Министерство образования и науки РФ
Иркутский национальный исследовательский технический университет

Распараллеливание потоков

Методические указания
по выполнению лабораторных работ

Издательство
Иркутского национального исследовательского технического университета
2018

УДК *****

Рецензент
Доктор техн. наук, зав.каф. кафедры Информационные системы и защита информации ФГБОУ ВО «ИрГУПС» Л.В.Аршинский.

Распараллеливание потоков: Метод. указания по выполнению лабораторных работ / сост.: З.А.Бахвалова – Иркутск: Изд-во ИРНИТУ, 2018 - 43с.

Соответствуют требованиям ФГОС ВО 09.03.01- Информатика и вычислительная техника и 09.03.02 - Информационные системы и технологии.

Данные методические материалы содержат теоретические материалы и задания для выполнения лабораторной работы, посвященной реализации параллельных процессов в современных языках программирования. Даны базовые примеры использования многопоточных приложений при построении программ на языке Java. Приведенные материалы используются при выполнении работ по курсам «Технологии разработки программных комплексов» и «Технология разработки программных комплексов».

Учебное издание

Распараллеливание потоков

Методические указания
по выполнению лабораторных работ

Составители:
Бахвалова Зинаида Андреевна

Оглавление

Введение 5
Лабораторная работа «Распараллеливание потоков» 7
Основные понятия 7
Параллельные процессы 7
Понятие ресурса 8
Организация программ как системы процессов 9
Понятие взаимодействия 11
Понятие тупика 16
Описание возможных изменений программы 19
Введение в Java 20
Параллелизм с точки зрения ОС 20
Поток 21
Потоки в Java 22
Отладка многопоточных программ 26
Проблема остановки 29
Практические примеры использования потоков 34
Задания для лабораторной работы 37
Требование к отчету 41
Контрольные вопросы 42
Список используемой литературы 43

Введение

Параллельные вычисления— способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно).

Существуют различные способы реализации параллельных вычислений. Например, каждый вычислительный процесс может быть реализован в виде процесса операционной системы, либо же вычислительные процессы могут представлять собой набор потоков выполнения внутри одного процесса ОС. Параллельные программы могут физически исполняться либо последовательно на единственном процессоре— перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно— выделяя каждому вычислительному процессу один или несколько процессоров (находящихся рядом или распределённых в компьютерную сеть).

Основная сложность при проектировании параллельных программ— обеспечить правильную последовательность взаимодействий между различными вычислительными процессами, а также координацию ресурсов, разделяемых между процессами.

При всем богатстве выбора существующих и создающихся вновь средств разработки параллельных программ, начиная от языков программирования и заканчивая библиотеками, предлагающими готовые реализации типовых вычислительных задач, реальных альтернатив на сегодняшний момент довольно немного, и каждой из них присущи свои ограничения.

Многопоточное программирование. Наиболее общим способом создания параллельных программ для систем с общей памятью является использование потоков. При этом функциональный параллелизм легко обеспечивается написанием разных потоковых функций, а параллелизм по данным реализуется благодаря общему виртуальному адресному пространству процесса, к которому все потоки имеют доступ. Работать с потоками программист может, как используя API операционной системы, так и создав собственную библиотеку потоков. Последний подход в ряде случаев может обеспечить большее быстродействие программы за счет меньших накладных расходов, но значительно более трудоемок.

Разработка параллельной программы на основе потоков, в особенности в случае параллелизма по данным, предполагает решение задачи синхронизации и преодоление ряда возможных проблем: взаимной блокировки, тупиков, гонок данных и т.д. Целью при этом является получение корректных результатов выполняемых вычислений, например, недопущение чтения из оперативной памяти порции данных одним потоком в тот момент, когда другой поток осуществляет запись этих данных.

Операционные системы предоставляют необходимые средства для решения всех задач, возникающих в многопоточном программировании: кри-

тические секции, признаки блокировки, семафоры, мьютексы, события и т.д. Однако грамотное использование этих механизмов требует существенных усилий, поскольку отсутствие необходимой синхронизации доступа к данным влечет за собой, как минимум, неверные результаты, а в худшем случае приводит к аварийному завершению программы. В то же время, чрезмерная синхронизация ведет к снижению эффективности и масштабируемости параллельной программы.

Технология OpenMP. Задачи, которые должен решить программист, разрабатывающий параллельную программу для системы с общей памятью, во многих случаях идентичны, а значит, их реализацию можно переложить на компилятор, что и предлагает стандарт OpenMP, специфицирующий набор директив компилятора (для языков C, C++ и Fortran), функций библиотеки (для тех же языков) и переменных окружения. OpenMP можно рассматривать как высокоуровневую надстройку над POSIX или Windows Threads (или аналогичными библиотеками потоков).

Создание параллельных программ с использованием стандарта openmp и соответствующих компиляторов во многих случаях дает не меньшую эффективность, чем программирование в потоках, и требует существенно меньше усилий от разработчика, однако OpenMP работает только в SMP-системах.

Технология MPI. Параллельное программирование для систем с распределенной памятью (кластеров) не имеет в достаточной степени высокоуровневой (наподобие потоков) поддержки в операционных системах. Прямую работу с сокетами для реализации обмена данными между процессами параллельной программы не отнесешь к удобным подходам.

Стандарт MPI предоставляет механизм построения параллельных программ в модели обмена сообщениями, характерной для разработки программ, ориентированных на кластерные системы. Стандарт специфицирует набор функций и вводит определенный уровень абстракций на основе сообщений, типов, групп и коммуникаторов, виртуальных топологий. Существуют стандартные “привязки” MPI к языкам C, C++, Fortran. Реализации стандарта MPI имеются практически для всех суперкомпьютерных платформ, а также

Кластеров на основе рабочих станций UNIX\LINUX и WINDOWS. В настоящее время MPI – наиболее широко используемый и динамично развивающийся интерфейс из своего класса.

Основным подходом к построению MPI -программ является явное распределение данных и вычислений между процессами, а также обмен сообщениями для передачи данных, в силу чего MPI-программа часто существенно отличается от программы последовательной, а в некоторых случаях даже не может выполняться в однопроцессном варианте. И конечно, создание и отладка MPI-программ требует значительно больших усилий, чем создание последовательной программы, решающей ту же задачу.

ЛАБОРАТОРНАЯ РАБОТА «РАСПАРАЛЛЕЛИВАНИЕ ПОТОКОВ»

Цель работы: формирование представления о способах параллельных вычислений на современных языках программирования.

Задача: разработка многопоточного приложения.

Требования к знаниям: при выполнении работы не накладывается ограничений на язык и среду разработки.

ОСНОВНЫЕ ПОНЯТИЯ

Параллельные процессы

Основой для построения моделей функционирования программ, реализующих параллельные методы решения задач, является понятие *процесса* как конструктивной единицы построения параллельной программы. Описание программы в виде набора процессов, выполняемых параллельно на разных процессорах или на одном процессоре в режиме разделения времени, позволяет сконцентрироваться на рассмотрении проблем организации *взаимодействия процессов*, определить моменты и способы обеспечения с*инхронизации и взаимоисключения процессов*, изучить условия возникновения или доказать отсутствие *тупиков* в ходе выполнения программ (ситуаций, в которых все или часть процессов не могут быть продолжены при любых вариантах продолжения вычислений).

Понятие *процесса* является одним из основополагающих в теории и практике параллельного программирования. Разброс в трактовке данного понятия является достаточно широким, но в целом большинство определений сводится к пониманию процесса как "*некоторой последовательности команд, претендующей наравне с другими процессами программы на использование процессора для своего выполнения*".

Конкретизация понятия процесса зависит от целей исследования параллельных программ. Для анализа проблем организации взаимодействия процессов процесс можно рассматривать как последовательность команд

$$p_n = (i_1, i_2, \cdots, i_n)$$

(для простоты изложения материала будем предполагать, что процесс описывается единственной командной последовательностью). Динамика развития процесса определяется моментами времен начала выполнения команд

$$t(p_n) = t_p = (\tau_1, \tau_2, \ldots, \tau_n),$$

где τ_j, $1 \le j \le n$, есть время начала выполнения команды τ_j. Последовательность t_p представляет временную *траекторию* развития процесса. Предполагая, что команды процесса исполняются строго последовательно, в ходе своей реализации не могут быть приостановлены (т.е. являются неделимыми) и имеют одинаковую длительность выполнения, равную 1 (в тех или иных временных единицах), получим, что моменты времени траектории процесса должны удовлетворять соотношениям

$$\forall i, \quad 1 \le i < n \quad \Rightarrow \quad \tau_{i+1} \ge \tau_i + 1.$$

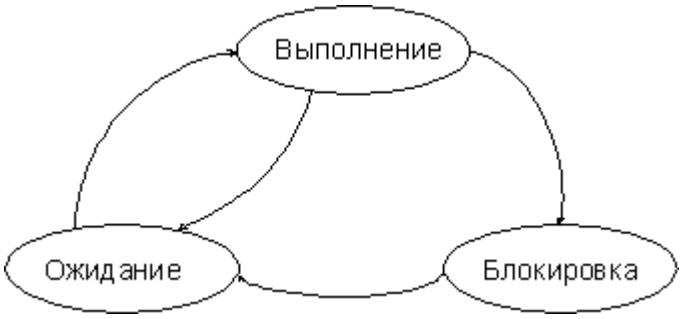

Рисунок 1 - Диаграмма переходов процесса из состояния в состояние

Равенство $\tau_{i+1} = \tau_i + 1$ достигается, если для выполнения процесса выделен процессор и после завершения очередной команды процесса сразу же начинается выполнение следующей команды. В этом случае говорят, что процесс является *активным* и находится в *состоянии выполнения*. Соотношение $\tau_{i+1} > \tau_i + 1$ означает, что после выполнения очередной команды процесс *приостановлен* и ожидает возможности для своего продолжения. Данная приостановка может быть вызвана необходимостью разделения использования единственного процессора между одновременно исполняемыми процессами. В этом случае приостановленный процесс находится в *состоянии ожидания* момента предоставления процессора для своего выполнения. Кроме того, приостановка процесса может быть вызвана и временной неготовностью процесса к дальнейшему выполнению (например, процесс может быть продолжен только после завершения операции ввода-вывода). В подобных ситуациях говорят, что процесс является *блокированным* и находится в *состоянии блокировки*.

В ходе своего выполнения состояние процесса может многократно изменяться; возможные варианты смены состояний показаны на диаграмме переходов

Понятие ресурса

Понятие *ресурса* обычно используется для обозначения любых объектов вычислительной системы, которые могут быть использованы процессом для своего выполнения. В качестве ресурса может рассматриваться процесс, память, программы, данные и т.п. По характеру использования могут различаться следующие категории ресурсов:

- *выделяемые* (монопольно используемые, неперераспределяемые) ресурсы характеризуются тем, что выделяются процессам в момент их возникновения и освобождаются только в момент завершения процессов; в качестве такого ресурса может рассматриваться, например, устройство чтения на магнитных лентах;
- *повторно распределяемые ресурсы* отличаются возможностью динамического запрашивания, выделения и освобождения в ходе выполнения процессов (таковым ресурсом является, например, оперативная память);
- *разделяемые* ресурсы, особенность которых состоит в том, что они постоянно остаются в общем использовании и выделяются процессам для использования в режиме разделения времени (как, например, процессор, разделяемые файлы и т.п.);
- *многократно используемые* (реентерабельные) ресурсы выделяются возможностью одновременного использования несколькими процессами (что может быть обеспечено, например, при неизменяемости ресурса при его использовании; в качестве примеров таких ресурсов могут рассматриваться реентерабельные программы, файлы, используемые только для чтения и т.д.).

Следует отметить, что тип ресурса определяется не только его конкретными характеристиками, но и зависит от применяемого способа использования. Так, например, оперативная память может рассматриваться как повторно распределяемый, так и разделяемый

ресурс; использование программ может быть организовано в виде ресурса любого рассмотренного типа.

Организация программ как системы процессов

Понятие процесса может быть использовано в качестве основного конструктивного элемента для построения параллельных программ в виде совокупности взаимодействующих процессов. Такая агрегация программы позволяет получить более компактные (поддающиеся анализу) вычислительные схемы реализуемых методов, скрыть при выборе способов распараллеливания несущественные детали программной реализации, обеспечивает концентрацию усилий на решение основных проблем параллельного функционирования программ.

