Глава 1. Concurrency, parallelism, multitasking

Существует две модели исполнения вычислений: sequential, concurrency и parallelism. В первой всё стандартно: строка за строкой, команда за командой. В concurrency код **запущен** одновременно, но выполняется по-очереди, происходит т.н. *context switching* (здесь я имею в виду любое переключение, не только между потоками или процессами). В parallelism модели код действительно может быть запущен одновременно (конкурренто) и по-настоящему параллельно, т.е. он ещё и **исполняется** одновременно. Также существует *multitasking (мультизадачность)*, она бывает *вытесняющей (preemptive)* и *невытесняющей (cooperative)*. В первом случае есть чёткий временной промежуток выполнения, после чего ОС отдаст время выполнения другой задаче, тогда как в cooperative исполняемый код самостоятельно принимает решение о передаче управления остальным в очереди. Преимущества cooperative в том, что не требуется context switching, и иногда полезнее знать как эффективнее переключать задачи.

Есть *blocking* и *non-blocking* операции, в первом случае интерпретатор остановится и будет ждать завершения, во втором продолжит выполнение кода дальше. В python есть GIL, идея в том, что он запрещает **одновременное** выполнение более одной инструкции **python кода** для конкретного запущенного интерпретатора. Так не происходит *race condition* (связанное с *reference counting* в менеджере памяти) в случае создания нескольких потоков, и поэтому нет издержек на дополнительные проверки. GIL отпускается на I/O задачах, используя средства ОС (например, epoll, kqueue, IOCP), а ещё он может отпускаться там, где исполняется не python код, например, у расширения на C.

В сердце asyncio лежит *event loop*, в сущности просто очередь событий, которую мы в бесконечном цикле обрабатываем. Загружаем события, затем на каждом шаге выполняем следующую задачу из очереди (снятую с паузы), пока не дойдём до блокирующей операции (ставим на паузу и делаем запрос к ОС: “скажи, когда придёт результат”), здесь же проверяем нет ли готовых задач после завершения блокирующих операций, снимаем их с паузы и повторяем цикл.

Глава 2. asyncio basics

event loop asyncio работает в одном потоке и использует *coroutines* (сопрограммы). coroutine – просто особенная функция (PEP 342 – Coroutines via Enhanced Generators, PEP 492 – Coroutines with async and await syntax), которая может останавливать своё выполнение при достижении *long-running* операции и продолжать после её завершения, в это время остальные сопрограммы могут продолжать своё выполнение в порядке очереди.

Для задания coroutine используется слово ***async***, при вызове такая функция будет возвращать объект типа coroutine, а не значение. Для выполнения используется цикл событий из asyncio. Получение результата (ожидание), а так же указание к исполнению задаётся через ***await***. Для одновременного запуска используются *task* – задачи, особый тип, который управляется asyncio и строится на *futures*. Task позволяет нам поместить корутины в цикл событий и быть запущенными (выставленными к исполнению) одновременно. Выполнение задач происходит по достижении первой инструкции await, но не ранее. Задачам можно выставлять *timeout (wait_for)* на выполнение, и их можно отменять *(.cancel(), CancelledError)*. *Future* – объект, который будет содержать какое-то значение в будущем, но не содержит его сейчас; у него можно проверить готовность (*.done()*), ему можно установить результат (*.set_result()*), получить результат (*.result()*). Если вызвать результата раньше, чем там на самом деле будет значение, то выпадет исключение. Coroutine и Future наследуются от Awaitable, а Task от Future (присутствует магический метод __await__).

```
1. import asyncio
2. from util import async_timed
3.
4. @async_timed()
5. async def delay(delay_seconds: int) -> int:
6.     print(f"sleeping for {delay_seconds} second(s)")
7.     await asyncio.sleep(delay_seconds)
8.     print(f"finished sleeping for {delay_seconds} second(s)")
9.     return delay_seconds
10.
11.@async_timed()
12.async def main():
```

```
13.     task_one = asyncio.create_task(delay(2))
14.     task_two = asyncio.create_task(delay(3))
15.
16.     await task_one
17.     await task_two
18.
19.
20.asyncio.run(main())
```

