

Еще!

спродажа


**Больше информации на сайте
рекламодателя**

Сроки проведения распродажи с 18.09.23 по 30.09.23. Товары, участвующие в распродаже, условия их распродажи определяются продавцами таких товаров и отмечаются значком «Распродажа». Представлен собирательный образ товара.


12+

2. Если необходимо получать результат работы нескольких потоков, но в той последовательности, в которой стоят задачи (ведь потоки могут возвращать результаты не по порядку), то используйте [очередь с приоритетом](#).
3. Если потоки трудятся над разными задачами и результаты работы потоков смешивать нельзя, то возвращаемые значения можно складывать в [словарь](#), где ключами могут быть имена потоков или простые [идентификаторы потоков](#)

РЕКЛАМА



БОГАТЫЙ ФАСАД
ПО ДОСТУПНОЙ ЦЕНЕ



УЗНАТЬ ПОДРОБНЕЕ

Задача, но запущено несколько экземпляров программы и каждая работает в несколько потоков, то работы будет [интегральный идентификатор текущего потока threading.get_native_id\(\)](#).
Если программы, потокам необходимо обмениваться результатами, то подойдет та же [многопоточная модель обмена информацией между потоками](#): в примере происходит чтение и обработка файлов из последующей передачи информации в 3-й поток, в котором она записывается в общий файл.
Если потоков нужно получать в реальном времени в основном потоке программы (хотя я не знаю зачем в реальном времени см. пункт 5.), то можно в цикле проверять, живы ли потоки, и пока они живы из той же очереди, в которую потоки будут складывать результаты.

Рабочий пример.

```
import queue, random

def worker(data, result):
    while not data.empty():
        # получаем задание из очереди с данными
        task = data.get()
        # для приличия, умножим хотя бы на 2
        res = task * 2
        # результаты будем возвращать как кортеж,
        # в котором будет (результат и ID_потока)
        result.put((res, threading.get_ident()))
        # имитируем нагрузку
        t_sleep = random.uniform(0.5, 2)
        time.sleep(t_sleep)
        # говорим очереди с данными 'data',
        # что задание выполнено
        data.task_done()

# заполняем очередь заданиями для потоков
# пускай это будет простой список чисел,
# которые потоки будут возвращать
data = queue.Queue()
for i in range(10, 20):
    data.put(i)

# очередь с возвращаемыми
# результатами работы потоков
result = queue.Queue()

# создаем и запускаем потоки
for _ in range(3):
    # имена потоков будут одинаковыми, что бы можно
    # было их отличить от основного потока программы
    thread = threading.Thread(name='worker',
                              target=worker,
                              args=(data, result,))


    thread.start()

# получаем результаты работы потоков в реальном
# времени в основном потоке программы.
t_start = time.time()
# цикл, пока жив хоть один поток 'worker'
while any(th.is_alive()
           for th in threading.enumerate()
           if th.name == 'worker'):

    # !Внимание! очередь с результатами при
    # работе потоков с разной нагрузкой,
    # короткое промежутки может быть пустой,
    # поэтому же мы сразу извлекаем результаты
```

```
if not result.empty():
    res, id_thread = result.get()
    # прошедшее время с момента запуска потоков
    tm = round(time.time() - t_start, 2)
    print(f'Поток {id_thread}: результат {res}, время: {tm}')
```

РЕКЛАМА



```
# льтат 24, время: 0.62
# льтат 20, время: 0.86
# льтат 26, время: 1.49
# льтат 22, время: 1.94
# льтат 28, время: 2.04
# льтат 30, время: 3.19
# льтат 34, время: 3.79
# льтат 32, время: 3.8
# льтат 36, время: 4.59
```

Планируя использование потоков в параллельной обработке файлов. В первую очередь создать каталог на предмет файлов с расширением .txt, а потом обрабатывать их например в 3-х потоках. В результате получится в изменении строк и запись измененных данных в другой каталог. Следовательно, для работы с файлами нужен новый каталог и текстовые файлы.

```
# prepare-data.py

import pathlib, random

path = pathlib.Path('.')
# название тестовой директории
test_dir = 'test_dir'
# Путь к тестовой директории
path_dir = path.joinpath(test_dir)
# создаем тестовый директорий
path_dir.mkdir(exist_ok=True)
# количество создаваемых файлов
n_files = 50

# скобочки {} - это шаблон для метода строки
# str.format() туда вставим имя файла
line = "{} - Эту строку будем писать в файл"

if path_dir.is_dir():
    for n in range(n_files):
        # название файла
        f_name = f'file-{n}.txt'
        # путь к файлу
        path_file = path_dir.joinpath(f_name)
        # Генерируем разное количество строк,
        # которые будут писаться в файл
        data = [line.format(f_name) for _ in range(random.randint(5000,15000))]
        # пишем данные в файл
        path_file.write_text('\n'.join(data))
```

Теперь сама программа многопоточной обработки файлов.

Предупреждение: При такой обработке файлов прирост производительности будет незначительным по сравнению с однопоточной обработкой, так как во-первых: 3 потока создают дополнительную загрузку файловой системы (одновременное чтение/запись 3-х файлов), следовательно файловая система будет работать медленнее, чем при однопоточной. И, во вторых: [GIL](#) еще ни кто не отменял.


```
import pathlib, threading, time, queue

def worker(que):
    while True:
        # Получаем задание (имя файла) из очереди
        job = que.get()
        # Путь к новому (обработанному) файлу
        file_write = path_dir_modified.joinpath(job.name)
        # открываем файл из очереди на чтение и
```

Вверх

```
# новый файл на запись
with open(job, 'r') as fr, open(file_write, 'w') as fw:
    # дописываем имя файла
    fw.write(f'\n\n=====> {file_write}\n\n')
```

РЕКЛАМА



построчно

заменяем букву y на 0
replace('y', '0')
ненные данные
е)
то задача выполнена

```
ра
#
тс
#
ра
#
l:
#
test_dir_modified = 'test_dir_modified'
path_dir_modified = path.joinpath(test_dir_modified)
path_dir_modified.mkdir(exist_ok=True)

# создаем и заполняем очередь именами файлов
que = queue.Queue()
for file in list_files:
    que.put(file)

if que.qsize():
    # Создаем и запускаем потоки
    n_thread = 3
    for _ in range(n_thread):
        th = threading.Thread(target=worker, args=(que,), daemon=True)
        th.start()

    # Блокируем дальнейшее выполнение
    # программы до тех пор пока потоки
    # не обслужат все элементы очереди
    que.join()
else:
    print('Файлы не найдены.')
```

Содержание раздела:

- [КРАТКИЙ ОБЗОР МАТЕРИАЛА.](#)
- [Получение общих сведений о потоках, модуль threading](#)
- [Класс Thread\(\) модуля threading](#)
- [Класс local\(\) модуля threading](#)
- [Класс Event\(\) модуля threading](#)
- [Класс Lock\(\) модуля threading](#)
- [Класс RLock\(\) модуля threading](#)
- [Класс Condition\(\) модуля threading](#)
- [Класс Semaphore\(\) модуля threading](#)
- [Класс Timer\(\) модуля threading](#)
- [Класс Barrier\(\) модуля threading](#)
- [Протокол управления контекстом в модуле threading](#)
- [Трассировка и профилирование потоков модулем threading](#)
- [Как перезапускать потоки?](#)