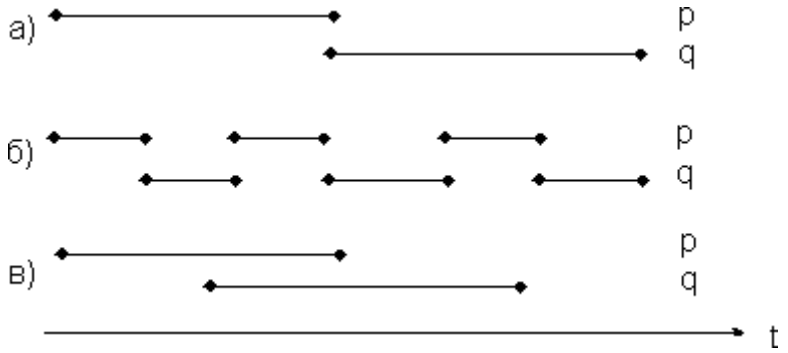

Рисунок 2 - Варианты взаиморасположения траекторий одновременно исполняемых процессов (отрезки линий изображают фрагменты командных последовательностей процессов)

Существование нескольких одновременно выполняемых процессов приводит к появлению дополнительных соотношений, которые должны выполняться для величин временных траекторий процессов. Возможные типовые варианты таких соотношений на примере двух процессов p и q состоят в следующем (см. рис.2):

- выполнение процессов осуществляется строго последовательно, т.е. процесс q начинает свое выполнение только после полного завершения процесса p (однопрограммный режим работы ЭВМ – см. рис.2. а),
- выполнение процессов может осуществляться одновременно, но в каждый момент времени могут исполняться команды только какого либо одного процесса (режим разделения времени или многопрограммный режим работы ЭВМ – см. рис.2. б),
- параллельное выполнение процессов, когда одновременно могут выполняться команды нескольких процессов (данный режим исполнения процессов осуществим только при наличии в вычислительной системе нескольких процессоров - см. рис.2. в).

Приведенные варианты взаиморасположения траекторий процессов определяются не требованиями необходимых функциональных взаимодействий процессов, а являются лишь следствием технической реализации одновременной работы нескольких процессов. С другой стороны, возможность чередования по времени командных последовательностей разных процессов следует учитывать при реализации процессов. Рассмотрим для примера два процесса с идентичным программным кодом.

Процесс 1	**Процесс 2**
N = N + 1	N = N + 1
печать N	печать N

Пусть начальное значение переменной N равно 1. Тогда при последовательном исполнении процесс 1 напечатает значение 2, процесс 2 – значение 3. Однако возможна и другая последовательность исполнения процессов в режиме разделения времени (с учетом того, что сложение N = N +1 выполняется при помощи нескольких машинных команд)

Время	Процесс 1	Процесс 2
1	Чтение N (1)	
2		Чтение N (1)
3		Прибавление 1 (2)
4	Прибавление 1 (3)	
5	Запись N (3)	
6	Печать N (3)	
7		Запись N (3)
8		Печать N (3)

(в скобках для каждой команды указывается значение переменной N).

Как следует из приведенного примера, результат одновременного выполнения нескольких процессов, если не предпринимать специальных мер, может зависеть от порядка исполнения команд.

Выполним анализ возможных командных последовательностей, которые могут получаться для программ, образованных в виде набора процессов. Рассмотрим для простоты два процесса

$$p_n = (i_1, i_2, \cdots, i_n), q_m = (j_1, j_2, \cdots, j_m).$$

Командная последовательность программы образуется чередованием команд отдельных процессов и, тем самым, имеет вид:

$$r_s = (l_1, l_2, \cdots, l_s), s = n + m.$$

Фиксация способа образования последовательности из команд отдельных процессов может быть обеспечена при помощи характеристического вектора

$$x_s = (\chi_1, \chi_2, \cdots, \chi_s),$$

в котором следует положить $\chi_k = p_n$, если команда l_k получена из процесса p_n (иначе $\chi_k = q_m$). Порядок следования команд процессов в r_s должен соответствовать порядку расположения этих команд в исходных процессах

$$\forall u, v : (u < v), (\chi_u = \chi_v = p_n) \Rightarrow p_n(l_u) < p_n(l_v),$$

где $p_n(l_k)$ есть команда процесса p_n, соответствующая команде l_k в r_s.

С учетом введенных обозначений, под программой, образованной из процессов p_n и q_m, можно понимать множество всех возможных командных последовательностей

$$R_s = \{< r_s, x_s >\}.$$

Данный подход позволяет рассматривать программу так же, как некоторый обобщенный (агрегированный) процесс, получаемый путем параллельного объединения составляющих процессов

$$R_s = p_n \otimes q_m.$$

Выделенные особенности одновременного выполнения нескольких процессов могут быть сформулированы в виде ряда принципиальных положений, которые должны учитываться при разработке параллельных программ:

− моменты выполнения командных последовательностей разных процессов могут чередоваться по времени;

− между моментами исполнения команд разных процессов могут выполняться различные временные соотношения (отношения следования); характер этих соотношений зависит от количества и быстродействия процессоров и загрузки вычислительной системы и, тем самым, не может быть определен заранее;

− временные соотношения между моментами исполнения команд могут различаться при разных запусках программ на выполнение, т.е. одной и той же программе при одних и тех же исходных данных могут соответствовать разные командные последовательности вследствие разных вариантов чередования моментов работы разных процессов;

− доказательство правильности получаемых результатов должно проводиться для любых возможных временных соотношений для элементов временных траекторий процессов;

− для исключения зависимости результатов выполнения программы от порядка чередования команд разных процессов необходим анализ ситуаций взаимовлияния процессов и разработка методов для их исключения.

Перечисленные моменты свидетельствуют о существенном повышении сложности параллельного программирования по сравнению с разработкой "традиционных" последовательных программ

Понятие взаимодействия

Одной из причин зависимости результатов выполнения программ от порядка чередования команд может быть разделение одних и тех же данных между одновременно исполняемыми процессами.

Данная ситуация может рассматриваться как проявление общей проблемы использования разделяемых ресурсов (общих данных, файлов, устройств и т.п.). Для организации разделения ресурсов между несколькими процессами необходимо иметь возможность:

− определения доступности запрашиваемых ресурсов (ресурс свободен и может быть выделен для использования, ресурс уже занят одним из процессов программы и не может использоваться дополнительно каким-либо другим процессом);

− выделения свободного ресурса одному из процессов, запросивших ресурс для использования;

− приостановки (блокировки) процессов, выдавших запросы на ресурсы, занятые другими процессами.

Главным требованием к механизмам разделения ресурсов является гарантированное обеспечение использования каждого разделяемого ресурса только одним процессом от момента выделения ресурса этому процессу до момента освобождения ресурса. Данное требование в литературе обычно именуется взаимоисключением процессов; командные последовательности процессов, в ходе которых процесс использует ресурс, называется критической секцией процесса. С использованием последнего понятия условие взаимоисключения процессов может быть сформулировано как требование нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного процесса.

Рассмотрим несколько вариантов программного решения проблемы взаимоисключения (для записи программ используется язык программирования C++). В каждом из вариантов будет предлагаться некоторый частный способ взаимоисключения процессов с целью демонстрации всех возможных ситуаций при использовании общих разделяемых ресурсов. Последовательное усовершенствование механизма взаимоисключения при рассмотрении вариантов приведет к изложению алгоритма Деккера, обеспечивающего взаимоисключение для двух параллельных процессов. Обсуждение способов взаимоисключе-

ния завершается рассмотрением концепции семафоров Дейкстра, которые могут быть использованы для общего решение проблемы взаимоисключения любого количества взаимодействующих процессов.

Первый способ взаимодействия

```
intProcessNum = 1; // номер процесса для доступа к ресурсу

Process_1()
{
  while (1)
  {  // повторять, пока право доступа к ресурсу у процесса 2
     while ( ProcessNum == 2 );
       < Использование общего ресурса >
       // передача права доступа к ресурсу процессу 2
       ProcessNum = 2;
  }
}

Process_2()
{
    while (1)
    { // повторять, пока право доступа к ресурсу у процесса 1
      while ( ProcessNum == 1 );
      < Использование общего ресурса >
      // передача права доступа к ресурсу процессу 1
      ProcessNum = 1;
    }
 }
```

Реализованный в программе способ гарантирует взаимоисключение, однако такому решению присущи два существенных недостатка:

- ресурс используется процессами строго последовательно (по очереди) и, как результат, при разном темпе развития процессов общая скорость выполнения программы будет определяться наиболее медленным процессом;
- при завершении работы какого-либо процесса другой процесс не сможет воспользоваться ресурсом и может оказаться в постоянно блокированном состоянии.

Решение проблемы взаимоисключения подобным образом известно в литературе как способ жесткой синхронизации

Второй способ взаимодействия

В данном варианте для ухода от жесткой синхронизации используются две управляющие переменные, фиксирующие использование процессами разделяемого ресурса.

```
  int ResourceProc1 = 0; // = 1 – ресурс занят процессом 1
  int ResourceProc2 = 0; // = 1 – ресурс занят процессом 2
  Process_1()
  {
    while (1)
   {
      // повторять, пока ресурс используется процессом 2
      while ( ResourceProc2 == 1 );
      ResourceProc1 = 1;
      < Использование общего ресурса >
      ResourceProc1 = 0;
    }
  }

 Process_2()
 {
    while (1)
    {
      // повторять, пока ресурс используется процессом 1
      while ( ResourceProc1 == 1 );
      ResourceProc2 = 1;
```

```
      < Использование общего ресурса >
      ResourceProc2 = 0;
    }
 }
```

Предложенный способ разделения ресурсов устраняет недостатки жесткой синхронизации, однако при этом *теряется гарантия взаимоисключения* – оба процесса могут оказаться одновременно в своих критических секциях (это может произойти, например, при переключении между процессами в момент завершения проверки занятости ресурса). Данная проблема возникает вследствие различия моментов проверки и фиксации занятости ресурса.

Следует отметить, что в отдельных случаях взаимоисключение процессов в данном примере может произойти и корректно - все определяется конкретными моментами переключения процессов. Отсюда следует два важных вывода:

- успешность однократного выполнения не может служить доказательством правильности функционирования параллельной программы даже при неизменных параметрах решаемой задачи;
- для выявления ошибочных ситуаций необходима проверка разных временных траекторий выполнения параллельных процессов

Третий способ взаимодействия

Возможная попытка в восстановлении взаимоисключения может состоять в установке значений управляющих переменных перед циклом проверки занятости ресурса.

```
int ResourceProc1 = 0; // = 1 – ресурс занят процессом 1
int ResourceProc2 = 0; // = 1 – ресурс занят процессом 2

Process_1()
{
  while (1)
  {
   // установить, что процесс 1 пытается занять ресурс
   ResourceProc1 = 1;
     // повторять, пока ресурс занят процессом 2
      while ( ResourceProc2 == 1 );
       < Использование общего ресурса >
       ResourceProc1 = 0;
  }
}

Process_2()
{
  while (1)
  {
     // установить, что процесс 2 пытается занять ресурс
     ResourceProc2 = 1;
       // повторять, пока ресурс используется процессом 1
       while ( ResourceProc1 == 1 );
       < Использование общего ресурса >
       ResourceProc2 = 0;
   }
}
```

Представленный вариант восстанавливает взаимоисключение, однако при этом возникает новая проблема – оба процесса могут оказаться заблокированными вследствие бесконечного повторения циклов ожидания освобождения ресурсов (что происходит при одновременной установке управляющих переменных в состояние "занято"). Данная проблема известна под названием ситуации тупика (дедлока или смертельного объятия) и исключение тупиков является одной из наиболее важных задач в теории и практике параллельных вычислений.