Глава 3. asyncio echo chat

Метод *.accept()* по умолчанию является блокирующим, и мы можем установить *.setblocking(False)*, но тогда он будет выбрасывать исключение, если нового соединения ещё не было получено (если на момент вызова accept() нет входящих соединений, то метод вернет исключение socket.error с кодом EWOULDBLOCK или EAGAIN, указывая на то, что операция блокирующей функции была вызвана в неблокирующем режиме и еще нет доступных данных). Тогда мы можем обернуть такое поведение в бесконечный цикл и блок *try…except*, однако так мы загрузим CPU на 100%. Чтобы избежать обработки исключений каждую секунду, в ОС существует система оповещений, та самая epoll, kqueue и т.д. Мы регистрируем сокеты на какие-то события. например, “чтение”, и можем получать обновления сразу как только они приходят не нагружая при этом процессор (т.к. реализована такая система на уровне железа и ОС). Затем каждое входящее соединение (сокет) мы делаем неблокирующим, регистрируем на событие чтения. Под капотом asyncio собственно и используется модуль *selectors* для работы с этой системой оповещений. Интересно, что у asyncio *AbstractEventLoop* есть свои методы, аналогичные методам socket, для работы с соединением через socket. Но к ним мы можем применять await, т.к. они возвращают coroutine. Для обработки сигналов, например, SIGINT (из модуля signal), мы можем добавлять обработчики прямо в event loop с помощью *.add_signal_handler()*. Таким образом можно обеспечить корректное завершение работы программы, например, запустив работу на исполнение после завершения главного цикла.

```
1. import socket
2. import signal
3. import logging
4. import asyncio
5. from asyncio import AbstractEventLoop
6.
7. async def echo(connection: socket.socket,
8.                loop: AbstractEventLoop) -> None:
9.     try:
10.         while data := await loop.sock_recv(connection, 1024):
```

```
            if data == b"boom\r\n":
                raise Exception("Unexpected network error")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()

echo_tasks = []


async def listen_for_connections(server_socket: socket.socket,
                                 loop: AbstractEventLoop) -> None:
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        echo_task = asyncio.create_task(echo(connection, loop))
        echo_tasks.append(echo_task)


class GracefulExit(SystemExit):
    pass

def shutdown():
    raise GracefulExit()

async def close_echo_tasks(echo_tasks: list[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_tasks]
    for task in waiters:
        try:
            await task
        except asyncio.TimeoutError:
            pass

async def main():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_socket.bind(("127.0.0.1", 8000))
    server_socket.setblocking(False)
```

```
52.     server_socket.listen()
53.
54.     for signame in {"SIGINT", "SIGTERM"}:
55.         loop.add_signal_handler(getattr(signal, signame), shutdown)
56.
57.     await listen_for_connections(server_socket,
    asyncio.get_event_loop())
58.
59. loop = asyncio.new_event_loop()
60.
61. try:
62.     loop.run_until_complete(main())
63. except GracefulExit:
64.     loop.run_until_complete(close_echo_tasks(echo_tasks))
65. finally:
66.     loop.close()
```

