:* диспетчер *не всегда* выполняет наилучшуют операцию, ибо в процессе выполнения может в *очереди появиться еще* одна, *лучшая операция,* после того, как был сделан выбор. Итак, здесь происходит естественное ослабление условий.

Аналогичным образом можно рассмотреть параллельный алгоритм, часто встречающийся при приближенном решении дифференциальных уравнений (например, методом сеток).

Пространство задач аппроксимируется конечной сеткой точек grid[n, n]. Каждой точке (или блоку точек) сетки назначается процесс

```
double grid[n, n];
process PDE[i = 1 to n − 1, j = 1 to n − 1] {
  while ([не сошлось]) {
    grid[i, j] = f([окружение точки (i, j)])
  }
}
```

Вместо индивидуального вычисления итерации цикла для точки с индексами `i`, `j` с помещением результата в `grid[i, j]` используется расширение:

1) берется другой массив, куда присваивается полученный результат;

2) производится барьерная синхронизация — шаг вычисления для всех процессов;

3) массивы меняются местами;

4) проводится следующий шаг вычисления.

§13 Глобальные инварианты

Эффективным способом избежать взаимного вмешательства являются глобальные инварианты.

Предположим, что I — предикат, который ссылается на глобальные переменные. Предикат I называется *глобальным инвариантом* по отношению к рассматриваемому *множеству процессов,* если

1) I является истинным в начале выполнения процессов,

2) I остается истинным при каждом присваивании.

Условие 1) удовлетворяется, если I — истина в начале каждого процесса, а условие 2) выполняется, если для любого действия присваивания I – истина после присваивания, если I был истинным до присваивания (математическая индукция).

Теперь предположим, что предикат I является глобальным инвариантом. Предположим, что любое критическое[1] утверждение C в обосновании каждого процесса P_j имеет вид $I \wedge L_j$, где L_j — предикат относительно локальных (для P_j) переменных, т.е. каждая переменная, на которую ссылается L_j, является либо локальной для процесса P_j, либо глобальной, но такой, которой делается лишь присваивание и присваивает только данный процесс.

Если все критические утверждения можно представить в виде $I \wedge L_j$, то процессы будут свободны от взаимного вмешательства.

[1] Напомним, что *критическое* утверждение — это предусловие или постусловие вне оператора await (т.е. это предусловие или постусловие, ссылающееся на переменные, которые *могут* быть изменены другими процессами). Если a — действие присваивания в другом процесса, то a *не вмешивается* в C, если истинна тройка $\{C \wedge pre(a)\}\, a\, \{C\}$.

Действительно, 1) предикат I инвариантен относительно любого присваивания, 2) ни одно действие присваивания в другом процессе не может повлиять на предикат L_j, ибо левые части отличаются от переменных в предикате L_j.

§14 Задача копирования массива

Эта задача нам уже встречалась. Приведем одно из ее программных решений.

```
int buf, p = 0, c = 0;
  # особенность: буфер может хранить лишь один элемент массива
  # p — число "поставок"
  # c — число "потребителей"
  # n — размер массива
process Producer {
  int a[n];
  while (p < n) {
    < await (p == c); >
    # дальнейшее выполнение процесса Producer откладывается
      до момента, когда выполнится условие p == c
      [процесс зациклен в операторе await]
    buf = a[p];
    p = p + 1;
  }
}
process Consumer {
  int b[n];
  while (c < n) {
    < await (p > c); >
    # выполнение процесса откладывается до момента,
      когда станет p > c
      [процесс зациклен в операторе await]
```

```
    b[c] = buf;
    c = c + 1;
  }
}
```

Условие синхронизации процессов имеет вид

$$\texttt{PC}: \ \texttt{c} <= \texttt{p} <= \texttt{c} + 1;$$

это означает, что Producer делает число "поставок" столько или на единицу больше числа "потреблений".

Замечание. Здесь представлен лишь фрагмент программы; можно считать, что процессы `Producer` и `Consumer` выполняются каждый на своем вычислительном модуле параллельной ВС.

Процессы `Producer` и `Consumer` по-очереди получают доступ к переменной `buf`. Сначала Producer помещает первый элемент массива `a` в переменную `buf`, затем Cinsumer извлекает его, а Producer помещает в `buf` следующий элемент массива и т.д.

В переменных `p` и `c` ведется *подсчет* числа *помещенных* и *извлеченных* элементов соответственно.

Для *синхронизации* доступа к переменной `buf` используются операторы await. Когда выполняется условие $\texttt{p} == \texttt{c}$ буфер пуст (последний помещенный в него элемент извлечен). Когда выполняется условие $\texttt{p} > \texttt{c}$, буфер заполнен.

Предположим, что сначала элементы массива `a[j]` содержат значения `A[j]` (здесь `A[j]` — целые числа). Наша цель — доказать, что при условии завершения программы значения элементов массива `b[n]` будут совпадать с `A[n]`, т.е. со значениями элементов массива `a[n]`.

Введем глобальный инвариант

$$\texttt{PC}: \ (\texttt{c} <= \texttt{p} <= \texttt{c} + 1) \wedge (\texttt{a}[0 : \texttt{n} - 1] = \texttt{A}[0 : \texttt{n} - 1])$$

$$\wedge\big((\texttt{p} == \texttt{c} + 1) \Rightarrow (\texttt{buf} == \texttt{A}[\texttt{p} - 1])\big).$$

Поскольку процессы чередуют доступ к `buf`, то в любой момент значение `p` равно `c` или больше `c` на 1. Массив `a[n]` не изменяется, и поэтому значением `a[i]` всегда является `A[i]`; наконец, если `p` на единицу больше, чем `c`, то буфер содержит $\texttt{A}[\texttt{p} - 1]$.

Предикат PC в начальном состоянии является истинным, поскольку p и c равны 0. Процесс доказательства имеет вид:

1) int buf, p = 0, c = 0;
2) {PC : (c <= p <= c + 1) ∧ (a[0 : n − 1] == A[0 : n − 1])∧
3) ((p == c + 1) ⇒ (buf == A[p − 1]))}
4) process Producer {
5) int a[n]; # предполагается, что $a[i]$ инициализирован с помощью $A[i]$
6) {IP : PC ∧ (p <= n)}
7) while (p < n) {
8) {PC ∧ (p < n)}
9) < await (p == c); > # ожидание опустошения буфера
10) {PC ∧ (p < n) ∧ (p == c)}
11) buf = a[p];
12) {PC ∧ (p < n) ∧ (p == c) ∧ (buf == A[p])}
13) p = p + 1;
14) {IP}
15) }
16) {PC ∧ (p == n)}
17) }
18) process Consumer {
19) int b[n];
20) {IC : PC ∧ (c <= n) ∧ (b[0 : c − 1] == A[0 : c − 1])}
21) while (c < n) {
22) {IC ∧ (c < n)}
23) < await (p > c); > # ожидание заполнения буфера
24) {IC ∧ (c < n) ∧ (p > c)}
25) b[c] = buf;
26) {IC ∧ (c < n) ∧ (p > c) ∧ (b[c] == A[c])}
27) c = c + 1;
28) {IC}

29) }
30) $\{\texttt{IC} \wedge (\texttt{c} == \texttt{n})\}$
31) }

В этом листинге перед каждым оператором и после него имеются предикаты, и каждая такая тройка $\{P\}\, S\, \{Q\}$ истинна. Тройки в каждом процессе образуются непосредственно вокруг оператора присваивания в каждом процессе.

Каждый процесс завершается после `n` итераций. Когда программа завершается, постусловия обоих процессов истинны, так что заключительное состояние программы удовлетворяет предикату

$$\texttt{PC} \wedge (\texttt{p} == \texttt{n}) \wedge \texttt{IC} \wedge (\texttt{c} == \texttt{n}),$$

откуда следует, что массив `b` содержит копию массива `a`.

Предикаты указанных процессов не оказывают взаимного влияния, ибо являются комбинациями `PC` и локальных предикатов, соответствующих требованиям взаимного невмешательства; исключениями являются предикаты, определяющие отношения между `p` и `c`, но они не подвержены влиянию благодаря операторам `await`.

§15 Стратегии планирования

При наличии нескольких процессов существует несколько возможных неделимых действий. *Стратегия планирования* определяет, какое из них будет выполнено следующим.

Рассмотрим программу

```
bool continue = true;
co while (continue);
 // continue = false;
oc
```

Допустим, что стратегия планирования *назначает процессор* для очередного процесса до тех пор, пока он не завершится, и *лишь затем* назначается процессор для следующего процесса.

Если в данной программе назначение процессора произойдет для первого процесса, то выполнение программы никогда не завершится; а если для второго процесса, то программа завершится.

Интуитивно чувствуется, что упомянутая стратегия не является справедливой.

Определение 1. Стратегия планирования называется *почти справедливой,* если любое допустимое (безусловное) неделимое действие в конце концов выполняется.[2]

Запуск и исполнение предыдущей программы при почти справедливой стратегии приведет в конце концов к ее окончанию.

Для этой программы почти справедливой стратегией ее выполнения на однопроцессорной системе явилась бы циклическая смена процессов (например, по времени) с сохранением состояния сменяемого процесса.

Однако, для программ, содержащих операторы `await` с условиями, стратегия почти справедливости недостаточна для завершения программы, т.к. условное неделимое действие не может быть выполнено, пока условие не станет истинным.

Определение 2. Стратегия планирования называется *справедливой в слабом смысле,* если 1) она почти справедлива, 2) каждое допустимое условное неделимое действие в конце концов выполняется, если *условие становится* и затем *остается истинным* в процессе реализации условного неделимого действия.

Однако, при стратегии, справедливой в слабом смысле, нет гарантии, что любой допустимый оператор `await` в конце концов выполнится; это связано с тем, что в период задержки условие может многократно и быстро меняться, принимая значения `false` и `true`, а проверка этого условия задержанным процессом может приходиться всякий раз на значение `false`, так что процесс не будет продвигаться.

Поэтому необходима более сильная стратегия.

Определение 3. Стратегия называется *справедливой в сильном смысле,* если 1) она *почти справедлива*; 2) любое условное неделимое действие в конце концов выполняется, если его *условие бывает истинным бесконечно часто.*

Рассмотрим программу

```
bool continue = true, try = false;
co while (continue) {try = true; try = false;}
 // < await (try) continue = false; >
```

[2]Иногда эту стратегию называют *безусловно справедливой,* см. [7].

При стратегии, справедливой в сильном смысле, программа в конце концов завершится, ибо переменная `try` принимает значение `true` бесконечно часто, и потому (по принуждению) сработает второй процесс.

Однако, при стратегии, справедливой в слабом смысле, программа может не завершиться, ибо переменная `try` принимает значения `false` также бесконечно много раз, и проверки во втором процессе могут всегда совпадать с моментами, когда `try` = `false`.

Если все циклы активного ожидания зациклились навсегда, то говорят, что программа вошла *в активный тупик.*

Глава 2 Критические секции и барьеры

§1 Задача критической секции. Крупномодульное решение

Классической задачей при параллельном программировании является *задача критической секции*[3].

В этой задаче n процессов многократно выполняют сначала критическую, а затем — некритическую секцию кода.

Критической секции соответствует *протокол входа,* а за ней следует *протокол выхода.* Итак, процесс имеет вид

```
process CS[i = 1 to n] {
  while (true) {
    [протокол входа];
    [критическая секция];
    [протокол выхода];
    [некритическая секция];
  }
}
```

Протоколы входа должны удовлетворять следующим свойствам.

(1) "Взаимное исключение": в любой момент только один процесс может выполнять свою критическую секцию.

[3]Критическая секиця — неделимая последовательность действий, которую нельзя прервать другими процессами.

(2) "Отсутствие взаимной блокировки": если несколько процессов пытаются войти в свои критические секции, хотя бы один это осуществит.

(3) "Отсутствие излишних задержек": если процесс хочет войти в свою критическую секцию, а другие выполняют некритические секции или завершены, то процессу разрешается вход в упомянутую критическую секцию.

(4) "Возможность входа": процесс, который пытается войти в критическую секцию, когда-нибудь это сделает.

Свойство "взаимное исключение" нарушается, если два процесса находятся в своих критических секциях.

Для "отсутствие излишних задержек" нарушается, если единственный процесс, пытающийся войти в критическую секцию, не может этого сделать.

Пример. Пусть `in1` и `in2` — логические переменные. Процесс `S1` (`S2`) находится в критической секции в то время как `in1` (`in2`) присваивается "истина".

Для взаимного исключения требуется истинность состояния `MUTEX` : $\neg(\texttt{in1} \wedge \texttt{in2})$.