Четвёртый способ взаимодействия

Предлагаемый подход для устранения тупика состоит в организации временного снятия значения занятости управляющих переменных процессов в цикле ожидания ресурса.

```
int ResourceProc1 = 0; // =1 – ресурс занят процессом 1
int ResourceProc2 = 0; // =1 – ресурс занят процессом 2

Process_1()
{
  while (1)
   {
      ResourceProc1 = 1;       // процесс 1 пытается занять ресурс
      // повторять, пока ресурс занят процессом 2
      while ( ResourceProc2 == 1 )
      {
        ResourceProc1 = 0;     // снятие занятости ресурса
        < временная задержка >
        ResourceProc1 = 1;
      }
      < Использование общего ресурса >
      ResourceProc1 = 0;
   }
}

Process_2()
{
  while (1)
   {
      ResourceProc2 = 1; // процесс 2 пытается занять ресурс
      // повторять, пока ресурс используется процессом 1
      while ( ResourceProc1 == 1 )
       {
        ResourceProc2 = 0; // снятие занятости ресурса
        < временная задержка >
        ResourceProc2 = 1;
       }
       < Использование общего ресурса >
       ResourceProc2 = 0;
   }
}
```

Длительность временной задержки в циклах ожидания должна определяться при помощи некоторого случайного датчика. При таких условиях реализованный алгоритм обеспечивает взаимоисключение и исключает возникновение тупиков, но опять таки не лишен существенного недостатка (перед чтением следующего текста попытайтесь определить этот недостаток).

Проблема состоит в том, что потенциально решение вопроса о выделении может откладываться до бесконечности (при синхронном выполнении процессов). Данная ситуация известна под наименованием бесконечное откладывание (starvation).

Алгоритм Деккера

В алгоритме Деккера предлагается объединение предложений вариантов 1 и 4 решения проблемы взаимоисключения.

```
int ProcessNum=1; // номер процесса для доступа к ресурсу
int ResourceProc1 = 0; // = 1 – ресурс занят процессом 1
int ResourceProc2 = 0; // = 1 – ресурс занят процессом 2

Process_1()
{
  while (1)
  {
```

```
   ResourceProc1 = 1;  // процесс 1 пытается занять ресурс
    //* цикл ожидания доступа к ресурсу */
    while ( ResourceProc2 == 1 )
    {
       if ( ProcessNum == 2 )
       {
          ResourceProc1 = 0;
         // повторять, пока ресурс занят процессом 2
          while ( ProcessNum == 2 );
          ResourceProc1 = 1;
       }
    }
    < Использование общего ресурса >
    ProcessNum  = 2;
    ResourceProc1 = 0;
  }
}

 Process_2()
 {
    while (1)
    {
      ResourceProc2 = 1;        // процесс 2 пытается занять ресурс
      /* цикл ожидания доступа к ресурсу */
      while ( ResourceProc1 == 1 )
      {
        if ( ProcessNum == 1 )
        {
          ResourceProc2 = 0;
          // повторять, пока ресурс используется процессом 1
          while ( ProcessNum == 1 );
          ResourceProc2 = 1;
        }
      }
      < Использование общего ресурса >
      ProcessNum = 1;
      ResourceProc2 = 0;
    }
 }
```

Алгоритм Деккера гарантирует корректное решение проблемы взаимоисключения для двух процессов. Управляющие переменные ResourceProc1, ResourceProc2 обеспечивают взаимоисключение, переменная ProcessNum исключает возможность бесконечного откладывания. Если оба процесса пытаются получить доступ к ресурсу, то процесс, номер которого указан в ProcessNum, продолжает проверку возможности доступа к ресурсу (внешний цикл ожидания ресурса). Другой же процесс в этом случае снимает свой запрос на ресурс, ожидает своей очереди доступа к ресурсу (внутренний цикл ожидания) и возобновляет свой запрос на ресурс.

Алгоритм Деккера может быть обобщен на случай произвольного количества процессов, однако, такое обобщение приводит к заметному усложнению выполняемых действий. Кроме того, программное решение проблемы взаимоисключения процессов приводит к нерациональному использованию процессорного времени ЭВМ (процессу, ожидающему освобождения ресурса, постоянно требуется процессор для проверки возможности продолжения – активное ожидание (busy wait)).

Семафоры Дейкстры

Под семафором S обычно понимается переменная особого типа, значение которой может опрашиваться и изменяться только при помощи специальных операций P(S) и V(S), реализуемых в соответствии со следующими алгоритмами:

операция P(S)

```
если S>0
  то S = S - 1
иначе < ожидать S >
```

операция V(S)

```
если < один или несколько процессов ожидают S >
  то < снять ожидание у одного из ожидающих процессов >
иначе  S = S + 1
```

Принципиальным в понимании семафоров является то, что операции **P(S)** и **V(S)** предполагаются неделимыми, что гарантирует взаимоисключение при использование общих семафоров (для обеспечения неделимости операции обслуживания семафоров обычно реализуются средствами операционной системы).

Различают два основных типа семафоров. Двоичные семафоры принимают только значения 0 и 1, область значений общих семафоров – неотрицательные целые значения. В момент создания семафоры инициализируются некоторым целым значением.

Семафоры широко используются для синхронизации и взаимоисключения процессов. Так, например, проблема взаимоисключения при помощи семафоров может иметь следующее простое решение.

```
Semaphore Mutex=1; // семафор взаимоисключения процессов

Process_1()
{
  while (1)
  {
    // проверить семафор и ждать, если ресурс занят
    P(Mutex);
    < Использование общего ресурса >
    // освободить один из ожидающих ресурса процессов
    // увеличить семафор, если нет ожидающих процессов
    V(Mutex);
  }
}

Process_2()
{
  while (1)
   {
     // проверить семафор и ждать, если ресурс занят
     P(Mutex);
     < Использование общего ресурса >
     // освободить один из ожидающих ресурса процессов
     // увеличить семафор, если нет ожидающих процессов
     V(Mutex);
   }
}
```

Приведенный пример рассматривает взаимоисключение только двух процессов, но, как можно заметить, совершенно аналогично может быть организовано взаимоисключение произвольного количества процессов

Понятие тупика

В самом общем виде тупик может быть определен как ситуация, в которой один или несколько процессов ожидают какого-либо события, которое никогда не произойдет. Важно отметить, что состояние тупика может наступить не только вследствие логических ошибок, допущенных при разработке параллельных программ, но и в результате возникновения тех или иных событий в вычислительной системе (выход из строя отдельных устройств, нехватка ресурсов и т.п.). Простой пример тупика может состоять в следую-

щем. Пусть имеется два процесса, каждый из которых в монопольном режиме обрабатывает собственный файл данных. Ситуация тупика возникнет, например, если первому процессу для продолжения работы потребуются файл второго процесса и одновременно второму процессу окажется необходимым файл первого процесса (см. рис.3).

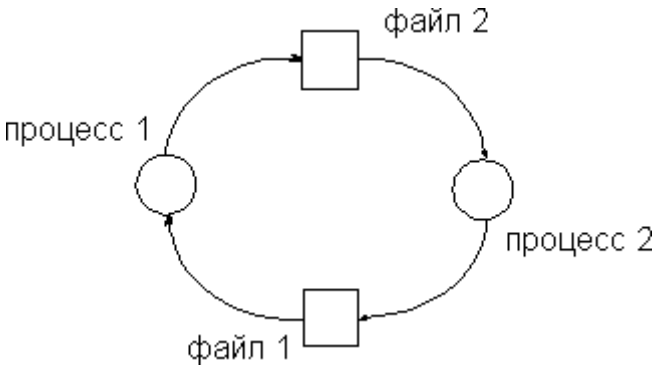

Рисунок 3- Пример ситуации тупика

Проблема тупиков имеет многоплановый характер. Это и сложность диагностирования состояния тупика (система выполняет длительные расчеты или "зависла" из-за тупика), и необходимость определенных специальных действий для выхода из тупика, и возможность потери данных при восстановлении системы при устранении тупика.

В данном разделе будет рассмотрен один из аспектов проблемы тупика – анализ причин возникновения тупиковых ситуаций при использовании разделяемых ресурсов и разработка на этой основе методов предотвращения тупиков.

Могут быть выделены следующие необходимые **условия тупика**:

- процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются (условие *взаимоисключения*);
- процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (условие *ожидания ресурсов*);
- ресурсы нельзя отобрать у процессов, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (условие *неперераспределяемости*);
- существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу цепи (условие *кругового ожидания*).

Как результат, для обеспечения отсутствия тупиков необходимо исключить возникновение, по крайней мере, одного из рассмотренных условий. Далее будет предложена модель программы в виде графа "процесс-ресурс", позволяющего обнаруживать ситуации кругового ожидания

Определение состояния системы

Состояние программы может быть представлено в виде ориентированного графа (V,E) со следующей интерпретацией и условиями:

1. Множество V разделено на два взаимно пересекающихся подмножества P и R, представляющие процессы $P = (p_1, p_2, \cdots, p_n)$ и *ресурсы* $R = (R_1, R_2, \cdots, R_m)$ программы.
2. Граф является "двудольным" по отношению к подмножествам вершин P и R, т.е. каждое ребро $e \in E$ соединяет вершину P с вершиной R. Если ребро e имеет вид $e = (p_i, R_j)$, то e есть ребро *запроса* и интерпретируется как запрос от процесса p_i на

единицу ресурса R_j. Если ребро e имеет вид $e=(R_j,p_i)$, то e есть ребро *назначения* и выражает назначение единицы ресурса R_j процессу p_i.

3. Для каждого ресурса $R_j \in R$ существует целое $k_j \geq 0$, обозначающее количество единиц ресурса R_j.

4. Пусть $|(a,b)|$ - число ребер, направленных от вершины a к вершине b. Тогда при принятых обозначениях для ребер графа должны выполняться условия:

- Может быть сделано не более k_j назначений (распределений) для ресурса R_j, т.е.

$$\sum_i |(R_j,p_i)| \leq k_j, \quad 1 \leq j \leq m;$$

- Сумма запросов и распределений относительно любого процесса для конкретного ресурса не может превышать количества доступных единиц, т.е.

$$|(R_j,p_i)| + |(p_i,R_j)| \leq k_j, \quad 1 \leq i \leq n, \quad 1 \leq j \leq m.$$

Граф, построенный с соблюдением всех перечисленных правил, именуется в литературе как *граф "процесс-ресурс"*. Для примера, на рис. приведен граф программы, в которой ресурс 1 (файл 1) выделен процессу 1, который, в свою очередь, выдал запрос на ресурс 2 (файл 2). Процесс 2 владеет ресурсом 2 и нуждается для своего продолжения в ресурсе 1.

Состояние программы, представленное в виде графа "процесс-ресурс", изменяется только в результате *запросов, освобождений* или *приобретений* ресурсов каким-либо из процессов программы.

Запрос. Если программа находится в состоянии S и процесс не p_i имеет невыполненных запросов, то p_i может запросить любое число ресурсов (в пределах ограничения 4). Тогда программа переходит в состояние T

$$S \xrightarrow{i} T$$

Состояние T отличается от S только дополнительными ребрами запроса от p_i к затребованным ресурсам.

Приобретение. Операционная система может изменить состояние программы S на состояние T в результате операции приобретения ресурсов процессом p_i тогда и только тогда, когда p_i имеет запросы на выделение ресурсов и все такие запросы могут быть удовлетворены, т.е. если

$$\forall R_j : (p_i,R_j) \in E \quad \Rightarrow \quad (p_i,R_j) + \sum_i |(R_j,p_i)| \leq k_j.$$

Граф T идентичен S за исключением того, что все ребра запроса (p_i,R_j) для p_i обратны ребрам (R_j,p_i), что отражает выполненное распределение ресурсов.

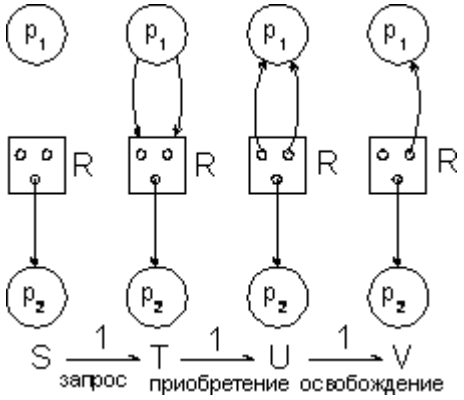

Рисунок 4-Пример переходов программы из состояния в состояние

Освобождение. Процесс p_i может вызвать переход из состояния S в состояние T с помощью освобождения ресурсов тогда и только тогда, когда p_i не имеет запросов, а имеет некоторые распределенные ресурсы, т.е.

