****

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ МОСКОВСКОЙ ОБЛАСТИ**

**Государственное бюджетное профессиональное образовательное**

**учреждение Московской области**

**«Люберецкий техникум имени Героя Советского Союза,**

**лётчика-космонавта Ю.А. Гагарина»**

|  |
| --- |
|  |
|  |
|  |
|  |

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**(ДИПЛОМНАЯ РАБОТА)**

На тему: «Проблемы современного программирования:

многопоточность, BigData, NP-полная задача и

безопасность.»

по специальности: 46.02.01 Документационное обеспечение управления и архивоведение

|  |  |  |  |
| --- | --- | --- | --- |
| Выполнил |  | М.В. Алексеев |  |
| Руководитель |  | Ю.В. Жирнова |  |

ДОПУЩЕН К ЗАЩИТЕ

Заместитель директора по информационным технологиям

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / А.В.Капранова /

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2022г.

**Содержание**

**1.Теоретическая часть**

**1.1Многопоточное программирование**

**1.1.1Многозадачность и многопоточность**

**Многозадачность (multitasking)** — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач.

**Многопоточность (multithreading)** — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что **процесс**, порождённый в операционной системе, может состоять из нескольких **потоков**, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

По-настоящему **параллельное выполнение задач** возможно только в многопроцессорной системе, поскольку только в них присутствуют несколько системных конвейеров для исполнения команд.  
В однопроцессорной многозадачной системе поддерживается так называемое **псевдопараллельное исполнение**, при котором создается видимость параллельной работы нескольких процессов. В таких системах, однако, процессы выполняются последовательно, занимая малые кванты процессорного времени.

**С точки зрения пользователя:**  
**Процесс** — экземпляр программы во время выполнения.  
**Потоки** — ветви кода, выполняющиеся «параллельно», то есть без предписанного порядка во времени.

**С точки зрения операционной системы:**  
**Процесс** — это абстракция, реализованная на уровне операционной системы. Процесс был придуман для организации всех данных, необходимых для работы программы.  
**Процесс** — это просто контейнер, в котором находятся ресурсы программы:

* адресное пространство
* потоки
* открытые файлы
* дочерние процессы
* и т.д.

**Поток** — это абстракция, реализованная на уровне операционной системы. Поток был придуман для контроля выполнения кода программы.

**Поток** — это просто контейнер, в котором находятся:

* Счётчик команд
* Регистры
* Стек

Поток легче, чем процесс, и создание потока стоит дешевле. Потоки используют адресное пространство процесса, которому они принадлежат, поэтому потоки внутри одного процесса могут обмениваться данными и взаимодействовать с другими потоками.

Когда одна программа выполняет множество задач, поддержка множества потоков внутри одного процесса позволяет:

* Разделить ответственность за разные задачи между разными потоками
* Повысить быстродействие

Кроме того, часто задачам необходимо обмениваться данными, использовать общие данные или результаты других задач. Такую возможность предоставляют потоки внутри процесса, так как они используют адресное пространство процесса, которому принадлежат. Конечно, можно было бы создать под разные задачи дополнительные процессы, но:

* у процесса будет отдельное адресное пространство и данные, что затруднит взаимодействие частей программы
* создание и уничтожение процесса дороже, чем создание потока

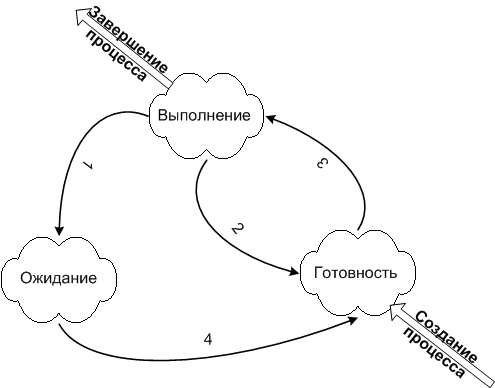
**Отличие процесса от потока**  
Процесс рассматривается ОС, как заявка на все виды ресурсов (память, файлы и пр.), кроме одного — процессорного времени. Поток — это заявка на процессорное время. Процесс — это всего лишь способ сгруппировать взаимосвязанные данные и ресурсы, а потоки — это единицы выполнения (unit of execution), которые выполняются на процессоре.