Глава 4. Concurrent web requests

aiohttp даёт такую функциональность, т.к. requests является блокирующей. Существуют и асинхронные менеджеры контекста *async with* (которые задаются в классе методами *__aenter__* и *__aexit__*). В aiohttp есть объект сессии *ClientSession*, и *ClientTimeout*. Первый задаёт параметры сессии, например, cookie, а второй задаёт timeout, например на подключение. Если мы захотим запустить несколько Task одновременно, то для удобства есть *asyncio.gather*, который принимает список Awaitables и ждёт завершения всех (gather не отменит автоматически выполнение оставшихся корутин даже при возникновении исключения). Мы можем отследить корутины, которые выбросили исключение задав параметр *return_exceptions=True*, и тогда вместо падения он вернёт список корутин, в которых возникло исключение вместе с теми, которые завершились корректно. Чтобы не ждать завершения всех корутин, мы можем обрабатывать их сразу при завершении одна за одной итерируясь с помощью *asyncio.as_completed*. Но мы не будем знать завершилась Future уже или нет, а для этого есть *asyncio.wait*, который возвращает *done* и *pending* множества, чтобы мы могли легко понять завершилась корутина или нет. Есть параметры, например, asyncio.ALL_COMPLETED, asyncio.FIRST_COMPLETED и asyncio.FIRST_EXCEPTION, позволяющие управлять ожиданием выполнения: ждать или не ждать завершения остальных корутин; начать обрабатывать завершённые.

```
1.  import asyncio
2.  import aiohttp
3.  from util import async_timed
4.  from chapter_04 import fetch_status
5.
6.  @async_timed()
7.  async def main():
8.      async with aiohttp.ClientSession() as session:
9.          pending = \
10.             [asyncio.create_task(
11.                 fetch_status(session, "https://example.com")),
12.              asyncio.create_task(
13.                 fetch_status(session, "https://example.com")),
14.              asyncio.create_task(
15.                 fetch_status(session, "https://example.com"))]
```

```
16.
17.         while pending:
18.             done, pending = await asyncio.wait(pending,
19.
    return_when=asyncio.FIRST_COMPLETED)
20.             print(f"Done task count: {len(done)}")
21.             print(f"Pending task count: {len(pending)}")
22.
23.             for done_task in done:
24.                 print(await done_task)
25.
26. asyncio.run(main())
```

```
1.  import asyncio
2.  import aiohttp
3.  from aiohttp import ClientSession
4.  from util import async_timed
5.
6.  @async_timed()
7.  async def fetch_status(session: ClientSession,
8.                         url: str,
9.                         delay: int = 0) -> int:
10.     await asyncio.sleep(delay)
11.     async with session.get(url) as result:
12.         return result.status
13.
14. @async_timed()
15. async def main():
16.     async with aiohttp.ClientSession() as session:
17.         fetchers = [fetch_status(session, "https://example.com", 1),
18.                     fetch_status(session, "https://example.com", 1),
19.                     fetch_status(session, "https://example.com", 10)]
20.
21.         for finished_task in asyncio.as_completed(fetchers, timeout=2):
22.             try:
23.                 status_code = await finished_task
24.                 print(status_code)
25.             except asyncio.TimeoutError:
26.                 print("We got a timeout error!")
27.
28.             for task in asyncio.all_tasks():
29.                 print(task)
30.
31.
32. asyncio.run(main())
```

Глава 5. Non-blocking database driver

Для работы с Postgres в PyPI существует библиотека *asyncpg*, которая позволяет работать асинхронно с подключениями к БД. Сначала мы создаём *connection* или *pool*: в первом случае будем всегда иметь одно подключение и не сможем параллельно отправлять на него несколько запросов, во втором случае можем иметь набор подключений, которые отдаются по требованию. Методы connection это просто корутины, например *.fetch()*, или *.execute()*. fetch, например, вернёт объект/ы типа *Record*. После завершения работы с БД соединение надо закрыть методом *.close()*. Есть метод *.executemany()*, который работает с помощью переменных SQL:

```
brands = generate_brand_names(common_words)
insert_brands = "INSERT INTO brand VALUES(DEFAULT, $1)"
return await connection.executemany(insert_brands, brands)
```

pool соединений можно создать через *asyncpg.create_pool()*, и работать как с контекстным менеджером; к запросу можно будет применить, например, *asyncio.gather(query_1(pool), query_2(pool))*. Конечно, имеются и транзакции, *connection.transaction()*, с которыми можно работать и как с контекстными менеджерами в том числе. Для ручного управления есть *Transaction* класс и соответствующие методы *.start()*, *.rollback()*, *.commit()*. Есть **вложенные транзакции**: если внутри произошло исключение, то будет отменено всё, что касалось этой транзакции, а всё, что было успешно выполнено у транзакции выше (по вложенности) останется (это механизм SAVEPOINTS в Postgres).

