

Оглавление

1. Отличия синхронного кода от асинхронного.	3
2. Потоки выполнения.	3
3. Основные функции библиотеки TPL.	4
4. Шаблон TAP. (Task Asynchronous Pattern)	5
5. Задачи и фабрики задач.	6
6. Продолжения задач.	6
7. Планировщик. Функционал планировщика.	7
8. Дочерние задачи.	8
9. Вложенные задачи.	8
10. Ключевые слова <code>async</code> <code>await</code>	9
11. Асинхронные методы.	9
12. Типы возвращаемых значений асинхронных методов.	9
13. Объект ожидания завершения асинхронной задачи.	10
14. Асинхронные операции.	10
15. Асинхронные операции ввода-вывода.	11
16. История ЭВМ до параллельного программирования.	11
17. Мультипрограммирование.	13
18. Виды параллелизма.	13
19. Процесс. Контекст процесса.	15
20. Планирование и диспетчеризация.	16
21. Разделяемые ресурсы.	16
22. Взаимоблокировка.	17
23. Механизмы межпроцессного взаимодействия.	19
24. Примитивы синхронизации. Разделяемая память.	19
25. Примитивы синхронизации. Критическая секция.	19
26. Примитивы синхронизации. Семафоры.	21
27. Примитивы синхронизации. Мьютексы.	22

28.	Примитивы синхронизации. Мониторы.	24
29.	Примитивы синхронизации. Сигналы.	25
30.	Примитивы синхронизации. Барьеры.	25
31.	Потоки. Создание и уничтожение потока.....	26
32.	Алгоритм Деккера.....	27
33.	Алгоритм Петерсона.	29
34.	Алгоритм пекарни Лемпорта.....	30
35.	Потоки и пулы потоков в C#.	30
36.	Parallel LINQ.....	32
37.	Потокобезопасные коллекции в C#.....	33
38.	Streams в Java.....	33
39.	Потоки и пулы потоков в Java.....	34
40.	Интерфейс CompletableFuture.	35
41.	Потокобезопасные коллекции в Java.	35

1. Отличия синхронного кода от асинхронного.

В синхронном коде, при вызове метода, основной поток ожидает пока он закончится.

Происходит удержание потока. Пользовательский интерфейс может перестать отвечать на запросы. В то время как в асинхронном коде, основной поток мог бы продолжать делать что-то полезное.

(По сути, недостатки синхронного кода от асинхронного и есть их отличие между собой.)

Недостатки синхронного кода

- Если один компонент заблокирован, то блокируется вся программа
- Пользовательский интерфейс может перестать отвечать на запросы
- Отсутствие использования преимуществ многоядерных систем

Асинхронное программирование — подход к написанию кода, который позволяет выполнять второстепенные и долго выполняемые задачи, не блокируя основной поток выполнения

Асинхронное программирование увеличивает пропускную способность. (Не производительность) Увеличивается количество одновременно выполняемых задач.

2. Потоки выполнения.

Поток выполнения — наименьшая единица выполнения кода. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы программы, тогда как процессы изолированы друг от друга.

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.

3. Основные функции библиотеки TPL.

Библиотека параллельных задач (TPL) — это набор общедоступных типов в пространствах имен `System.Threading` и `System.Threading.Tasks`

Цель TPL — повысить продуктивность разработчиков за счет упрощения процесса добавления параллелизма.

Основная концепция библиотеки TPL — использование задач

Основные функции:

```
Task task = new Task(Counter); //Инициализация  
task.Start(); //Старт
```

```
Task task = Task.Factory.StartNew(Counter);
```

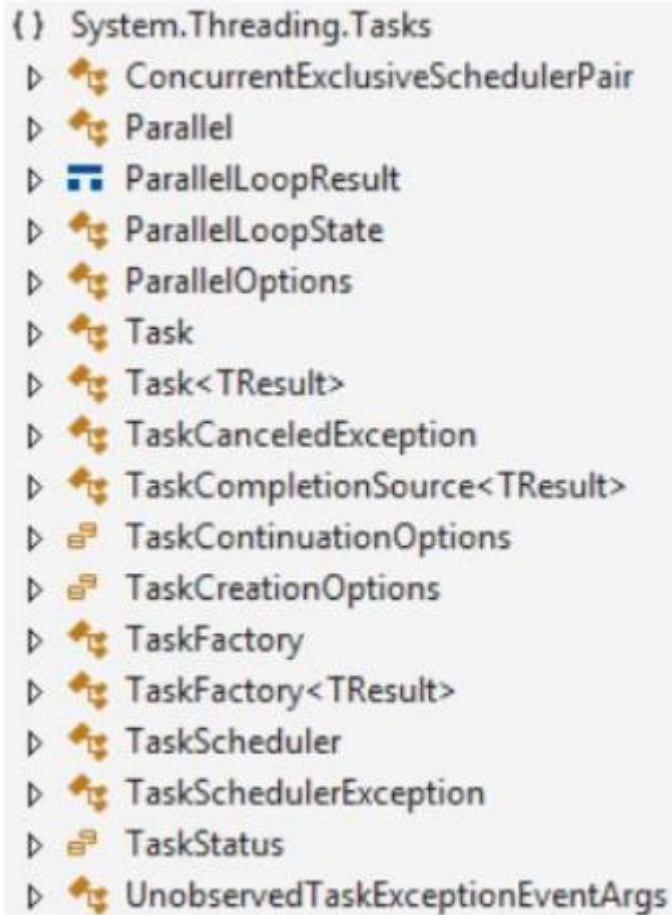
```
Task task = Task.Run(Counter); //Инициализация и старт
```

Все три способа запуска задач, по сути, идентичны — выполнение задачи осуществляется в отдельном фоновом потоке из пула потоков.

```
task.Wait(); // Ожидание выполнения задачи
```

4. Шаблон TAP. (Task Asynchronous Pattern)

Сам паттерн состоит из двух частей: набора классов из пространства имен System.Threading.Tasks и конвенций написания своих асинхронных классов и методов.



```
{ } System.Threading.Tasks
  ▷ ConcurrentExclusiveSchedulerPair
  ▷ Parallel
  ▷ ParallelLoopResult
  ▷ ParallelLoopState
  ▷ ParallelOptions
  ▷ Task
  ▷ Task<TResult>
  ▷ TaskCanceledException
  ▷ TaskCompletionSource<TResult>
  ▷ TaskContinuationOptions
  ▷ TaskCreationOptions
  ▷ TaskFactory
  ▷ TaskFactory<TResult>
  ▷ TaskScheduler
  ▷ TaskSchedulerException
  ▷ TaskStatus
  ▷ UnobservedTaskExceptionEventArgs
```

Что нам дает асинхронный подход в контексте TAP:

- Реализации фасадов по согласованной работе с задачами, а именно:
 - Запуск задач.
 - Отмена задач.
 - Отчет о прогрессе.
 - Комбинация цепочек задач, комбинаторы.
 - Неблокирующие ожидания (механизм async/await).
- Конвенции по именованию и использованию асинхронных методов:
 - В конец добавляем постфикс Async.
 - В аргументы метода можем передавать или не передавать CancellationToken & IProgress имплементацию.
 - Возвращаем только запущенные задачи.

5. Задачи и фабрики задач.

Задача - Task

- Task (задача) – конструкция, которая реализует модель параллельной обработки, основанной на обещаниях (Promise). Задача «обещает», что работа будет выполнена позже, позволяя взаимодействовать с помощью обещания с чистым API.
- Для работы с задачами в .NET используют класс Task.

Фабрика задач

- Фабрика задач – механизм, позволяющий настроить набор сгруппированных задач, которые находятся в одном состоянии.
- Классы для работы с фабрикой у TaskFactory и TaskFactory <TResult>.
- Можно создать экземпляр класса TaskFactory и настроить его нужными параметрами для создания экземпляров класса Task с этими параметрами. Удобность применения TaskFactory состоит в отсутствии необходимости указания этих параметров при каждом создании экземпляров класса Task.

6. Продолжения задач.

