Национальный исследовательский университет “Высшая школа экономики”.

Факультет компьютерных наук. Программная инженерия.

Архитектура вычислительных систем.

Индивидуальное домашнее задание №4 студента группы БПИ213 Абрамова Александра Сергеевича.

Вариант 37.

37. ***Задача о сельской библиотеке.*** В библиотеке имеется **N** книг, Каждая из книг в $\mathbf{K_i}$ экземплярах. **M** читателей регулярно заглядывают в библиотеку, выбирая для чтения **от одной до трех** книг и читая их некоторое количество дней. Если желаемой книги нет, то читатель, взяв существующие, дожидается от библиотекаря информации об ее появлении и приходит в библиотеку, чтобы специально забрать ее. Возможна ситуация, когда несколько читателей конкурируют из-за этой популярной книги. ***Создать многопоточное приложение, моделирующее заданный процесс.***

СОДЕРЖАНИЕ

Поведение объектов как взаимодействующих субъектов

В условии задания предполагается три типа сущностей: библиотека, читатели и книги, которые передаются между объектами первых двух типов. Каждая книга описывается своим идентификатором, а каждый читатель имеет уникальный номер.

Читатели могут брать копии книг из библиотеки. Для удобства организации работы библиотека поддерживает список всех книг с количеством имеющихся в данный момент копий каждой из них. Для обращения к библиотеке читатель должен встать в очередь. При этом каждое обращение регистрируется отдельно, а в любой момент библиотека способна обрабатывать не более одного запроса от одного читателя.

Для получения книги читатель должен прийти в библиотеку, зарегистрировать обращение и встать в очередь. Когда обращение становится первым в очереди, оно поступает в обработку. При этом библиотека проверяет, есть ли копия запрашиваемой книги в наличии. Если таковая есть, она отдаётся читателю, который при этом обязуется вернуть её строго через определенное количество дней с момента получения. Если копии книги нет в наличии, но таковые бывают в библиотеке, то читателю предлагается подождать её появления. При этом библиотека обязуется сообщить о появлении читателю, который должен будет снова прийти в библиотеку для получения.

Для возврата книги читатель также должен прийти в библиотеку, зарегистрировать обращение и встать в очередь. Как только библиотека приступает к обработке обращения, она проверяет, правильную ли книгу возвращает пользователь. Если книга верная, то библиотека принимает книгу, обновляет счетчик доступного количества копий и сообщает всем читателям, которые ожидают появления книги, что она появилась и необходимо явиться для получения. При этом обработка обращений будет происходить в порядке очереди, то есть копию книги получит тот, кто явится первым.

Модель параллельных вычислений

При разработке программы используется модель параллельных вычислений “клиенты и серверы”, где роль клиентов выполняют читатели, а серверов - библиотека.

Клиенты отправляют запросы серверу (на получение или возврат книги) и адаптируют своё поведение в зависимости от полученных ответов.

Сервер же только ожидает запросы клиентов и обрабатывает их. При этом сервер может требовать открытия долгосрочного соединения с клиентом для передачи данных (информации о появлении копии книги в библиотеке).

Для передачи запросов обе стороны организуют очередь обработки - общий ресурс, обращения к которому требуют синхронизации. При обращении к серверу клиент лишь помещает описание желаемого действия в очередь сервера, который производит регулярное чтение и обработку первого элемента очереди, если такой есть. Аналогично, сервер может добавлять элементы в очередь клиента для передачи в обратном направлении данных, которые будут обработаны как только клиент “освободится”.

Также для корректной совместной работы клиента и сервера производится синхронизация текущего времени путём прикрепления соответствующих данных к каждому запросу или путём установки общей точки отсчёта при старте.

Описание программы

Для определённости установим следующие требования ко входным и выходным данным:

1. Первый параметр командной строки при запуске программы должен быть флагом, который указывает на способ ввода:
   1.1. `-c` указывает на ввод из командной строки. Далее должны следовать непосредственно выходные данные (см п.2)
   1.2. `-s` указывает на ввод из консоли после запуска.
   1.3. `-f` указывает на ввод из файла. Вторым параметром должно быть указано расположение файла со входными данными относительно исполняемого файла.
   1.4. `-r` указывает на необходимость случайной генерации входных данных.
2. Входные данные должны удовлетворять следующему формату (программа может не проверять их корректность):
   2.1. Количество книг в библиотеке n;
   2.2. Время работы библиотеки;
   2.3. n пар значений - идентификатор очередной книги и количество её копий k_i;
   2.4. Количество читателей m;
   2.5. m значений - номера читателей;
   2.6. Количество q событий взятия книги из библиотеки;
   2.7. q наборов значений, описывающих очередное событие взятия книги из библиотеки:
      2.7.1. Время, в которое должно произойти событие;
      2.7.2. Номер читателя, который должен взять книги;
      2.7.3. Количество книг a_i, которые читатель должен взять (целое число от 1 до 3);
      2.7.4. a_i пар значений - идентификатор книги, которую читатель хочет взять, и время, на которое читатель хочет взять книгу;

Программа состоит из следующих частей:

1. Директория `Utilities`
   1.1. Пространство имён $threading$, содержащее реализации классов-обёрток над объектами библиотеки $POSIX\ Threads$:
      1.1.1. $Barrier$, реализующий создание барьера в конструкторе, его удаление в деструкторе и предоставляющий возможность ожидания на барьере.

1.1.2. $Mutex$, реализующий создание мьютекса в конструкторе, его удаление в деструкторе и предоставляющий возможность его блокировки и разблокировки

1.1.3. $Thread$, реализующий выполнение функции, переданной в конструктор, с указанными аргументами в отдельном потоке. Для этого используется вариативный шаблон, который должен получать список типов аргументов функции. Для организации вызова реализован метод Run, с сигнатурой $void$ * ($void$ *), указатель на который может быть передан функции $pthread\_create$ создания потока. Для передачи функции, которую необходимо выполнить в отдельном потоке, и её аргументов используется контейнер $std::tuple$ стандартной библиотеки шаблонов, выделяемый в динамической памяти.

1.2. Пространство имён io (реализованное в виде класса для более удобного поддержания состояния), содержащее реализации классов, позволяющих удобно производить ввод данных разными способами.

1.2.1. Классы $Input$ и $Output$ описывают интерфейсы ввода и вывода соответственно. Для удобства использования интерфейсы требуют переопределения соответствующего оператора $>>$ или $<<$ при наследовании для ввода и вывода данных определённых типов.

1.2.2. Классы вида * $Input$ предоставляют реализации интерфейса $Input$ с использованием соответствующего способа ввода: из командной строки, из консоли, из файла или случайно. Подробные комментарии о работе реализованных подпрограмм представлены в соответствующих файлах с исходным кодом.

1.2.3. Классы вида * $Output$ предоставляют реализации интерфейса $Output$ с использованием соответствующего способа вывода: в консоль или в файл.

1.2.4. Класс $IoWrapper$, реализующий обёртку над одной версией ввода и произвольным количеством версий вывода. Для удобства использования для класса также переопределены операторы $>>$ и $<<$ для ввода и вывода данных определённых типов.

1.2.5. Переменная $stream$ непосредственно содержит инстанцию класса $IoWrapper$ и может использоваться в других частях программы для ввода и вывода данных.

1.2.6. Функция $Init$ принимает на вход информацию о переданных программе аргументах командной строки и, обрабатывая её, устанавливает соответствующие значения переменных инстанции $stream$ класса $IoWrapper$ для дальнейшего использования.

2. Директория `Book` и одноимённый заголовочный файл реализуют тип для хранения сущностей вида “книга”. В данной реализации книга описывается лишь одним значением - идентификатором 64-битного беззнакового целочисленного типа. Для удобства указано переопределение оператора <=>, что позволяет использовать все операторы сравнения для значений типа $Book$. Сравнение при этом происходит по значению id.
3. Директория `Actions`, содержащая определения типов событий, которые используются для организации взаимодействия потоков, и обёртки над ними:
    3.1. Запрос читателя на получение книг представлен событием $TakeAction$, которое содержит информацию о книгах, которые читатель хочет взять. Использование структуры данных $std::optional$ позволяет не хранить количество запрашиваемых книг, а вычислять его динамически.
    3.2. Запрос читателя на возврат книги представлен событием $ReturnAction$, которое содержит информацию о книге, которую читатель хочет вернуть.
    3.3. Сообщение библиотекой читателю о получении книги (успешный ответ на событие $TakeAction$) представлено событием $TakenAction$, которое содержит информацию о взятой книге.
    3.4. Сообщение библиотекой читателю об отсутствии запрашиваемой книги и необходимости подождать её появления (безуспешный ответ на событие $TakeAction$) представлено событием $WaitAction$, которое содержит информацию о запрошенной книге и дублирует время, на которое читатель хочет взять книгу.
    3.5. Сообщение библиотекой читателю о появлении ранее запрошенной книги и возможности явиться для её получения представлено событием $AppearAction$, которое содержит информацию о появившейся книге.
    3.6. Структура $Action$ реализует объект, содержащий информацию о событии и ряд деталей о его выполнении: время $time$ и исполнителя $executor$. Для реализации хранения только одного исполнителя и только одного события использована структура $std::variant$. Для удобства проверки типа обрабатываемого события реализованы перечисление $Type$ и метод $GetType$, а для обеспечения возможности приоритезации событий определена операция сравнения двух событий по времени.
4. Директория `Reader`, реализующая сущность “читатель”. Подробное описание всех переменных, методов и их реализаций представлено в соответствующих файлах с исходным кодом. Здесь