$\forall R_j : (p_i, R_j) \notin E$.

В этой операции p_i может освободить любое непустое подмножество своих ресурсов. Результирующее состояние T идентично исходному состоянию S за исключением того, что в T отсутствуют некоторые ребра приобретения из S (из S удаляются ребра (R_j, p_i) каждой освобожденной единицы ресурса R_j).

Для примера на рис. показаны состояния программы с одним ресурсом емкости 3 и двумя процессами после выполнения операций запроса, приобретения и освобождения ресурсов для первого процесса.

При рассмотрении переходов программы из состояния в состояние важно отметить, что поведение процессов является недетерминированным – при соблюдении приведенных выше ограничений выполнение любой операции любого процесса возможно в любое время

Описание возможных изменений программы

Определение состояния программы и операций перехода между состояниями позволяет сформировать модель параллельной программы следующего вида.

Под программой будем понимать систему

$\langle \Sigma, P \rangle$,

где $\{\Sigma\}$ есть множество состояний программы (S, T, U,…), а P представляет множество процессов $(p_1, p_2, \cdots, p_n)$. Процесс $p_i \in P$ есть частичная функция, отображающая состояния программы в непустые подмножества состояний

$p_i : \Sigma \to \{\Sigma\}$,

где $\{\Sigma\}$ есть множество всех подмножеств Σ. Обозначим множество состояний, в которые может перейти программа при помощи процесса p_i (область значений процесса p_i) при нахождении программы в состоянии S через $p_i(S)$. Возможность перехода программы из состояния S в состояние T в результате некоторой операции над ресурсами в процессе p_i (т.е. $T \in p_i(S)$) будем пояснять при помощи записи

$S \xrightarrow{i} T$.

Обобщим данное обозначение для указания достижимости состояния T из состояния S в результате выполнения некоторого произвольного количества переходов в программе

$$S \xrightarrow{*} T \quad \Leftrightarrow \quad (S = T) \vee (\exists p_i \in P : S \xrightarrow{i} T) \vee (\exists p_i \in P, U \in \Sigma : S \xrightarrow{i} U, U \xrightarrow{*} T)$$

Введение в Java

В данном разделе рассматриваются особенности создания параллельных приложений на Java. Чтобы освоить материал данного раздела необходимо иметь представление о синтаксисе и особенностях языка.

Параллелизм с точки зрения ОС

Параллельные программы появились достаточно давно. Даже такие примитивные системы как MS-DOS, были способны параллельно выполнять несколько программ. Конечно, если в машине всего один процессор, то полного параллелизма, конечно, достичь не удастся. Процессор в каждый момент времени может выполнять команды только одной программы. Тут уже задача операционной системы обеспечить правильное планирование выполнения различных программ, так чтобы и различные устройства (диск, процессор) использовались эффективно, и каждой программе «честно» бы выделялся свой квант времени.

Классический алгоритм планирования выполнения программ использует одну или несколько очередей с задачами, готовыми к выполнению. Каждой задаче назначается свой приоритет выполнения. Обычно интерактивным задачам (задачам, взаимодействующим с пользователями) следует назначать более высокий приоритет по сравнению с преимущественно вычислительными задачами. Также высокий приоритет нужен критичным по времени задачам, таким, как управление записью на компакт диск или проигрыватель музыки. Приоритет программы может меняться в процессе работы – давно находящейся в режиме ожидания задаче можно поднять приоритет, а активно выполняющейся – наоборот снизить. Это способствует более «честному» планированию задач и быстрой реакции на команды пользователя. Планировщик задач выбирает задачу из начала списка наиболее высокоприоритетных задач, выделяет ей квант времени для выполнения и передаёт ей управление. Далее возможно следующее:

- Программа полностью отрабатывает свой квант времени. Тогда планировщик заносит её в конец списка готовых к выполнению задач и выбирает для выполнения следующую задачу с наибольшим приоритетом.
- Программа инициирует операцию, которая не может быть немедленно выполнена (например, чтение с диска). В этом случае планировщик переносит задачу в список ждущих задач и выбирает следующую, готовую для выполнения. По окончании операции задача будет опять помещена в список готовых к выполнению.

Происходит системное прерывание или появляется задача с большим приоритетом, готовая к выполнению. Планировщик определяет, может ли программа продолжить работу или следует запустить более высокоприоритетный процесс.

Подобная схема ещё называется «преимущественной многозадачностью» (preemptive multitasking). При такой схеме никакая задача не может полностью блокировать выполнение задач с таким же или большим приоритетом. Некоторые старые ОС (например, Windows 95) имели более простую в реализации модель параллелизма (называемую «не выталкивающей многозадачностью»), когда перепланирование процессов происходит только при желании активного процесса. Т.е. задача должна сама информировать ОС, что её можно прервать. Естественно не правильно работающая программа при такой схеме может полностью блокировать работу системы, что и является следствием гораздо менее стабильной работы Win 95, по сравнению, например, с семейством Windows NT.

Поток

Итак, параллельные процессы используются уже достаточно давно. Почему же их нельзя использовать для решения наших задач? Ответ: можно, но неудобно. Дело в том, что процесс представляет собой достаточно замкнутую систему. У процесса есть своя память (команд и данных), свой стек, свои дескрипторы для доступа к файлам и другим устройствам и т.д. При этом средства взаимодействия процессов между собой достаточно ограничены предоставляемым ОС API. Обычно средства межпроцессорного взаимодействия включают в себя очереди для обмена сообщениями между процессами, разделяемую память (shared memory) и синхронизационные примитивы: семафоры, события, критические секции. Т.е. если вы хотите выполнить фоновое сохранение данных на диск, вам нужно будет создать новый процесс, использовать один из существующих механизмов межпроцессорного взаимодействия (IPC) для передачи этому процессу имени файла и данных, которые должны быть сохранены, после этого, опять-таки с использованием IPC, получить от процесса уведомление о завершении операции. В общем, простейшая операция выливается во множество строчек кода.

Проблема ещё состоит в том, что, так как с процессом связано достаточно много данных, то и порождение процесса и переключение контекста процессов представляет собой достаточно трудоёмкую операцию. Таким образом, использовать различные процессы для распараллеливания работы можно только если эти работы слабо связаны с друг другом. В этом случае процессы могут работать почти независимо друг от друга. Если же для выполнения необходим интенсивный обмен данными, то накладные расходы от межпроцессного взаимодействия и переключения контекстов часто оказываются настолько большими, что всякий выигрыш от параллельной работы процессов теряется.

Поэтому не удивительно, что почти во всех современных операционных системах поддерживаются облегчённые процессы или потоки **(threads).** На самом деле реализации потоков в разных ОС может существенно отличаться. Но мы не будем здесь заострять на этом внимание. Определим поток как самостоятельную активность внутри процесса. Поток имеет свой стек, но не имеет своей собственной памяти. Вместо этого все потоки в процессе разделяют общую память (т.е. один поток может получить доступ к данным другого потока). Это значительно упрощает передачу данных между потоками (*механизм передачи данных ничем не отличается от передачи параметров в подпрограмму*). Но при этом, в отличие от процесса, данные которого изолированы от других процессов, данные потока могут быть изменены любым другим потоком, что, конечно, требует повышенного внимания при программировании многопоточных приложений и аккуратного использования синхронизации.

За счёт того, что у потока гораздо меньше собственных ресурсов, чем у процесса, создание потока требует гораздо меньше времени, чем запуск процесса. Также переключение контекста между различными потоками в рамках одного процесса выполняется гораздо быстрее, чем переключение между различными процессами (все потоки внутри одного процесса имеют общую память, поэтому при переключении контекста не надо менять таблицу отображения страниц).

Планирование выполнения потоков система осуществляет точно так же, как планирование процессов. Точнее, во многих операционных системах, поддерживающих потоки, единицей планирования является именно поток, а не процесс. Важно отметить, что операционная система способна организовать параллельное выполнение потоков: либо распределением потоков по различным процессорным устройствам (на многопроцессорной машине), либо используя перепланировку по истечении отведённого потоку кванта времени. В обоих случаях надо быть готовым к тому, что выполнение потока может быть прервано в любой момент и управление передано другому потоку (это не совсем так – есть понятие атомарной, т.е. неделимой операции, но какая операция атомарная, а какая нет, зависит от архитектуры системы, используемого языка, типа данных и даже опций компилятора, по этому в этой работе эта проблема не обсуждается).

Потоки в Java

Механизм потоков в Java использует усовершенствованную схему мониторов Хоара. Потоку соответствует класс **java.lang.Thread.** Для создания своего потока можно либо унаследовать свой класс из **Thread,** либо реализовать интерфейс **Runnable**. И в том, и в другом случае необходимо реализовать собственный метод **run(),** который, собственно, и будет выполнять нужное действие. Запустить поток на выполнение можно методом **start().** Если необходимо дождаться завершения потока, надо использовать метод **join().** Ниже приведены примеры для двух способов создания потока:

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Thread is running");
  }
}

  public static void main(String args[])
  {
    MyThread thread = new MyThread();
    thread.start();
    try {
       thread.join();
    } catch (InterruptedException x) {}
 }
```

или

```
class MyActivity implements Runnable {
  public void run() {
     System.out.println("Thread is running");
  }
}

public static void main(Strnig args[]) {
  MyActivity myActivity = new MyActivity();
  Thread thread = new Thread(myActivity);
  thread.start();
  try {
     thread.join();
  } catch (InterruptedException x) {}
}
```

А что будет, если не дожидаться завершения потока? Программа будет активна, пока работает по крайней мере один поток. Т.е., другими словами, программа ждёт завершения всех запущенных потоков. Иногда это не нужно. В этом случае надо пометить поток как «демон» с помощью метода **setDaemon(true).** Завершения "демонов" программа не дожидается и просто их останавливает. Вызвать **setDaemon** метод нужно до запуска потока.

Но все потоки разделяют общую память. Чем это чревато. Допустим, мы пишем банковскую систему для обслуживания банкоматов, и у нас есть метод, который проверяет, достаточно ли у клиента денег на счету и, если да, снимает запрошенную сумму со счёта и даёт банкомату "добро" на выдачу денег:

```
class Account {
   private int balance;

   public boolean withdraw(int amount)  {
      if (amount > balance) {
```

```
            return false;
        }
        balance -= amount;
        return true;
    }
}
```

Так как банкоматов много, следует реализовать параллельную обработку запросов и для этого использовать потоки. Вроде бы всё прекрасно работает (в этом и заключается одна из основных проблем параллельного программирования – ошибки очень трудно воспроизводимы). Теперь внимательно проанализируем код. Итак, метод withdraw() может параллельно выполняться несколькими потоками. Как мы уже отмечали, поток может быть прерван в любой момент времени. Поэтому возможет такой сценарий:

1. Имеется два потока П1 и П2, в которых вызывается метод withdraw() Пусть на счету находится сумма в 100уе, П1 запрашивает 80уе, а П2 – 60уе.
2. Поток П1 проверяет условие (amount < balance). Оно истинно (80 < 100).
3. Происходит перепланировка потоком, управление получает поток П2
4. Поток П2 проверяет условие (amount < balance). Оно тоже истинно (60 < 100).
5. Поток П2 уменьшает баланс на затребованную величину и возвращает true. Теперь баланс равен 40уе.
6. Управление возвращается потоку П1.
7. Он тоже уменьшает значение баланса и возвращает true. Баланс равен… -40уе!

Но так с деньгами обращаться нельзя. Параллелизм - это конечно здорово, но целостность данных от этого страдать не должна. Кусок кода программы, в котором недопустимо влияние других потоков, называется критической секцией. В критическую секцию дозволено входить только одному потоку. Остальные будут ждать, пока этот поток не покинет критическую секцию. В языке Java для оформления критической секции предусмотрено ключевое слово **synchronized**(синхронизированный). Можно объявить синхронизированными как весь метод, так и отдельный блок операторов:

```
boolean synchronized withdraw(int amount)  {
   if (amount > balance) {
        return false;
   }
   balance -= amount;
   return true;
}
```

или

