# 17. Алгоритм выполнения многопоточной программы. Блокировка потоков.

- потоки управления (threads) образуются и работают в рамках одного процесса.

- в однопоточной программе используется только 1 главный поток (некоторая “нить” последовательно исполняемых операторов, которая начинается при запуске программы, является непрерывной и обрывается только по завершении программы)

- при использовании многопоточности, «нить» (главный поток) в некоторой точке программы делится на две, три и т.д. нити, которые в дальнейшем так же могут порождать другие потоки от себя. Но при этом **важно знать**, что один из потоков **всегда остается главным**, и его завершение означает завершение всей программы!

- в каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Несмотря на внешнюю простоту такого механизма, деятельность потоков при этом должна быть полностью согласованна. Одновременная попытка изменения двумя и более потоками объекта скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы.

- Классическим средством согласования потоков являются **семафоры**. Самый простой – **lock(замок)**  или **mutex(взаимоисключения)**. Если в некоторой части программы нам необходимо сохранить целостность данных согласовав потоки, то логично использовать замки. Принцип их работы можно описать как эстафету с палочкой. Пока данными пользуется один поток, он забирает «эстафетную палочку» , в то время как следующий поток ожидает передачи «палочки» от первого потока, чтобы потом начать использовать те же данные.

# 18. Доступ к общим ресурсам в многопоточной программе. Механизмы блокировки ресурсов модуля threading.

Одновременная попытка изменения двумя и более потоками объекта в многопоточной программе скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы.

Поддержка многопоточности в языке Python доступна через использование ряда модулей. В стандартном модуле threading определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров (классы замков Lock, RLock и класс Semaphore ) и другие механизмы взаимодействия между потоками (классы Event и Condition ), класс Timer для запуска функции по прошествии некоторого времени. Модуль Queue реализует очередь, которой могут пользоваться сразу несколько потоков.

**Замки** Простейший замок может быть реализован на основе класса Lock модуля threading. Замок имеет два состояния: он может быть или открыт, или заперт. В последнем случае им владеет некоторый поток. Объект класса Lock имеет следующие методы:

• acquire([blocking=True]) Делает запрос на запирание замка. Если параметр blocking не указан или является истиной, то поток будет ожидать освобождения замка. Если параметр не был задан, метод не возвратит значения. Если blocking был задан и истинен, метод возвратит True (после успешного овладения замком). Если блокировка не требуется (то есть задан blocking=False ), метод вернет True, если замок не был заперт и им успешно овладел данный поток. В противном случае будет возвращено False.

• release() Запрос на отпирание замка.

• locked() Возвращает текущее состояние замка ( True - заперт, False - открыт). Следует иметь в виду, что даже если состояние замка только что проверено, это не означает, что он сохранит это состояние до следующей команды.

Имеется еще один вариант замка - threading.RLock, который отличается от threading.Lock тем, что некоторый поток может запрашивать его запирание много раз. Отпирание такого замка должно происходить столько же раз, сколько было запираний. Это может быть полезно, например, внутри рекурсивных функций.

Замки позволяют ограничивать вход в некоторую область программы одним потоком. Замки могут потребоваться для обеспечения целостности структуры данных. Например, если для корректной работы программы требуется добавление определенного элемента сразу в несколько списков или словарей, такие операции в многопоточном приложении следует обставить замками. Вокруг атомарных операций над встроенными типами (операций, которые не вызывают исполнение какого-то другого кода на Python) замки ставить необязательно. Например, метод append() (встроенного) списка является атомарной операцией, а тот же метод, реализованный пользовательским классом, может требовать блокировок. В случае сомнений, конечно, лучше перестраховаться и поставить замки, однако следует минимизировать общее время действия замка, так как замок останавливает другие потоки, пытающиеся попасть в ту же область программы. Отсутствие замка в критической части программы, работающей над общими для двух и более потоков ресурсами, может привести к случайным, трудноуловимым ошибкам.

**Семафоры** являются более общим механизмом синхронизации потоков, нежели замки. Семафоры могут допустить в критическую область программы сразу несколько потоков. Семафор имеет счетчик запросов, уменьшающийся с каждым вызовом метода acquire() и увеличивающийся при каждом вызове release(). Счетчик не может стать меньше нуля, поэтому в таком состоянии потокам приходится ждать, как и в случае с замками, пока значение счетчика не увеличится.