Продолжения - Continuations

- Продолжения – это асинхронная задача, вызываемая другой задачей при своем завершении. Это некий вариант метода обратного вызова (Callback method).

```
Task task = new Task(new Action(Download));  
Task continuation = task.ContinueWith(new Action<Task>(ShowData));
```

- Метод ContinueWith() возвращает новый экземпляр класса Task, что позволит выстраивать цепочки продолжений.

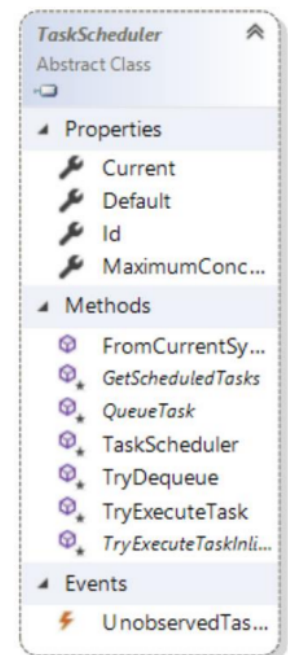
7. Планировщик. Функционал планировщика.

Планировщик

- Планировщик задач – это механизм, который позволяет настроить выполнение задач указанным вами способом и методами.
- Для работы с планировщиком в .NET используют класс TaskScheduler

TaskScheduler

- Планировщик является базовым (абстрактным) классом.
- Реализация конкретной логики работы планировщика полностью ложится на программиста-пользователя.
- В библиотеке .NET есть стандартные варианты планировщиков. К примеру стандартный планировщик построенный на ThreadPool из статического свойства TaskScheduler.Default



Основные методы планировщика

- Абстрактные методы:
 - QueueTask(Task task) – помещает переданную задачу в очередь выполнения.
 - GetScheduledTasks() – возвращает очередь задач в виде коллекции.
 - TryExecuteTaskInline(Task task, bool taskWasPreviouslyQueued) – запрашивает возможность выполниться синхронно.
- Другие методы: :
 - TryExecuteTask(Task task) – попытка выполнить задачу.
 - TryDequeue(Task task) – попытка удалить задачу из очереди выполнения.
 - FromCurrentSynchronizationContext() – создает планировщик, связанный с текущим элементом SynchronizationContext.

8. Дочерние задачи.

Дочерние задачи (Child Tasks)

- Дочерние задания – создание задач и их дальнейшее прикрепление к другой задачи, которую будут считать родителем. Другими словами – это способ настроить связь Родитель-Потомок.
- Задача может иметь любое количество дочерних задач.
- Несколько дочерних задач могут иметь общего родителя. Пока дочерние задачи полностью не выполнятся, родитель не вернет результат.
- Для присоединения к родительской задачи дочерней, нужно передать флаг перечисления `TaskCreationOptions.AttachedToParent`
- Можно запретить присоединять к задачи дочерние. При создании передать флаг перечисления `TaskCreationOptions.DenyChildAttach`

9. Вложенные задачи.

Лекция:

Вложенные задачи (Nested Tasks)

- Вложенные и задачи – создание задач в теле другой задачи, которые выполняются независимо от родительского объекта.
- Родительская задача может иметь любое количество вложенных задач. Но, родительская задача абсолютно не зависит от работы вложенных в нее задач и может выполниться гораздо раньше, чем вложенные.

Синица:

- Вложенные задачи – задачи, которые запускаются внутри другой задачи.
- Родительская задача может иметь сколько угодно вложенных задач, но от них она абсолютно не зависит.
- Родительская задача не ожидает завершения вложенных задач.
- Родительская задача не отслеживает исключения вложенных задач.

Пример вложенной задачи

```
await Task.Factory.StartNew(() =>
{
    var inner = Task.Factory.StartNew(() =>
    {
        Thread.Sleep(2000);
    });
});
```


10. Ключевые слова `async await`.

Основная идея `async/await` в том, чтобы писать асинхронный код в синхронной манере и не задумываться, как это работает.

Асинхронный метод выполняется синхронным образом до тех пор, пока не будет достигнуто первое выражение `await`, после чего метод приостанавливается, пока не будет завершена ожидаемая задача.

11. Асинхронные методы.

Ключевые слова `async await`

- Ключевое слово `async` – является модификатором для методов. Указывает, что метод асинхронный.
- Модификатор `async`:
 - указывает компилятору, что необходимо создать конечный автомат для обеспечения работы асинхронного метода. Основная задача конечного автомата – приостановка и затем асинхронное возобновление работы в точках ожидания.
 - позволяет использовать в теле асинхронного метода ключевое слово `await`.
 - позволяет записать возвращаемое значение (если метод возвращает `Task<TResult>`) или необработанное исключение в результирующую задачу.

12. Типы возвращаемых значений асинхронных методов.

Ключевые слова `async await`

- Типы возвращаемых значений асинхронных методов
- Асинхронные методы могут иметь следующие типы возвращаемых значений:
 - Тип `void` – используется только для обработчиков событий.
 - Тип `Task` – для асинхронной операции, которая не возвращает значение.
 - Тип `Task<TResult>` – для асинхронной операции, которая возвращает значение.
 - Тип `ValueTask` – для асинхронной операции, которая не возвращает значения.
 - Тип `ValueTask<TResult>` – для асинхронной операции, которая возвращает значение.

13. Объект ожидания завершения асинхронной задачи.

Объект ожидания TaskAwaiter/TaskAwaiter<TResult>

TaskAwaiter/TaskAwaiter<TResult> - объект ожидания завершения асинхронной задачи. Объект ожидания поддерживает полноценный функционал для работы оператора await.

```
public struct TaskAwaiter : ICriticalNotifyCompletion, INotifyCompletion
{
    public bool IsCompleted { get; }

    public void GetResult();
    public void OnCompleted(Action continuation);
    public void UnsafeOnCompleted(Action continuation);
}
public struct TaskAwaiter<TResult> : ICriticalNotifyCompletion, INotifyCompletion
{
    public bool IsCompleted { get; }

    public TResult GetResult();
    public void OnCompleted(Action continuation);
    public void UnsafeOnCompleted(Action continuation);
}
```

14. Асинхронные операции.

Результат асинхронной операции

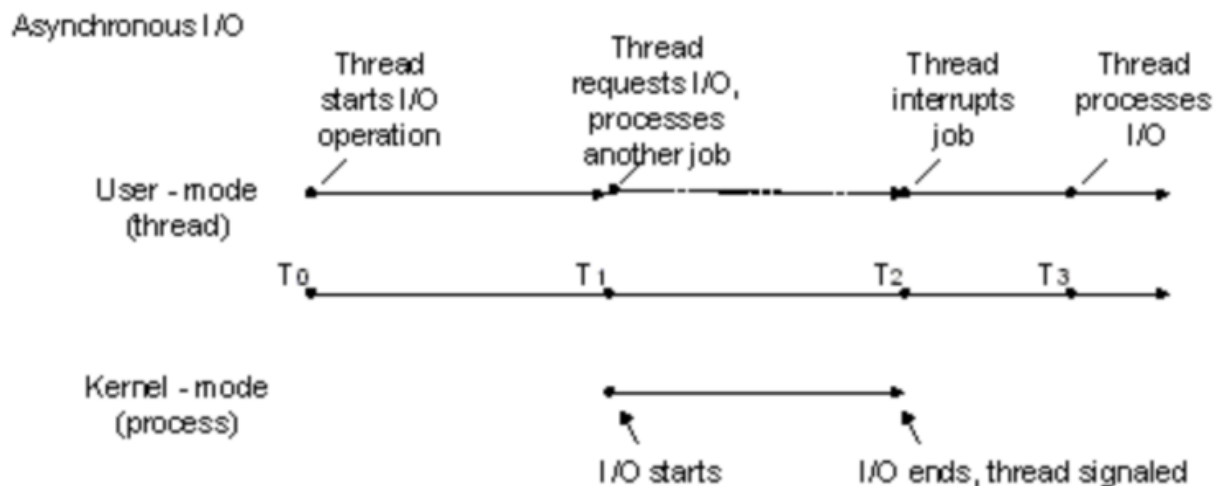
- Асинхронные операции способны возвращать результат своей работы.
- При работе с задачами есть несколько способов для извлечения результата из асинхронной задачи:
- Свойство Result, вызванное на экземпляре класса Task<TResult>.
- Метод GetResult(), вызванный на экземпляре структуры TaskAwaiter<TResult>.
- Оператор await.

```
private static async Task Main()
{
    int a = OperationAsync().Result;
    int b = OperationAsync().GetAwaiter().GetResult();
    int c = await OperationAsync();
}
```

- Сейчас, для получения результата асинхронной задачи, необходимо по возможности использовать только оператор await. Он делает это максимально эффективно и безопасно.