```
boolean withdraw(int amount)  {
   synchronized(this) {
      if (amount > balance) {
         return false;
      }
      balance -= amount;
      return true;
   }
}
```

В данном случае оба способа эквивалентны. Первый представляется более удобным, а второй обеспечивает большую гибкость.

Что означает аргумент **this** в конструкции **synchronized(this**)? В Java с каждым объектом неявно связан монитор – т.е. нечто, отвечающее за синхронизацию доступа к данному объекту. Вся синхронизация в Java осуществляется на уровне объектов. Т.е. внутри **synchronized** блока блокируется данный экземпляр объекта. Метод, помеченный

как **synchronized**, осуществляет блокировку **this** объекта для методов экземпляра класса или блокировку самого класса для статических методов. Т.е объявляя метод как **synchronized**, мы говорим системе, что этот метод требует эксклюзивного доступа к объекту. И система гарантирует, что из всех синхронизированных участков кода, для каждого конкретного экземпляра объекта в каждый момент времени может выполняться только один такой участок. При этом:

- Синхронизированный метод может выполняться параллельно для разных объектов, например – метод **withdraw** может параллельно производить операции с различными счетами.
- Для одного экземпляра запрещено параллельное выполнение не только одного и того же синхронизированного метода, но и любых других методов и синхронизированных блоков, использующих данный экземпляр объекта. Например, если бы у нас в классе **Account** был синхронизированный метод **deposit** (положить деньги на счёт), то и он бы не мог выполняться параллельно с методом **withdraw** для данного счёта.
- Конструкция **synchronized** не препятствует параллельному выполнению методов, не отмеченных как синхронизированные. Т.е., если бы в нашем классе **Account** был метод deposit(), который мы забыли бы пометить как **synchronized,** то он вполне мог выполняться параллельно с методом **withdraw().** Причём, последствия могли бы быть не менее печальными, чем при выполнении двух параллельных **withdraw().**

Итак, мы научились предотвращать нежелательное влияние потоков друг на друга. Но этого ещё недостаточно. Часто хочется уметь оповещать другой поток, например, о готовности данных для него. И, соответственно, ждать прихода такого сообщения. Для этого в **Java** в классе **Object** предусмотрены методы **wait, notify, notifyAll**. Для использования этих методов поток должен быть эксклюзивным владельцем объекта, т.е. использовать эти методы для **this** объекта внутри метода, объявленного как **synchronized,** или внутри **synchronized** блока с этим объектом в качестве параметра. Эти методы работают следующим образом:

- При выполнении метода **wait** система снимает блокировку с объекта и переводит поток в режим ожидания.
- При выполнении метода **notify** система переводит в точности один поток, ожидающий наступления данного события, в режим готовности. Если такого потока нет, то выполнение этого метода не имеет никакого эффекта. Следует заметить, что если наступления события ожидают несколько потоков, то будет выбран **любой** из них. При этом, для возврата из метода **wait,** потоку придётся конкурировать за доступ к данному объекту с другими потоками. Только установив вновь эксклюзивную блокировку объекта, ожидающий поток может вновь продолжить выполнение.
- При выполнении метода **notifyAll** пробуждаются все потоки, ждущие наступления данного события.

Кажется не совсем понятным, зачем требуется блокировать объект перед вызовом **wait**, чтобы потом **wait** снял эту блокировку и пытался установить её вновь после получения уведомления. Но это сделано для того, чтобы проверка условия и переход в режим ожидания выполнялись атомарно (не делимым образом). Обычно поток, ожидающий наступления какого-то события, проверяет переменную, связанную с этим событием. Если результат проверки отрицательный, то поток переходит в режим ожидания. Например, простейший класс "событие" (event), можно реализовать на Java следующим образом:

```
class AutoResetEvent { // событие с автосбросом

    public synchronized void waitEvent() {
       try {
           while (!signaled) { // ждать наступления события
               wait();
           }
```

```
            signaled = false;
        } catch (InterruptedException x) {}
    }

    public synchronized void signalEvent() {
        signaled = true;
        notify(); // пробудить спящий поток
    }

    private boolean signaled;
}
```

Обратите внимание, что проверка условия **signaled** делается в цикле. Это необходимо, т.к., после того как **signaled** был установлен в true методом **signal** и ждущий поток помещён в очередь готовых к выполнению процессов, поток, вызвавший **wait**, ещё не стал владельцем блокировки объекта. Поэтому, если в данный момент другой поток вызовет метод **waitEvent**, вполне может быть, что именно он завладеет блокировкой и продолжит выполнение. Так как переменная **signaled** имеет значение **true**, то этот поток не будет ждать, а сбросит **signaled** в false и продолжит выполнение. А поток, выполнивший **wait** и завладевший наконец блокировкой, обнаружит, что signaled сброшен. Поэтому он вынужден будет выполнить ещё одну итерацию цикла и опять ждать наступления следующего события.

Следует также заметить, что метод **wait** снимает блокировку только с одного объекта (а именно с того, для которого был позван метод **wait**). Если вы заблокировали какие-либо еще объекты, то они так и останутся заблокированными на время, пока поток находится в состоянии ожидания. Возможно, вы того и хотели, но чаще всего это свидетельствует об ошибке в программе. Для повышения степени параллелизма (и, соответственно, производительности) и избежания тупиковых ситуаций следует устанавливать блокировки на как можно более короткий срок. Следующий пример иллюстрирует проблему:

```
class MyClass {
   void synchronized foo(Object event) {
      synchronized(event) {
          event.wait();
      }
   }

}
```

В этом примере метод **wait** снимает блокировку с объекта **event**, но собственный объект (**this**) остаётся заблокированным.

А вот пример реализации на Java события с ручным (явным) сбросом:

```
class ManualResetEvent { // событие с автосбросом

    public synchronized void waitEvent() {
        try {
             while (!signaled) { // ждать наступления события
                  wait();
             }
        } catch (InterruptedException x) {}
    }

    public synchronized void resetEvent() {
        signaled = false;
    }
```

```
    public synchronized void signalEvent() {
        signaled = true;
        notifyAll(); // пробудить все спящие потоки
    }

    private boolean signaled;

}
```

И еще один пример - реализация семафора на Java. Семафор - это классический синхронизационный примитив, предложенный Э.В. Дейкстрой, который можно использовать для распределения некоторого ограниченного количества ресурсов. У классического семафора есть две операции **P**(занять ресурс) и **V**(освободить ресурс). Операция **P** проверяет счётчик доступных ресурсов и, если он больше нуля, то вычитает единицу и возвращает управление. В противном случае поток, выполняющий операцию **P**, блокируется до тех пор, пока счётчик не станет больше нуля. А операция **V** используется для оповещения о том, что ресурс потоку больше не нужен. При этом счётчик доступных ресурсов увеличивается на единицу.

```
class Semaphore {

    public synchronized void p() {
       while (counter == 0) {
           wait();
       }
       counter -= 1;
    }

    public synchronized void v() {
        counter += 1;
        notify();
    }
}
```

Реализация потоков в Java зависит от используемой виртуальной машины. В некоторых случаях используется реализация потоков на библиотечном уровне. То есть всё управление потоками осуществляется внутри самой виртуальной машины без участия операционной системы. Такие потоки имеют наименьшие накладные расходы, так как для переключения контекста не требуется системных вызовов. Но такая реализация не позволяет использовать все имеющиеся ресурсы на многопроцессорной машине и параллельно выполнять несколько потоков. Кроме того, в большинстве случаев такая реализация использует не вытесняющую стратегию для перепланировки потоков. То есть, для того, чтобы управление было передано другому потоку, активный в данный момент поток должен сам захотеть этого (например, вызов блокирующего системного метода приводит к тому, что поток переводится в очередь ждущих потоков, а из очереди готовых к выполнению потоков выбирается новый поток). Поэтому большинство текущих реализаций виртуальных машин использует системные потоки (например, Win32 или pthread) для реализации над ними потоков Java.

Простота конструкций, используемых в Java для управления потоками, создаёт иллюзию, что написание многопоточных приложений ничуть не сложнее написания обычных приложений: расставил всюду **synchronized,** вставил где необходимо **wait/notify,** создал нужное количество потоков и всё - многопоточная программа готова. По сравнения с библиотеками, используемыми, например, в C++ для создания и управления потоками, в Java всё кажется простым и понятным. К сожалению, эта простота кажущаяся. Проблемы, возникающие при написании параллельных программ, никуда не исчезли.

Отладка многопоточных программ

Как происходит отладка программы на C/C++? "Гуляющие указатели", не инициализированные переменные, содержащие мусор, висящие ссылки и утечки памяти, не от-

лавливаемые выходы индекса за границу массива… Программа "грохающаяся" в непредсказуемом месте, потому что совсем в другом участке программы кто-то неправильно обошёлся с указателем. Программа то работающая, то не работающая (в зависимости от того, какой мусор встретился в памяти. Не даром для C++ создано столько всяких CodeGuardов и BoundsCheckerов. После этого ошибки в программах на языке Java кажутся чуть ли не самоустраняющимися - все переменные инициализированы, выход индекса за границу массива отслеживается и пресекается мгновенно, гуляющих и висячих указателей нет в принципе, как и утечек памяти. Программа ведёт себе абсолютно детерминированным образом - если она "сваливалась" на каком-то определённом наборе входных данных, то можно быть уверенным, что, будучи запущенной ещё раз с тем же набором данных, мы получим тот же самый результат. В общем, полное торжество защищённых программных систем и языков.

Но вот мы написали многопоточную программу. И что мы видим? Опять программа ведёт себя абсолютно непредсказуемым образом. Будучи отлажена, оттестирована и запущена 1000 раз, на тысячу первый она ни с того ни с сего виснет. Или ещё хуже - вдруг оказываются "испорченными" данные. Попытка воспроизвести ошибку ни к чему не приводит - программа отлично работает...до следующего падения. Случаются и не столь катастрофические, но не менее загадочные явления - программа, прекрасно работающая на однопроцессорной машине, на вдвое более мощной двухпроцессорной машине вдруг начинает работать на порядок (десятичный, а не двоичный) медленнее, чем на однопроцессорной. В общем, мы опять оказываемся в первобытном дремучем лесу отладки недетерминировано работающей программы. С той лишь разницей, что продуктов типа BoundsCheckera для поиска ошибок в синхронизации очень мало, да и качество их работы оставляет желать лучшего

На самом деле, все ошибки, приводящие к неправильной работе многопоточного приложения можно разбить на два класса: конкурентный доступ (английский термин "race condition" не поддаётся разумному переводу) и взаимные блокировки. Ошибки первого рода являются следствием недостатка синхронизации, второго - её избытка (или неправильного применения).

С ошибками первого рода мы уже сталкивались - пример с банковским счётом. Если два или больше потоков начинают одновременно изменять (или даже один изменяет, а другой только смотрит) одни и те же данные, то обычно ничего хорошего из этого не получается. Что же делать? Понаставить всюду synchronized? Во-первых, тогда весь выигрыш от многопоточности может испариться - если все объекты доступны только в эксклюзивном режиме, то работать в каждый момент времени сможет только один поток. А во вторых, чрезмерное и необдуманное использование блокировок приводит к проблемам второго рода - тупиковым ситуациям. Рассмотрим следующий пример:

```
class Pipe { // канал

 Queue src; // источник
 Queue dst; // приёмник

 void forward() { // переметить элемент из источника в приёмник
    synchronized(src) { // блокируем приёмник
      synchronized(dst) { // блокируем приёмник
         Object elem = src.dequeue(); // взять элемент из очереди src
         dst.enqueue(elem); // и поместить в очередь dst
   }
      }
 }

 void backward() { // переместить элемент в обратном направлении –
                   // из приёмника в источник
   synchronized(dst) { // блокируем приёмник
      synchronized(src) { //  блокируем приёмник
```