зафиксирую лишь основные моменты, непосредственно относящиеся к реализации взаимодействия потоков сущностей:

4.1. Для организации работы использована приоритетная очередь событий $std::priority\_queue < Action >$, для синхронизации обращения к которой используется мьютекс.

4.2. Для организации работы читателя в отдельном потоке реализованы методы $Start$, Run и $Stop$, использующие реализацию $Thread$ пространства имён $threading$.

4.3. Для синхронизации времени между читателями и библиотекой используется барьер, проход через который возможен только если все запущенные потоки дошли до него (запустились). Как только барьер пройден, все читатели запоминают текущее время - время начала работы - в переменной $epoch$. При этом погрешность оказывается пренебрежимо мала.

4.4. Для обработки событий метод Run последовательно, пока не истекло время работы, считывает первое событие из очереди и, если текущее время превышает время, когда необходимо выполнить событие, определяет его тип и выполняет соответствующие действия, фиксируя изменения или “записываясь” в библиотеку.

4.4.1. Для обеспечения корректного считывания и удаления событий читатель блокирует доступ к очереди с помощью мьютекса до операции и разблокирует его по завершении.

4.5. Метод $GetId$ возвращает идентификатор читателя для организации удобного вывода.

4.6. Методы вида $Enqueue$ * добавляют соответствующее событие в очередь. Для этого происходит блокировка доступа к очереди до операции и разблокировка по её завершении.

5. Директория `Library`, реализующая сущность “библиотека”. Подробное описание всех переменных, методов и их реализаций представлено в соответствующих файлах с исходным кодом. Здесь зафиксирую лишь основные моменты, непосредственно относящиеся к реализации взаимодействия потоков сущностей:

5.1. Для организации работы использована очередь событий без приоритетов (раньше обрабатывается тот, кто “записался” раньше) $std::queue < Action >$, для синхронизации обращения к которой используется мьютекс.

5.2. Для организации работы читателя в отдельном потоке реализованы методы $Start$, Run и $Stop$, использующие реализацию $Thread$ пространства имён $threading$.

5.3. Для синхронизации времени между читателями и библиотекой используется барьер, проход через который возможен только если все запущенные потоки дошли до него (запустились).

Как только барьер пройден, библиотека запоминает текущее время - время начала работы - в переменной $epoch$. При этом погрешность оказывается пренебрежимо мала.

5.4. Для обработки событий метод Run последовательно, пока не истекло время работы, считывает первое событие из очереди, определяет его тип и выполняет соответствующие действия, фиксируя изменения и отправляя ответ читателю.

5.4.1. Для обеспечения корректного считывания и удаления событий читатель блокирует доступ к очереди с помощью мьютекса до операции и разблокирует его по завершении.

5.5. Методы вида $Enqueue*$ добавляют соответствующее событие в очередь. Для этого происходит блокировка доступа к очереди до операции и разблокировка по её завершении.

5.6. Метод Log и перечисление $LogType$ для организации удобного вывода информации о выполненных событиях.

6. Файл `index.cpp` организует ввод данных, инстанцирование всех сущностей, а также запуск и остановку библиотеки и читателей. Для этого программа опирается на реализацию класса io (см. п.1.2).