ПРОГРАММА ДЛЯ КРИТИЧЕСКОЙ СЕКЦИИ (КРУПНОМОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ)

```
bool in1 = false, in2 = false;
## MUTEX : ¬(in1 ∧ in2) — глобальный инвариант
 1) process CS1 {
 2)   while (true) {
 3)     < await (!in2) in1 = true; >  # вход
 4)     [критическая секция];
 5)     in1 = false; # выход
 6)     [некритическая секция];
 7)   }
 8) }
 9) process CS2 {
10)   while (true) {
```

```
11)     < await (!in1) in2 = true; >  # вход
12)     [критическая секция];
13)     in2 = false;  # выход
14)     [некритическая секция];
15)   }
16) }
```

§2 Активные блокировки

Можно рассмотреть произвольное число `n` процессов, но использовать одну логическую переменную (вместо `n` логических переменных в каждом процессе).

Пусть `lock` — логическая переменная. Если один из процессов находится вкритической секции, то в предыдущей программе `in1` $\vee$ `in2` == `true`. Пусть `lock` принимает значение `true`, если процесс находится в критической секции, а если нет процессов в критической секции, то `true` = `false`. Проиллюстрируем использование этой переменной для двух процессов ($n = 2$).

Замечание. Инвариант, рассмотренный выше (а именно, $\neg$(`in1`$\wedge$`in2`)) нельзя сформулировать при помощи переменной `lock`. Однако, использование `lock` позволяет поддерживать этот инвариант; это вытекает из следующего примера.

```
bool lock = false;
process CS1 {
  while (true) {
    < await (!lock) lock = true; >  # если !lock, то вход
                                    # разрешен: входим...
    [критическая секция];
    lock = false;  # выходим из критической секции и сразу
                   # устанавливаем lock = false
    [некритическая секция];
  }
}
process CS2 {
```

```
  while (true) {
    < await (!lock) lock = true; >  # если !lock, то вход
                                    # разрешен: входим...
    [критическая секция];
    lock = false; # выходим из критической секции и сразу
                  # устанавливаем lock = false
    [некритическая секция];
  }
}
```

Замечание. Использование логической переменной `lock` при произвольном числе процессов аналогично.

§3 Инструкция “проверить–установить”

Инструкция “проверить–установить” (test and set — кратко TS) получает в качестве аргумента переменную `lock`, в неделимом действии присваивает ей значение `true`, а затем возвращает предыдущее сохраненное значение переменной `lock`. Это описывается так:

```
bool TS(bool lock) {
  < bool initial = lock; # сохраняем начальное значение
    lock = true; # устанавливаем lock внутри неделимого действия
    return initial; > # возвращаем предыдущее значение
}
```

Здесь приведем программу с использованием инструкции `TS`. Такое ее использование называется *циклической блокировкой.*

КРИТИЧЕСКИЕ СЕКЦИИ НА ОСНОВЕ
“ПРОВЕРИТЬ–УСТАНОВИТЬ”
(ЦИКЛИЧЕСКАЯ БЛОКИРОВКА)

```
1) bool lock = false;
2) process CS[i = 1 to n] {
```

```
3)  while (true) {
4)    while (TS(lock)) skip; # протокол входа
5)    [критическая секция];
      # ввиду свойства TS здесь lock == true
6)    lock = false; # восстанавливаем возможность доступа
                    # к критическим секциям
7)    [некритическая секция];
8)  }
9) }
```

Свойства этой программы состоят в следующем.

(1) "Взаимное исключение" выполнено, ибо если `lock = false`, и несколько процессов хотят войти в критическую секцию, то только один войдет в нее и сделает присваивание `lock = true` (а другие не войдут).

(2) "Отсутствие взаимной блокировки" выполнено: если оба процесса находятся во входных протоколах, то `lock` имеет значение `false`, и лишь один из них, войдя в критическую секцию, сразу изменит (с помощью `TS`) значение переменной `lock` на `true` (второй же процесс войти в критическую секцию не сможет).

(3) Излишние задержки не возникают: если оба процесса вне критических секций, то один из них может войти.

(4) Если используется стратегия планирования, справедливая в слабом смысле, то выполнение возможности входа *не гарантируется* (возможен поток значений `false` у переменной `lock` как раз в моменты проверок этого значения процессом); однако, если стратегия планирования справедлива в сильном смысле, то *возможность входа гарантируется.*

Правило: при решении задачи критической секции с циклической блокировкой протокол выхода *должен присваивать* разделяемым переменным их *начальные значения.*

§4 Протокол “проверить–проверить–установить”

Применение стратегии “проверить–установить” приводит к снижению производительности из-за частого обращения к памяти, что

приводит к конфликтам при использовании переменной `lock`.

Заменив протокол входа на протокол "проверить-провоерить-установить", можно уменьшить затраты на конфликты.

ПРОТОКОЛ НА ОСНОВЕ “ПРОВЕРИТЬ–ПРОВЕРИТЬ–УСТАНОВИТЬ”

```
while (lock) skip;
# пока установлено lock = true повторять цикл
while (TS(lock)); {
# попытка захватить lock [, равное false]
  while (lock) skip; }
  # повторять цикл, если попытка не удалась
  #(lock успела сменить свое значение на true)
```

Этот протокол называется “проверить–проверить–установить”, поскольку в двух циклах `lock` лишь только проверяется и это не влияет на другие процессы. Если `lock = false` (т.е. флаг блокировки сброшен), то хотя бы один приостановленный процесс проверит инструкцию `TS` (продолжаться будет лишь один из них).

КРИТИЧЕСКАЯ СЕКЦИЯ НА ОСНОВЕ ПРОТОКОЛА “ПРОВЕРИТЬ–ПРОВЕРИТЬ–УСТАНОВИТЬ”

```
   bool lock = false; # разделяемая блокировка
1) process CS[i = 1 to n] {
2)   while (true) {
3)     while (lock) skip; # протокол входа
4)     while (TS(lock)) {
5)       while (lock) skip;
6)     }
7)     [критическая секция];
8)     lock = false; # протокол выхода
9)     [некритическая секция];
10)  }
11) }
```

§5 Реализация оператора await

Решение задачи критической секции можно использовать для реализации безусловного неделимого действия $<$ `S`; $>$, *скрывая внутренние контрольные точки* от других процессов.

Пусть `CSenter` — входной протокол, `CSexit` — выходной протокол. Тогда действие $<$ `S`; $>$ реализуется так

```
CSenter;
S;
CSexit;
```

Здесь предполагается, что *все секции кодов* процессов, которые изменяют или ссылаются на переменные, изменяемые в `S`, *защищены* аналогичными протоколами (критическими секциями).

Теперь нам доступна реализация оператора $<$ `await (B) S`; $>$ — условного неделимого действия, которое начинает осуществляться при `B = true` и далее выполняет `S`.

Для циклической проверки условия `B` можно использовать такой цикл (тело цикла, отмеченное вопросительными знаками, обсуждается далее):

```
1) CSenter;
2) while (!B) {???};
3) S;
4) CSexit;
```

Здесь опять-таки предполагается, что критические секции всех процессов, *изменяющих* переменные `B` или `S`, или *использующих* переменные, изменяемые в `S`, защищены *такими же* протоколами входа и выхода.

Возникает вопрос, *как реализовать тело цикла,* выделенного выше вопросительными знаками. Если цикл выполняется, значит `B = false`; единственный способ сделать условие `B` истинным — изменить в другом процессе значения переменных, входящих в это условие. Но поскольку операторы, изменяющие эти переменные находятся в критических секциях, а эти критические секции не могут исполняться, когда рассматриваемый процесс находится в своей критической секции, то *единственный* способ продвижения

этого процесса - *выход из критической секции.* Этим определяется желаемая программа, которая может иметь вид

```
CSenter;
while (!B) { CSexit; CSenter; };
# цикл = процесс многократно выходит из критической секции
# в ожидании события B = true
S;
CSexit;
```

Если стратегия планирования является *в слабом смысле,* то цикл завершится при условии, что B станет истинным и будет *оставаться* таковым, пока процесс проходит критическую секцию.

В случае реализации стратегии, справедливой в *сильном смысле,* цикл завершится и в случае, когда B становится *истинным бесконечное число раз.*

Написанная выше программа верна, но *не эффективна* из-за того, что процесс *многократно* выходит из критической секции в ожидании того, что какой-то другой процесс в этот момент войдет в свою критическую секцию и изменит переменные так, что B станет истинным. Это может привести к *конфликтам* при обращении к памяти или к ресурсам коммуникационной среды.

Поэтому *для сокращения числа попыток* раздобыть B = true вводят задержку Delay, замедляющую скорость выполнения процесса. Получаемый протокол называется протоколом "отхода" ("back-off").

ПРОТОКОЛ ОТХОДА (BACK-OFF)

```
CSenter;
while (!B) { CSexit; Delay; CSenter; };
S;
CSexit;
```

Замечание 1. Задержкой Delay может быть, например, пустой цикл, выполняющий случайное количество повторений.

Замечание 2. Протокол, аналогичный описанному, применяется в сетях *Ethernet:* для передачи сообщения контроллер Ethernet

посылает его в сеть и *следит*, не возникнет ли конфликт с другими сообщениями от других контроллеров. Если *конфликтов не наблюдается*, то считается, что *сообщение послано удачно;* в противном случае посылка *повторяется* через *случайный* промежуток времени. Такой протокол называется "двоичным экспоненциальным протоколом отхода".

Замечание 3. Если `S` состоит из одного оператора `skip`, то протокол упрощается; в частности, если условие `B` удовлетворяет условию "не больше одного", то оператор $<$ `await (B);` $>$ реализуется в виде `while (!B) skip;`.

§6 Алгоритм разрыва узла

Вспомним крупномодульное решение задачи о критической секции.

```
1) bool in1 = false, in2 = false;
2) ## INY : ¬(in1 ∧ in2) — глобальный инвариант
3) process CS1 {
4)   while (true) {
5)     < await (!in2) in1 = true; >  # вход
6)     [критическая секция];
7)     in1 = false;  # выход
8)     [некритическая секция];
9)   }
10) }
11) process CS2 {
12)   while (true) {
13)     < await (!in1) in2 = true; > # вход
14)     [критическая секция];
15)     in2 = false;  # выход
16)     [некритическая секция];
17)   }
18) }
```

Недостаток такого решения задачи о критической секции в том, что *в критическую секцию может попасть любой* из процессов (безконтрольно). Например, один процесс может несколько раз входить в свою критическую секцию, а другой — ни разу. Это — несправедливое решение. Для того, чтобы решение было справедливым, должна соблюдаться очередность входа в критическую секцию, если несколько процессов пытаются туда войти.

Расмотрим следующую программу.

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
 while (true) {
  last = 1; in1 = true; # протокол входа
  < await (!in2 or last == 2); >
  [критическая секция];
  in1 = false; # протокол выхода
  [некритическая секция];
 }
}
process CS2 {
 while (true) {
  last = 2; in2 = true; # протокол входа
  < await (!in1 or last == 1); >
  [критическая секция];
  in2 = false; # протокол выхода
  [некритическая секция];
 }
}
```

Здесь введена пременная `last`, которая принимает значение 1, если последним входил в критическую секцию процесс `CS1`, и принимает значение 2, если последним входил в критическую секцию процесс `CS2`.

Приведенная только что программа представляет один из алгоритмов "разрыва узла" для двух процессов.

Однако, эта программа не исключает одновременное появление процессов в своих критических секциях; действительно, если после присваивания в `CS1` `last` $=$ `1` процесс `CS2` вошел в свою критическую секцию, то он успевает присвоить `last` $=$ `2` и проверить, что `in1` $=$ `false` (а после этого процесс `CS1` присваивает `in1` $=$ `true`). В этом случае процесс `CS1` сможет войти в свою критическую секцию (в то время как `CS2` еще не вышел из своей критической секции).

Таким образом, корректная реализация алгоритма разрыва узла существенно зависит от работы оборудования: существенная *разница в скорости обработки процессов* может нарушить правильность реализации приведенной программы, так что *оба процесса окажутся в своих критических секциях*.

Читателю рекомендуется предложить свои варианты "разрыва узла" и проанализировать их с точки зрения реализуемости при тех или иных предположениях о работе оборудования.

§7 Алгоритм билета

Здесь получим одно из возможных обобщений алгоритма разрыва узла на случай `n` процессов.

Моделью приводимого решения является вытягивание билетов (номеров) и последующее ожидание своей очереди в предположении, что последовательное вытягивание билетов сопровождается возрастанием их номеров; это решение называется *алгоритмом билета*.

АЛГОРИТМ БИЛЕТА: КРУПНОМОДУЛЬНОЕ РЕШЕНИЕ

```
1) int number = 1, next = 1, turn[1 : n] = ([n]0);
2) process CS[i = 1 to n] {
3)   while (true) {
4)     < turn[i] = number; number = number + 1; >
5)     < await (turn[i] == next); >
6)     [критическая секция];
```