```
            Object elem = dst.dequeue(); // взять элемент из очереди dst
            src.enqueue(elem); // и поместить в очередь src
        }
    }
  }
}
```

Допустим, что с данным классом параллельно работают несколько потоков. При этом возможен такой сценарий:

1. Поток П1 вызывает метод forward.
2. В потоке П1 выполняется первый synchronized оператор и устанавливается блокировка src.
3. В результате перепланировки потоков управление получает поток П2.
4. Поток П2 вызывает метод backward.
5. В результате выполнения метода backward в потоке П2 происходит блокирование объекта dst.
6. Попытка блокировать объект src не удаётся, так как этот объект уже блокирован П1. Поток П2 переходит в режим ожидания.
7. Управление вновь получает П1 и пытается блокировать dst. И эта попытка оканчивается неудачей, так как dst уже блокирован П2. Потоку П1 ничего не остаётся, как ждать...потока П2, который в свою очередь ждёт П1. Выйти из этой ситуации потоки не смогут. Поэтому она и зовётся тупиком.

В данном случае проблема решается достаточно просто: нужно, чтобы метод **backward** блокировал объекты в том же порядке, что и метод **forward.** Тупиковая ситуация при этом возникнуть не может. Этот простой прием, кстати, является одним из самых основных способов предотвращения тупиковых ситуаций - старайтесь всегда блокировать объекты в одном и том же порядке. Можно также ассоциировать с объектами-ресурсами приоритеты, и блокировать объекты в порядке убывания приоритета (иерархические блокировки). К сожалению, тупиковую ситуацию можно получить, даже если у вас есть один единственный метод с **synchronized** атрибутом. Рассмотрим следующий пример:

```
class SomeClass {
    void synchronized foo(SomeClass a) {
        ...
        a.foo();
        ...
    }
}
```

Теперь допустим, что у нас есть объекты класса SomeClass **o1** и **o2** и два потока **П1** и **П2**, параллельно выполняющие следующие вызовы:

```
П1: o1.foo(o2);
П2: o2.foo(o1);
```

С большой вероятностью этот код приведёт к взаимной блокировке: П1 блокирует o1, П2 блокирует o2 и оба будут ждать друг друга).

Что же делать? Как научиться определять возможные блокировки и пытаться избегать их? К сожалению, Java компилятор нам тут ничем не поможет. Придётся брать в руки карандаш и бумагу и начинать рисовать граф блокировок. Вершинам в этом графе соответствуют объекты, которые могут быть заблокированы (объект синхронизации в **synchronized** блоке или экземпляр класса для **synchronized** метода). Конечно, этих объектов может быть неопределённо много (например, в случае **synchronized** метода, каждый экземпляр объекта данного класса может быть заблокирован). Для начала, представим все экземпляры класса одной вершиной. Итак, вершины мы нарисовали. Теперь внимательно

изучаем код и ищем зависимости между объектами. Для этого строим граф вызовов, т.е. определяем, какие другие методы может позвать данный метод. Нужно построить замыкание этого отношения (т.е. иными словами учесть не только те методы, которые могут быть непосредственно позваны, но и методы, которые позовут эти методы, и.т.д.) Не забудьте учесть наследование - в общем случае вызов метода базового класса или интерфейса может привести к вызову метода любого выведенного класса, переопределяющего или реализующего данный метод. Граф вызовов для большой программы может получиться просто непотребно большим, поэтому лучше его не рисовать явно, а строить в уме. Но тут главное никого не забыть. После того, как мы выяснили, кто кого может позвать, проводим рёбра в нашем графе блокировок. Если внутри участка кода, блокировавшего объект **"x",** блокируется объект **"y"** или вызывается метод, который в свою очередь блокирует объект **"y"**, то от вершины, помеченной **"x"**, мы проводим ребро в вершину помеченную **"y".**

Теперь изучаем наш граф и ищем в нём циклы. Не нашли - поздравляю: возникновение тупиковой ситуации в вашей программе невозможно (если вы, конечно, не напутали что-то с графом). Нашли? - ну тут возможны варианты. Помните, что мы все экземпляры данного класса пометили одной единственной вершиной? Может так оказаться, что цикл, существующей в нашем графе на самом деле невозможен, потому что блокируются различные экземпляры объектов. Тут уже нужно более тонкое изучение, использующее семантику конкретной программы.

К счастью, в отличие от конкурентного доступа, тупиковая ситуация (будучи воспроизведённой под отладчиком), хорошо поддаётся анализу. Просто смотрим список "застрявших" потоков, изучаем стек вызова каждого такого потока и определяем, какие объекты данный поток блокировал и какие пытается заблокировать. После чего надо только понять, как разрушить цикл.

К сожалению, для поиска ошибок связанных с конкурентным доступом сложно дать какие-то рекомендации. Очевидно, что каждые разделяемые и изменяемые данные должны быть защищены от конкурентного доступа. То есть синхронизация имеет смысл только в том случае, если все потоки используют для синхронизации одни и те же объекты. Если поток **П1** для обращения к объекту о блокировал объект **m1**, а поток **П2** для обращения к тому же объекту о использует блокировку другого объекта **m2**, то ничего хорошего от такой "синхронизации" не получится. Поэтому можно попробовать для каждого совместно используемого объекта или переменной определить объект, который отвечает за синхронизацию доступа к нему. Если получилось больше одного синхронизирующего объекта для какого-то совместно используемого элемента - то, скорее всего, тут ошибка.

И ещё одна рекомендация по воспроизведению ошибок. Хотя, как мы уже говорили, даже на однопроцессорной машине поток может быть прерван в любой момент и поэтому возможно параллельное выполнение почти любых операторов (если только они не находятся в критической секции), на многопроцессорной машине вероятность этого значительно больше. Кроме того, тут так же сказывается параллелизм на уровне инструкций - инструкции, который могли рассматриваться как атомарные в однопроцессорной конфигурации, перестают быть такими в многопроцессорной системе. Поэтому следует прогнать свою программу на многопроцессорной системе - возможно гораздо быстрее наткнётесь на скрытые ошибки. Кроме того, при прогоне на многопроцессорной машине могут проявиться проблемы с падением производительности, вызванные конфликтами блокировок разных потоков. В этом случае придётся либо менять схему блокировок, либо изменять распределение данных между потоками, чтобы свести к минимуму использование глобальных (разделяемых) данных, доступ к которым требует синхронизации.

Проблема остановки

Запуск потока в Java не представляет собой проблемы – достаточно создать объект **Thread** и позвать метод **start.** Однако, оказывается, что, запустив поток на выполнение,

его не так-то просто остановить. Хотя метод **stop** в классе **Thread** есть, но он помечен как **deprecated** (устаревший). Т.е. пользоваться им настоятельно не рекомендуется. И на то есть веские причины. Дело в том, что поток, остановленный в произвольный момент времени, может оставить в некорректном состоянии системные ресурсы. Например, если поток был владельцем каких либо блокировок, то эти объекты так и окажутся не разблокированными. Что же делать, если «убивать» поток нельзя? Надо заставить его завершиться добровольно. Для этого в потоках, которые выполняют некоторое циклическое действие, в условие цикла надо поставить проверку того, что потоку не пора завершать работу. Например:

```
class MyThread extends Thread {
    boolean running = true;
      public void run() {
      while (running) {
        ... // делаем что-то
      }
    }
}
```

Однако обычно поток не только производит вычисления, но и совершает операции ввода/вывода. Соответственно очень часто поток блокирован в результате выполнения системного вызова, который не может быть выполнен мгновенно – например, **read**.

Чтобы заставить такой поток закончить выполнение, надо воспользоваться методом **interrupt()** класса **Thread.** Если поток был блокирован во время выполнения метода **Object.wait(),** то в потоке будет инициирована исключительная ситуация **InterruptedException.** Если поток выполнял блокирующую операцию ввода/вывода, то выполнение операции будет прервано с помощью исключительной ситуации **java.nio.channels.ClosedByInterruptException.** При этом всё равно полезно использовать флаг остановки, чтобы отличать ситуацию, когда операция прервана в результате ошибки ввода/вывода и когда причина – остановка потока.

Иногда в потоке стоит задержка выполнения на некоторое время. В классе **Thread** имеется метод **Sleep,** который "усыпляет" поток на заданное время в миллисекундах. Но если поток нужно будет останавливать, то более правильным представляется использование метода **Object.wait** с задержкой. В этом случае объекту можно послать уведомление, чтобы прервать его сон:

```
class MyThread extends Thread {
    boolean  running = true;
    Object timer = new Object();
    static final int DELAY=1000; // одна секунда
    public void run() {
         while (running) {
          ...  // делаем что-то
          synchronized(timer) {
            try {
               timer.wait(DELAY); // ждём заданное время,
                                  // либо сигнала о завершении
            } catch (InterruptedException x) {}
          }
         }
    }

    public void terminate() {
         synchronized(timer) {
             running = false; // устанавливаем флаг завершения
             timer.notify(); // и посылаем сигнал, чтобы пробудить спящий
процесс.
         }
    }
```

}

Хотя поток и является более легковесным ресурсом, чем процесс (т.е. создание и переключение потоков требует меньше времени и памяти), но всё равно создание потока достаточно сложная операция, особенно в операционных системах, поддерживающих потоки на уровне ядра (а только в этом случае можно получить выигрыш от использования параллельных потоков на многопроцессорной машине). Поэтому, если программе необходимо периодически выполнять какие-то параллельные действия, неэффективно каждый раз создавать новый поток. Вместо этого лучше завести пул потоков. Идея очень проста: вместо того, чтобы каждый раз создавать новый поток, а потом его завершать, мы заводим некоторое число готовых потоков, которые ждут момента, когда они понадобятся. После выполнения работы поток не завершается, а опять попадает в список свободных потоков и ждёт следующего задания. Посмотрим, как это можно реализовать:

```
/**
 * Пул потоков
 */

public class ThreadPool {
    /**
    * Получить экземпляр пула потоков
    */
    public static ThreadPool getInstance() {
      return theInstance;
    }

   /**
    * Поток для повторного использования
    */

    static class PooledThread extends Thread {
      PooledThread next;
      Object       ready; // объект, используемый для ожидания нового задания
      Object       done;  // объект, используемый для сигнализации завершения
работы
      boolean      busy; // флажок отмечающий, что поток занят
      boolean      doneNotificationNeeded; // ждёт ли кто-нибудь завершения
работы
      Runnable     task; // что нужно сделать
      ThreadPool   pool; // пул потоков

      public void run() {
        try {
          synchronized(ready) { // блокируем ready – готовимся к ожиданию ра-
боты
            while (!pool.closed) { // проверяем не закрыт ли пул
              synchronized(done) { // блокировка done
                busy = false;    // мы закончили работу начатую на предыдущей
                // итерации цикла
                if (doneNotificationNeeded) { // если кто-то ждёт завершения
работы…
                  done.notify();     // то пошлём уведомление, что мы её за-
кончили
                }
              }
              ready.wait();
              // ждём нового задания
              if (task != null) {
                // если мы его получили…
                task.run();
                // то выполняем
```

```
              } else {
                break;
                // иначе завершаем работу
              }
            }
          }
        } catch (InterruptedException x) {}
      }

      final void wakeUp(Runnable t) { // запустить задание на выполнение
        synchronized(ready) { // блокировать ready, чтобы можно было послать
уведомление
          busy = true; // установить флаг занятости
          doneNotificationNeeded = false;
          task = t;
          ready.notify(); // послать уведомление свободному потоку
        }
      }

      final void waitCompletion() throws InterruptedException {
        synchronized(done) { // блокировать done для ожидания
          if (busy) { // если задание ещё не завершилось…
            doneNotificationNeeded = true; // то поставить флаг ожидания за-
вершения
            done.wait(); // и подождать
          }
        }
      }

      PooledThread(ThreadPool pool) {
        this.pool = pool;
        ready = new Object();
        done = new Object();
        busy = true;
        setDaemon(true); // запускаем поток как демона, чтобы не ждать его
завершения
        // при выходе из программы
        this.start();
      }
    }

   /**
    * Найти свободный поток и запустить в нём задание на выполнение\
    * @param task объект реализующий интерфейс Runnable,
    * метод run которого будет выполнен потоком
    * @return поток, выделенный для данного задания
    * (его можно использовать только в методе ThreadPool.join)
    */
    public Thread start(Runnable task)    {
      PooledThread thread;
      synchronized (this) { // для работы со списком нужна блокировка
        while (availableThreadList == null) { // если список свободных пото-
ков пуст
          try {
            if (nActiveThreads == maxThreads) { // и если достигнуто ограни-
чение на
              // максимальное число потоков
              deficit += 1; // то отметить, что есть задачи, ждущие выполне-
ния
              wait();  // и подождать пока какой-нибудь из потоков не освобо-
дится
            } else {
              availableThreadList = new PooledThread(this); // создать новый
поток
              availableThreadList.waitCompletion(); // и дождаться момента,
```

