Билет 11.

Алгоритмы, ограниченные процессором и вводом-выводом. Основные характеристики, особенности выполнения и распараллеливания.

Процессы можно классифицировать как те, которые ограничены скоростью ввода-вывода (I/O-bound), и те, которые ограничены скоростью процессора(processor-bound). К первому типу относятся процессы, которые большую часть своего времени выполнения тратят на отправку запросов на ввод-вывод информации и на ожидание ответов на эти запросы. Следовательно, такие процессы часто готовы к выполнению, но могут выполняться только в течение короткого периода времени, так как в конце концов они блокируются в ожидании выполнения ввода-вывода (имеются в виду не только дисковые операции ввода-вывода, но и любой другой тип ввода-вывода информации, как, например, работа с клавиатурой).

Процессы, ограниченные скоростью процессора, наоборот, большую часть времени исполняют программный код. Такие процессы обычно выполняются до того момента, пока они не будут вытеснены, так как эти процессы не блокируются в ожидании на запросы ввода-вывода. Поскольку такие процессы не влияют на скорость ввода-вывода, то для обеспечения нормальной скорости реакции системы не требуется, чтобы они выполнялись часто. Стратегия планирования процессов, ограниченных скоростью процессора, поэтому предполагает, что такие процессы должны выполняться реже, но более продолжительный период времени. Конечно, оба эти класса процессов взаимно не исключают друг друга. Пример процесса, ограниченного скоростью процессора, — это выполнение бесконечного цикла.

Распараллеливание программ — процесс адаптации [алгоритмов](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), записанных в виде программ, для их эффективного исполнения на вычислительной системе [параллельной архитектуры](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) (в последнее время, как правило, на [многопроцессорной](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) [вычислительной системе](https://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C)). Заключается либо в переписывании программ на специальный язык, описывающий параллелизм и понятный [трансляторам](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80) целевой вычислительной системы, либо к вставке специальной разметки (например, инструкций [MPI](https://ru.wikipedia.org/wiki/Message_Passing_Interface) или [OpenMP](https://ru.wikipedia.org/wiki/OpenMP" \o "OpenMP)).

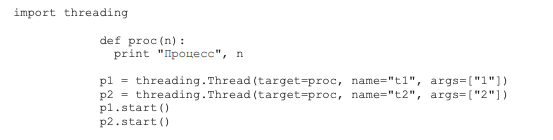
Распараллеливание может быть ручным, автоматизированным и полуавтоматизированным. Для оценки эффективности его качества применяются следующие критерии:

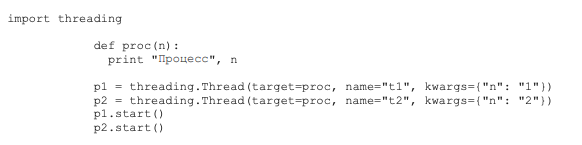
Ускорение  — время исполнения распараллеленной программы на p процессорах,  T1— время исполнения исходной программы. В идеальном случае (отсутствие накладных расходов на организацию параллелизма) равна p.

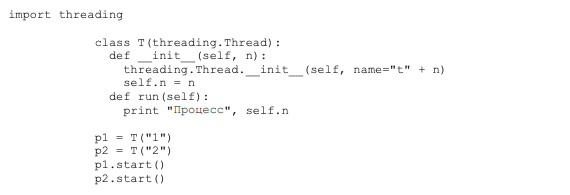
Загруженность , показывающая долю использования процессоров. В идеальном случае равна 1, или 100 %. Эта величина зачастую гораздо более наглядно характеризует эффективность параллелизма в серии испытаний при разных p, чем Sp, особенно на графиках.

Билет 12.

Особенности реализации многопоточности в Python. Модуль threading.