```
7)     < next = next + 1; >
8)    [некритическая секция];
9)  }
10) }
```

Недостатки алгоритма. 1) Общий недостаток для алгоритмов с неограниченно увеличивающимися счетчиками: увеличение счетчиков может привести к переполнению (к счастью, на практике это бывает редко). 2) Первое из фигурирующих в этой программе неделимых действий — *чтение* числа `number` и его *увеличение* — реализовать трудно.

В некоторых вычислительных системах имеются инструкции, возвращающие одновременно и старое значение переменной и увеличенное (или уменьшенное) на 1 ее значение.

Пусть у нас имеет инструкция `FA` (Fetch and Add — извлечь и сложить):

```
FA (var, incr) :
 < int tmp = var; var = var + 1; return(tmp); >
```

Тогда можно предложить *мелкомодульное* решение *алгоритма билета.*

АЛГОРИТМ БИЛЕТА: МЕЛКОМОДУЛЬНОЕ РЕШЕНИЕ

```
1) int number = 1, next = 1, turn[1 : n] = ([n]0);
2) process CS[i = 1 to n] {
3)   while (true) {
4)     turn[i] = FA (number, 1); # протокол входа
5)     while (turn[i]! = next) skip;
6)     [критическая секция];
7)     next = next + 1; # протокол выхода
8)     [некритическая секция];
9)   }
10) }
```

Если же осутствует операция `FA`, но есть инструкция неделимого увеличения, то протокол входа может иметь вид

$$\texttt{turn}[\texttt{i}] = \texttt{number};\ < \texttt{number} = \texttt{number} + 1; > .$$

В последнем случае недостаток состоит в том, что хотя переменная `number` будет увеличиваться правильно, но процессы могут не получить уникальных номеров: например, каждый из процессов может выполнять первое присваивание примерно в одно и то же время и, таким образом, получить один номер. Таким образом, важно, чтобы *оба присваивания* выполнялись *в одном неделимом действии.*

Можно воспользоваться общим подходом создания критической секции. Пусть `CSenter` — протокол входа, `CSexit` — протокол выхода из критической секции, рассмотренные ранее. Тогда инструкцию `FA` можно заменить последовательностью

$$\texttt{CSenter};\ \texttt{turn}[\texttt{i}] = \texttt{number};\ \texttt{number} = \texttt{number} + 1;\ \texttt{CSexit};$$

Такой подход на практике удовлетворителен, если доступна инструкция типа “проверить–установить”; однако, при этом процессы получают номера *не всегда в том* порядке, в котором они пытаются это сделать, и теоретически *процесс может зациклиться навсегда.* Но с большой вероятностью каждый процесс получит свой номер, и большинство процессов получит номера по порядку.

§8 Справедливое планирование без использования специальных инструкций

Предыдущий алгоритм можно легко реализовать на машинах, имеющих неделимую операцию “извлечь и сложить”. Если такой операции нет, то приходится использовать еще один протокол критической секции, но это *может нарушить* справедливость алгоритма.

По алгоритму билета, посетитель получает уникальный номер и ждет, пока подойдет его очередь. В отличие от упомянутого алгоритма, в котором посетители *сверяют* свои номера *с общим счетчиком,* здесь посетители сверяются *друг с другом.*

Предположим, что $\texttt{turn}[\texttt{1} : \texttt{n}]$ — массив целых чисел, инициализированный нулями. Для входа в критическую секцию процесс

CS[i] сначала turn[i] присваивает значение на 1 больше, чем максимальное значение элементов массива turn, а затем ждет, пока значение turn[i] станет наименьшим среди ненулевых элементов массива turn.

Этот алгоритм называется *алгоритмом поликлиники.*

Соответствующий предикат имеет вид

$$\begin{aligned}&\texttt{CL}: \ \big(\forall \texttt{i}: \ 1 \le \texttt{i} \le \texttt{n}: \\ &(\texttt{CS}[\texttt{i}] \text{ в своей критической секции}) \Rightarrow (\texttt{turn}[\texttt{i}] > 0) \wedge \\ &(\texttt{turn}[\texttt{i}] > 0) \Rightarrow (\forall \texttt{j}: \ 1 \le \texttt{j} \le \texttt{n}, \ \texttt{j} != \texttt{i}: \\ &\ \ \texttt{turn}[\texttt{j}] == 0 \vee \texttt{turn}[\texttt{i}] < \texttt{turn}[\texttt{j}])\big)\end{aligned}$$

КРУПНОМОДУЛЬНОЕ РЕШЕНИЕ АЛГОРИТМА ПОЛИКЛИНИКИ

```
int turn[1 : n] = ([n]0);
## глобальный инвариант — предикат CL
process CS[i = 1 to n] {
  while (true) {
    < turn[i] = max(turn[1 : n]) + 1; >
    # поскольку это неделимая операция, то все получат
    # уникальные номера
    for [j = 1 to n st j != i]
    < await (turn[j] == 0 or turn[i] < turn[j]);
    # или 0 или наша очередь меньше —
    # значит, можно "крутнуть" цикл:
    # если цикл пройдет, то наша очередь меньше всех
    [критическая секция];
    turn[i] = 0;
    [некритическая секция];
  }
}
```

Первое неделимое действие обеспечивает уникальность всех ненулевых значений массива turn. Второе неделимое действие обес-

печивает условие входа в критическую секцию (протокол входа): вход в критическую секцию возможен лишь в случае, если наш номер меньше всех ненулевых.

К сожалению, написанный алгоритм не реализуем непосредственно на существующих ЭВМ, ибо для того, чтобы осуществить присваивания переменной `turn[i]` требуется найти максимальное из `n` значений разделяемого массива `turn`. Для реализации этой операции неделимым образом необходимо использовать *еще один* протокол критической секции (например, алгоритм разрыва узла); однако, такое усложнение является неэффективным.

Рассмотрим два процесса `CS1` и `CS2`. Пусть

```
turn1 = turn2 + 1;
while (turn2! = 0 and turn1 > turn2) skip;
```

— протокол входа для процесса `CS1`, а

```
turn2 = turn1 + 1;
while (turn1! = 0 and turn2 > turn1) skip;
```

— аналогичный протокол для процесса `CS2`. Каждый процесс присваивает значение своей переменной `turn` в соответствии с упомянутой программой (при `n = 2`). Однако, при таком решении оба процесса могут одновременно оказаться в своей критической секции: например, если процесс `CS1` определит, что `turn2` имеет значение 0, то он присвоит `turn1` единицу и начнет входить в критическую секцию. Но до присваивания единицы первым процессом второй процесс может сделать присвавание `turn2 = 0 + 1` и войти в свою критическую секцию. Таким образом, хотя `turn1 = 1` и `turn2 = 1` оба процесса оказываются в своих критических секциях одновременно, что недопустимо. В результате получается состояние "гонок" и нарушение основного принципа: в критической секции единовременно может находиться лишь один процесс.

Для того, чтобы не допустить состояния гонок каждый процесс может присвоить единицу переменным `turn1` и `turn2` соответственно в самом начале протокола входа, а именно, для процесса `CS1`

```
turn1 = 1; turn1 = turn2 + 1;
while (turn2! = 0 and turn1 > turn2) skip;
```

а для процесса CS2

turn2 = 1; turn2 = turn1 + 1;

while (turn1! = 0 and turn2 >= turn1) skip;

В результате если процесс CS1 к моменту проверки в CS2 *уже изменил* turn1 (так что turn1 положительно) и при этом присваивание turn2 = turn1 + 1 вторым процессом оказалось позже упомянутого изменения turn1, то второй процесс *задержится* на цикле while до тех пор, пока первый процесс не пройдет свою критическую секцию.

Замечание 1. Поскольку условия turn1 > turn2 и turn1 ≤ turn2 — взаимно исключающие и дополняющие друг друга, то обязательно произойдет нарушение лишь одного из них, и поэтому *точно один процесс* войдет в свою критическую секцию.

Замечание 2. Протоколы входа двух процессов *несимметричны.* Для того, чтобы им придать симметричный вид введем отношение сравнения двух чисел формулой

(a, b) > (c, d) == true ⇔ (a > c) ∨ (a == c ∧ b > d),

(a, b) > (c, d) == false — в остальных случаях.

Тогда условия turn1 > turn2 и turn1 ≤ turn2 могут быть представлены в виде

turn1 > turn2 ⇔ (turn1, 1) > (turn2, 2),

turn2 ≥ turn1 ⇔ (turn2, 2) > (turn1, 1).

Обобщим теперь алгоритм поликлиники для двух процессов: каждый из процессов показывает, что он собирается войти в свою критическую секцию, присваивая переменной turn число 1. Затем он найдет максимальное значение из всех turn[i] и прибавит к нему 1. Далее он запускает цикл for, и так же, как в крупномодульном решении, ждет своей очереди.

АЛГОРИТМ ПОЛИКЛИНИКИ: МЕЛКОМОДУЛЬНОЕ РЕШЕНИЕ

int turn[1 : n] = ([n]0); # инициализация

process CS[i = 1 to n] {

```
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1 : n]) + 1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and (turn[i], i) > (turn[j], j)) skip;
      # либо turn[i] > turn[j], либо i > j, если turn[i] = turn[j]
    [критическая секция];
    turn[i] = 0;
    [некритическая секция];
  }
}
```

Заметим, что максимальное значение в массиве определяется считыванием всех его элементов и отысканием наибольшего. Поскольку эти действия не являются неделимыми, то точный результат не гарантируется. Но если несколько процессов получают одно и то же "значение очереди" (т.е. номер), то происходит их упорядочивание по правилу, описанному в замечании 2.

§9 Барьерная синхронизация в цикле

Многие задачи решаются с помощью *итерационных алгоритмов,* которые последовательно вычисляют приближение к решению и завершаются после выполнения некоторого условия. В ряде случаев обрабатываются некоторые массивы, причем в основном к повторяются одни и те же действия.

Рассмотрим следующую схему параллельной реализации подобных итераций:

```
while (true) {
  co [i = 1 to n]
  [тело решаемой задачи с номером i]
  oc
}
```

Реализация по этой схеме неэффективна, ибо при каждой итерации создаются и уничтожаются процессы. Другая схема реализации состоит в том, что процессы создаются один раз, а итерации

программируются в каждом процессе. В конце каждой итерации происходит синхронизация всех процессов с помощью барьера: закончившие очередную итерацию процессы ждут момента, когда все процессы закончат эту итерацию, и только после этого каждый переходит к следующей итерации.

```
process Worke[i = 1 to n] {
  while (true) {
    [код решения задачи с номером i];
    [ожидание завершения всех задач];
  }
}
```

§10 Требования к барьеру

В итерациях, реализуемых `n` процессами с использованием барьера, можно рассмотреть *разделяемый* целочисленный счетчик `count` с нулевым начальным значением. Будем считать, что если процесс достигает барьера, то он увеличивает счетчик на 1 и останавливается в ожидании значения счетчика, равного `n`; как только счетчик достигнет значения `n`, все процессы могут продолжить работу. Эта идея иллюстрируется следующей схемой:

```
int count = 0;
process Worker[i = 1 to n] {
  while (true) {
    [программа реализации задачи i];
    < count = count + 1; >
    < await (count == n); >
  }
}
```

Если имеется инструкция `FA` (“извлечь и сложить”, то барьер можно реализовать по схеме:

```
FA(count, 1);
while (count! = n) skip;
```

Однако, приведенная схема реализации все-таки не соответствует поставленной задаче, ибо значение `count` должно быть нулем в начале каждой интерации, т.е. переменной `count` нужно присваивать 0 после того, как все процессы подойдут к барьеру, но это обстоятельство не отражено в упомянутой схеме.

Можно было бы эту проблему решить, введя два счетчика, один из которых уменьшается, а другой возрастает, причем роли счетчиков меняются при переходе к следующей итерации. Однако, это ведет к новым трудностям:

1) увеличивать и уменьшать нужно неделимым образом (значит, кроме `FA` нужно иметь неделимую операцию "извлечь и вычесть" — назовем ее `FM`);

2) когда процесс приостанавливается, он непрерывно проверяет значение переменной `count`; поскольку число ожидающих у барьера процессов доходит до $\mathtt{n}-1$, то при больших значениях `n` это может привести к *конфликтам при обращении к памяти* (за значением `count`). Если же у каждого процесса имеется быстрая память (*кэш*), то возникает проблема поддержания *когерентности* кэшей (под когерентностью кэшей подразумевается тождественность их содержания).

Вывод: рассмотренный здесь способ следует использовать лишь при *малом* `n`.

§11 Управляющие процессы

Пусть имеется массив $\mathtt{arrive}[1:\mathtt{n}]$ с нулевыми начальными значениями. Заменим операцию увеличения счетчика `count` присваиванием $\mathtt{arrive}[\mathtt{i}]=1$. Глобальным инвариантом служит предикат

$$\mathtt{count} == (\mathtt{arrive}[1] + \ldots + \mathtt{arrive}[\mathtt{n}]).$$

Если элементы массива `arrive` хранятся в разных строках кэш-памяти, то конфликтов в памяти при обращении к ним не будет, но вместо неделимого действия

$$< \mathtt{await}\ (\mathtt{count} == \mathtt{n}); >$$

потребуется неделимое действие

$$< \mathtt{await}\big((\mathtt{arrive}[1] + \ldots + \mathtt{arrive}[\mathtt{n}]) == \mathtt{n}\big); >,$$

которое опять-таки приведет к конфликтам в памяти, ибо сумму вычисляет каждый процесс. Таким образом, это решение неэффективно.

Обе проблемы (конфликтов обращения к памяти и обнуления массива) могут быть решены использованием управляющего процесса.

Пусть имеется набор `n` процессов `Worker[i]`, `i` = 1, ..., `n`, и один процесс `Coordinator`. Каждый процесс `Worker[i]` ждет, пока элемент `continue[i]` примет значение 1, где `continue[1 : n]` — дополнительный массив с нулевыми начальными значениями: сначала (по прибытию к барьеру) процесс `Worker[i]` делает присваивание `arrive[i]` = 1, а затем ожидает момента, когда окажется, что `continue[i]` == 1, и лишь после этого он продолжает работу. Приведем соответствующий фрагмент программы для процесса `Worker[i]`:

```
arrive[i] = 1;
< await (continue[i] == 1); >
```

Что касается процесса `Coordinator`, то он сначала *ждет*, пока все значения элементов массива `arrive` станут равными 1, а затем — *присваивает* всем элементам `continue` значения 1, тем самым пропуская через барьер процессы `Worker[i]`. Соответствующий фрагмент программы для процесса `Coordinator` таков:

```
for [i = 1 to n] < await (arrive[i] == 1); >
for [i = 1 to n] continue[i] = 1;
```

Операторы `await` можно реализовать в виде циклов `while ... skip`, т.к. в каждом случае ссылка происходит лишь на одну разделяемую переменную.

Заметим, что `Coordinator` может проверять установку элементов `arrive[i]` в единицу *в любом порядке*; при этом конфликтов обращения к памяти не будет, поскольку процессы ожидают изменения *различных* переменных, каждая из которых хранится в своей строке кэш-памяти.

Фактически переменные `arrive[i]`, `continue[i]` являются примерами *флагов,* которые "поднимаются" и "сбрасываются" процессами, что соответствует их установке в 1 и в 0 соответственно. В

рассматриваемой ситуации эти флаги называются “флагами синхронизации”.

Правила работы с такими флагами состоят в следующем:

1) флаг синхронизации сбрасывается *лишь* процессом, *ожидающим его установки*;

2) флаг можно устанавливать *только* в том случае, если он уже сброшен (другим процессом), т.к. в противном случае возможна взаимная блокировка (процесс, которому выставляется флаг, не успевает его сбросить и не обрабатывает соответствующую часть своей работы).

В нашем случае это означает, что флаг `continue[i]` должен сбрасываться процессом `Worker[i]`, а флаг `arrive[i]` должен сбрасываться процессом `Coordinator`. Последний это может делать в цикле.

БАРЬЕРНАЯ СИНХРОНИЗАЦИЯ С УПРАВЛЯЮЩИМ ПРОЦЕССОМ

```
int arrive[1 : n] = ([n]0), continue[1 : n] = ([n]0);
process Worker[i = 1 to n] {
  while (true) {
    [код решения задачи i];
    arrive[i] = 1;
    < await (continue[i] == 1); >
    continue[i] = 0
  }
}
process Coordinator {
  while (true) {
    for [i = 1 to n] {
      < await (arrive[i] == 1); >
      arrive[i] = 0;
    }
    for [i = 1 to n] continue[i] = 1;
```

```
  }
}
```

Положительное качество приведенной программы: *конфликты* при обращении к памяти *исключаются.*

Отрицательные ее качества:

1) требуется *дополнительный процесс* — `Coordinator` (и, возможно, дополнительный процессор);

2) время выполнения каждой итерации процессом `Coordinator` *пропорционально числу процессов* `Worker`[`i`], поскольку `Coordinator` проверяет флаги `arrive`[`i`] по-очереди; заметим, что на самом деле в прикладных программах работы, выполняемые `Worker`[`i`], часто эквивалентны по объему, и потому все процессы `Worker` приходят к барьеру *почти одновременно,* так что все флаги `arrive`[`i`] будут установлены в `1` практически одновременно.

Однако, обе трудности легко преодолеваются построением *бинарного дерева процессов* `Worker` (если количество `n` процессов меньше числа $1 + 2 + \ldots + 2^{k-1} = 2^k - 1$, то некоторые из них можно сделать фиктивными).

Сигнал о том, что процесс подошел к барьеру распространяется от "листьев" к "корню", т.е. вверх (см. схему), а сигнал о разрешении продолжения работы — в обратном направлении.

Рис. 2.

БАРЬЕРНАЯ СИНХРОНИЗАЦИЯ С ПОМОЩЬЮ БИНАРНОГО ДЕРЕВА (БАРЬЕР С ОБЪЕДИНЯЮЩИМ ДЕРЕВОМ)

```
ROOT : < await (arrive[left] == 1); >
     arrive[left] = 0;
     < await (arrive[right] == 1); >
     arrive[right] = 0;
     continue[left = 1; continue[right] = 1;
INTERMEDIATE :
     < await (arrive[left] == 1); >
     arrive[left] = 0;
     < await (arrive[right] == 1); >
     arrive[right] = 0;
     arrive[I] = 1;
     < await (contineu[I] == 1); >
     continue[I] = 0;
     continue[left] = 1; continue[right] = 1;
LIST :  arrive[L] = 1;
     < await (continue[L] == 1); >
     continue[L] = 0;
```

Замечание. Каждый из процессов (узлов дерева) может выполнять также предписанную для него работу (здесь она не описана).

Легко видеть, что здесь используется *столько же* переменных, сколько и в *централизованной версии барьера,* но высота дерева пропорциональна $\log_2 n$. Можно сделать *программу еще эффективнее,* если *корневой узел* будет посылать *единственное* сообщение о возможности продолжения работы всем остальным узлам (например, флаг `continue`), а для сбрасывания флага можно провести *двойную буферизацию:* иметь *два флага* и переключаться между ними или изменять смысл флага на четных и нечетных итерациях (0 на 1 и 1 на 0).

§12 Построение симметричных барьеров

Недостаток барьера с объединяющим деревом в том, что процессы имеют неодинаковую загрузку: 1) средние узлы загружены больше, чем листья и корень; 2) кроме того, корень должен ждать, пока сигналы прибытия пройдут через все дерево.

Симметричный барьер для n процессов строится из *пар простых двухпроцессорных барьеров.* Пусть каждый процесс по достижении барьера устанавливает собственный флаг; тогда симметричный барьер для них выглядит так:

```
# Фрагмент для барьера в W[i]
< await (arrive[i] == 0); >
# эта строка необходима, чтобы W[i] не установил флаг до того,
# как W[j] использовал эту становку флага на предыдущем
# использовании барьера
arrive[i] = 1;
< await (arrive[j] == 1); >
arrive[j] = 0;


# Фрагмент для барьера в W[j]
< await (arrive[j] == 0); >
# см. предыдущий комментарий
arrive[] = 1;
< await (arrive[i] == 1); >
arrive[i] = 0;
```

Возникает вопрос: как связать эти двухпроцессорные барьеры для того, чтобы получить барьер для n процессов.

Один из способов объединения (называемый *барьер-бабочка*) представлен на следующем рисунке.

Рис. 3. Барьер-бабочка для 8 процессов.

Замечание. Ошибка в синхронизации может возникнуть в следующих обстоятельствах. Предположим, что процесс 2 работает значительно медленнее остальных. Пусть процесс 1 подошел к барьеру: `arrive[1] = 1`, а процесс 2 задерживается так, что `arrive[2] = 0`; пусть еще процессы 3 и 4 подошли к барьеру: теперь процесс 3 будет синхронизироваться с процессом 1; в результате процесс 3 переходит на уровень 3, хотя флаг `arrive[1] = 1` был установлен для процесса 2 (который задерживается). Теперь может отработать третий уровень так, как будто все процессы подошли к барьеру (однако, процесс 2 еще не подошел к барьеру): в результате некоторые процессы пройдут барьер раньше, чем нужно, а другие будут бесконечно ждать у барьера.

Для того, чтобы избежать такой ошибки, можно использовать целочисленные флаги как счетчики, которые увеличивают свой номер в соответствии с прохождением очередного уровня барьера. Будем считать,что начальное значение флага `arrive[i]` равно 0, а при переходе на новый уровень барьер `arrive[i]` увеличивается на 1. Предположим, что рабочий процесс с i-м номером определяет номер j своего партнера на достигнутом уровне и ожидает, пока `arrive[j]` станет таким же, как `arrive[i]`. Таким образом, получаем следующую программную схему.

БАРЬЕР ДЛЯ РАБОЧЕГО ПРОЦЕССА С НОМЕРОМ i

```
for [s = 1 to num_stages] { # "до числа уровней барьера"
  arrive[i] = arrive[i] + 1;
```

[определение номера партнера j на уровне s];

`while` (`arrive`[`j`] < `arrive`[`i`]) `skip`;

} # этим определяется, что процесс j зашел так же далеко, как i

Здесь не требуется:

1) ожидать переустановки собственного флага;

2) переустанавливать флаг соседа.

Этой схеме свойственнен обычный недостаток программ с возрастающими счетчиками: теоретически возможно их переполнение (на практике это бывает редко).

§13 Распараллеливание префиксных вычислений

Пусть дан числовой массив `a`[`n`]; требуется вычислить сумму первых `i` элементов массива и занести ее в элемент `s`[`i`] массива `s`[`n`], `i` = 0, 1, . . . , `n`−1. Подобные вычисления называются *префиксными*. Решение этой задачи может быть реализовано программой

`s`[0] = `a`[0];

`for` [`i` = 1 `to` `n` − 1]

  `s`[`i`] = `s`[`i` − 1] + `a`[`i`];

Наша задача — дать параллельную версию таких вычислений. Если бы требовалось вычислить лишь всех элементов, то можно было бы использовать алгоритм сдваивания; для его реализации при $\mathtt{n} = 2^{\mathtt{k}}$ потребовалось бы `k` параллельных сложений.

Подобный подход можно использовать и для вычисления всех префиксов.

Нужный алгоритм имеет следующие этапы:

1) сначала производится присваивание `s`[`i`] = `a`[`i`];

2) далее складываются `s`[`i` − 1] и `s`[`i`] для $i \geq 1$;

3) далее складываются `s`[`i` − 2] и `s`[`i`] для $i \geq 2$;

4) далее складываются `s`[`i` − 4] и `s`[`i`] для $i \geq 4$;

и т.д.

При $\mathtt{n} = 2^{\mathtt{k}}$ описываемый алгоритм реализуется за `k` тактов синхронной параллельной системы. Следующий рисунок иллюстрирует работу алгоритма в случае, когда `n` = 8, а слагаемыми являются натуральные числа: `s`[`i`] = `i` + 1, `i` = 0, 1, . . . , 7.

Рис. 4. Иллюстрация распараллеливания префиксных вычислений

Пусть `i` — идентификатор процесса, `barrier(i)` — точка синхронизации, `old[n]` — массив, где сохраняется старое значение.