15. Асинхронные операции ввода-вывода.

Асинхронный ввод-вывод также называется перекрывающимся вводом-выводом. В синхронном файле ввода-вывода поток запускает операцию ввода-вывода и немедленно переходит в состояние ожидания до завершения запроса ввода-вывода. Поток, выполняющий асинхронный ввод-вывод файлов, отправляет запрос ввода-вывода в ядро, вызывая соответствующую функцию. Если запрос принимается ядром, вызывающий поток продолжает обработку другого задания, пока ядро не сигнализирует потоку о завершении операции ввода-вывода. Затем он прерывает текущее задание и обрабатывает данные из операции ввода-вывода по мере необходимости. В ситуациях, когда ожидается, что запрос ввода-вывода занимает большое количество времени, например, обновление или резервное копирование большой базы данных или канал медленного взаимодействия, асинхронный ввод-вывод обычно является хорошим способом оптимизации эффективности обработки. Однако для относительно быстрых операций ввода-вывода затраты на обработку запросов ввода-вывода ядра и сигналов ядра могут сделать асинхронные операции ввода-вывода менее полезными, особенно если требуется выполнить много быстрых операций ввода-вывода. В этом случае синхронный ввод-вывод будет лучше. Механизмы и сведения о реализации выполнения этих задач зависят от типа используемого дескриптора устройства и конкретных потребностей приложения. Другими словами, обычно существует несколько способов решения проблемы.



16. История ЭВМ до параллельного программирования.

Первые компьютеры были построены в соответствии с принципами, сформулированными фон Нейманом. Они имели три главных компонента - память, процессор и некоторый набор внешних устройств, обеспечивающих ввод и вывод информации. Программы для первых компьютеров

представляли последовательность команд, входящих в допустимый набор команд процессора.

Выполнение программы на компьютере осуществлялось достаточно просто. В каждый момент времени на компьютере выполнялась одна программа. Процессор, в соответствии с программой, последовательно выполнял одну команду за другой. Все ресурсы компьютера - память, время процессора, все устройства - были в полном распоряжении программы, и ничто не могло вмешаться в ее работу (не считая конечно человека). Параллелизма не было и в помине.

С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы — полупроводниковых элементов. В эти годы появились первые алгоритмические языки, а, следовательно, и первые системные программы — компиляторы.

Высокая стоимость процессорного времени требовало уменьшения затрат времени между запусками программы. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программы за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом.

Появление операционных систем в 1964 году привело к переходу от однопрограммного режима работы к мультипрограммному, когда на одном компьютере одновременно выполняются несколько программ. Мультипрограммирование — это еще не параллельное программирование, но это шаг в направлении параллельных вычислений.

17. Мультипрограммирование.

Революционным шагом было появление в 1964 году операционной системы фирмы IBM - OS 360, все представители которой должны были использовать одинаковые инструкции и архитектуру ввода/вывода. Появившаяся у компьютера операционная система стала его полновластным хозяином-распорядителем всех его ресурсов.

Теперь программа пользователя могла быть выполнена только под управлением операционной системы. Операционная система позволяла решить две важные задачи - с одной стороны обеспечить необходимый сервис всем программам, одновременно выполняемым на компьютере, с другой - эффективно использовать и распределять существующие ресурсы между программами, претендующими на эти ресурсы.

Появление операционных систем привело к переходу от однопрограммного режима работы к мультипрограммному, когда на одном компьютере одновременно выполняются несколько программ. *Мультипрограммирование — это еще не параллельное программирование, но это шаг в направлении параллельных вычислений.*

Мультипрограммирование - параллельное выполнение нескольких программ. Мультипрограммирование позволяет уменьшить общее время их выполнения.

18. Виды параллелизма.

Под параллельными вычислениями понимается параллельное выполнение одной и той же программы. Параллельные вычисления позволяют уменьшить время выполнения одной программы.

Параллельные вычисления следует отличать от многозадачных (многопрограммных) режимов работы ЭВМ.

Необходимость параллельных вычислений

- Теоретическая ограниченность роста производительности последовательных компьютеров
- Резкое снижение стоимости многопроцессорных (параллельных) вычислительных систем
- Смена парадигмы построения высокопроизводительных процессоров - многоядерность

Формы параллелизма:

Параллельные вычисления существуют в нескольких формах:

- **параллелизм на уровне битов** (Эта форма параллелизма основана на увеличении размера машинного слова. Увеличение размера машинного слова уменьшает количество операций, необходимых процессору для выполнения действий над переменными, чей размер превышает размер машинного слова.)
- **параллелизм на уровне инструкций** (Компьютерная программа – это, по существу, поток инструкций, выполняемых процессором. Но можно изменить порядок этих инструкций, распределить их по группам, которые будут выполняться параллельно, без изменения результата работы всей программы),
- **параллелизм данных** (Основная идея подхода, основанного на параллелизме данных, заключается в том, что одна операция выполняется сразу над всеми элементами массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции – перевода исходного текста программы в машинные команды. Роль программиста в этом случае обычно сводится к заданию настроек векторной или параллельной оптимизации компилятору, директив параллельной компиляции, использованию специализированных языков для параллельных вычислений.),
- **параллелизм задач** (Стиль программирования, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Существуют всего две цели, которые преследуют программисты, используя потоки: •позволить приложению параллельно работать над несколькими относительно независимыми задачами; • использовать преимущества многопроцессорных систем для повышения производительности приложения.).

19. Процесс. Контекст процесса.

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения (значения регистров, программного счетчика, состояния стека и значения переменных), находящуюся под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра – при использовании системных вызовов, обработке внешних прерываний.

С позиции данной абстрактной модели, у каждого процесса есть собственный виртуальный центральный процессор.

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является изоляция одного процесса от другого. Виртуальное адресное пространство процесса – это совокупность адресов, которыми может манипулировать программный модуль процесса. Операционная система отображает виртуальное адресное пространство процесса на отведенную процессу физическую память.

Процесс имеет:

- собственные области памяти под код и данные, включая значения регистров и счетчика команд;
- собственный стек;
- собственное отображение виртуальной памяти (в системах с виртуальной памятью) на физическую.

Всю необходимую для деятельности процесса информацию операционная система хранит в некоем описателе процесса, который является моделью процесса для операционной системы. Например, управляющий блок процесса (PCB - Process Control Block).

Контекст процесса

Информацию, для хранения которой предназначен PCB, удобно разделить на две части. Содержимое всех регистров процесса, включая значение программного счетчика, называется регистровым контекстом процесса, а все остальное – системным контекстом процесса. Код и данные, находящиеся в адресном пространстве процесса, называются его пользовательским контекстом. Совокупность регистрового, системного и пользовательского контекстов принято называть просто контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

20. Планирование и диспетчеризация.

Планирование и диспетчеризация

- Работа ОС по определению того, в какой момент необходимо прервать выполнение текущего активного процесса и какому процессу предоставить возможность выполняться, называется планированием. Планирование осуществляется на основе информации, хранящейся в описателях процессов. Планирование, по существу, включает в себя решение двух задач:
 - определение момента времени для смены текущего активного процесса;
 - выбор для выполнения процесса из очереди готовых процессов.

Планирование и диспетчеризация

Диспетчеризация заключается в реализации найденного в результате планирования (динамического или статистического) решения, то есть в переключении процессора с одного процесса на другой. Диспетчеризация сводится к следующему:

- сохранение контекста текущего потока, который требуется сменить;
- загрузка контекста нового потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

21. Разделяемые ресурсы.