Для итерации по запросу (для ленивого вычисления, сокращения издержек и выгрузки данных в память частями) есть **асинхронные генераторы** и специальный *async for* синтаксис. Вместо значения он будет генерировать coroutine, которую можно будет потом ожидать (await). Работать оно будет с помощью *streaming cursor*. **cursor** – в asyncpg это и асинхронный генератор и awaitable. Мы можем перемещаться курсором по записям, например, с помощью *.forward()* вперёд. Если захотим получить определённое число элементов и завершить работу генератора, то нам придётся определить свой асинхронный генератор.

```
1. import asyncio
```

```
import asyncpg
from asyncpg.transaction import Transaction

import logging

async def main():
    connection = await asyncpg.connect(host="localhost",
                                       port=5432,
                                       user="postgres",
                                       database="products",
                                       password="password")
    transaction: Transaction = connection.transaction()
    await transaction.start()
    try:
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_1')")
        await connection.execute("INSERT INTO brand "
                                 "VALUES(DEFAULT, 'brand_2')")
    except asyncpg.PostgresError:
        print('Errors, rolling back transaction!')
        await transaction.rollback()
    else:
        print('No errors, committing transaction!')
        await transaction.commit()

    query = """SELECT brand_name FROM brand
               WHERE brand_name LIKE 'brand%'"""
    brands = await connection.fetch(query)
    print(brands)

    await connection.close()

asyncio.run(main())
```

Асинхронный генератор:

```
import asyncio
from util import delay, async_timed

async def positive_integers_async(until: int):
    for integer in range(1, until):
        await delay(integer)
        yield integer

@async_timed()
async def main():
    async_generator = positive_integers_async(3)
```

```
    print(type(async_generator))
    async for number in async_generator:
        print(f"Got number {number}")

asyncio.run(main())
```

```
import asyncio
import asyncpg

async def main():
    connection = await asyncpg.connect(host="localhost",
                                       port=5432,
                                       user="postgres",
                                       database="products",
                                       password="password")
    async with connection.transaction():
        query = "SELECT product_id, product_name FROM product"
        cursor = await connection.cursor(query)
        await cursor.forward(500)
        products = await cursor.fetch(100)
        for product in products:
            print(product)

    await connection.close()

asyncio.run(main())
```