```
int a[n], s[n], old[n]; # предполагается, что массив a[n]
                        #  инициализирован
process Sum[i = 0 to n − 1] {
  int d = 1;
  s[i] = a[i];
  barrier(i); # I барьер
  while (d < n) {
    old[i] = s[i]; # сохранение старых значений
    barrier(i); # II барьер
    if ((i − d) >= 0)
      s[i] = old[i − d] + s[i];
    barrier(i); # III барьер
    d = d + d; # удваивание расстояния
  }
}
```

Замечание. Барьеры здесь необходимы для устранения взаимного влияния:

I барьер: элементы `s[i]` должны быть проинициализированы до начала выполнения программы;

II барьер: старые значения используются, и потому должны быть сохранены;

III барьер: перед переходом на следующий уровень нужно провести все необходимые вычисления.

Замечания. 1) Алгоритм может быть изменен для любой ассоциативной бинарной операции (коммутативность не обязательна).

2) Программа может быть адаптирована к числу процессов меньше `n`; при этом каждому процессу придется поручить больше работы.

§14 Операции со связанными списками

Рассмотрим задачу об отыскании *конца* связанного списка с `n` элементами, у которого связи хранятся в массиве `link`[`n`], а данные списка хранятся в массиве `data`[`n`].

Пусть на начало списка указывает переменная `head`, а конец списка характеризуется указателем `null` (см. рис. 5).

Рис. 5. Иллюстрация связей в списке

Задача состоит в организации систематической перестройки указателей так, чтобы *все* они указывали в `null`.

Последовательное решение этой задачи требует `n` тактов, а при параллельном решении потребуется $\lceil \log_2 n \rceil$ тактов (здесь $\lceil a \rceil$ означает наименьшее целое `k`, для которого $k \geq a$).

Пусть `end`[`n`] — разделяемый массив типа `integer`. Для простоты будем считать, что `link`[`i`] — также целочисленный массив, что пустой указатель `null` интерпретируется числом -1, и при $i < n-1$ элемент `link`[`i`] содержит индекс следующего элемента списка, т.е. число $i + 1$; кроме того, пусть `link`[`n` $-$ `1`] == `null`.

Первоначально каждый процесс присваивает элементу `end`[`i`] значение `link`[`i`] (т.е. индекс следующего элемента списка). Процессы выполняют несколько этапов, на каждом из которых элементу `end`[`i`] присваивается значение `end`[`end`[`i`]], если такое вычисление имеет смысл и дает индекс элемента, имеющегося в списке `data`[`n`], или `null` — в противном случае.

Итак, первоначально end[i] указывает на следующие элементы списка, на *первом этапе* — на элементы, находящиеся на расстоянии *в две связи*, на втором этапе — *в 4 связи* и т.д.(см. рис. 6).

Рис. 6. Схема изменений связей

Эта схема реализуется следующей программой.

ПОИСК КОНЦА ПОСЛЕДОВАТЕЛЬНОГО СПИСКА

```
int link[n], end[n];
process Find[i = 0 to n − 1] {
  int new, d = 1;
  end[i] = link[i]; # инициализация элементов end
  barrier(i); # первоначальный барьер
  while (d < n) {
    new = null; # проверяем, нужно ли обновление end[i]
    if (end[i]! = null and end[end[i]]! = null)
      new = end[end[i]]; # присваивание для new
    barrier(i); # барьерная синхронизация: нельзя обновлять
                # end[i] раньше времени: нужно ждать
                # другие процессы
    if (new! = null) # обновить end[i]
```

```
    end[i] = new;
  barrier(i); #после полного обновления end[i]
              # продолжаем работу
  d = d + d; # удваиваем расстояние
 }
}
```

§15 Итерации Якоби

Многие вычисления связаны с сеткой, накладываемой на некоторую область пространства, причем вычисления представляют собой итерационный процесс, проводимый до момента выполнения того или иного условия завершения. Примерами подобных вычислений являются

— обработка изображений, где с сеткой ассоциируются пиксели, имеющие различные градации яркости, а искомыми являются координаты соседних пикселей с похожими градациями яркостей,

— решение уравнений в частных производных методом сеток, где значения в граничных точках сетки определяются граничными условиями исходной задачи, а искомыми являются приближенные значения решения во внутренних узлах сетки, и т.д.

Вычисления идут по следующей схеме:

```
[инициализация матрицы];
while (условие завершения задачи не выполнено) {
[для каждой точки вычислить новое значение];
[проверить условия завершения];
}
```

Характерные черты методов этого рода состоят в том, что на каждой итерации вычисляются новые значения в точках, и эти вычисления можно проводить параллельно с использованием барьеров.

В качестве конкретного примера рассмотрим решение уравнения Лапласа с условиями Дирихле на квадрате. Пусть `grid[n+1, n+1]` — матрица приближенных значений искомого решения (заметим, что в соответствии с принятыми обозначениями элементами этой

матрицы служат элементы grid[i, j], i, j = 0, 1, . . . , n). Для вычисления решения применяется метод Якоби, состоящий в реализации итераций вида

$$\texttt{newgrid}[\texttt{i},\texttt{j}] = (\texttt{grid}[\texttt{i}-1,\texttt{j}] + \texttt{grid}[\texttt{i}+1,\texttt{j}]+$$

$$+\texttt{grid}[\texttt{i},\texttt{j}-1] + \texttt{grid}[\texttt{i},\texttt{j}+1])/4$$

с последующей пересылкой

$$\texttt{grid}[\texttt{i},\texttt{j}] = \texttt{newgrid}[\texttt{i},\texttt{j}],$$

где i, j = 1, 2, . . . , n − 1. Условием окончания процесса является условие |grid[i, j] − newgrid[i, j]| < EPSILON для всех внутренних узлов квадрата, т.е. для упомянутых выше значений i, j = 1, 2, . . . , n − 1; здесь EPSILON — заданное число.

РЕШЕНИЕ ЗАДАЧИ ДИРИХЛЕ ДЛЯ УРАВНЕНИЯ ЛАПЛАСА МЕТОДОМ ЯКОБИ

```
real grid[n + 1, n + 1], newgrid[n + 1, n + 1];
bool converged = true;
[инициализация матрицы grid[n + 1, n + 1]];
process Grid[i = 1 to n − 1, j = 1 to n − 1] {
 while (¬converged) {
  newgrid[i, j] = (grid[i − 1, j] + grid[i + 1, j]+
                + grid[i, j − 1] + grid[i, j + 1])/4;
  if (abs(newgrid[i, j] − grid[i, j]) < EPSILON)
   converged = converged ∧ TRUE
  else
   converged = FALSE;
  barier(i, j);
  if (¬converged) grid[i, j] = newgrid[i, j];
  barier(i, j);
 }
}
```

Замечание 1. Приведенная программа правильна, но допускает улучшение: будет меньше пересылок, если сначала обновить `newgrid` (как это здесь и было сделано), а затем обновить `grid` тем же приемом, и эту последовательность постоянно повторять (на нечетных шагах обновлять `newgrid`, а на четных — `grid`).

Замечание 2. Быстрее описанного метода сходится метод верхней релаксации; однако, здесь на вопросах ускорения сходимости останавливаться не будем.

§16 Замечание о синхронном выполнении

До сих пор допускалась возможность, что вычислительные модули (ВМ) рассматриваемой вычислительной системы (ВС) имеют различные типы и работают различными скоростями (система гетерогенная); таким образом, процессы могли идти с разной скоростью.

В случае, когда все ВМ работают синхронно, выполняют одну и ту же последовательность команд, а в каждый момент времени — одну и ту же операцию, то многие приведенные ранее программы упрощаются. Например, программа вычисления всех частичных сумм массива (см. §13) принимает вид

```
int a[n], s[n];
process Sum[i = 0 to n − 1] {
  s[i] = a[i]; # инициализация элементов массива s
  while (d < n) {
    if ((i − d) >= 0) # одновление s
      s[i] = old[i − d] + s[i];
    d = d + d; # удваивание расстояния
  }
}
```

Здесь программные барьеры не нужны, ибо процессы выполняют одни и те же команды, точно так же не нужны дополнительные переменные. Параллельные инструкции в этой ситуации становятся неделимыми.

Рассматриваемые ВС называются синхронными мультипроцессорными (SIMD-машинами); они привлекательны по следующим

причинам:

1) их легко конструировать;

2) в ряде случаев легко поддерживать неделимость инструкций;

3) они удобны для решения больших задач.

Недостатки таких вычислительных систем:

1) SIMD-машины в определенном смысле — специализированные (в каждый момент времени рашется одна задача);

2) трудно обеспечить высокую загрузку (в частности, в приведенном примере все ме́ньшее число процессоров обновляет значения `sum[i]`, а остальные не делают полезной работы); часто загрузка таких ВС не превышает 20%.

§17 Умножение матриц

При перемножении двух квадратных матриц используем прием, называемый *вычислениями с портфелем задач*. Обычно такой прием применяют по отношению к совокупности независимых задач или независимых подзадач, возникающих в исходной задаче. Освободившийся вычислительный модуль параллельной вычислительной системы "подхватывает" задачу из "портфеля" и начинает ее решение. Благодаря такому подходу, нагрузка хорошо распределяется между вычислительными модулями (получается *хорошая балансировка* загрузки).

Пусть матрицу `a[n,n]` следует умножить на матрицу `b[n,n]`, а результат получить в массиве `c[n,n]`.

Будем считать, что имеется `n` процессоров, и что матрицы `a` и `b` *хранятся по столбцам* (в частности, это предположение обычно справедливо при использовании языка Фортран). В данном случае схема перемножения матриц с помощью портфеля задач принимает вид:

```
int nextRow = 0; # портфель задач
double a[n,n], b[n,n], c[n,n];
process Worker[w = 1 toP] {
  int row;
  double sum; # используется для промежуточных вычислений
```

```
  while(true) {
    [получить задачу из портфеля];
    < row = nextRow; nextRow + +; >
    if (row >= n)
      break;
    [вычислить скалярные произведения строки a[row, ∗] на
     столбцы b[∗, ∗] для получения строки c[row, ∗]];
  }
}
```

Фрагменты программы, относящиеся к вычислению скалярных произведений строк на столбцы, очевидны, и потому не приводятся.

§18 Адаптивная квадратура с портфелем задач

Задача состоит в приближенном адаптивном вычислении интеграла

$$\int_a^b f(x)\,dx$$

методом трапеций с априори заданной точностью $\varepsilon > 0$; здесь a, b, ε — заданные числа, $f(x)$ — заданная функция, непрерывная на отрезке $[a, b]$.

Метод вычислений состоит в следующем. Пусть $[\alpha, \beta]$ — некоторый отрезок, лежащий в $[a, b]$, а $\gamma = (\alpha+\beta)/2$. Если $(f(\alpha)+f(\beta))(\beta-\alpha)/2$ (площадь прямолинейной трапеции над отрезком $[\alpha, \beta]$) отличается от $(f(\alpha) + f(\gamma))(\gamma - \alpha)/2 + (f(\gamma) + f(\beta))(\beta - \gamma)/2$ (т.е. от суммы площадей трапеций над отрезками $[\alpha, \gamma]$ и $[\gamma, \beta]$) менее, чем на $\varepsilon \cdot (\beta - \alpha)$, то вычисления заканчиваются; в противном случае для каждого из отрезков $[\alpha, \gamma]$ и $[\gamma, \beta]$ проводятся вычисления по той же схеме, что и для $[\alpha, \beta]$.

Таким образом, в нашем случае согласно этому методу сначала вычисляется середина m отрезка $[a, b]$, а затем приближения на отрезках $[a, b]$, $[a, m]$ и $[m, b]$; если приближение на отрезке $[a, b]$ отличается от суммы приближений на отрезках $[a, m]$ и $[m, b]$ менее,

чем на $\varepsilon \cdot (b-a)$, то вычисления прекращаются; в противном случае для каждого из отрезков $[a, m]$ и $[m, b]$ проводятся вычисления по той же схеме, что и для $[a, b]$ (см. рис. 7).

Рис. 7.

ПРОГРАММА АДАПТИВНОЙ КВАДРАТУРЫ С ИСПОЛЬЗОВАНИЕМ ПОРТФЕЛЯ ЗАДАЧ

```
type task (double left, right, fleft, fright, lrarea);
queue bag(task); # портфель задач
int size; # число задач в портфеле
int idle = 0; # число простаивающих процессов
double total = 0.0; # общая площадь
[вычислить аппроксимацию площади из отрезка от a до b];
[добавить в портфель задачу (a, b, f(a), f(b), area)];
size = 1;
process Worker[w = 1 to n] {1
 double left, right, fleft, fright, lrarea;
 double mid, fmid, larea, rarea;
 while (true) {2
  # проверить завершение
  < idle++;
    if (idle == n&&size == 0) break; > # неделимая операция
  # получить задачу из портфеля
  < await (size > 0)
    [забрать задачу из портфеля];
    size--; idle--; >
```