Разделяемые и неразделяемые ресурсы, особенности параллелизма.

Ресурс – это объект (аппаратный или программный), который может одновременно использоваться только одной задачей.

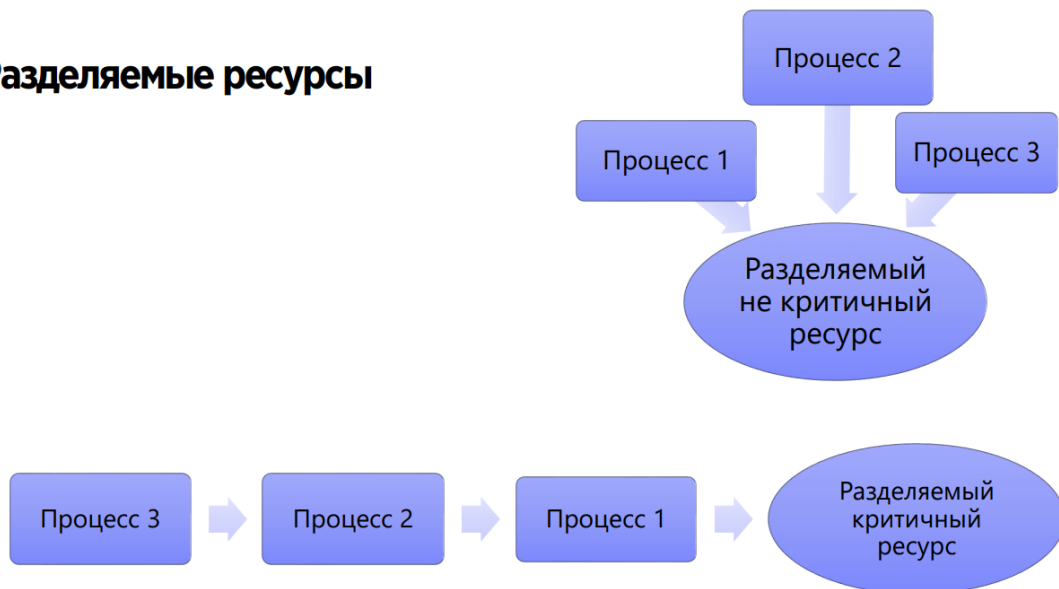
Разделяемые ресурсы – это ресурсы, которые используются несколькими процессами или потоками и существуют, пока хотя бы 1 процесс их использует.

Разделяемые ресурсы бывают критичные и не критичные.

Критичные ресурсы могут использоваться только одним процессом в один момент времени, и пока один процесс использует его, ресурс не доступен другим процессам.

Не критичные ресурсы могут использоваться несколькими процессами одновременно.

Разделяемые ресурсы

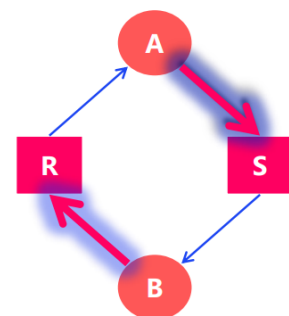


- По типу взаимодействия различают:
 - сотрудничающие процессы:
 - процессы, разделяющие только коммуникационный канал, по которому один передает данные, а другой получает их;
 - процессы, осуществляющие взаимную синхронизацию: когда работает один, другой ждет окончания его работы
 - конкурирующие процессы:
 - процессы, использующие совместно разделяемый ресурс;
 - процессы, использующие критические секции;
 - процессы, использующие взаимные исключения.

Для работы с разделяемыми ресурсами необходимо использовать средства синхронизации. С этим связано большинство проблем разработки параллельных программ. По различным причинам в программе могут оказаться проблемы синхронизации, такие как состояние состязания, deadlock, starvation, mutual-lock и другие.

22. Взаимоблокировка.

- Часто процесс для выполнения задачи нуждается в исключительном доступе не к одному, а к нескольким ресурсам.
 - Предположим, что имеется два процесса А и В. Процесс А имеет в исключительном доступе ресурс R, процесс В – ресурс S.
- Такая ситуация называется тупиком, тупиковой ситуацией или взаимоблокировкой.



Как избавиться от deadlock:

- Каждая критическая секция захватывает все общие ресурсы.
 - Это означает, что вход в каждую критическую секцию закрывается одним ключом. Недостатком такого подхода является увеличение общего времени ожидания. Во многих ситуациях неразумно, когда все ресурсы принадлежат одному владельцу, а он не пользуется ими одновременно.
- Если в критических секциях работа с ресурсами ведется последовательно, а не одновременно, то ресурс следует освобождать, как только работа с ним закончена.
 - Это общее правило работы с разделяемыми ресурсами. Оно не всегда работает, поскольку часто необходимы одновременно несколько ресурсов в каждой из критических секций.
- Захват не возникает, если есть только одна критическая секция.
 - По сути, это также захват всех ресурсов, означающий, ожидание в очереди всех потоков, пока не отработает поток, вошедший в критическую секцию.
- В основном потоке можно использовать форму ожидания, позволяющую ожидать завершения работы запущенного потока в течение времени, заданного параметром t .
 - Используя этот механизм, основной поток может корректно обработать возникшую ситуацию, возможно связанную с клинчем. В любом случае приложение не зависнет.
- Применение мягких методов блокировки, когда блокируется только запись, но не чтение ресурса.
 - В то же время, если ресурс используется только для чтения, то возможно его одновременное использование.

23. Механизмы межпроцессного взаимодействия.

Межпроцессное взаимодействие

- Хотя каждый процесс в системе, как правило, выполняет какую-либо отдельную функцию, в параллельном программировании возникает необходимость в согласовании (синхронизации) действий, выполняемых различными процессами. Процессы могут взаимодействовать только путем обмена информацией.
- Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть кооперативными или взаимодействующими процессами.
- Межпроцессное взаимодействие – это тот или иной способ передачи информации из одного процесса в другой.
- Наиболее распространенными формами взаимодействия являются:
 - Разделяемая память;
 - Критические секции;
 - Семафоры;
 - Мьютексы;
 - Мониторы;
 - Сообщения:
 - Сигналы;
 - Почтовые ящики;
 - Барьеры;

24. Прimitives синхронизации. Разделяемая память.

Разделяемая память – два или более процесса могут иметь доступ к одному и тому же блоку памяти.

В системах с виртуальной памятью организация такого вида взаимодействия требует поддержки со стороны операционной системы, поскольку необходимо отобразить соответствующие блоки виртуальной памяти на один и тот же блок физической памяти.

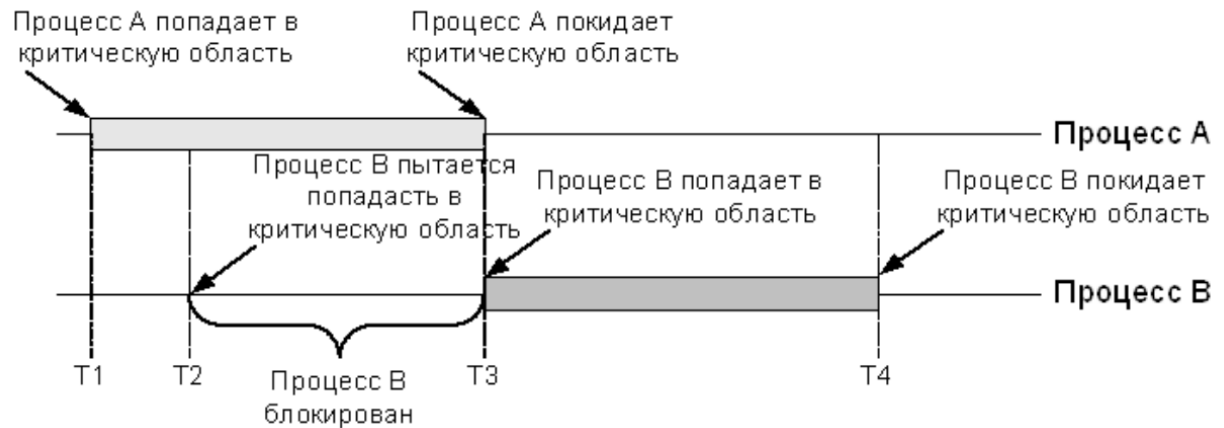
25. Прimitives синхронизации. Критическая секция.