```
            // когда он стартует и будет
            // готов к получению задания
          }
        } catch (InterruptedException x) {
          return null;
        }
      }
      thread = availableThreadList; // взять поток из списка свободных
      availableThreadList = thread.next;
      nActiveThreads += 1; // увеличит счётчик активных потоков
    }
    thread.wakeUp(task); // запустить задание на выполнение
    return thread;
  }

  /**
   * Дождаться завершения задания. Поток при этом возвращается в список сво-
бодных
   * @param thread поток, возвращённый методом ThreadPool.start
   */
  public void join(Thread thread) throws InterruptedException    {
    PooledThread t = (PooledThread)thread;
    t.waitCompletion(); // дождаться завершения задания
    synchronized (this) { // для работы со списком нужна блокировка
      t.next = availableThreadList; // добавить поток в список свободных
      availableThreadList = t;
      nActiveThreads -= 1; // уменьшить значение счётчика активных потоков
      if (deficit > 0) { // если есть потоки ждущие уведомления…
        notify();  // то послать уведомление о том, что поток освободился
        deficit -= 1;
      }
    }
  }

  /**
   * Закрытие пула потоков. Подождать окончания работы всех активных потоков
и
   * завершить  все свободные потоки
   */
  public synchronized void close()    {
    closed = true; // ставим флаг прекращения работы
    while (nActiveThreads > 0) { // пока есть активные потоки
      try {
        deficit += 1; // мы нуждаемся в уведомлении о завершении потока
        wait(); // ждём уведомления
      } catch (InterruptedException x) {}
    }
    while (availableThreadList != null) { // пока список свободных потоков
не пуст
      availableThreadList.wakeUp(null); // потоку запрос на завершение
      availableThreadList = availableThreadList.next; //  и исключаем его
из списка
    }
  }

  /**
   * Конструктор пула потоков с ограничением на максимальное число активных
потоков
   * @param maxThreads максимальное число одновременно работающих потоков.
   */
  public ThreadPool(int maxThreads) {
    this.maxThreads = maxThreads;
  }

  /**
```

```
    * Конструктор пула потоков с неограниченным числом потоков
    */
    public ThreadPool() {
      this(Integer.MAX_VALUE);
    }

    PooledThread availableThreadList; // список свободных потоков
    int          nActiveThreads; // число активных потоков
    int          deficit; // количество потоков, заинтересованных в получении
                          // уведомления о завершении работы
    int          maxThreads; // ограничение на максимальное число одновремен-
но
                             // работающих потоков
    boolean  closed; // флаг прекращения работы

    static ThreadPool theInstance = new ThreadPool();

}
```

Обратите внимание в этом примере на то, что уведомления посылаются не всегда, а только тогда, когда их кто-то ждёт. Подобная оптимизация с использованием дополнительной переменной позволяет избежать лишних вызовов **notify.**

Практические примеры использования потоков

Рассмотрим использование потоков в реальных и достаточно простых приложениях. Будут рассматрены примеры приложений под J2ME (Java для мобильных и встроенных устройств) по причине исключительной простоты MIDP интерфейса. Но способы работы с потоками в J2ME ничем не отличаются от других платформ Java.

Первый пример.

Реализуем научный калькулятор, который должен уметь рисовать графики функций. Для этого следует вычислить значение функции во всех точках указанного интервала с заданным шагом. Их может быть довольно много, а вычисление функции (особенно на слабеньком процессоре мобильного телефона), может занимать много времени. С другой стороны, пока не будут вычислены значение функции для всех точек, нельзя определить максимум и минимум функции на этом интервале и вычислить коэффициент для вертикального масштабирования графика функции (чтобы график поместился на экран). Конечно, можно просто вывести на экран "Подождите минуточку..." и заняться вычислениями. Но это не очень хорошее решение:

- Во-первых пользователь не может прервать процесс вычислений (допустим вам в это время звонят, а у вас телефон занят вычислениями).
- Во-вторых пользователь не знает сколько ещё времени ему осталось ждать (если бы он знал, что график будут построен только через час, он бы давно остановил работу и задал другой интервал или шаг).

Ясно, что вычисление функции должно производится независимо от работы остальной программы. То есть нам нужен поток:

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class Plot extends Form implements CommandListener, Runnable  {
    Calculator calculator;
    Gauge      progressIndicator; // "градусник"
    StringItem completed; // а сюда мы будем записывать строчку с процентом
выполнения
    Float      from; // начальная точка
    Float      till; // конечная точка
    Float      dx;   // шаг
    Float[]    values; // сюда мы будем заносить вычисленные значения функции
    int        percent; // процент выполнения
    Thread     thread;  // поток в котором будет всё делаться
```

```
    boolean    running; // флажок нужный для "аварийной" остановки потока
    Compiler.FunctionExpression f; // вычисляемая функция

    Plot(Calculator calculator, Compiler.PlotExpression expr, int screen-
Width) {
      super("Plot");
      this.calculator = calculator;
      from = expr.from.evaluate(calculator.bindings);
      till = expr.till.evaluate(calculator.bindings);
      dx = expr.step != null
           ? expr.step.evaluate(calculator.bindings)
           : till.Sub(from).Div(screenWidth);
      f = expr.f;
      int n = (int)till.Sub(from).Div(dx).toLong() + 1;
      if (n < 2) {
        calculator.showAlert(AlertType.ERROR, "Error", "Bad interval");
        return;
      }
      values = new Float[n];
      i = 0;
      x = from;
      progressIndicator = new Gauge("Progress indicator", false, n, 0);
      completed = new StringItem("Completed:", "0%");
      append(progressIndicator);
      append(completed);
      setCommandListener(this);
      addCommand(Calculator.STOP_CMD);
      Display.getDisplay(calculator).setCurrent(this);
      running = true;
      thread = new Thread(this);
      thread.start(); // запускаем
   }

   public void run() {
      try {
        Float x = from;
        for (int i = 0, n = values.length; running && i < n; i++) { // прове-
ряем флажок running, чтобы остановить работу
                                                                    //потока
при выполнении команды STOP
          f.arguments[0] = x;          // вычисляем значение функции в оче-
редной точке
          values[i] = f.evaluate(calculator.bindings);
          int newPercent = i*100/n;          // оптимизация: обновляем инди-
каторы прогресса только при изменении процента выполнения
          if (newPercent != percent) {
            progressIndicator.setValue(i);
            percent = newPercent;
            completed.setText(Integer.toString(newPercent) + '%');
          }
          x = x.Add(dx);
        }
      } catch (CompileError x) {
        calculator.showAlert(AlertType.ERROR, "Error", x.getMessage());
        return;
      }
      if (running) {        // рисуем график, только если не была выполнена
команда Stop
        new Graph(calculator, from, till, values);
      }
    }

    public void commandAction(Command c, Displayable d) {
      if (c == Calculator.STOP_CMD) {        // устанавливаем флажок для за-
вершения потока
```

```
        running = false;
        Display.getDisplay(calculator).setCurrent(calculator.mainMenu);
      }
    }
}
```

Пример написания простейшей игрушки.

Допустим, кто-то должен (для определённости назовём его НЛО) перемещаться по экрану с заданной скоростью. Рисование в MIDP выполняется в переопределённом методе **paint(Graphics g)** класса **Canvas.** Чтобы инициировать его выполнения нужно с помощью метода **repaint** попросить систему перерисовать какую-то область экрана (или весь экран). Т.е. если НЛО ползёт по экрану, нужно перерисовать прямоугольник, в котором находится НЛО. Дальше стираем старый образ НЛО, вычисляем его новые координаты и рисуем его на новом месте. Осталась самая малость - кто-то должен периодически (с периодом определяемым скоростью движения НЛО) вызывать метод **repaint.** Можно конечно для этого завести поток, но к счастью есть и более простое решение - использовать класс **Timer:**

```
import javax.microedition.lcdui.*;
import java.util.Timer;
import java.util.TimerTask;

public class GameCanvas extends Canvas implements CommandListener {
    class GameTask extends TimerTask {
      public void run() {
        repaint(); // запрос на перерисовку
      }
    }

    GameCanvas(Game game) {      // вычисляем интервал перерисовки (скорость
движения НЛО) в зависимости от уровня
      delay = MAX_DELAY - game.settings.level*MAX_DELAY/Settings.LEVELS;
// создаём таймер
      timer = new Timer();      // и объект котрый будет запускаться таймером
      task = new GameTask();
      ...
    }

    // запустить игру
    void start() {      // запускаем таймер (с нулевой задержкой и заданным
интервалом в миллисекундах)
      timer.schedule(task, 0, delay);
    }

    //  приостановить игру
    void stop() {      // остановить таймер
      timer.cancel();
    }
    ...

    static final long MAX_DELAY = 1000;
    GameTask task;
    Timer    timer;
    long     delay;
}
```

Следует заметить, что внутри таймер реализован с помощью потока, который осуществляет задержку с помощью метода **wait(long delay).**

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ

Задания к лабораторным работам выполняются в соответствии с номером варианта. Номер варианта, соответствует порядковому номеру студента в списке преподавателя.

Чётные варианты заданий подразумевают использование **2 (двух)** потоков. **Нечётные - 3 (трёх)** потоков.

Все варианты заданий подразумевают использование промежуточных буферов, представляющих собой динамические массивы. Максимальный размер буферов - **N** чисел. **N** определяется для каждого варианта. Потоки, помещающие числа в буферы, следят за переполнение буферов. Потоки, извлекающие числа из буферов, могут производить данную операцию в произвольный момент времени вне зависимости от того, заполнен ли буфер полностью или нет.

Варианты заданий:

1. Значения констант и реализуемые потоками функции:
   N=10
   Первый поток - генерирует в первый буфер 1000 чисел последовательно от 1 до 1000.
   Второй поток - находит минимальное среди текущего набора чисел в первом буфере и перемещает его во второй буфер. Данная операция производится 995 раз.
   Третий поток - извлекает из второго буфера максимальное число и выводит его на экран. Третий поток завершает своё выполнение по окончании чисел во втором буфере.

2. Значения констант и реализуемые потоками функции:
   N=3
   Первый поток - генерирует в первый буфер 1000 случайных чисел в интервале от 10 до 20.
   Второй поток - извлекает второе число в буфере, возводит его в квадрат и выводит его на экран. Второй поток завершает своё выполнение на 999 итерации.

3. Значения констант и реализуемые потоками функции:
   N=5
   Первый поток - генерирует в первый буфер 30 чисел ряда Фибоначчи.
   Второй поток - извлекает числа из первого буфера, начиная с максимального делит извлечённое число на 2 и помещает во второй буфер.
   Третий поток - извлекает из второго буфера максимальное число и выводит его на экран. Третий поток завершает своё выполнение по окончании чисел во втором буфере.

4. Значения констант и реализуемые потоками функции:
   N=10
   Первый поток - генерирует в первый буфер 100 случайных чисел в интервале от 1 до
   Второй поток - извлекает из буфера последнее число - x и выводит на экран значение функции sin(x/Pi). Второй поток завершает свое выполнение после 100 итераций.

5. Значения констант и реализуемые потоками функции:
   N=300
   Первый поток - генерирует в первый буфер 1000 чисел от 1000 до 1.
   Второй поток - извлекает числа из первого буфера, начиная с первого, вычисляет тангенс этого числа и помещает его во второй буфер.

Третий поток - извлекает из второго буфера первое число, вычисляет его арктангент и выводит результат на экран. Третий поток завершает своё выполнение по окончании чисел во втором буфере.

6. Значения констант и реализуемые потоками функции:

N=2

Первый поток - генерирует в первый буфер случайную последовательность из 0 и 1. Число итераций в первом потоке равно 100.