```
    # уменьшены число задач в портфеле и число
    # простаивающих процессов
  mid = (left + right)/2;
  fmid = f(fmid);
  larea = (flaft + fmid) * (mid − left)/2;
  rarea = (fmid + fright) * (right − mid)/2;
  if (abs((larea + larea) − lrarea) > EPSILON * (right − left)) {3
    < поместить в портфель (left, mid, fleft, fmid, larea),
    < поместить в портфель (mid, right, fmid, fright, rarea),
    size = size + 2; >
  }3
  else
    < total = total + lrarea; >
 }2
 if (w == 1) # рабочий процесс №1 выводит результат
  printf("общая площадь ", total);
}1
```

Задачи и упражнения

1. Пусть имеются два последовательных связанных списка. Требуется написать алгоритм (программу), параллельный по данным, который ставит в соответствие элементы массивов с одинаковыми номерами: в результате элементы списков должны указывать друг на друга. Если один список длиннее другого, то лишние элементы более длинного списка должны содержать пустые указатели. Результаты сохранить в дополнительных массивах.

2. Пусть последовательный связанный список *согласован с возрастанием полей* данных. Какую *оценку* работы имеет последовательный алгоритм вставки нового элемента в такой список (с сохранением возрастания полей)? *Требуется* построить параллельный алгоритм вставки с улучшенной оценкой работы.

3. В *обработке изображений* возникает *задача выделения* свя-

заных областей с *одинаковой интенсивностью пикселей* (считая, что соседями пикселя P являются левый, правый, верхний и нижний:

Рис. 8.

Задача: найти все такие области и присвоить каждому пикселю уникальную (в глобальном смысле) метку (номер) так, чтобы номера в пределах каждой области представляли собой отрезок натурального ряда. Написать программу, считая, что область имеет $n \times n$ пикселей.

4. Программу для предыдущей задачи преобразовать в параллельную так, чтобы время ее решения имело порядок $O(\ln n)$.

Глава 3 Синхронизация с помощью семафоров

§1 Некоторые определения

Семафор — это разделяемая переменная, имеющая неотрицательные целочисленные значения и обрабатываемая с помощью двух неделимых операций `P` и `V`:

$\texttt{P(s)} : < \texttt{await (s} > \texttt{0) s} = \texttt{s} - \texttt{1;} >$

$\texttt{V(s)} : < \texttt{s} = \texttt{s} + \texttt{1;} >$

Операция `P(s)` в *неделимом действии* выполняет следующее: 1) проверяет условие $\texttt{s} > \texttt{0}$; 2) приостанавливает процесс при $\texttt{s} = \texttt{0}$ до увеличения значения `s`; 3) если `s` оказывается положительным, то в том же действии эта операция уменьшает `s` на единицу и пропускает процесс.

Операция `V(s)` увеличивает `s` на единицу в *неделимом действии.*

Обычный семафор может принимать любые значения, а двоичный семафор — только значения 0 и 1.

§2 Взаимное исключение. Барьеры

Задача критической секции ставится так: в каждый момент времени лишь один из n процессов *выполняет критическую секцию*, в которой он имеет доступ к разделяемому ресурсу (именно поэтому критические секции должны выполняться со взаимным исключением). Затем процесс выполняет *некритическую секцию,* в которой он работает с локальными объектами. При этом схема выполняемой программы такова:

```
sum s = 1;
process CS[j = 1 to n] {
  while (true) {
    P(s);
    [критическая секция];
    V(s);
    [некритическая секция];
  }
}
```

В барьерной синхронизации семафор используется как *флаг синхронизации.* Процесс *устанавливает* флаг, выполняя *операцию* `V`; при выполнении операции `P` он *ждет установки флага*, а после его установки *прекращает* ожидание и *сбрасывает* флаг.

Сигнализирующий семафор — это семафор с *нулевым значением.* Процесс сигнализирует о событии, устанавливая флаг с помощью операции `V(s)`, а другие процессы ожидают этого сигнала, выполняя операцию `P(s)`. При двухпроцессорном барьере имеется два существенных события: 1) *прибытие* к барьеру *первого процесса;* 2) *прибытие* к барьеру *второго процесса;* поэтому требуется два флага для сигнализации об этих событиях. Воспользуемся флагами `arrive1` и `arrive2`.

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
  ...
  V(arrive1); # прибытие I процесса
  P(arrive2); # ожидание прибытия II процесса
  ...
}
process Worker2 {
  ...
  V(arrive2); # прибытие II процесса
  P(arrive1); # ожидание прибытия I процесса
```

```
  ...
}
```

В случае n процессов можно реализовать *барьер–бабочку* или *барьер с распространением* (см. предыдущую главу). В этих случаях потребуется *массив семафоров* `arrive[n]`. При наличии управляющего процесса можно обойтись одинм (простым) семафором.

§3 Разделенные двоичные семафоры

Предположим, имеется несколько производителей и несколько потребителей, которые общаются через *разделяемый буфер*.

Производители пользуются этим буфером, помещая туда результаты своей деятельности ("товар") операцией `deposite` — *поместить,* а пользователи забирают этот "товар" с помощью операции `fetch` — *извлечь*.

Предполагается, что *буфер заполняется в одном акте* процедурой `deposite`, а опустошение его также *проводится за один акт* процедурой `fetch`. Для того, чтобы *заполнение* было возможно, необходимо извлечение, т.е. заполнение и извлечение должны производиться по-очереди.

Пусть `empty` и `full` — разделяемые двоичные семафоры.

$$\texttt{empty} = \begin{cases} 1, & \text{если буфер опустошен,} \\ 0, & \text{если буфер заполнен;} \end{cases}$$

$$\texttt{full} = \begin{cases} 0, & \text{если буфер пуст,} \\ 1, & \text{если он заполнен.} \end{cases}$$

СХЕМА ПРОГРАММЫ "ПРОИЗВОДИТЕЛИ И ПОТРЕБИТЕЛИ"

```
typeT buf; # буфер некоторого типа T
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
  while (true) {
    ... # произвести data
```

```
   P(empty); # и поместить результат в буфер
   buf = data;
   V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {

   P(full); # извлечь результат
   result = buf;
   V(empty);
   ... # и его использовать
  }
}
```

§4 Задача "об обедающих философах"

Задача "об обедающих философах" иллюстрирует борьбу процессов за разделяемые ресурсы. Она формулируется следующим образом.

Пять философов сидят возле круглого стола. Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола стоит блюдо спагетти. Спагетти длинные и запутанные, с ними тяжело управиться, и потому каждому философу во время еды нужно использовать две вилки. По недосмотру положили всего пять вилок, причем каждая лежит между философами. Ввиду этого они договорились, что будут есть двумя вилками, а те, у кого не будет двух вилок, подождут, пока насытятся соседи. Задача состоит в том, чтобы написать программу, моделирующую поведение философов; в ней следует предусмотреть невозможность тупиковой ситуации, при которой каждый философ успел взять одну вилку, но ни один из них не успел взять вторую.

При компьютерной реализации можно считать, что периоды приемов пищи и периоды раздумий случайны: для этого можно использовать датчик случайных чисел.

Рассматриваемые здесь процессы можно представить так:

```
process Philosopher[i = 1 to 4] {
  while (true) {
    [поразмыслить];
    [взять вилки];
    [поесть];
    [отдать вилки];
  }
}
```

Каждая вилка *может идентифицироваться с критической секцией:* ее может взять только один философ. Поэтому ее можно представить семафором `fork[i]`, `i = 0`, ..., `4`, которые в начальный момент инициализируются значением 1 (*все вилки доступны*).

Однако, каждый философ может взять сначала левую вилку: тогда все вилки будут разобраны, но ни один философ не сможет приступить к еде — *тупиковая ситуация.*

Для того, чтобы такая ситуация не могла возникнуть, условимся, что все философы, кроме одного, сначала берут левую вилку, а затем — правую; но один философ, например, четвертый, всегда берет сначала правую вилку, а потом левую; таким образом, тупикodая ситуация *не может* возникнуть.

СХЕМА РЕШЕНИЯ ЗАДАЧИ “ОБ ОБЕДАЮЩИХ ФИЛОСОФАХ”

```
sem fork[5] = {1, 1, 1, 1, 1};
process Philosopher[i = 0 to 3] {
  while (true) {
    P(fork[i]); # взял левую вилку
    P(fork[i + 1]); # взял правую вилку
    [ест (датчик с.ч.)];
    V(fork[i]); # освободил левую вилку
    V(fork[i + 1]); # освободил правую вилку
```

```
   [размышляет (датчик с.ч.)];
  }
}
process Philosopher[4] {
  while (true) {
   P(fork[0]); P(fork[4]);
   # взял правую вилку, потом — левую
   [ест (датчик с.ч.)];
   V(fork[0]); V(fork[4]);
   # освободил правую, потом — левую вилку
   [размышляет (датчик с.ч.)];
  }
}
```

§5 Задача “о читателях и писателях” как задача исключения

В базе данных (БД) имеется лишь *два типа процессов:* читатели и писатели. Читатели просматривают записи (транзакции чтения), а писатели просматривают и изменяют записи (транзакции чтения и записи). Исходное состояние базы данных считается *непротиворечивым* (осмысленным). Каждая транзакция переводит ее из одного непротиворечивого состояния в другое. Для *отсутствия взаимного вмешательства* процесс-писатель должен иметь исключительный доступ к базе данных. Если ни один процесс-писатель не обращается к базе данных, то ее могут читать любое число процессов-читателей.

Пусть `rw` — семафор взаимного исключения с начальным значением 1. Читатели должны блокировать группу писателей, но при этом лишь *первый* из них устанавливает блокировку `P(rw)`; остальные читатели могут сразу обращаться к базе данных. Последний активный процесс-читатель должен снимать блокировку. Однако, сначала приведем решение с *дополнительным* ограничением: не только каждый писатель, но и любой читатель имеет исключительный доступ к БД.

СХЕМА РЕШЕНИЯ ЗАДАЧИ С ДОПОЛНИТЕЛЬНЫМ ОГРАНИЧЕНИЕМ

```
sem rw = 1;
process Reader[i = 1 to M] {
  while (true) {
    ...
    P(rw); # захват блокировки исключительного доступа
    [читать базу данных];
    V(rw); # освобождение блокировки
  }
}
process Write[i = 1 to K] {
  while (true) {
    ...
    P(rw); # захват блокировки
    [записать в базу данных];
    V(rw); # освобождение блокировки
  }
}
```

Следующая схема решения предусматривает блокировку группы писателей со стороны читателей, однако, лишь *первый* из них устанавливает блокировку P(rw); остальные читатели могут сразу обращаться к базе данных (последний из них снимает блокировку).

СХЕМА РЕШЕНИЯ ЗАДАЧИ "О ЧИТАТЕЛЯХ И ПИСАТЕЛЯХ" БЕЗ ДОПОЛНИТЕЛЬНЫХ ОГРАНИЧЕНИЙ

```
int nr = 0 # число активных читателей
sem rw = 1; # блокировка исключения
process Reader[i = 1 to M] {
```

```
  while (true) {
    ...
    < nr = nr + 1;
    if (nr == 1) P(rw); # блокировку получает первый
    >
    [читать базу данных]
    < nr = nr − 1;
    if (nr == 0) V(rw); #снимает блокировку последний
    >
  }
}
process Writer[i = 1 to K] {
  while (true) {
    ...
    P(rw);
    [ записать в базу данных];
    V(rw);
  }
}
```

Теперь для организации неделимости используем семафор, который обозначим s. Тогда можно предложить следующую схему решения рассматриваемой задачи.

СХЕМА РЕШЕНИЯ ЗАДАЧИ “О ЧИТАТЕЛЯХ И ПИСАТЕЛЯХ” С ИСПОЛЬЗОВАНИЕМ СЕМАФОРОВ

```
int nr = 0; # число активных читателей
sem rw = 1; # блокировка доступа в БД
sem s = 1; # блокировка читателей при доступе к nr
process Reader[i = 1 to M] {
```

```
  while (true) {
    ...
    P(s);
    nr = nr + 1;
    if (nr == 1) P(rw); # первый читатель получает блокировку
    V(s);
    [чтение БД];
    P(s);
    nr = nr - 1;
    if (nr == 0) V(rw); #последний снимает блокировку
    V(s);
  }
}
process Write[i = 1 to K] {
  while (true) {
    ...
    P(rw);
    [запись БД]
    V(rw);
  }
}
```

Замечание. Если первый процесс обратится к базе данных, а далее подходят к протоколам входа читатель и писатель, то второму читателю ничего проверять не нужно — он беспрепятственно входит (после выполнения критической секции первым); все остальные читатели также не нуждаются в проверках. Поэтому создается неравноправная ситуация: возможен поток читателей, который *заблокирует* возможность входа писателя. Таким образом, ситуация несправедливая: налицо преимущество читателей.

§6 Использование условной синхронизации

Используем в предыдущей задаче условную синхронизацию. Для сохранения непротиворечивости (целостности) базы данных писателям необходим исключительный доступ, а читатели могут работать в любом количестве.

Если `nr` — число читателей, получивших доступ к БД, а `nw` — число писателей, получивших доступ к БД, то должно выполняться условие

$$\texttt{RW}: \ (\texttt{nr} == 0 \vee \texttt{nw} == 0) \wedge \texttt{nw} \leq 1,$$

означающее, что 1) читатели и писатели не могут иметь доступ к БД одновременно, 2) количество писателей не превосходит единицы.