В параллельных программах используются два основных типа синхронизации: **взаимное исключение и условная синхронизация.**

Взаимное исключение обеспечивает, чтобы критические секции операторов не выполнялись одновременно.

Условная синхронизация задерживает процесс до тех пор, пока не выполнится определенное условие.

Критическая секция (области) – это участок программы, в котором есть обращение к совместно используемым данным. На этом участке запрещается переключение задач для обеспечения исключительного использования ресурсов процессом. Говоря иными словами, необходимо взаимное исключение.



Объекты, представляющие критические секции, доступны в пределах одного процесса. Процедура входа и выхода из критических секций обычно занимает меньшее время, нежели аналогичные операции с мьютексами или семафорами, что связано с отсутствием необходимости обращаться к ядру ОС.

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

Каждая критическая секция является последовательностью операторов, имеющих доступ к некоторому разделяемому объекту. Предполагается, что процесс, вошедший в критическую секцию, обязательно когда-нибудь из нее выйдет. Таким образом, процесс может завершиться только вне критической секции.

Необходимо разработать **протоколы входа и выхода**, которые удовлетворяют следующим условиям:

- **Взаимное исключение.** В любой момент времени только один процесс может выполнить свою критическую секцию.
- **Отсутствие взаимной блокировки (живой блокировки).** Если несколько процессов пытаются войти в свои критические секции, хотя бы один это осуществит. (Отметим, что живой блокировкой называется состояние, когда все процессы работают, но навсегда заиклены в попытке войти в

критическую секцию. Возможна при решении с активным ожиданием).

- **Отсутствие излишних задержек.** Если один процесс пытается войти в свою критическую секцию, а другие выполняют свои некритические секции или завершены, первому процессу разрешается вход в критическую секцию.
- **Возможность входа.** Процесс, который пытается войти в критическую секцию, когда-нибудь это сделает.

Первые три свойства являются свойствами безопасности, четвертое – свойством живучести.

Что нужно знать о критических секциях

Критические секции работают быстро и не требуют большого количества системных ресурсов. Для синхронизации доступа к нескольким (независимым) переменным лучше использовать несколько критических секций, а не одну для всех. Код, ограниченный критическими секциями, лучше всего свести к минимуму. Находясь в критической секции, не стоит вызывать методы "чужих" объектов.

26. Прimitives синхронизации. Семафоры.

Первым средством способным быть использованным для блокирования приостанавливаемых процессов, не потерявшим до сих пор актуальности, являются семафоры, предложенные Дейкстрой в 1965 году.

Семафор – это объект синхронизации, задающий количество пользователей (задач, процессов), имеющих одновременный доступ к некоторому ресурсу. Идея семафоров взята из методов синхронизации движения поездов на железной дороге. Железнодорожный семафор – это «сигнальный флажок», показывающий, свободен путь впереди или занят другим поездом. Семафор может быть установлен на время, достаточное, пока поезд идет по занятому пути, и сбрасывается, когда путь свободен.

С каждым семафором связаны счетчик (значение семафора) и очередь ожидания (процессов, задач, ожидающих принятие счетчиком определенного значения).

Различают:

- двоичные (булевские) семафоры – это механизм взаимного исключения для защиты критичного разделяемого ресурса;
- счетные семафоры – это механизм взаимного исключения для защиты ресурса, который может быть использован не более чем ограниченным фиксированным числом задач n .

Операции над семафорами

Над семафорами определены следующие элементарные операции:

- **взять k единиц из семафора**, т.е. уменьшить счетчик на k. Если в счетчике нет k единиц, то эта операция переводит процесс в состояние ожидания наличия как минимум k единиц в семафоре, и добавляет его в конец очереди ожидания этого семафора. Значение счетчика в таком случае становится равным нулю (отрицательных значений счетчика быть не может).
- **вернуть k единиц в семафор**, т.е. увеличить счетчик на k. Если с семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию взять, один из них выбирается системой и ему разрешается завершить свою операцию. Таким образом, после операции вернуть, примененной к семафору, связанному с несколькими ожидающими процессами, значение счетчика так и остается равным 0, но число ожидающих процессов уменьшается. Активизированный процесс в случае более высокого приоритета может вытеснить текущую задачу.
- **попробовать взять k единиц из семафора**. Если в счетчике k единиц, то взять k единиц из него, иначе вернуть признак занятости семафора без перевода задачи в состояние ожидания. проверить семафор, т.е. получить значение счетчика. заблокировать семафор, т.е. взять из него столько единиц, сколько в нем есть. При этом иногда бывают две разновидности этой операции: взять столько, сколько есть в данный момент, или взять столько, сколько есть в начальный момент, т.е. максимально возможное количество. Именно последнее обычно называют блокировкой, поскольку такая задача будет монопольно владеть ресурсом.
- **разблокировать семафор**, т.е. вернуть столько единиц, сколько всего было взято данной задачей по команде заблокировать. Основными операциями являются операции взять ($\text{down}(s)$, $P(s)$) и вернуть ($\text{up}(s)$, $V(s)$).

Все операции над семафорами выполняются как атомарная операция. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции.

Мьютекс отличается от булевского семафора тем, что только владеющий им поток может его освободить, т.е. перевести в свободное состояние.

27. Прimitives синхронизации. Мьютексы.

Мьютекс (mutex, сокращение от mutual exclusion - взаимное исключение) – объект ядра ОС, служащий для управления взаимным исключением доступа к совместно используемым ресурсам и кодам.

Мьютекс фактически состоит из пары:

булевского семафора

идентификатора задачи – текущего владельца семафора (т.е. той задачи, которая успешно выполнила функцию взять и стала владельцем разделяемого ресурса).

Мьютекс не способен считать, он может лишь управлять взаимным исключением доступа к совместно используемым ресурсам или кодам.

Освободить мьютекс может только его владелец.

Операции над мьютексом

Для доступа к объекту типа мьютекс определены три примитивные операции:

Lock(m) – заблокировать мьютекс m. Если m уже заблокирован другой задачей, то эта операция переводит задачу в состояние ожидания разблокирования m.

Unlock(m) – разблокировать мьютекс m. Если m ожидается другой задачей, то она может быть активизирована, удалена из очереди ожидания и может вытеснить текущую задачу, если ее приоритет выше. Если вызвавшая эту операцию задача не является владельцем m, то операция не имеет никакого эффекта.

TryLock(m) – попробовать заблокировать мьютекс m. Если m не заблокирован, то эта операция эквивалентна Lock(m), иначе возвращается признак неудачи. Как и для семафоров, эти операции являются неделимыми. Поток может завладеть одним и тем же ресурсом несколько раз (в зависимости от параметров создания мьютекса), и при этом не будет блокироваться даже в тех случаях, когда уже владеет данным ресурсом. В конечном счете, поток должен освободить мьютекс столько раз, сколько он его захватывал.

Мьютекс, владевший которым поток завершился, не освободив его, называют покинутым (abandoned), и его дескриптор переходит в сигнальное состояние. Обнаружение покинутого мьютекса может означать наличие дефекта в коде, организующем работу потоков, поскольку потоки должны программироваться таким образом, чтобы ресурсы всегда освобождались, прежде чем поток завершит свое выполнение. Возможно также, что выполнение данного потока было прервано другим потоком.

Мьютексы и Критические секции

Мьютексам можно присваивать имена, и их могут совместно использовать потоки, принадлежащие разным процессам. (+)

Мьютексы, покинутые завершающимися потоками, переходят в сигнальное состояние, в результате чего другие потоки не будут блокироваться на неопределенное время. (+)

Имеется возможность организовать ожидание мьютекса с использованием конечного интервала ожидания, тогда как в случае объектов критических секций возможен только опрос их состояния. (+)

Поток, создающий мьютекс, может сразу же указать, что он становится его владельцем. В случае объектов критических секций за право владения объектом могут состязаться несколько потоков. (+)

Обычно, хотя и не всегда, использование объектов критических секций обеспечивает более высокую производительность по сравнению с той, которая достигается при использовании мьютексов. (-)

Мьютексы и Семафоры

Мьютекс отличается от булевого семафора тем, что только владеющий им поток может его освободить, т.е. перевести в свободное состояние.