Глава 6. Handling CPU-bound work

Для CPU-bound задач существует модуль *multiprocessing* в python. В общем и целом, он вызывает *fork*, но предоставляет удобное API для работы в родительском процессе. Есть модуль *concurrent.futures*, который позволяет создавать pool процессов, которыми можно пользоваться вместе с asyncio. Например, создать event loop, создать pool в контекстном менеджере и отправлять задачи из asyncio на исполнение в *ProcessPoolExecutor*: *loop.run_in_executor()*. Для предоставления общего доступа к памяти используются примитивы *Value* и *Array* из модуля multiprocessing, которые имеют методы *.acquire()* и *.release()*, что позволяет избежать *race condition* – ситуации, когда результат выполнения набора операций зависит от того какая исполнится первой, т.е. первый процесс (или поток в общем случае) попытался сделать считать значение, а второй в это время увеличил его, тогда у первого будет старое значение в памяти (https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe) . Value и Array – *mutex (mutual exclusion)* и контекстные менеджерами в придачу. Ещё важно отметить, что для передачи, что даже в рамках получения результата от дочернего процесса к родительскому передаваемые данные должны быть *pickled (serializable)*, каким, например, не является *asyncpg.Record*, но является dict. Т.к. часто нельзя передать функцию вместе с параметрами напрямую, в python есть модуль functools, который позволяет заменить вызов: func(100) => f = functools.partial(func, 100) => f(); Объект модуля concurrent.future *Future*, **отличается** от привычного в рамках asyncio.

```
1. import asyncio
2. import functools
3. from multiprocessing import Value
4. from concurrent.futures import ProcessPoolExecutor
5. from chapter_06.listing_6_8 import partition, merged_dictionaries
6.
7. map_progress: Value
8.
9.
10. def init(progress: Value):
11.     global map_progress
12.     map_progress = progress
```

```
13.
14.
15. def map_frequencies(chunk: list[str]) -> dict[str, int]:
16.     counter = {}
17.     for line in chunk:
18.         word, _, count, _ = line.split("\t")
19.         if counter.get(word):
20.             counter[word] += int(count)
21.         else:
22.             counter[word] = int(count)
23.
24.     with map_progress.get_lock():
25.         map_progress.value += 1
26.
27.     return counter
28.
29.
30. async def progress_reporter(total_partitions: int):
31.     while map_progress.value < total_partitions:
32.         print(f"Finished {map_progress.value} / {total_partitions} map
    operations")
33.         await asyncio.sleep(1)
34.
35.
36. async def main(partition_size: int):
37.     global map_progress
38.
39.     with open("chapter_06/googlebooks-eng-all-1gram-20120701-a",
    encoding="utf-8") as f:
40.         contents = f.readlines()
41.         loop = asyncio.get_running_loop()
42.         tasks = []
43.         map_progress = Value("i", 0)
44.
45.         with ProcessPoolExecutor(initializer=init, initargs=(map_progress,))
    as pool:
46.             total_partitions = len(contents) // partition_size
47.             reporter =
    asyncio.create_task(progress_reporter(total_partitions))
48.
49.             for chunk in partition(contents, partition_size):
50.                 tasks.append(
51.                     loop.run_in_executor(
52.                         pool, functools.partial(map_frequencies, chunk)
53.                     )
```

```
                )

            counters = await asyncio.gather(*tasks)

            await reporter

            final_result = functools.reduce(merged_dictionaries, counters)

            print(f"Aardvark has appeared {final_result['Aardvark']} times.")


if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))
```

```
import asyncio
import asyncpg
from util import async_timed
from concurrent.futures import ProcessPoolExecutor


product_query = """
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_color_id JOIN
product_size as ps on ps.product_size_id = s.product_size_id WHERE
p.product_id = 100"""


async def query_product(pool: ProcessPoolExecutor):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)

@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)


```

```
30. def run_in_new_loop(num_queries: int) -> list[dict]:
31.     async def run_queries():
32.         async with asyncpg.create_pool(
33.             host="localhost",
34.             port=5432,
35.             user="postgres",
36.             password="password",
37.             database="products",
38.             min_size=6,
39.             max_size=6,
40.         ) as pool:
41.             return await query_products_concurrently(pool, num_queries)
42. 
43.     # One thing to note here is that we convert our results into dictionaries
44.     # because asyncpg record objects cannot be pickled.
45.     # Converting to a data structure that is serializable ensures that
46.     # we can send our result back to our parent process.
47.     results = [dict(result) for result in asyncio.run(run_queries())]
48.     return results
49. 
50. 
51. @async_timed()
52. async def main():
53.     loop = asyncio.get_running_loop()
54.     pool = ProcessPoolExecutor()
55.     tasks = [loop.run_in_executor(pool, run_in_new_loop, 10000) for _ in
    range(4)]
56.     all_results = await asyncio.gather(*tasks)
57. 
58.     total_queries = sum([len(result) for result in all_results])
59.     print(f"Retrieved {total_queries} products from product database.")
60. 
61. 
62. if __name__ == "__main__":
63.     asyncio.run(main())
```