КРУПНОМОДУЛЬНОЕ РЕШЕНИЕ ЗАДАЧИ О ЧИТАТЕЛЯХ И ПИСАТЕЛЯХ

```
int nr = 0, nw = 0;
## RW: (nr == 0 ∨ nw == 0) ∧ nw ≤ 1
process Reader[i = 1 to M] {
 while (true) {
  ...
  < await (nw == 0) nr = nr + 1; >
  # необходимость защиты неделимой операции возникает
  # из-за того, что доступ у читателей возможен в
  # условиях, когда нет писателей
  [чтение БД];
  < nr = nr − 1; >
  # при освобождении ресурса нет необходимости ждать
 }
}
process Writer[i = 1 to K] {
 while (true) {
  ...
```

```
    < await (nr == 0 and nw == 0) nw = nw + 1; >
    # здесь защита связана с условием nw ≤ 1: ресурс
    #может быть захвачен не более, чем одним писателем
    [запись в БД];
    < nw = nw − 1; >
    # освобождение не требует задержки
  }
}
```

Замечание. Условия защиты при захвате ресурса определяются предикатом. При освобождении ресурса никаких условий соблюдать не требуется.

§7 Метод "передачи эстафеты"

Описываемый метод "передачи эстафеты" позволяет реализовать *любой оператор* < await; >.

Рассмотрим двоичный семафор e с начальным значением 1; он будет управлять входом в любое неделимое действие.

С каждым условием защиты связывается один семафор и один счетчик, причем их начальные значения равны 0.

Пусть r — семафор защиты в процессе-читателе, dr — счетчик приостановленных процессов-читателей, w — семафор защиты в процессе-писателе, dw — счетчик приостановленных процессов-писателей. Поскольку сначала нет ожидающих процессов-читателей и процессов-писателей, то все их начальные значения равны 0.

СХЕМА ВЗАИМОДЕЙСТВИЯ ЧИТАТЕЛЕЙ И ПИСАТЕЛЕЙ

```
int nr = 0, ## RW : (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1, #семафор управления входом в критические секции
     r = 0, #семафор для приостановки читателей
     w = 0; #семафор для приостановки писателей
```

```
int dr = 0, # число приостановленных читателей
    dw = 0; # число приостановленных писателей
process Reader[i = 1 to M] {
  while (true) {
    # реализация неделимой
    #  операции < await (nw == 0) nr = nr + 1; >
    P(e);
    if (nw > 0) {dr = dr + 1; V(e); P(r);}
    # если работает писатель, то читатель приостановлен
    nr = nr + 1;
    SIGNAL;
    [чтение базы данных];
    # реализация неделимой операции < nr = nr − 1; >
    P(e);
    nr = nr − 1;
    SIGNAL;
  }
}
process Writer[i = 1 to N] {
  while (true) {
    # реализация неделимой
    # операции < await (nr == 0 and nw == 0) nw = nw + 1; >
    P(e);
    if (nr > 0 or nw > 0) {dw = dw + 1; V(e); P(w);}
    nw = nw + 1;
    SIGNAL;
    [записать в БД];
    # реализация неделимой операции < nw = nw − 1; >
    P(e);
    nw = nw − 1;
```

```
   SIGNAL;
  }
}
```

Здесь SIGNAL представляет собой фрагмент программы, действие которого сводится к следующему:

— если нет активных процессов-писателей, но есть приостановленный процесс-читатель, то последний активизируется и продолжается;

— если нет активных процессов-читателей или процессов-писателей, но есть приостановленный процесс-писатель, то этот процесс-писатель активизируется и продолжается;

— если нет приостановленных процессов-писателей или процессов-читателей, то входной семафор получает сигнал с помощью операции V(e).

ФРАГМЕНТ ПРОГРАММЫ SIGNAL

```
if (nw == 0 and dr > 0) {
  dr = dr − 1; V(r);
  # возобновление процесса-читателя
}
elseif (nr == 0 and nw == 0 and dw > 0) {
  dw = dw − 1; V(w);
  # возобновление процесса-писателя
}
else
  V(e); # разблокирование входа
```

Таким образом, SIGNAL сигнализирует одному из трех семафоров.

Эти три семафора образуют *разделенный двоичный семафор*: в любой момент времени только один из них может иметь значение 1 и все выполняемые ветви начинаются операцией P и заканчиваются операцией V, так как операторы каждой ветви выполняются со взаимным исключением.

Инвариант RW является истинным в начале работы программы и перед каждой операцией V и потому он истинен, если один из семафоров имеет значение 1.

При выполнении каждого защищенного оператора условия упомянутый инвариант оказывается истинным, ибо его истинность установил либо сам процесс, либо семафор, сигнализирующий о приостановке данного процесса.

Заметим, что нет также и взаимных блокировок, т.к. семафор задержки получает сигнал только, если некоторый процесс находится в состоянии ожидания или должен в него войти (процесс может увеличить счетчик ожидающих процессов и выполнять операцию V(e), но не может выполнить операцию P для семафора задержки).

Описанный метод называется *методом передачи эстафеты*; он состоит в следующем:

1) процесс, находящийся в критической секции, получает эстафету, подтверждающую его право на выполнение критической секции;

2) передача эстафеты происходит при достижении SIGNAL;

3) эстафета передается (недетерминированно) одному из процессов, ожидающих условия (становящегося теперь истинным);

4) если нет ожидающих процессов, эстафета передается одному из процессов, который попытается войти в критическую секцию впоследствии.

§8 Об упрощении программы и о справедливости стратегий планирования

Включим фрагмент SIGNAL непосредственно в текст программы предыдущего параграфа и полученную в результате программу несколько упростим. В результате получается программа с теми же свойствами, но более простую, хотя и более длинную.

```
1) int nr = 0, ## RW: (nr == 0 or nw == 0) and nw <= 1
2)     nw = 0;
3) sem e = 1, #семафор e управляет входом в критические секции
4)       r = 0, #семафор r служит для приостановки читателей
```

```
5)      w = 0; #семафор w служит для приостановки писателей
                #всегда выполнено условие 0 ≤ (e + r + w) ≤ 1
6) int dr = 0, # dr — число приостановленных читателей
7)     dw = 0; # dw — число приостановленныъ писателей
8) process Reader[i = 1 to M] {
9)   while (true) {
       # реализация неделимой
       #  операции < await (nw == 0) nr = nr + 1; >
10)    P(e);
11)    if (nw > 0) {dr = dr + 1; V(e); P(r);}
12)    nr = nr + 1;
13)    if (dr > 0) {dr = dr − 1; V(r);}
14)    else V(e);
15)    [чтение базы данных];
       # реализация неделимой операции < nr = nr − 1; >
16)    P(e);
17)    nr = nr − 1;
18)    if (nr == 0 and dw > 0) {dw = dw − 1; V(w);}
19)    else V(e);
20)  }
21) }
22) process Writer[i = 1 to N] {
23)   while (true) {
       # реализация неделимой
       # операции < await (nr == 0 and nw == 0) nw = nw + 1; >
24)    P(e);
25)    if (nr > 0 or nw > 0) {dw = dw + 1; V(e); P(w);}
26)    nw = nw + 1;
27)    V(e);
28)    [запись в БД];
```

```
        # реализация неделимой операции < nw = nw − 1; >
29)     P(e);
30)     nw = nw − 1;
31)     if (dr > 0) {dr = dr − 1; V(r);}
32)     elseif (dw > 0) {dw = dw − 1; V(w);}
33)     else V(e);
34)   }
35) }
```

В рассматриваемых случаях преимущество отдается процессу-читателю. Однако, можно осуществить другие способы планирования, если изменить программу.

Для того, чтобы отдавалось предпочтение процессам-писателям, необходимо, чтобы

— если ждет процесс-писатель, то новые процессы-читатели должны его пропустить (а сами должны приостановиться);

— приостановленный процесс-читатель возобновляет работу только в том случае, если нет приостановленных процессов-писателей.

Первое условие можно выполнить, заменив строку 11 в предыдущей программе строкой

```
if (nw >= 0 or dw > 0) {dr = dr + 1; V(e); P(r);},
```

а второе условие выполнится, если изменить порядок ветвей оператора `if` процессов-писателей (см. строки 31–33 предыдущей программы) следующим образом:

```
if (dw > 0) {dw = dw − 1; V(w);}
elseif (dr > 0) {dr = dr − 1; V(r);}
else V(e);
```

Вывод. Преимущество метода передачи эстафеты в том, что для управления порядком запуска процессов можно менять условия защиты, не влияя на правильность решения.

§9 Задача распределения ресурсов и общая схема ее решения

Задача о распределении ресурсов — это задача о *предоставлении доступа каждому процессу* к тому или иному ресурсу в тот или иной момент времени.

Ресурсом может служить любой разделяемый процессами *объект:* область памяти (база данных, переменные, ячейка буфера), принтер или какое-либо другое устройство, а также некоторые разделяемые программные единицы, например, критические секции.

Простейшая схема распределения ресурсов состоит в том, что если некоторый процесс хочет захватить ресурс, и

1) если ресурс свободен и больше нет желающих, то процесс его захватывает;

2) если ресурс занят, то процесс ожидает его освобождения;

3) если ресурс свободен и еще несколько процессов хотят захватить ресурс, то ресурс предоставляется (недетерминированно) одному из желающих; при этом остальные процессы ждут его освобождения.

Более сложная схема использования ресурса сводится к различным способам учета дополнительных обстоятельств:

— *способа использования* ресурса (например, запись или чтение при работе с памятью),

— *приоритета* процесса в конкурентной борьбе при захвате ресурса и т.п.

Различные модификации упомянутой простейшей схемы могут допускать использование ресурса *несколькими процессами* одновременно или в недетерминированной последовательности; могут быть введены специальные средства для поддержания *"справедливой стратегии"* обслуживания процессов.

Замечание 1. Разделяемый ресурс может распадаться на *несколько элементов,* за обладание которыми идет конкурентная борьба между процессами. В некоторых случаях процессам могут быть нужны наборы элементов, причем разным процессам — возможно, разные наборы таких элементов. Если процессу потребовался некоторый набор этих элементов, то может оказаться, что часть этих элементов занята: процесс вынужден приостановиться. После освобождения нужных ему элементов процесс их захватыва-

ет и продолжает работу. Другая особенность, которая может возникнуть: процесс освобождает элементы *не сразу*, а некоторыми *группами*. Третья особенность может состоять в том, что процесс захватывает элементы ресурса одними группами, а освобождает другими группами. Такая обработка ресурса может быть и у других процессов. Однако, нет особой необдходимости рассматривать ресурс, распадающийся на отдельные элементы, ибо всегда элементы ресурса можно рассматривать, как отдельные виды ресурсов; поэтому в дальнейшем будем считать тот или иной ресурс единым целым, не допускающим дробления на отдельные элементы.

Общую схему операции `request` (запросить)/ `release` (освободить) можно представить в виде:

```
request([параметры]) :
  < await ([условия удовлетворения запроса выполнены])
    [получить требуемое]; >
release([параметры]) :
  < [возвратить полученное]; >
```

Заметим, что *неделимость* этих операций определяется тем, что как при получении требуемого, так и при возврате полученного *необходим доступ к ресурсу.*

Пусть семафор `e` управляет входом в критическую секцию.

Метод передачи эстафеты позволяет реализовать это в следующей форме

```
request([параметры]) :
  P(e);
  if ([условия удовлетворения запроса не выполнены])
    DELAY;
  [получить требуемое];
  SIGNAL;
release([параметры]) :
  P(e);
  [возвратить затребованное];
  SIGNAL;
```

Здесь SIGNAL представляет фрагмент кода, который

1) запускает приостановленные процессы, если их запрос может быть удовлетворен, снимая блокировку с семафора задержки r;

2) снимает блокировку e критической секции (разрешая ожидающему процессу вход в критическую секцию), если запрос не может быть удовлетворен

Таким образом SIGNAL имеет вид

```
SIGNAL :
 if ([выполнены условия возобновления процессов])
  V(r);
   # запустить приостановленные процессы
   # ⇛ посылка сигнала процессам
 else
  V(e);
   #освободить блокировку входа
```

Замечание 2. Фактически, роль SIGNAL — *сигнализировать* процессам, связанным с семафором задержки r, что они могут продолжать работу. Может быть также несколько групп процессов G_1, G_2, ..., G_k, причем каждая группа G_j связана со своим семафором r_j. В этом случае SIGNAL можно рассматривать как сигнал одному из семафоров r_1, ..., r_k, e (см. предыдущий параграф); при этом говорят, что с помощью SIGNAL реализуется *разделяемый семафор*.

Далее, DELAY также представляет собой фрагмент кода, который

1) *запоминает* число запросов n,

2) *снимает* блокировку критической секции с помощью операции V(e),

3) *блокирует* процесс на семафоре задержки r (с помощью P(r)).