28. Прimitives синхронизации. Мониторы.

Мониторы – примитивы синхронизации высокого уровня, представляющие собой набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет.

Монитор при создании автоматически иницирует число ресурсов и включает процедуры, позволяющие блокировать и активизировать процессы. Вход в монитор находится под жестким контролем системы и только через монитор осуществляется взаимоисключение процессов. Если процесс обращается к монитору и требуемый ресурс занят, то процесс переводится в состояние ожидания. Со временем некоторый процесс обращается к монитору для возвращения ресурса и монитор оповещает процесс о том, что может выделить ресурс и покинуть очередь. Режимом ожидания управляет сам монитор, который для гарантии получения ресурса процессом повышает приоритеты процессов критических областей.

Монитор состоит из:

- набора процедур, взаимодействующих с общим ресурсом
- мьютекса
- переменных, связанных с этим ресурсом
- инварианта, который определяет условия, позволяющие избежать состояние гонки

Процедура монитора захватывает мьютекс перед началом работы и держит его или до выхода из процедуры, или до момента ожидания условия (см. ниже). Если каждая процедура гарантирует, что перед освобождением мьютекса инвариант истинен, то никакая задача не может получить ресурс в состоянии, ведущем к гонке.

Простой пример. Рассмотрим монитор, выполняющий транзакции банковского счёта.

```
monitor account {  
  int balance = 0  
  function withdraw(int amount) {  
    if amount < 0 then error "Счёт не может быть отрицательным"  
    else if balance < amount then error "Недостаток средств"  
    else balance = balance - amount  
  }  
  
  function deposit(int amount) {  
    if amount < 0 then error "Сумма не может быть отрицательной"  
    else balance = balance + amount  
  }  
}}
```

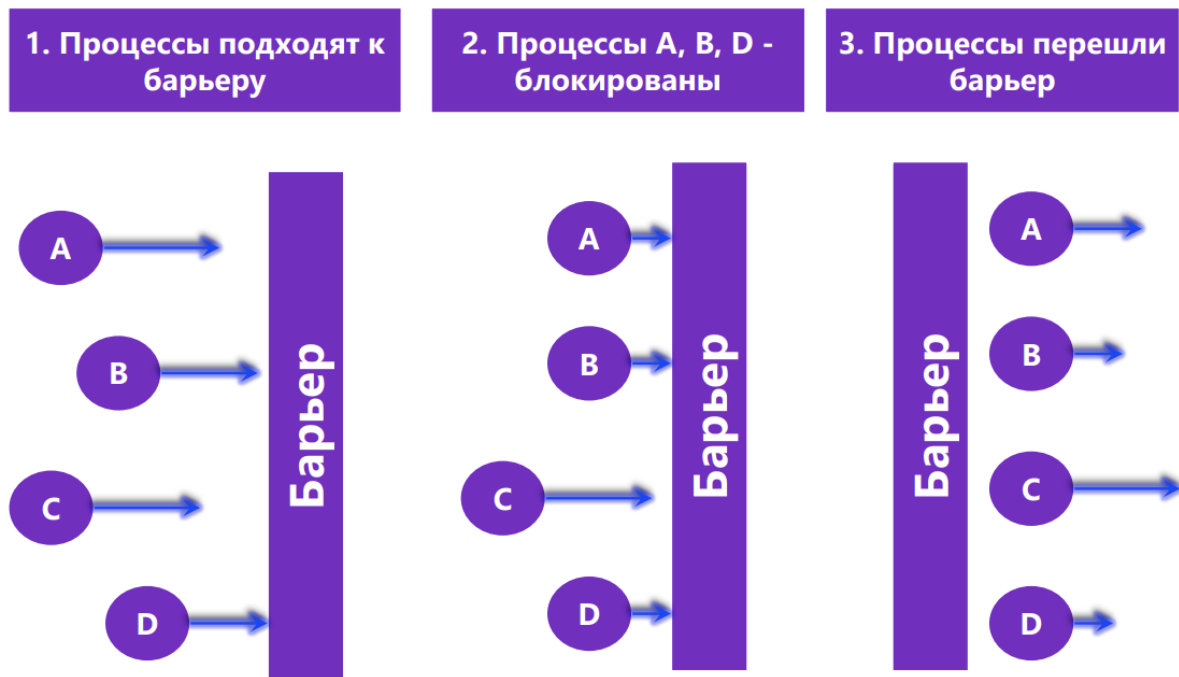
Инвариант здесь просто утверждает, что баланс должен отразить все прошедшие операции до того, как начнётся новая операция. Обычно это не выражено в коде, но подразумевается и может быть упомянуто в комментариях. Блокировка добавлена компилятором. Это делает мониторы безопаснее и удобнее, чем другие подходы, требующие от программиста вручную добавлять операции блокировки-разблокировки, — поскольку программист может забыть добавить их.

29. Примитивы синхронизации. Сигналы.

Сигналы – это сообщения, доставляемые посредством операционной системы процессу. Процесс должен зарегистрировать обработчик этого сообщения у операционной системы, чтобы получить возможность реагировать на него. Часто операционная система извещает процесс сигналом о наступлении какого-либо сбоя, например, делении на 0, или о каком-либо аппаратном прерывании, например прерывании таймера.

30. Примитивы синхронизации. Барьеры.

Барьеры – используются для организации взаимодействия групп процессов. Пусть, например, работа каждого процесса разбита на фазы и существует правило, что процесс не может перейти в следующую фазу, пока к этому не готовы все остальные процессы. Этого можно добиться, разместив в конце каждой фазы барьер. Когда процесс доходит до барьера, он блокируется, пока все процессы не достигнут барьера.



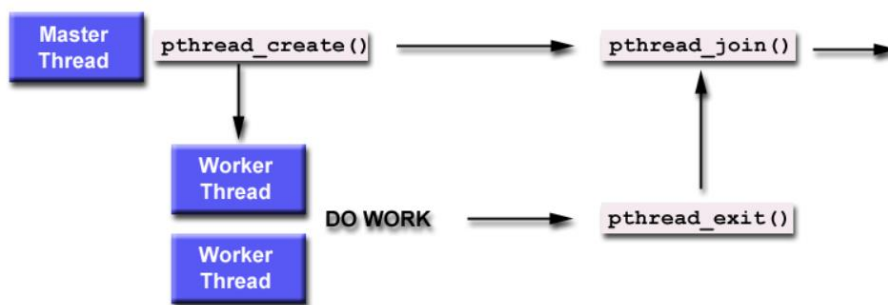
31. Потоки. Создание и уничтожение потока.

1. Стандарт POSIX.

Создание функцией «`pthread_create`».

Завершение:

1. Самостоятельное «`pthread_exit`»
2. Извне «`pthread_cancel`» и «`pthread_kill`» (где мы передаём некоторый сигнал `sig`).



2. Windows

Создание функцией «`CreateThread`», в которую передаются множество параметров, таких как:

1. Атрибут защиты потока (если параметра нет, то используется системный атрибут защиты)
2. Размер стека в байтах (если 0, то размер = размеру главного потока)

3. Адрес функции
4. Число-параметр передаваемое в поток
5. Дополнительные флаги (если 0, то код потока выполняется сразу после создания)
6. Указатель на переменную, в которую будет возвращен идентификатор потока.

Завершение:

1. Самостоятельное завершение. Функция `ExitThread` (код завершение потока);. Этот метод является предпочтительным, так как отчищает стек потока, отсоединяет динамически загружаемые библиотеки, освобождает память.
2. Извне уничтожить поток можно, используя функцию `TerminateThread`: `BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode)`, где «`hThread`» – идентификатор уничтожаемого потока; «`wExitCode`» – код завершения потока.

32. Алгоритм Деккера.