Глава 7. Handling blocking works with threads

Вообще говоря GIL отпускается всякий раз, когда идёт речь не о работе с объектами python или байт-кодом python. Это значит, что мы можем так же эффективно использовать потоки, как и процессы, а точнее целый пул потоков. Для этого есть *ThreadPoolExecutor* из concurrent.futures. В asyncio, когда мы вызываем run_in_executor, то по-умолчанию будет и так стоять ThreadPoolExecutor, вместо ProcessPoolExecutor. Более того, чтобы ещё сэкономить запись, можно воспользоваться методом *.to_thread()*, вместо объявления пула и вызова run_in_executor. С использованием потоков появляется и проблема *race condition* и *deadlocks*. Для предотвращения условий, когда переменная должна быть обработана атомарно есть *Lock*, из модуля threading (именно для потоков!), который позволяет захватить мьютекс и затем его отпустить. Проблема в том, что дважды захватить Lock, который **уже захвачен** (acquired) **нельзя** из одного и того же потока, для этого есть *RLock* – reentrant lock, который устанавливает счётчик захватов для одного потока, и уменьшает каждый раз, когда отпускаем, что позволяет писать, например так:

```
1. def sum_list(int_list: list[int]) -> int:
2.     print("Waiting to acquire lock...")
3.     with list_lock:
4.         print("Acquired lock.")
5.         if len(int_list) == 0:
6.             print("Finished summing.")
7.             return 0
8.         else:
9.             head, *tail = int_list
10.            print("Summing rest of list.")
11.            return head + sum_list(tail)
```

где list_lock есть RLock, потому что при использовании Lock мы бы пытались сделать acquire снова на уже acquired lock и зависли бы. Вторая проблема *deadlocks* – ситуация, при которой два и более потока пытаются захватить ресурсы для двух критических секций. Т.е. I захватил A, II захватил B, I требует ещё и B, II требует ещё и A. В итоге каждый держит свой кусок и ждёт от второго ещё один и программа зависает. Для борьбы с этим недугом можно перестроить порядок захвата ресурсов (например, пусть оба сначала требуют B,

потом оба A), можно переписать код так, чтобы использовать только один Lock и т.д.

Для потоко безопасных приложений (например, как Tkinter), чтобы внедрить асинхронность или просто другой поток на выполнение можно создавать event loop в отдельном потоке, тогда как главный поток будет отдан для управления event loop Tkinter. Важно, что данные между потоками тоже должны передаваться аккуратно, во избежании состояний гонки, deadlock и т.д. Например, можно использовать Queue, для передачи данных между потоками, тогда один поток туда кладёт, второй читает и никто не пишет напрямую в чужой поток.

listing 7.13:

```
1.  import asyncio
2.  from concurrent.futures import Future
3.  from asyncio import AbstractEventLoop
4.  from typing import Callable, Optional
5.  from aiohttp import ClientSession
6.
7.
8.
9.  class StressTest:
10.     def __init__(
11.         self,
12.         loop: AbstractEventLoop,
13.         url: str,
14.         total_requests: int,
15.         callback: Callable[[int, int], None],
16.     ):
17.         self._completed_requests: int = 0
18.         self._load_test_future: Optional[Future] = None
19.         self._loop = loop
20.         self._url = url
21.         self._total_requests = total_requests
22.         self._callback = callback
23.         self._refresh_rate = total_requests // 100 + 1
24.
25.     def start(self):
26.         future = asyncio.run_coroutine_threadsafe(self._make_requests(),
    self._loop)
27.         self._load_test_future = future
28.
```

```
    def cancel(self):
        if self._load_test_future:
            self._loop.call_soon_threadsafe(self._load_test_future.cancel)  # B

    async def _get_url(self, session: ClientSession, url: str):
        try:
            # response = requests.get(url)
            response = await session.get(url)
            # print(response.status)
        except Exception as e:
            print(e)
        self._completed_requests = self._completed_requests + 1  # C
        if (
            self._completed_requests % self._refresh_rate == 0
            or self._completed_requests == self._total_requests
        ):
            self._callback(self._completed_requests, self._total_requests)

    async def _make_requests(self):
        async with ClientSession() as session:
            reqs = [
                self._get_url(session, self._url) for _ in range(self._total_requests)
            ]
            await asyncio.gather(*reqs)
```