Приведем схему фрагмента кода DELAY:

```
DELAY :
 {n = n + 1; V(e); P(r);}
```

Замечание 3. Приведенные фрагменты, конечно, не отражают полностью необходимый программный код, который может варьироваться в значительной степени; они лишь служат для создания общей схемы решения задачи о распределении ресурсов.

§10 Распределение ресурсов по методу “кратчайшее задание”

Пусть разделяемый ресурс состоит из одного элемента. Распределение ресурса по методу “кратчайшее задание” состоит в следующем.

Предположим, что несколько процессов находятся в конкурентной борьбе за разделяемый ресурс, причем запрос ресурса процессом производится операцией `request(time, id)`, где целочисленный параметр `time` определяет приоритет запроса (чем меньше `time`, тем выше приоритет), а параметр `id` идентифицирует процесс.

Если ресурс свободен, то он выделяется процессу с идентификатором `id` немедленно; в противном случае упомянутый процесс приостанавливается.

Освобождение ресурса, захваченного процессом, происходит с помощью операции `release`.

Освобожденный ресурс тут же отдается приостановленному процессу с наименьшим значением параметра `time` (если такие процессы имеются). В том же случае, когда указанное значение `time` оказалось наименьшим у нескольких процессов, то ресурс отдается тому, кто ждал дольше всех.

В частности, параметр приоритета `time` может являться предполагаемым временем работы процесса с затребованным ресурсом (в некоторых случаях можно определить даже точное значение такого времени, а в других случаях время `time` может быть указано ориентировочно).

Замечание 1. Если `time` — время работы процесса с ресурсом, то метод “кратчайшее задание” часто *минимизирует* затраты времени на выполнение всей задачи.

Замечание 2. Рассматриваемый метод в чистом виде *не является справедливым,* ибо при наличии потока запросов с малым значением `time`, некоторые запросы с бóльшим значением `time` могут

вообще никогда *не получить* этот ресурс; естественно, что в этом случае вся задача не будет выполнена.

Пусть `free` — булевская переменная, принимающая значение `TRUE`, когда ресурс доступен, а `FALSE`, когда он занят. Для простоты будем считать, что все запросы имеют различные значения `time`. Обозначим `pairs` — множество пар (`time`, `id`), упорядоченное по возрастанию параметра `time`; тогда глобальный инвариант метода "кратчайшее расстояние" может быть записан в виде

`INV` : ([`pairs` — упорядоченный набор])∧
(`free` ⇒ (`pairs` == ∅));

таким образом, `pairs` — упорядоченный набор, и если `free` = `TRUE`, то `pairs` — пустой набор.

Вначале предикат `INV` истинен, т.к. `pairs` — пустое множество, и `free` = `TRUE`.

Поскольку запрос может быть удовлетворен, как только ресурс станет доступным, то можно предложить такое решение задачи о распределении ресурсов по упомянутому методу:

```
bool free = true; #bool free разделяемая переменная
request(time, id) : < await (free) free = false; >
release( ) : < free = true; >
```

Операции `request`/`release` реализуем следующим способом:

```
request(time, id) :
  P(e);
  if (!free) DELAY;
  free = false;
  SIGNAL;
release( );
  P(e);
  free = true;
  SIGNAL;
```

В данном случае во фрагменте `DELAY` процесс должен

1) вставить параметры запроса в набор `pairs` (т.е. учесть запрос);

2) освободить критическую секцию с помощью `V(e)`;

3) заблокироваться на семафоре задержки.

В предыдущих примерах (например, в задаче о читателях и писателях) были два условия задержки и два семафора задержки. В данном случае у каждого процесса есть *свое условие задержки,* которое зависит от его приоритета (от позиции в наборе `pairs`), так что все процессы упорядочены; поэтому каждый процесс должен ждать на своем семафоре задержки.

Если имеется n процессов, то рассматривается массив `b[n]` семафоров с начальными значениями 0. Пусть идентификаторы `id` этих процессов уникальны и значения идентификаторов находятся во множестве $\{0, 1, \ldots, n-1\}$; тогда можно считать, что процесс с идентификатором `id` приостанавливается на семафоре `b[id]`.

В результате получим следующее решение рассматриваемой задачи.

РЕШЕНИЕ ЗАДАЧИ РАСПРЕДЕЛЕНИЯ РЕСУРСОВ ПО МЕТОДУ “КРАТЧАЙШЕЕ ЗАДАНИЕ”

```
bool free = true;
sem e = 1, b[n] = ([n]0);
# семафоры входа и задержек
typedef P = set of (int, int);
P pairs = ∅;
## INV : (pairs — упорядоченный набор∧
          (free ⇒ (pairs == ∅))
request(time, id) :
 P(e);
 if (!free) {
  [вставить (time, id) в pairs];
  V(e); # снять блокировку входа
  P(b[id]); # ждать возобновления процесса
 }
```

```
  free = false;
  [предоставление ресурса];
  V(e);  #снимается блокировка, т.к. free == false
         # [остальные процессы тоже могут попытаться]
         # [попасть в pairs]
release( ) :
  P(e);
  free = true;
  if (P! = ∅) {
    [удалить первую пару (time, id) из pairs];
    V(b[id]);  # передать эстафету процессу
               # с идентификатором id
  }
  else V(e);
```

Определение. Семафор называется *частным,* если операция `P` над ним выполняется лишь одним процессом.

Замечание 3. Семафоры `b[i]` предыдущего примера являются частными.

Замечание 4. В некоторых случаях *число условий задержки меньше,* чем *число процессов:* так происходит, когда порядок удовлетворения запросов не важен (см. задачу о читателях и писателях). В этом случае экономнее использовать меньшее количество семафоров по сравнению с числом процессов: достаточно применить столько же семафоров, сколько имеется условий задержки (к ним, конечно, добавляется семафор `e` входа в критическую секцию).

Если ресурс состоит из ряда элементов, а процесс может затребовать `amount` таких элементов, то можно изменить предыдущее решение следующим образом:

1) введем параметр `amount` в операции `request` и `release`;

2) булеву переменную `free` заменим целочисленной переменной `avail` для хранения количества доступных элементов;

3) в `request` вместо `!free` будем проверять условие `amount` > `avail` (число затребованных элементов больше числа доступных).

§11 Некоторые возможности библиотеки Pthreads

Для написания переносимых многопоточных приложений в середине 1990-х годов группой под эгидой организации POSIX (*Portable Operating System Interface* — интерфейс переносимой операционной системы) была разработана библиотека Pthreads. Она распространяется с различными операционными системами (в частности, с операционной системой Unix).

Опишем здесь базовый набор функций этой библиотеки.

При использовании языка *C* прежде всего необходимо:

1) подключить стандартный заголовочный файл

```
# include < pthread.h >
```

2) сделать объявления

```
pthread_attr_t tathr; /* атрибуты потока */
pthread_t tid; /* дескриптор потока */
```

3) инициализировать атрибуты с помощью функций

```
pthread_attr_init(&tattr);
pthread_attr_setscope(&tattr, PTHREAD_SCOPY_SYSTEM);
```

4) создать поток, установив предварительно его начальные атрибуты (см. ниже).

Атрибуты потока включают: 1) размер его стека, 2) его приоритет, 3) область планирования (локальная или глобальная).

Обычно достаточно значений атрибутов, установленных по умолчанию (исключением является область планирования: здесь значение следует указывать явно).

Если поток должен конкурировать в борьбе за процессор со всеми потоками, а не только с родительским, то следует произвести вызов функции `pthread_attr_setscope` (см. выше).

Новый поток создается вызовом

```
pthresd_create(&tid, &tattr, start_func, arg);
```

Первый аргумент — адрес дескриптора потока, второй аргумент — адрес дескриптора атрибутов потока. Поток начинает свою работу с вызова функции `start_func` с одним аргументом `arg`. Поток

завершает свою работу вызовом

```
pthread_exit(value);
```

где `value` — скалярное возвращаемое значение или `null`.

Если требуется, чтобы родительский поток *ждал* завершения работы сыновнего процесса, то следует установить

```
pthread_join(tid, value_ptr);
```

где `tid` — идентификатор сыновнего процесса, а `value_ptr` — адрес для возвращаемого значения, которое определяется, когда сыновний процесс вызывает функцию `exit`.

Потоки взаимодействуют через *глобальные* переменные; они синхронизируются с помощью *активного ожидания, блокировок, семафоров* или с помощью *условных переменных*.

Заголовочный файл `semaphore.h` содержит *определения* и *прототипы операций* для семафоров.

Дескрипторы семафоров определены как *глобальные* относительно использующих их потоков `sem_t S`; дескриптор инициализируется вызовом

```
sem_init(&S, SHARED, 1);
```

который позволяет семафору присвоить значение 1.

Если параметр `SHARED` *не равен нулю,* то процессы могут разделять данный семафор; в противном случае (т.е. если параметр `SHARED` равен нулю) — семафор может использоваться потоками только одного процесса.

Эквивалентом операции `P` в библиотеке Pthread служит функция `sem_wait`, а эквивалентом операции `V` является функция `sem_post`.

Этот параграф закончим одним из вариантов *защиты критической секции* с использованием семафоров из библиотеки Pthread:

```
sem_wait(&s); /* применение операции P к семафору s */
[критическая секция];
sem_post(&s); /* применение операции V к семафору s */
```

§12 Программа “производитель–потребитель”, использующая библиотеку Pthreads

В приводимой здесь программе рассматриваются независимые потоки `Producer` и `Consumer`, которые общаются через разделяемый буфер `data`; для попеременного доступа к последнему используются семафоры `empty` и `full`. Производитель помещает в буфер числа 1, 2, . . . , `numIters`, а потребитель извлекает и складывает их.

Головная программа `main` инициализирует дескрипторы и семафоры, создает упомянутые потоки и ожидает окончания их работы. В этой программа аргументы потокам не передаются: в функции `pthread_create` использован адрес `NULL`.

```
1) # include < pthread.h >
2) # include < semaphore.h >
3) # define SHARED 1
4) # include < stdio.h >
5) void *Producer(void*); /* потоки */
6) void *Consumer(void*);
7) sem_t empty, full; /* дескрипторы глобальных семафоров */
8) int data; /* разделяемый буфер */
9) int numIters;
10) /* main — прочитать командную строку, создать потоки */
11) int main(int args, char.*argv[ ]) {1
12)   pthread_t pid,cid; /* дескрипторы */
13)   pthread_attr_t attr; /* атрибуты потока */
14)   pthread_attr_init(&attr); /* инициализация атрибутов потока */
15)   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
16)   sem_init(&empty, SHARED, 1);
      /* инициализация семафора sem empty = 1 */
17)   sem_init(&full, SHARED, 0);
      /* инициализация семафора sem full = 0 */
18)   numIters = atoi(argv[1]);
19)   pthread_create(&pid, &attr, Producer, NULL);
```

```
    /* создание новых потоков */
20) pthread_create(&cid, &attr, Consumer, NULL);
21) pthread_join(pid, NULL);
22) pthread_join(cid, NULL);
    /* родительские процессы ожидают завершения */
    /* сыновних процессов */
23) }1
24) /* поместить в буфер данных числа 1, ..., numIter */
25) void *Producer(void *arg); {1
26)  int produced;
27)  for (produces = 1; produced <= numIters; produced++) {2
28)   sem_wait(&empty);
29)   data = produsedd;
30)   sem_post(&full);
31)  }2
32) }1
33) /* извлечь numIters элементов из буфера и сложить */
34) void *Consumer(void *arg) {1
35)  int total = 0, consumed;
36)  for (consumed = 1; consumed <= numIters; cunsumed++) {2
37)   sem_wait(&full); # P(full)
38)   total = total + data;
39)   sem_post(&empty); # V(empty)
40)  }2
41)  printf("Сумма равна" ,total);
42) }1
```

СПИСОК ТЕРМИНОВ

Литература

Список литературы

[1] *Воеводин В.В.* Математические модели и методы в параллельных процессах. М., 1986. 296 с.

[2] *Корнеев В.В.* Параллельные вычислительные система. М., 1999. 320 с.

[3] *Yukiya Aoyama, Jun Nakato.* RS/6000 SP:Practical MPI Programming. IBM. Tachnical Support Organization., 2000. 221 p. www.redbook.ibm.com

[4] *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб., 2002. 608 с.

[5] *Бурова И.Г., Демьянович Ю.К.* Лекции по параллельным вычислениям. СПб., 203. 132 с.

[6] *Грегори Р. Эндрюс.* Основы многопоточного параллельного и распределенного программирования. М., 2003. 512 с.

[7] Математическая Энциклопедия. М., 1977. Т.1, 1152 с.

Содержание