Конструктор класса threading.Semaphore принимает в качестве (необязательного) аргумента начальное состояние счетчика (по умолчанию оно равно 1, что соответствует замку класса Lock ).

Методы acquire() и release() действуют аналогично описанным выше одноименным методам у замков. Семафор может применяться для охраны ограниченного ресурса. Например, с его помощью можно вести пул соединений с базой данных. Пример такого использования семафора (заимствован из документации к Python) дан ниже: from threading import BoundedSemaphore maxconnections = 5 # Подготовка семафора pool\_sema = BoundedSemaphore(value=maxconnections) # Внутри потока: pool\_sema.acquire() conn = connectdb() # ... использование соединения ... conn.close() pool\_sema.release() Таким образом, применяется не более пяти соединений с базой данных. В примере использован класс threading.BoundedSemaphore. Экземпляры этого класса отличаются от экземпляров класса threading.Semaphore тем, что не дают сделать release() больше, чем сделан acquire().

**События.** Еще одним способом коммуникации между объектами являются события. Экземпляры класса threading.Event могут быть использованы для передачи информации о наступлении некоторого события от одного потока одному или нескольким другим потокам. Объекты-события имеют внутренний флаг, который может находиться в установленном или сброшенном состоянии. При своем создании флаг события находится в сброшенном состоянии. Если флаг в установленном состоянии, ожидания не происходит: поток, вызвавший метод wait() для ожидания события, просто продолжает свою работу. Ниже приведены методы экземпляров класса threading.Event:

• set() Устанавливает внутренний флаг, сигнализирующий о наступлении события. Все ждущие данного события потоки выходят из состояния ожидания.

• clear() Сбрасывает флаг. Все события, которые вызывают метод wait() этого объекта-события, будут находиться в состоянии ожидания до тех пор, пока флаг сброшен, или по истечении заданного таймаута.

• isSet() Возвращает состояние флага.

• wait([timeout]) Переводит поток в состояние ожидания, если флаг сброшен, и сразу возвращается, если флаг установлен. Аргумент timeout задает таймаут в секундах, по истечении которого ожидание прекращается, даже если событие не наступило. Составить пример работы с событиями предлагается в качестве упражнения.

**Условия.**

Более сложным механизмом коммуникации между потоками является механизм условий. Условия представляются в виде экземпляров класса threading.Condition и, подобно только что рассмотренным событиям, оповещают потоки об изменении некоторого состояния. Конструктор класса threading.Condition принимает необязательный параметр, задающий замок класса threading.Lock или threading.RLock. По умолчанию создается новый экземпляр замка класса threading.RLock. Методы объекта-условия описаны ниже:

• acquire(...) Запрашивает замок. Фактически вызывается одноименный метод принадлежащего объекту-условию объекта-замка.

• release() Снимает замок.

• wait([timeout]) Переводит поток в режим ожидания. Этот метод может быть вызван только в том случае, если вызывающий его поток получил замок. Метод снимает замок и блокирует поток до появления объявлений, то есть вызовов методов notify() и notifyAll() другими потоками. Необязательный аргумент timeout задает таймаут ожидания в секундах. При выходе из ожидания поток снова запрашивает замок и возвращается из метода wait().

• notify() Выводит из режима ожидания один из потоков, ожидающих данные условия. Метод можно вызвать, только овладев замком, ассоциированным с условием. Документация предупреждает, что в будущих реализациях модуля из целей оптимизации этот метод будет прерывать ожидание сразу нескольких потоков. Сам по себе метод notify() не приводит к продолжению выполнения ожидавших условия потоков, так как этому препятствует занятый замок. Потоки получают управление только после снятия замка потоком, вызвавшим метод notify().

• notifyAll() Этот метод аналогичен методу notify(), но прерывает ожидание всех ждущих выполнения условия потоков.