listing 7.14:

```
from queue import Queue
from tkinter import Tk
from tkinter import Label
from tkinter import Entry
from tkinter import ttk
from typing import Optional
from chapter_07.listing_7_13 import StressTest


class LoadTester(Tk):
    def __init__(self, loop, *args, **kwargs):  # A
        Tk.__init__(self, *args, **kwargs)
        self._queue = Queue()
        self._refresh_ms = 25

        self._loop = loop
        self._load_test: Optional[StressTest] = None
        self.title("URL Requester")
```

```
19.
20.         self._url_label = Label(self, text="URL:")
21.         self._url_label.grid(column=0, row=0)
22.
23.         self._url_field = Entry(self, width=10)
24.         self._url_field.grid(column=1, row=0)
25.         self._url_field.insert(0, "https://example.com") # DEFAULT
26.
27.         self._request_label = Label(self, text="Number of requests:")
28.         self._request_label.grid(column=0, row=1)
29.
30.         self._request_field = Entry(self, width=10)
31.         self._request_field.grid(column=1, row=1)
32.         self._request_field.insert(0, "5") # DEFAULT
33.
34.         self._submit = ttk.Button(self, text="Submit", command=self._start)
    # B
35.         self._submit.grid(column=2, row=1)
36.
37.         self._pb_label = Label(self, text="Progress:")
38.         self._pb_label.grid(column=0, row=3)
39.
40.         self._pb = ttk.Progressbar(
41.             self, orient="horizontal", length=150, mode="determinate"
42.         )
43.         self._pb.grid(column=1, row=3, columnspan=2)
44.
45.     def _update_bar(self, pct: int):  # C
46.         if pct == 100:
47.             self._load_test = None
48.             self._pb["value"] = pct
49.             self._submit["text"] = "Submit"
50.         else:
51.             self._pb["value"] = pct
52.             self.after(self._refresh_ms, self._poll_queue)
53.
54.     def _queue_update(self, completed_requests: int, total_requests: int):  #
    D
55.         self._queue.put(int(completed_requests / total_requests * 100))
56.
57.     def _poll_queue(self):  # E
58.         if not self._queue.empty():
59.             percent_complete = self._queue.get()
60.             self._update_bar(percent_complete)
61.         else:
```

```
            if self._load_test:
                self.after(self._refresh_ms, self._poll_queue)

    def _start(self):  # F
        if self._load_test is None:
            self._submit["text"] = "Cancel"
            test = StressTest(
                self._loop,
                self._url_field.get(),
                int(self._request_field.get()),
                self._queue_update,
            )
            self.after(self._refresh_ms, self._poll_queue)
            test.start()
            self._load_test = test
        else:
            self._load_test.cancel()
            self._load_test = None
            self._submit["text"] = "Submit"
```

listing 7.15:

```
import asyncio
from asyncio import AbstractEventLoop
from threading import Thread
from chapter_07.listing_7_14 import LoadTester


class ThreadedEventLoop(Thread):  # A
    def __init__(self, loop: AbstractEventLoop):
        super().__init__()
        self._loop = loop
        self.daemon = True

    def run(self):
        self._loop.run_forever()


loop = asyncio.new_event_loop()

asyncio_thread = ThreadedEventLoop(loop)
asyncio_thread.start()  # B

app = LoadTester(loop)  # C
app.mainloop()
```