**Планирование, состояния потоков, приоритеты**

Выбор текущего потока из нескольких активных потоков, пытающихся получить доступ к процессору называется **планированием**. Процедура планирования обычно связана с весьма затратной **процедурой диспетчеризации** — переключением процессора на новый поток, поэтому планировщик должен заботиться об эффективном использовании процессора.

Поток может находиться в одном из трёх состояний:

* Выполняемый (Executing) — поток, который выполняется в текущий момент на процессоре.
* Готовый (Runnable) — поток ждет получения кванта времени и готов выполнять назначенные ему инструкции. Планировщик выбирает следующий поток для выполнения только из готовых потоков.
* Ожидающий (Waiting) — работа потока заблокирована в ожидании блокирующей операции



В реальных задачах важность работы разных потоков может сильно различаться. Для контроля этого процесса был придуман **приоритет** работы. У каждого потока есть такое числовое значение приоритета. Если есть несколько спящих потоков, которые нужно запустить, то ОС сначала запустит поток с более высоким приоритетом. ОС управляет потоками так, как посчитает нужным. Потоки с низким приоритетом не будут простаивать, просто они будут получать меньше времени, чем другие, но выполняться все равно будут. Потоки с одинаковыми приоритетами запускаются в порядке очереди. Приоритет потока может меняться в процессе выполнения. Например, после завершения операции ввода-вывода могут увеличивать приоритет потока, чтобы дать ему возможность быстрее начать выполнение и, может быть, вновь инициировать операцию ввода-вывода. Таким способом система поощряет интерактивные потоки и поддерживает занятость устройств ввода-вывода.

Потоки могут быть созданы не только в **режиме ядра**, но и в **режиме пользователя**, в зависимости от того, какой планировщик потоков используется:

* Центральный планировщик ОС режима ядра, который распределяет время между любым потоком в системе.
* Планировщик библиотеки потоков. У библиотеки потоков режима пользователя может быть свой планировщик, который распределяет время между потоками различных процессов режима пользователя.
* Планировщик потоков процесса. К примеру свой Thread Manager есть у каждого процесса Mac OS X, написанного с использованием библиотеки Carbon.

#### Системные вызовы, режимы доступа

**Системный вызов** — это вызов функции ядра, из приложения пользователя.

Чтобы защитить жизненно важные системные данные от доступа и (или) внесения изменений со стороны пользовательских приложений, в WIndows и Linux используются два процессорных режима доступа (даже если процессор поддерживает более двух режимов): **пользовательский режим** и **режим ядра**.  
Код пользовательского приложения запускается в пользовательском режиме, а код операционной системы (например, системные службы и драйверы устройств) запускается в режиме ядра. Режим ядра — такой режим работы процессора, в котором предоставляется доступ ко всей системной памяти и ко всем инструкциям центрального процессора. Предоставляя программному обеспечению операционной системы более высокий уровень привилегий, нежели прикладному программному обеспечению, процессор гарантирует, что приложения с неправильным поведением не смогут в целом нарушить стабильность работы системы.  
Также следует отметить, что в случае выполнения системного вызова потоком и перехода из режима пользователя, в режим ядра, происходит смена стека потока на стек ядра. При переключении выполнения потока одного процесса, на поток другого, ОС обновляет некоторые регистры процессора, которые ответственны за механизмы виртуальной памяти (например CR3), так как разные процессы имеют разное виртуальное адресное пространство. Здесь я специально не затрагиваю аспекты относительно режима ядра, так как подобные вещи специфичны для отдельно взятой ОС.  
Старайтесь не злоупотреблять средствами синхронизации, которые требуют системных вызовов ядра (например мьютексы).

### **Задачи и проблемы многопоточности**

К достоинствам многопоточной реализации той или иной системы перед однопоточной можно отнести следующее:

* Упрощение программы в некоторых случаях, за счёт вынесения механизмов чередования выполнения различных слабо взаимосвязанных подзадач, требующих одновременного выполнения, в отдельную подсистему многопоточности.
* Повышение производительности процесса за счёт распараллеливания процессорных вычислений и операций ввода-вывода.

К достоинствам многопоточной реализации той или иной системы перед многопроцессной можно отнести следующее:

* Упрощение программы (взаимодействия её параллельных частей) в некоторых случаях за счёт использования общего адресного пространства.
* Меньшие относительно процесса временные затраты на создание потока.