Алгоритм Деккера – первое известное корректное решение проблемы взаимного исключения в конкурентном программировании. Он позволяет двум потокам выполнения совместно использовать неразделяемый ресурс без возникновения конфликтов, используя только общую память для коммуникации.

```
shared bool flag[N]; // флаг интереса
shared int turn = 0; // от 1 - очередь
Process CS (int process){
    flag[process] = true;

    // пока другой заинтересован в попадание в критич. секцию
    while (flag[process-1] = true) {
        if (turn = process-1) {           // очередь другого потока
            flag[process] = false;
            while (turn = process-1) {} // активное ожидание;
        }
        // как только до нас дошла очередь вернуть интерес к критич. секции.
        flag[process] = true;
    }
}
// критическая секция
turn = process-1; // сменить значение очереди на противоположный поток
flag[process] = false;
// конец критической секции
}
```

Потоки последовательно пытаются войти в критич. секцию

Время	Поток 0	Поток 1
t ₀	flag[0]=false; flag[1]=false; turn=0;	
t ₁	flag[0]=true;	
t ₂	while (flag[1]=true);	
t ₃	<i>Критическая секция</i>	flag[1]=true;
t ₄		while (flag[0]=true){
t ₅		if (turn = 0) {
t ₆		flag[1] := false;
t ₇		while (turn = 0);
t ₈		while (turn = 0);
t ₉	turn := 1;	
t ₁₀	flag[0] := false;	
t ₁₁		while (turn = 0);
t ₁₂		flag[1]=true;}}
t ₁₃		while (flag[0]=true);
t ₁₄		<i>Критическая секция</i>
t ₁₅		
t ₁₆		turn := 0;
t ₁₇		flag[1] := false;

Потоки почти одновременно пытаются войти в критич. секцию

Время	Поток 0	Поток 1
t ₀	flag[0]=false; flag[1]=false; turn=0;	
t ₁	flag[0]=true;	
t ₂		flag[1]=true;
t ₃	while (flag[1]=true){	
t ₄		while (flag[0]=true){
t ₅	if (turn = 1);	
t ₆	while (flag[1]=true){	
t ₇		if (turn = 0) {
t ₈		flag[1] := false;
t ₉	if (turn = 1);	
t ₁₀	while (flag[1]=true);	
t ₁₁	<i>Критическая секция</i>	while (turn = 0);
t ₁₂		while (turn = 0);
t ₁₃		while (turn = 0);
t ₁₄	turn := 1;	
t ₁₅	flag[0] := false;	
t ₁₆		while (turn = 0);
t ₁₇		flag[1]=true;}}
t ₁₈		while (flag[0]=true);
t ₁₉		<i>Критическая секция</i>
t ₂₀		
t ₂₁		turn := 0;
t ₂₂		flag[1] := false;

В данном случае очередность определяет тот поток, который покинул критическую секцию – он передает «эстафету» следующему потоку

33. Алгоритм Петерсона.

Рассмотрим решение этой задачи с использованием активного ожидания, предложенный в 1981 году Г.Л. Петерсоном. Для простоты предположим, что число процессов $N=2$:

```
shared int turn;           // Чья сейчас очередь?
shared bool interested[N]; // Индикатор

void enter_region(int process) // Протокол входа
{
    int other = 1 - process;    // Номер противоположного процесса
    interested[process] = true; // Индикатор интереса
    turn = process;             // Установка флага
    while(turn == process && interested[other]); // Ожидание входа
}

void leave_region(int process) // Протокол выхода
{
    // Индикатор выхода из критической секции
    interested[process] = false;
}
```

- Поток вызывает enter_region последовательно:

Время	Поток 0	Поток 1
t ₀	interested [0]=false; interested [1]=false; turn=0;	
t ₁	other=1;	
t ₂	interested[0]=true;	
t ₃	turn=0;	
t ₄	while (turn==0 && interested[1]);	
t ₅	<i>Критическая секция</i>	other=0;
t ₆		interested[1]=true;
t ₇		turn=1;
t ₈		while (turn==0 && interested[0]);
t ₉		while (turn==0 && interested[0]);
t ₁₀	interested[0]=false;	
t ₁₁		while (turn==0 && interested[0]);
t ₁₂		<i>Критическая секция</i>
t ₁₃		
t ₁₄		interested[1]=false;

Потоки вызывают enter_region почти одновременно:

Время	Поток 0	Поток 1
t ₀	interested [0]=false; interested [1]=false; turn=0;	
t ₁	other=1;	
t ₂		other=0;
t ₃		interested[1]=true;
t ₄	interested[0]=true;	
t ₅	turn=0;	
t ₆		turn=1;
t ₇		while(turn==0 && interested[0]);
t ₈	while(turn==0 && interested[1]);	
t ₉	<i>Критическая секция</i>	while(turn==0 && interested[0]);
t ₁₀		while(turn==0 && interested[0]);
t ₁₁		while(turn==0 && interested[0]);
t ₁₂	interested[0]=false;	
t ₁₃		while(turn==0 && interested[0]);
t ₁₄		<i>Критическая секция</i>
t ₁₅		
t ₁₆		interested[1]=false;

34. Алгоритм пекарни Лемпорта.

Для процессов, число которых больше 2, разработан аналогичный алгоритм, называемый алгоритмом булочной (Bakery algorithm, Алгоритм пекарни Лемпорта) или алгоритм билета. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. Алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени.

Обычно количество процессов больше количества процессоров. А активное ожидание, по существу, означает, что процесс для проверки возможности входа в критическую секцию входит в бесконечный цикл. Это приводит к непроизводительному расходованию ресурсов процессора.

35. Потоки и пулы потоков в C#.

Thread (поток):

Основной функционал для использования потоков в C# сосредоточен в пространстве имен System.Threading. В нем определен класс, представляющий отдельный поток - класс Thread.

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

1. ExecutionContext: позволяет получить контекст, в котором выполняется поток

2. **IsAlive**: указывает, работает ли поток в текущий момент
3. **IsBackground**: указывает, является ли поток фоновым
4. **Name**: содержит имя потока
5. **ManagedThreadId**: возвращает числовой идентификатор текущего потока
6. **Priority**: хранит приоритет потока - значение перечисления **ThreadPriority**:

ThreadState возвращает состояние потока - одно из значений перечисления **ThreadState** (завершен, приостановлен, запущен и работает и тд).

Также класс **Thread** определяет ряд методов для управления потоком. Основные из них:

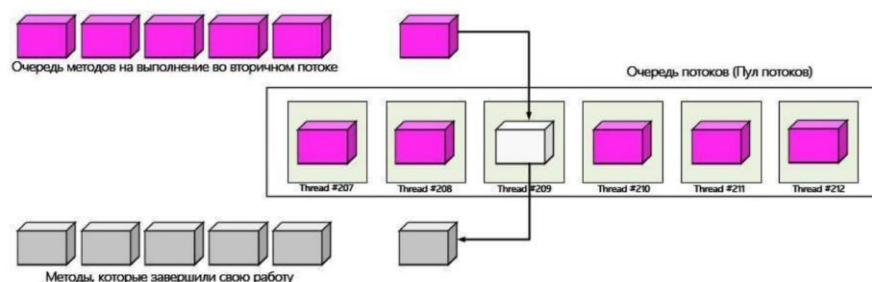
1. Статический метод **Sleep** останавливает поток на определенное количество миллисекунд.
2. Метод **Interrupt** прерывает поток, который находится в состоянии **WaitSleepJoin**.
3. Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод.
4. Метод **Start** запускает поток.

ThreadPool (пул потоков):

Предоставляет пул потоков, который можно использовать для выполнения задач, отправки рабочих элементов, обработки асинхронного ввода-вывода, ожидания от имени других потоков и обработки таймеров.

Многие приложения создают потоки, которые проводят много времени в спящем состоянии, ожидая возникновения события. Другие потоки могут переходить в спящее состояние только для периодического пробуждения для опроса сведений об изменении или обновлении сведений о состоянии. Пул потоков позволяет более эффективно использовать потоки, предоставляя приложению пул рабочих потоков, управляемых системой.