В следующем примере условия используются для оповещения потоков о прибытии новой порции данных (организуется связь производитель - потребитель, producer - consumer):

import threading

cv = threading.Condition()

class Item:

"""Класс-контейнер для элементов, которые будут потребляться в потоках"""

def \_\_init\_\_(self):

self.\_items = []

def is\_available(self):

return len(self.\_items) > 0

def get(self):

return self.\_items.pop()

def make(self, i):

self.\_items.append(i)

item = Item()

def consume():

"""Потребление очередного элемента (с ожиданием его появления)"""

cv.acquire()

while not item.is\_available():

cv.wait()

it = item.get()

cv.release()

return it

def consumer():

while True:

print consume()

def produce(i):

"""Занесение нового элемента в контейнер и оповещение потоков"""

cv.acquire()

item.make(i)

cv.notify()

cv.release()

p1 = threading.Thread(target=consumer, name="t1")

p1.setDaemon(True)

p2 = threading.Thread(target=consumer, name="t2")

p2.setDaemon(True)

p1.start()

p2.start()

produce("ITEM1")

produce("ITEM2")

produce("ITEM3")

produce("ITEM4")

p1.join()

p2.join()

В этом примере условие cv отражает наличие необработанных элементов в контейнере item. Функция produce() "производит" элементы, а consume(), работающая внутри потоков, "потребляет". Стоит отметить, что в приведенном виде программа никогда не закончится, так как имеет бесконечный цикл в потоках, а в главном потоке - ожидание завершения этих потоков. Еще одна особенность - признак демона, установленный с помощью метода setDaemon() объекта-потока до его старта.

**Очередь.** Процесс, показанный в предыдущем примере, имеет значение, достойное отдельного модуля. Такой модуль в стандартной библиотеке языка Python есть, и он называется Queue.

Помимо исключений - Queue.Full (очередь переполнена) и Queue.Empty (очередь пуста) - модуль определяет класс Queue, заведующий собственно очередью.

Собственно, здесь можно привести аналог примера выше, но уже с использованием класса Queue.Queue:

Изображение выглядит как снимок экрана

Автоматически созданное описание

# 19. Работа с файловой системой в Python. Основные операции.

Модуль OS в python предоставляет функции для взаимодействия с операционной системой. OS, поставляется под стандартные служебные модули Python. Этот модуль предоставляет портативный способ использования функциональных возможностей, зависящих от операционной системы. Модули \* os \* и \* os.path \* включают в себя множество функций для взаимодействия с файловой системой.

**os.name** - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

**os.environ** - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

**os.getlogin**() - имя пользователя, вошедшего в терминал (Unix).

**os.getpid**() - текущий id процесса.

**os.uname**() - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

**os.access**(path, mode, \*, dir\_fd=None, effective\_ids=False, follow\_symlinks=True) - проверка доступа к объекту у текущего пользователя. Флаги: **os.F\_OK** - объект существует, **os.R\_OK** - доступен на чтение, **os.W\_OK** - доступен на запись, **os.X\_OK** - доступен на исполнение.

**os.chdir**(path) - смена текущей директории.

**os.chmod**(path, mode, \*, dir\_fd=None, follow\_symlinks=True) - смена прав доступа к объекту (mode - восьмеричное число).

**os.chown**(path, uid, gid, \*, dir\_fd=None, follow\_symlinks=True) - меняет id владельца и группы (Unix).

**os.getcwd**() - текущая рабочая директория.

**os.link**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None, follow\_symlinks=True) - создаёт жёсткую ссылку.

**os.listdir**(path=".") - список файлов и директорий в папке.

**os.mkdir**(path, mode=0o777, \*, dir\_fd=None) - создаёт директорию. OSError, если директория существует.

**os.makedirs**(path, mode=0o777, exist\_ok=False) - создаёт директорию, создавая при этом промежуточные директории.

**os.remove**(path, \*, dir\_fd=None) - удаляет путь к файлу.

**os.rename**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает файл или директорию из src в dst.

**os.renames**(old, new) - переименовывает old в new, создавая промежуточные директории.

**os.replace**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает из src в dst с принудительной заменой.

**os.rmdir**(path, \*, dir\_fd=None) - удаляет пустую директорию.

**os.removedirs(path)** - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

**os.symlink**(source, link\_name, target\_is\_directory=False, \*, dir\_fd=None) - создаёт символическую ссылку на объект.

**os.sync**() - записывает все данные на диск (Unix).

**os.truncate**(path, length) - обрезает файл до длины length.

**os.utime**(path, times=None, \*, ns=None, dir\_fd=None, follow\_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

**os.walk**(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

**os.system**(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

**os.urandom**(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

[os.path](https://pythonworld.ru/moduli/modul-os-path.html) - модуль, реализующий некоторые полезные функции на работы с путями.