Для сборки и запуска программы был создан shell-скрипт `run.sh`, который собирает программу с использованием утилиты `g++` и запускает полученный исполняемый файл с указанными аргументами командной строки. При сборке компилятору дополнительно передаётся ряд параметров командной строки:

1. `-lpthread` для подключения библиотеки $POSIX\ Threads$;
2. `-std=c++20` для указания используемой спецификации языка C++ (C++20);
3. `-fsanitize=address,undefined` для отлова возможных ошибок (неверной работы с памятью, в том числе её утечек, неопределённого поведения) во время выполнения;
4. `-fno-sanitize-recover=all` для немедленного прекращения работы программы при обнаружении ошибок во время выполнения
5. `-Wall`, `-Werror`, `-Wsign-compare` для улучшения отлова возможных ошибок во время компиляции и рассмотрения всех предупреждений компилятора как ошибок.

Тестирование программы

Наборы тестовых данных представлены в директории `data`.

1. Описание формата входных данных представлено в файле `input-format.txt`;
2. Наборы тестовых данных для теста $i = \overline{1,\ 4}$ представлены в директории с именем i:
   2.1. Файл `in.in` содержит входные данные;
   2.2. Файл `description.txt` содержит входные данные с пояснениями;
   2.3. Файл `out.out` содержит примеры корректных выходных данных - списка событий - с пояснениями (при этом возможны другие выходные данные, которые также будут корректны);

Для тестирования программа была собрана и запущена с помощью скрипта `run.sh` на всех тестах 1 - 4, а также с использованием аргумента командной строки `-r`. При этом использовалось устройство с $12th\ Gen\ Intel\ Core\ i5\ -\ 12600K\ @\ 3.6GHz\ 16\ CPU,\ 32GB\ DDR5\ RAM$ под управлением $WSL\ Debian\ 11$ на $Windows\ 11$. Результаты запусков представлены в папке `report`. Нетрудно проверить, что полученные списки событий корректны, то есть действительно могли быть получены с учётом конкуренции читателей. Следовательно, программа работает корректно.

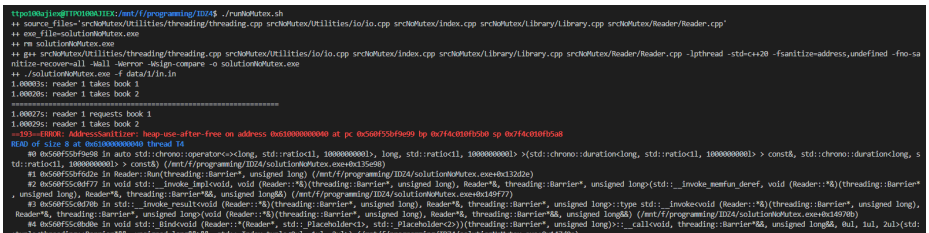
Поведение программы при отключении синхропримитивов

Для анализа поведения программы при отсутствии синхропримитивов были созданы копии исходного кода в следующих директориях:

1. `srcNoMutex`, в котором отключено создание мьютексов и их ожидание путём удаления реализаций соответствующих методов класса $Mutex$ пространства имён $threading$.
2. `srcNoReaderSync`, в котором отключена синхронизация обращений к очереди событий читателей. Для этого в классе $Reader$ подменён тип переменной $actions\_access$: интерфейс $BrokenMutex$ совпадает с интерфейсом $Mutex$ пространства имён $threading$, но не содержит реализаций методов.
3. `srcNoLibrarySync`, в котором отключена синхронизация обращений к очереди событий библиотеки аналогичным `srcNoReaderSync` образом.
4. `srcNoBarrier`, в котором отключено создание барьеров и их ожидание путём удаления реализаций соответствующих методов класса $Barrier$ пространства имён $threading$.
5. Дополнительно были созданы shell-скрипты для сборки новых версий программы. Запуск производится с использованием считывания из файла.