Распараллеливать работу приложения бывает удобно в самых разных ситуациях. Вот несколько примеров:

* Многопоточность широко используется в приложениях с пользовательским интерфейсом. В этом случае за работу интерфейса отвечает один поток, а какие-либо вычисления выполняются в других потоках. Это позволяет пользовательскому интерфейсу не подвисать, когда приложение занято другими вычислениями.
* Многие алгоритмы легко разбиваются на независимые подзадачи, которые можно выполнять в разных потоках для повышения производительности. Например, при фильтрации изображения разные потоки могут заниматься фильтрацией разных частей изображения.
* Если некоторые части приложения вынуждены ждать ответа от сервера/пользователя/устройства, то эти операции можно выделить в отдельный поток, чтобы в основном потоке можно было продолжать работу, пока другой поток ждёт ответа.
* и т.д.

Кроме того, многопоточную систему можно реализовать с возможностью масштабирования производительности. Например, при распараллеливании алгоритма количество создаваемых потоков может зависеть от количества процессорных ядер. Это позволит ускорять работу программы в определённых пределах, улучшая железо и не изменяя код.

Когда потоки должны взаимодействовать друг с другом или работать с общими данными, могут возникать проблемы. Часто проблемы многопоточности иллюстрируются на следующих задачах:

* [Задача об обедающих философах](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE%D0%B1_%D0%BE%D0%B1%D0%B5%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D1%85_%D1%84%D0%B8%D0%BB%D0%BE%D1%81%D0%BE%D1%84%D0%B0%D1%85)
* [Проблема спящего парикмахера](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D0%B0_%D1%81%D0%BF%D1%8F%D1%89%D0%B5%D0%B3%D0%BE_%D0%BF%D0%B0%D1%80%D0%B8%D0%BA%D0%BC%D0%B0%D1%85%D0%B5%D1%80%D0%B0)
* [Задача о курильщиках](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D0%BA%D1%83%D1%80%D0%B8%D0%BB%D1%8C%D1%89%D0%B8%D0%BA%D0%B0%D1%85)
* [Задача о читателях-писателях](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D1%87%D0%B8%D1%82%D0%B0%D1%82%D0%B5%D0%BB%D1%8F%D1%85-%D0%BF%D0%B8%D1%81%D0%B0%D1%82%D0%B5%D0%BB%D1%8F%D1%85)
* [Другие задачи](http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf)

Рассмотрим некоторые проблемы синхронизации.

Состояние гонки (race condition)

Состояние гонки — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.  
Состояние гонки — «плавающая» ошибка (гейзенбаг), проявляющаяся в случайные моменты времени и «пропадающая» при попытке её локализовать.

Рассмотрим пример.  
Допустим, каждый из двух потоков должен увеличить значение глобальной переменной 1. В идеальной ситуации последовательность операций должна быть следующая:

Скрин\*

В результате мы получаем значение 2, как и ожидали. Однако, если два потока работают одновременно, и их работа не синхронизируется, результат операции может быть неправильным. Возможна следующая последовательность операций:

Скрин\*

В этом случае результат будет равен 1, хотя ожидалось значение 2.

Код на C++, приводящий к состоянию гонки:

Скрин\*

В данном примере решить проблему можно либо использованием атомарных операций вместо нескольких инструкций чтение-изменение-запись, либо ограничивая доступ потоков к переменной так, чтобы в один момент времени только один поток мог изменять переменную.

Использование атомарных операций:

Скрин\*

Подробнее про atomic:  
[std::atomic](https://en.cppreference.com/w/cpp/atomic/atomic)  
[std::atomic. Модель памяти C++ в примерах](https://habr.com/ru/post/517918/)

Ограничение доступа к переменной так, чтобы только один поток в один момент времени мог изменять переменную:

Скрин\*

В этом примере поток перед тем как изменить переменную захватывает mutex (устанавливает флаг о том, что переменная занята), а другой поток, пытаясь захватить тот же mutex в это же время, обнаруживает, что первый поток уже работает с переменной, и дожидается её освобождения.