Второй поток - Извлекает все единицы из буфера и выводит их число на экран. Если в буфере остаются лишь 0, то второй поток очищает буфер. Второй поток завершает свое выполнение по завершению доступных чисел в буфере**.**

7. Значения констант и реализуемые потоками функции:

N=3

Первый поток - генерирует в первый буфер 50 чисел из ряда 1/x, где x принимает значения от 1 до 50.

Второй поток - извлекает числа из первого буфера, начиная с последнего в буфере, извлекает из него корень и помещает его во второй буфер.

Третий поток - извлекает из второго буфера первое число, вычисляет значение функции 1/x, где x - извлечённое число, и выводит это значение на экран. Третий поток завершает своё выполнение по окончании чисел во втором буфере.

8. Значения констант и реализуемые потоками функции:

N=16

Первый поток - генерирует в первый буфер 10000 случайных чисел в интервале от 1 до 16.

ности. Для этого он извлекает из буфера максимальное число, сравнивает его с текущим максимумом, хранящимся во втором потоке. После этого второй поток очищает буфер. Второй поток завершает выполнение по окончанию чисел, доступных в буфере.

9. Значения констант и реализуемые потоками функции:

N=10

Первый поток - генерирует в первый буфер 500 случайных чисел в интервале от 1 до 100.

Второй поток - извлекает числа из первого буфера, начиная с последнего в буфере, и если извлечено число больше 50, то помещает во второй буфер 1. В противном случае во второй буфер помещается 0.

Третий поток - извлекает из второго буфера числа и подсчитывает суммарное число 0 и 1 и очищает буфер. Третий поток по окончании чисел во втором буфере выводит число 0 и 1, которые были занесены во второй буфере, на экран и завершает выполнение.

10. Значения констант и реализуемые потоками функции:

N=3

Первый поток - генерирует в буфер 1000 чисел ряда sin(x/Pi), где x принимает значения от 0,01 до 10 с шагом 0,01**.**

Второй поток - Накапливает сумму чисел в буфере и по окончанию доступных чисел в буфере выводит полученную сумму на экран. Буфер очищается вторым потоком на каждой итерации.

11. Значения констант и реализуемые потоками функции:

N=50

Первый поток - генерирует в первый буфер 100 чисел от 1 до 100.

Второй поток - извлекает числа из первого буфера, начиная с первого. Если число делится на 2, то из него берётся квадратный корень, а результат помещается во второй буфер. В противном случае во второй буфер помещается квадрат числа.
Третий поток - извлекает из второго буфера числа, начиная с максимального и выводит их на экран.

12. Значения констант и реализуемые потоками функции:
N=15
Первый поток - генерирует в буфер 100 чисел ряда cos(x*Pi), где x принимает значения от 0,01 до 1 с шагом 0,01.
Второй поток - Накапливает произведение чисел в буфере и по окончанию доступных чисел в буфере выводит полученную сумму на экран. Буфер очищается вторым потоком на каждой итерации.

13. Значения констант и реализуемые потоками функции:
N=1000
Первый поток - генерирует в первый буфер 1001 случайных чисел в интервале от 100 до 105.
Второй поток - извлекает числа из первого буфера. Обозначим извлечённое число переменной x. Поток вычисляет значение функции 3(x-100) и помещает его во второй буфер
Третий поток - извлекает из второго буфера числа, начиная с минимального, делит их на на 3 и выводит на экран.

14. Значения констант и реализуемые потоками функции:
N=10
Первый поток - генерирует в буфер 100000 случайных чисел из интервала от 0 до 2.
Второй поток - переводит получившееся в буфере число из троичной системы счисления в десятичную, выводит полученный результат на экран и очищает буфер.

15. Значения констант и реализуемые потоками функции:
N=8
Первый поток - генерирует в первый буфер 97 случайных чисел в интервале от 1 до 100000.
Второй поток - извлекает числа из первого буфера. Обозначим извлечённое число переменной x. Поток вычисляет значение функции 18cos(x*Pi) и помещает его во второй буфер
Третий поток - извлекает из второго буфера числа, начиная с минимального и выводит на экран.

16. Значения констант и реализуемые потоками функции:
N=4
Первый поток - генерирует в буфер 100000 случайных чисел из интервала от 0 до 7.
Второй поток - переводит получившееся в буфере число из восьмеричной системы исчисления в шестнадцатеричную, выводит полученный результат на экран и очищает буфер.

17. Значения констант и реализуемые потоками функции:
N=10
Первый поток - генерирует в первый буфер 100 чисел от 100000 до 1000 с шагом 1000.
Второй поток - извлекает числа из первого буфера. Обозначим извлечённое число переменной x. Поток вычисляет значение функции x/1000 и помещает его во второй буфер
Третий поток - извлекает из второго буфера числа, начиная с максимального и выводит их на экран.

18. Значения констант и реализуемые потоками функции:
N=4
Первый поток - генерирует в буфер 800 случайных чисел из интервала от 0 до 1.
Второй поток - переводит получившееся в буфере число из двоичной системы исчисления в восьмеричную, выводит полученный результат на экран и очищает буфер.

19. Значения констант и реализуемые потоками функции:

N=5
Первый поток - генерирует в первый буфер 1000 случайных чисел в интервале от 1 до 5.
Второй поток - извлекает все числа из первого буфера. Перемножает между собой числа. И помещает результат во второй буфер. После этого первый буфер очищается.
Третий поток - извлекает из второго буфера число, начиная с максимального, вычисляет из него квадратный корень и выводит результат на экран.

20. Значения констант и реализуемые потоками функции:
N=4
Первый поток - генерирует в буфер 150 случайных чисел из интервала от 0 до 15.
Второй поток - переводит получившееся в буфере число из шестнадцатеричной системы исчисления в троичную, выводит полученный результат на экран и очищает буфер.

21. Значения констант и реализуемые потоками функции:
N=25
Первый поток - генерирует в первый буфер 100 случайных чисел в интервале от 20 до 23.
Второй поток - извлекает последние числа из первого буфера. Вычитает из них 20 и помещает результат во второй буфер.
Третий поток - извлекает из второго буфера число, начиная с минимального. Это число делится на 3 и выводится на экран.

22. Значения констант и реализуемые потоками функции:
N=8
Первый поток - генерирует в буфер 1200 случайных чисел из интервала от 0 до 4.
Второй поток - переводит получившееся в буфере число из пятеричной системы исчисления в десятеричную, выводит полученный результат на экран и очищает буфер.

23. Значения констант и реализуемые потоками функции:
N=14
Первый поток - генерирует в первый буфер 10000 случайных чисел из ряда 1, 2, 4, 8, 16, 32.
Второй поток - извлекает первое число из первого буфера. Вычисляет логарифм по основанию 2 и помещает результат во второй буфер.
Третий поток - извлекает из второго буфера число, начиная с минимального. Это число умножается на 2 и выводится на экран.

24. Значения констант и реализуемые потоками функции:
N=3
Первый поток - генерирует в буфер 167 случайных чисел из интервала от 10 до 184.
Второй поток - извлекает числа из буфера, начиная со стоящих в позиции с большим значением индекса, и вычисляет для них значение косинуса. Результат выводится на экран.

25. Значения констант и реализуемые потоками функции:
N=10
Первый поток - генерирует в первый буфер 100 случайных чисел из ряда 0,1,2,3,4,5.

Второй поток - извлекает первое число из первого буфера. Умножает на 2 и помещает результат во второй буфер.
Третий поток - извлекает из второго буфера число, начиная с минимального. Это число делится на 2 и выводится на экран.

26. Значения констант и реализуемые потоками функции:
N=15
Первый поток - генерирует в буфер 30 случайных чисел из интервала от 0 до 2.
Второй поток - извлекает числа из буфера по 5 штук, начиная с первого и переводит полученное число из 3-й системы счисления в 10-ю. Результат выводится на экран.

27. Значения констант и реализуемые потоками функции:
N=3
Первый поток - генерирует в первый буфер 1000 случайных чисел 0 или 1**.**
Второй поток - извлекает первое число из первого буфера. Если оно равно 0, то он генерирует случайное число в интервале от -10 до -1, а если 1 , то в интервале от 1 до 10 и помещает его во второй буфер.
Третий поток - извлекает из второго буфера число, начиная с минимального. Это число делится на 2 и выводится на экран.

28. Значения констант и реализуемые потоками функции:
N=2
Первый поток - генерирует в буфер 30 случайных чисел из интервала от -1 до 1.
Второй поток - извлекает числа из буфера, начиная с первого и возводит их в квадрат. Результат выводится на экран.

29. Значения констант и реализуемые потоками функции:
N=300
Первый поток - генерирует в первый буфер 1000 случайных чисел от 1 до 10.
Второй поток - извлекает первое число из первого буфера. Если оно меньше 5, то оно возводится в квадрат, а если больше или равно 5, то из него извлекается квадратный корень. Результат помещается во второй буфер.
Третий поток - извлекает из второго буфера число, начиная с минимального. Это число умножается на 2 и выводится на экран.

ТРЕБОВАНИЕ К ОТЧЕТУ

Все отчёты предоставляются на проверку в **печатном виде.**

Отчёт по лабораторной работе должен содержать:

- Титульный лист
- Задание к лабораторной работе
- Диаграмму классов разработанной программы
- Исходный код классов с комментариями, поясняющими выполнение лабораторной работы
- Результаты, выводимые лабораторной работой на экран. Результат должен содержать номер обрабатываемого числа, номер потока и значение результата.
- Примеры случаев, когда необходимо использование параллельных алгоритмов для решения задачи по разработке. (Данные примеры должны содержать краткое описание реальных задач, для которых могут быть использованы параллельные вычисления).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем отличие технологии OpenMp и MPI?
2. Каким образом на однопроцессорной машине исполняются многопоточные приложения?
3. Какие преимущества дает многопоточная архитектура?
4. Что такое приоритет потока?
5. Что такое демон-поток?
6. Когда закрывается виртуальная машина, выполняющая программу с несколькими потоками исполнения?
7. Для чего служит в Java класс Thread?
8. Поскольку интерфейс Runnable представляет собой альтернативный способ программировать потоки исполнения, можно ли в такой программе обойтись вовсе без класса Thread?
9. Что такое локальное и главное хранилища в Java? Чем они различаются?
10. Что называется критической секцией?
11. Если один поток начал исполнение synchronized-блока, указав ссылку на некий объект, может ли другой поток обратиться к полю этого объекта? К методу?
12. Если объявить метод synchronized, то какой эффект будет этим достигнут?
13. Как работают static synchronized методы?
14. Почему метод wait требует обработки InterruptedException, а методы notify и notifyAll – нет?
15. Может ли поток никогда не выйти из метода wait, даже если будет вызван метод notify? notifyAll?
16. Какой будет результат работы следующего кода?

```
public abstract class Test implements Runnable {
        private Object lock = new Object();
        public void lock() {
            synchronized (lock) {
                        try {
                                lock.wait();
                                System.out.println("1");
                        } catch (InterruptedException e) {
                        }
                }
            }
        public void unlock() {
                synchronized (lock) {
                        lock.notify();
                    System.out.println("2");
            }
        }
        public static void main(String s[]) {
                new Thread(new Test() {
                        public void run() {
                                lock();
                        }
                }).start();
                new Thread(new Test() {
                        public void run() {
                                unlock();
                        }
                }).start();
        }
  }
```

Список используемой литературы

1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Н.Новгород: ННГУ, 2000, 121 с. (2 изд. 2003).
2. ИНТУИТ. Параллельное программирование URL: https://www.intuit.ru/studies/courses/1110/153/info
3. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб: БХВ-Петербург, 2002. — 608 с.
4. Оленев Н. Н. Основы параллельного программирования в системе MPI. — М.: ВЦ РАН, 2005. — 80 с. URL: http://www.ccas.ru/mmes/educat/lab04k
5. Andrey Chernyshev (Intel). Введение в технологии параллельного программирования URL: https://software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction
6. ЮГИНФО, Сектор высокопроизводительных вычислительных систем, учебные материалы, В.Н. Дацюк, А.А. Букатов, А.И. Жегуло, Методическое пособие по курсу "Многопроцессорные системы и параллельное программирование" URL: http://rsusu1.rnd.runnet.ru/koi8/index.html
7. НИВЦ МГУ Лаборатория параллельных информационных технологий URL: http://www.parallel.ru