Пул потоков - Thread Pool



36. Parallel LINQ.

По умолчанию все элементы коллекции в LINQ обрабатываются последовательно, но начиная с .NET 4.0 в пространство имен System.Linq был добавлен класс `ParallelEnumerable`, который инкапсулирует функциональность PLINQ (Parallel LINQ) и позволяет выполнять обращения к коллекции в параллельном режиме.

При обработке коллекции PLINQ использует возможности всех процессоров в системе. Источник данных разделяется на сегменты, и каждый сегмент обрабатывается в отдельном потоке. Это позволяет произвести запрос на многоядерных машинах намного быстрее.

В то же время по умолчанию PLINQ выбирает последовательную обработку данных. Переход к параллельной обработке осуществляется в том случае, если это приведет к ускорению работы. Однако, как правило, при параллельных операциях возрастают дополнительные издержки. Поэтому если параллельная обработка потенциально требует больших затрат ресурсов, то PLINK в этом случае может выбрать последовательную обработку, если она не требует больших затрат ресурсов.

Поэтому смысл применения PLINQ имеется преимущественно на больших коллекциях или при сложных операциях, где действительно выгода от распараллеливания запросов может перекрыть возникающие при этом издержки.

Также следует учитывать, что при доступе к общему разделяемому состоянию в параллельных операциях будет неявно использоваться синхронизация, чтобы избежать взаимоблокировки доступа к этим общим ресурсам. Затраты на синхронизацию ведут к снижению производительности, поэтому желательно избегать или ограничивать применения в параллельных операциях разделяемых ресурсов.

Метод `AsParallel()` позволяет распараллелить запрос к источнику данных. Он реализован как метод расширения LINQ у массивов и коллекций. При вызове данного метода источник данных разделяется на части (если это возможно) и над каждой частью отдельно производятся операции.

Рассмотрим простейший пример нахождения квадратов чисел:

```
1 int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, };
2 var squares = from n in numbers.AsParallel()
3               select Square(n);
4
5 foreach (var n in squares)
6     Console.WriteLine(n);
7
8 int Square(int n) => n * n;
```

Фактически здесь обычный запрос LINQ, только к источнику данных применяется метод `AsParallel`.

`ForAll()` – метод, который выводит данные в том же потоке, в котором они обрабатываются.

37. Потокобезопасные коллекции в C#.

Лично не имел дела с данным стеком разработки, но теоретически знаю, что...

В .NET Framework 4 введено пространство имен `System.Collections.Concurrent`, включающее несколько потокобезопасных и масштабируемых классов коллекций. Несколько потоков могут безопасно и эффективно добавлять, и удалять элементы из таких коллекций, не требуя при этом дополнительной синхронизации в пользовательском коде.

Тип	Описание
<code>BlockingCollection<T></code>	Предоставляет возможности блокировки и ограничения для всех типов, реализующих интерфейс <code>IProducerConsumerCollection<T></code> . Дополнительные сведения см. в разделе Общие сведения о коллекции BlockingCollection .
<code>ConcurrentDictionary<TKey,TValue></code>	Потокобезопасная реализация словаря пар "ключ-значение".
<code>ConcurrentQueue<T></code>	Потокобезопасная реализация очереди с типом "первым поступил — первым обслужен" (FIFO).
<code>ConcurrentStack<T></code>	Потокобезопасная реализация стека с типом "последним поступил — первым обслужен" (LIFO).
<code>ConcurrentBag<T></code>	Потокобезопасная реализация неупорядоченной коллекции элементов.
<code>IProducerConsumerCollection<T></code>	Это интерфейс, тип которого должен быть реализован для использования в классе <code>BlockingCollection</code> .

38. Streams в Java.

Java Stream – это компонент, способный выполнять внутреннюю итерацию своих элементов, то есть он может выполнять итерацию своих элементов сам.

Потоки Java (Streams) используются для работы с коллекциями данных. Сами по себе они не являются структурой данных, но могут использоваться для ввода информации из других структур данных путем упорядочивания и конвейерной обработки для получения окончательного результата.

Примечание: Важно не путать Stream и Thread, Stream обозначает объект для выполнения операций (чаще всего передача данных или их накопление), тогда как Thread (дословный перевод — нить) обозначает объект, который позволяет выполнить определенный программный код параллельно с другими ветками кода.

39. Потоки и пулы потоков в Java.

Потоки:

Создание потоковых «элементов» возможно через:

1. класс, реализующий интерфейс Runnable;
2. классы, расширяющие Thread;
3. реализацию `java.util.concurrent.Callable`.

Первый вариант встречается на практике чаще всего. Связано это с тем, что Java реализует интерфейс не в единственном количестве. Это дает возможность наследования классов.

Метод Runnable

На практике все перечисленные способы создания threads не слишком трудно реализовать. В случае с Runnable потребуется:

1. создать объект класса thread;
2. сделать class object, который реализовывает интерфейс Runnable;
3. вызвать метод `start()` у объекта thread.

Все это поможет сделать new thread, а затем использовать его.

Метод Thread

Еще один вариант – наследование. Для этого предстоит:

1. сделать class object `ClassName extends Thread`;
2. обеспечить предопределение `run()` в соответствующем классе.

Позже будет приведен пример, в котором осуществляется передача имени потока «Second»

Через Concurrent

В этом случае потребуется:

1. сделать объект класса, работающего с интерфейсом Callable;
2. обеспечить создание ExecutorService, в котором пользователь указывает пул потокового характера;
3. создать Future object.

Последний шаг осуществляется путем внедрения метода submit.

Thread в Java встречается в нескольких состояниях:

new – создан;

runnable – запуск;

blocked – блокировка;

terminated – завершение;

waiting – ожидание.

Пул потоков:

Пул потоков — это набор объектов Runnable и постоянно работающих потоков. Коллекция объектов Runnable называется рабочей очередью. Постоянно запущенные потоки проверяют рабочий запрос на наличие новой работы, и если новая работа должна быть выполнена, то из рабочей очереди будет запущен объект Runnable.

Пул потоков полезен для организации серверных приложений, и очень важно правильно его реализовать, чтобы предотвратить любые проблемы, такие как взаимоблокировка и сложность использования для wait() или notify(). Поэтому рекомендуется рассмотреть возможность использования одного из классов Executor из util.concurrent, такого как ThreadPoolExecutor, а не писать пул потоков с нуля. Если требуется создать потоки для обработки краткосрочных задач, вы можете вместо этого использовать пул потоков.

40. Интерфейс CompletableFuture.

Интерфейс CompletableFuture — средство для передачи информации между параллельными потоками исполнения. По существу, это блокирующая очередь, способная передать только одно ссылочное значение. В отличие от обычной очереди, передает также исключение, если оно возникло при вычислении передаваемого значения.

41. Потокобезопасные коллекции в Java.

В ранних версиях Java можно было «сделать» коллекцию потокобезопасной, обернув в Collections.synchronizedXXX(...). Это сериализовывало любой доступ к внутреннему состоянию коллекции. Из-за поддержки обратной совместимости сейчас так тоже можно, но не нужно.

Цена такого решения — плохой параллелизм: конкуренция за блокировку (lock contention).

С версии 5 появились классы, специально разработанные для потокобезопасности, с меньшим количеством блокировок.

Их использование является предпочтительным.

1. CopyOnWriteArrayList и CopyOnWriteArraySet:

Это структуры данных на основе массива. Они пересоздают всё заново при каждой модификации. Это дорого, зато все читающие итераторы стабильны. Хороши, когда на одну операцию записи приходится много операций чтения.

2. ConcurrentLinkedQueue/Deque

Основаны на неблокирующем алгоритме (CAS-операции). poll() вернёт null, если очередь пуста. Под капотом — связные/двусвязные списки.

3. ConcurrentHashMap

Замена HashMap при разделённом доступе к данным. Не блокируется при чтении и редко блокируется при записи. Не позволяет использовать null в качестве ключа или значения.

4. ConcurrentSkipListMap

Замена TreeMap при разделённом доступе к данным. Не позволяет использовать null в качестве ключа или значения. Имеет атомарные методы.