О потоках управления В современной операционной системе, даже не выполняющей ничего особенного, могут одновременно работать несколько процессов (processes). Например, при запуске программы запускается новый процесс. Функции для управления процессами можно найти в стандартном модуле os языка Python. Здесь же речь пойдет о потоках. Потоки управления (threads) образуются и работают в рамках одного процесса. В однопоточном приложении (программе, которая не использует дополнительных потоков) имеется только один поток управления. Говоря упрощенно, при запуске программы этот поток последовательно исполняет встречаемые в программе операторы, направляясь по одной из альтернативных ветвей оператора выбора, проходит через тело цикла нужное число раз, выбирается к месту обработки исключения при возбуждении исключения. В любой момент времени интерпретатор Python знает, какую команду исполнить следующей. После исполнения команды становится известно, какой команде передать управление. Эта ниточка непрерывна в ходе выполнения программы и обрывается только по ее завершении. Теперь можно представить себе, что в некоторой точке программы ниточка раздваивается, и каждый поток идет своим путем. Каждый из образовавшихся потоков может в дальнейшем еще несколько раз раздваиваться. (При этом 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. Такое, казалось бы, незначительное усложнение механизма выполнения программы на самом деле требует качественных изменений в программе - ведь деятельность потоков должна быть согласована. Нельзя допускать, чтобы потоки одновременно изменяли один и тот же объект, результат такого изменения, скорее всего, нарушит целостность объекта. Одним из классических средств согласования потоков являются объекты, называемые семафорами. Семафоры не допускают выполнения некоторого участка кода несколькими потоками одновременно. Самый простой семафор - замок (lock) или mutex (от английского mutually exclusive, взаимоисключающий). Для того чтобы поток мог продолжить выполнение кода, он должен сначала захватить замок. После захвата замка поток выполняет определенный участок кода и потом освобождает замок, чтобы другой поток мог его получить и пройти дальше к выполнению охраняемого замком участку программы. Поток, столкнувшись с занятым другим потоком замком, обычно ждет его освобождения. 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. Для создания и (низкоуровневого) управления потоками в стандартном модуле thread определен класс Thread. Пример многопоточной программы В следующем примере создается два дополнительных потока, которые выводят на стандартный вывод каждый свое: 

Сначала получается два объекта класса Thread, которые затем и запускаются с различными аргументами. В данном случае в потоках работает одна и та же функция proc(), которой передается один аргумент, заданный в именованном параметре args конструктора класса Thread. Нетрудно догадаться, что метод start() служит для запуска нового потока. Таким образом, в приведенном примере работают три потока: основной и два дополнительных (с именами "t1" и "t2" ). Функции модуля threading В модуле threading, который здесь используется, есть функции, позволяющие получить информацию о потоках: • activeCount() Возвращает количество активных в настоящий момент экземпляров класса Thread. Фактически, это len(threading.enumerate()). • currentThread() Возвращает текущий объект-поток, то есть соответствующий потоку управления, который вызвал эту функцию. Если поток не был создан через модуль threading, будет возвращен объект-поток с сокращенной функциональностью (dummy thread object). • enumerate() Возвращает список активных потоков. Завершившиеся и еще не начатые потоки не входят в список. Класс Thread Экземпляры класса threading.Thread представляют потоки Python-программы. Задать действия, которые будут выполняться в потоке, можно двумя способами: передать конструктору класса исполняемый объект и аргументы к нему или путем наследования получить новый класс с переопределенным методом run(). Первый способ был рассмотрен в примере выше. Конструктор класса threading.Thread имеет следующие аргументы: Thread(group, target, name, args, kwargs) Здесь group - группа потоков (пока что не используется, должен быть равен None ), target - объект, который будет вызван в методе run(), name - имя потока, args и kwargs - последовательность и словарь позиционных и именованных параметров (соответственно) для вызова заданного в параметре target объекта. В примере выше были использованы только позиционные параметры, но то же самое можно было выполнить и с применением именованных параметров: 

То же самое можно проделать через наследование от класса threading.Thread с определением собственного конструктора и метода run():

Самое первое, что необходимо сделать в конструкторе - вызвать конструктор базового класса. Как и раньше, для запуска потока нужно выполнить метод start() объекта-потока, что приведет к выполнению действий в методе run(). Жизнью потоков можно управлять вызовом методов: • start() Дает потоку жизнь. • run() Этот метод представляет действия, которые должны быть выполнены в потоке. • join([timeout]) Поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван. Параметр timeout (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод join был вызван. Вызывать join() некоторого потока можно много раз. Поток не может вызвать метод join() самого себя. Также нельзя ожидать завершения еще не запущенного потока. Слово "join" в переводе с английского означает "присоединить", то есть, метод, вызвавший join(), желает, чтобы поток по завершении присоединился к вызывающему метод потоку. • getName() Возвращает имя потока. Для главного потока это "MainThread". • setName(name) Присваивает потоку имя name. • isAlive() Возвращает истину, если поток работает (метод run() уже вызван, но еще не завершился). • isDaemon() Возвращает истину, если поток имеет признак демона. Программа на Python завершается по завершении всех потоков, не являющихся демонами. Главный поток демоном не является. • setDaemon(daemonic) Устанавливает признак daemonic того, что поток является демоном. Начальное значение этого признака заимствуется у потока, запустившего данный. Признак можно изменять только для потоков, которые еще не запущены. В модуле Thread пока что не реализованы возможности, присущие потокам в Java (определение групп потоков, приостановка и прерывание потоков извне, приоритеты и некоторые другие вещи), однако они, скорее всего, будут созданы в недалеком будущем.