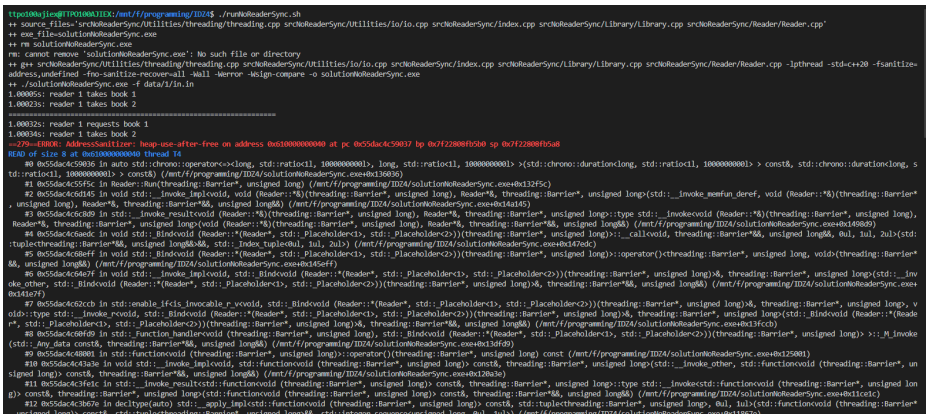
Сборка и запуск программ показали следующее:

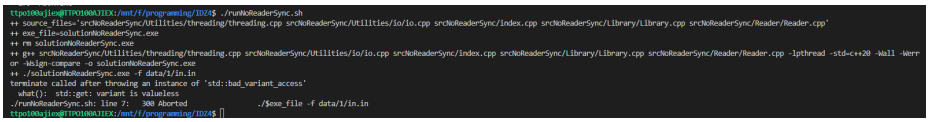
1. Отключение всех мьютексов приводит к ошибкам во время выполнения программы (например, на первом наборе тестовых входных данных).



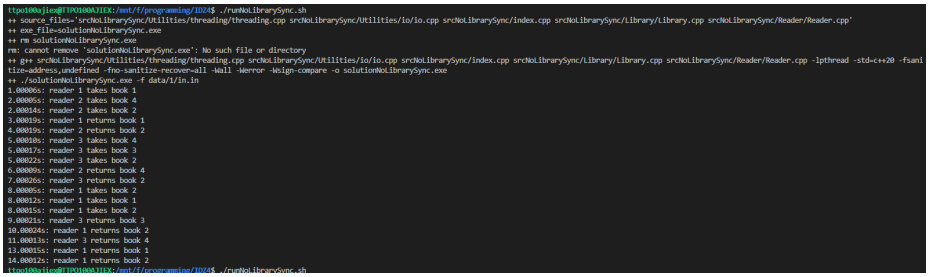
При отключении санитайзеров программа “держится” чуть дольше, но печатает неверные результаты, а в итоге всё равно завершается с ошибкой (в данном случае, с ошибкой сегментации).

2. Отключение синхронизации обращений к очереди событий читателей, ожидаемо, приводит к похожему результату за исключением того, что без санитайзеров на первом наборе тестовых входных данных программа завершается с другим исключением - некорректным обращением к значениям $std::variant$.

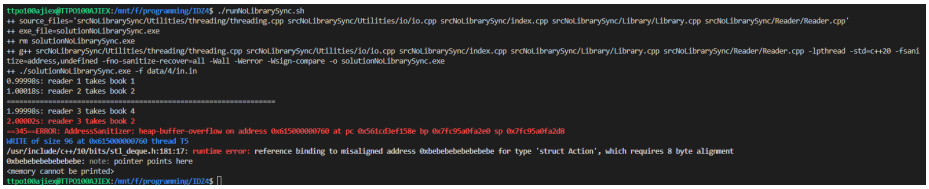


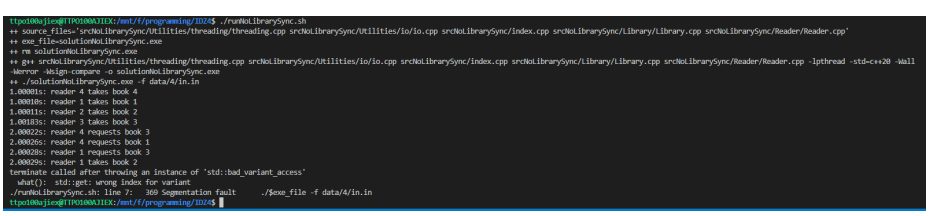


3. Тем не менее при отключение синхронизации обращений к очереди событий библиотеки программа работает корректно на первом наборе тестовых данных, что, на самом деле, ожидаемо: первый набор тестовых данных не требует одновременного обращения к памяти.