Подробнее про mutex:  
[std::mutex](https://en.cppreference.com/w/cpp/thread/mutex)

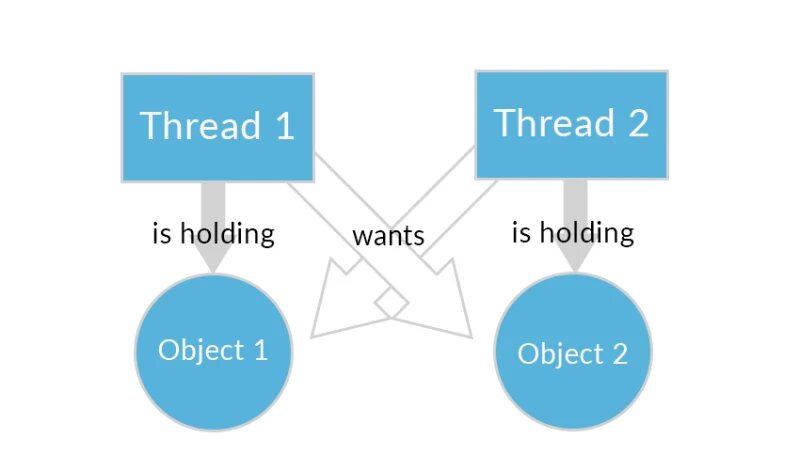
Используя mutex в примере выше, мы **синхронизируем** работу потоков. Mutex является **примитивом синхронизации**.  
Примитивы синхронизации — механизмы, позволяющие реализовать взаимодействие потоков, например, единовременный доступ только одного потока к критической области.  
Примитивы синхронизации преследуют различные задачи:

* Взаимное исключение потоков — примитивы синхронизации гарантируют то, что единовременно с критической областью будет работать только один поток.
* Синхронизация потоков — примитивы синхронизации помогают отслеживать наступление тех или иных конкретных событий, то есть поток не будет работать, пока не наступило какое-то событие. Другой поток в таком случае должен гарантировать наступление данного события.

Однако если взаимоотношения между потоками более сложные, то неаккуратные блокировки потоков могут приводить к новой проблеме — взаимным блокировкам (deadlock).

### **Взаимная блокировка (deadlock)**

Deadlock — ситуация, при которой несколько потоков находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать выполнение.



Представим, что поток-1 работает с каким-то Объектом-1, а поток-2 работает с Объектом-2. При этом программа написана так:

* Поток-1 перестанет работать с Объектом-1 и переключится на Объект-2, как только Поток-2 перестанет работать с Объектом 2 и переключится на Объект-1.
* Поток-2 перестанет работать с Объектом-2 и переключится на Объект-1, как только Поток-1 перестанет работать с Объектом 1 и переключится на Объект-2.

Даже не обладая глубокими знаниями в многопоточности легко понять, что ничего из этого не получится. Потоки никогда не поменяются местами и будут ждать друг друга вечно. Ошибка кажется очевидной, но на самом деле это не так. Допустить ее в программе можно запросто.

Кстати, на [Quora](https://www.quora.com/What-are-some-real-life-examples-of-deadlock) есть отличные примеры из реальной жизни, объясняющие что такое deadlock.

Пример возникновения взаимной блокировки в программе на C++:

Скрин\*

Менее наглядный, но более жизненный пример можно посмотреть [тут](https://leimao.github.io/blog/CPP-Deadlocks/).

Классический способ борьбы с взаимными блокировками состоит в том, чтобы захватывать несколько мьютексов всегда в одинаковом порядке.  
Более строго, это значит, что между блокировками устанавливается отношение сравнения и вводится правило о запрете захвата «большей» блокировки в состоянии, когда уже захвачена «меньшая». Таким образом, если процессу нужно несколько блокировок, ему нужно всегда начинать с самой «большой» — предварительно освободив все захваченные «меньшие», если такие есть — и затем в нисходящем порядке. Это может привести к лишним действиям (если «меньшая» блокировка нужна и уже захвачена, она освобождается только чтобы тут же быть захваченной снова), зато гарантированно решает проблему.

С учётом этого пример принимает следующий вид:

Скрин\*

В нашем простом примере легко было вручную задать верный порядок блокировки мьютексов, однако, это не всегда так легко. Например, в ситуации, когда два мьютекса передаются в функцию по ссылке и блокируются ею, порядок блокировки будет зависеть от порядка переданных аргументов. Поэтому для блокировки мьютексов одинаковом порядке стандартная библиотека предоставляет функцию std::lock (аналог std::mutex::lock()) и класс std::scoped\_lock (аналог std::lock\_guard).

std::scoped\_lock — это улучшенная версия std::lock\_guard, конструктор которого блокирует произвольное количество мьютексов в фиксированном порядке (как и std::lock). В новом коде следует использовать std::scoped\_lock, std::lock\_guard остался в языке для обратной совместимости. Пример:

Скрин\*