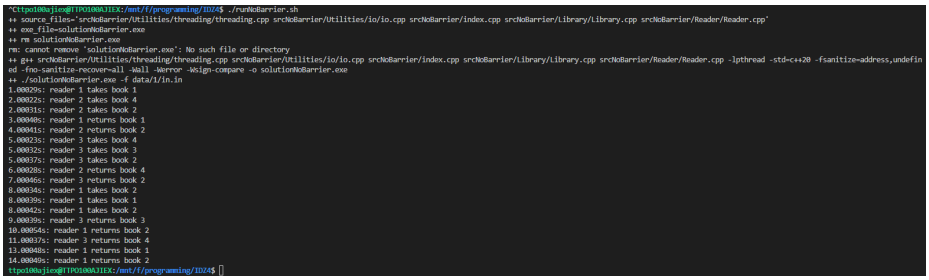


Но если программа требует одновременного выполнения действий (как, например, в 4 наборе тестовых данных), то выполнение также завершается ошибкой.

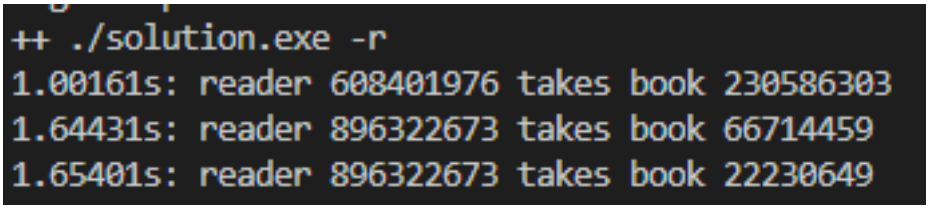


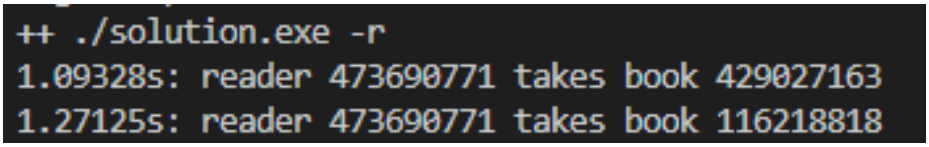


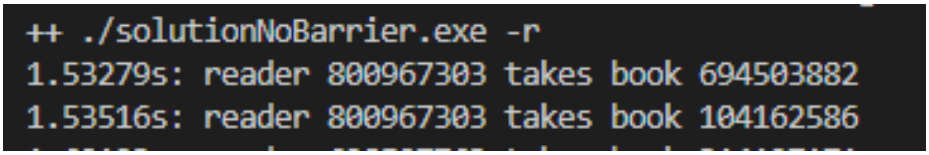
4. Отключение барьера же не приводит к значительному нарушению работы программы. Единственным отличием оказывается увеличение погрешности выводимого времени. Так, вместо $1.00005s$ программа вывела $1.00029s$, а вместо $14.00010s$ - $14.00049s$. Именно такая (нужно заметить, незначительная в виду небольшого размера входных данных) погрешность возникает при отсутствии синхронизации времени между потоками.

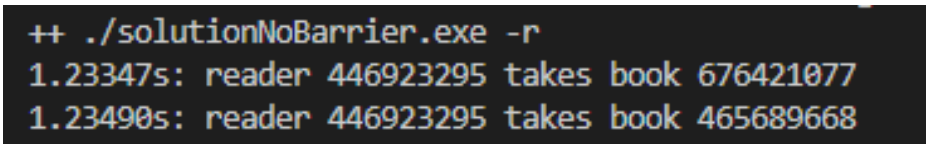


Погрешность ещё более заметна при запуске программы с большим количеством читателей (с помощью случайной генерации входных данных) - чем больше читателей, тем больше времени уходит на запуск всех потоков, тем больше погрешность. Несмотря на то, что генерируемые тесты случайны, это не должно значительно влиять на погрешность выводимого времени, что подтверждает повторный запуск программы без отключения синхропримитивов.









Альтернативное решение

В качестве другого синхропримитива были использованы семафоры. Известно, что мьютекс - двоичный семафор, что и было использовано для создания новой программы. При этом достаточно было изменить лишь реализацию класса *Mutex* пространства имён *threading*.

Программа представлена в директории `srcAlternative`, а для её сборки и запуска создан shell-скрипт `runAlternative.sh`.

Нетрудно заметить, что новая программа работает так же, как и предыдущая, на всех тестах. Полный лог запусков программы представлен в файле `alternative.out` директории `report`.