Глава 8. Streams

Streams – высокоуровневый (async await ready) набор классов и функций для работы с сетевыми соединениями, они позволяют пересылать данные без использования *callback* или низкоуровневых *Protocols* and *Transports*. Или же просто: упрощение работы с сетью. Абстрагирует работу с сокетами, SSL и всякое другое. Как уже говорил, Protocol и Transport – нужны для низкоуровневой работы, transport определяет **как** данные (байты) будут передаваться, а protocol определяет **какие** данные (байты) передавать. *Protocol* вызывает методы *transport* для **отправки** данных, а *transport* вызывает методы *protocol* для **передачи** ему полученных данных. Например, есть BaseTransport, WriteTransport, ReadTransport и другие. По сути это программные интерфейсы. Протоколы же есть, например, BaseProtocol, DatagramProtocol и другие. Машина состояние протокола: start => CM [=> DR*] [=> ER?] => CL => end, где * CM: connection_made(), DR: data_received(), ER: eof_received(), CL: connection_lost(). Всё это абстрагируется для нас в *Streams*. Есть *StreamReader*, и *StreamWriter*. Соответственно, первый отвечает за приём, второй за отправку. В случае с *write*, нужно учесть, что данные могут попасть в буфер и не будут отправлены немедленно, а в каких-то ситуациях буфер разрастётся так, что памяти на компьютере не останется, поэтому существует метод *drain*, который убедиться, что все данные отправлены на сокет (await).

Данные из терминала можно считывать асинхронно, например, используя *StreamReaderProtocol*, который с помощью *asyncio.connect_read_pipe()*, установит переданному *pipe (у которого есть методы read и write)* в соответствие *protocol factory* – функцию, которая создаёт инстанс *protocol*. Например, pipe будет sys.stdin, а *StreamReaderProtocol* примет наш StreamReader, делегирует ему управление и мы сможем считывать данные с помощью *StreamReader*.

```
async def create_stdin_reader() -> StreamReader:
      stream_reader = asyncio.StreamReader()
      protocol = asyncio.StreamReaderProtocol(stream_reader)
      loop = asyncio.get_running_loop()
      await loop.connect_read_pipe(lambda: protocol, sys.stdin)
```

```
return stream_reader
```

У терминала есть разные режимы работы, в том числе *сырой (raw)* и *готовый (cooked)*. Чтобы получить полное управление, можно поставить режим *tty.setcbreak(0)*, который оставит только CTRL+C обработку, а остальное возложит за нас. С помощью ASCII кодов мы сможем перемещаться курсором по терминалу и таким образом выводит информацию в нужном нам порядке.

WARNING: Важно отметить, что EOF в методах у StreamReader считает \n окончанием, и если StreamReader не получит \n, то может никогда не завершить обработку.

Server:

```
1. import asyncio
2. import logging
3. from asyncio import StreamReader, StreamWriter
4.
5.
6. class ChatServer:
7.     def __init__(self):
8.         self._username_to_writer = {}
9.
10.     async def start_chat_server(self, host: str, port: int):
11.         server = await asyncio.start_server(self.client_connected, host,
    port)
12.
13.         async with server:
14.             await server.serve_forever()
15.
16.     async def client_connected(self, reader: StreamReader, writer:
    StreamWriter):
17.         command = await reader.readline()
18.         print(f"CONNECTED {reader} {writer}")
19.         command, args = command.split()
20.         if command == b"CONNECT":
21.             username = args.replace(b"\n", b"").decode()
22.             self._add_user(username, reader, writer)
23.             await self._on_connect(username, writer)
24.         else:
25.             logging.error("Got invalid command from client, disconnecting.")
26.             writer.close()
27.             await writer.wait_closed()
28.
```

```
    def _add_user(self, username: str, reader: StreamReader, writer:
StreamWriter):
        self._username_to_writer[username] = writer
        asyncio.create_task(self._listen_for_messages(username, reader))

    async def _on_connect(self, username: str, writer: StreamWriter):
        writer.write(
            f"Welcome! {len(self._username_to_writer)} user(s) are
online!\n".encode()
        )
        await writer.drain()
        await self._notify_all(f"{username} connected!\n")

    async def _remove_user(self, username: str):
        writer = self._username_to_writer[username]
        del self._username_to_writer[username]
        try:
            writer.close()
            await writer.wait_closed()
        except Exception as e:
            logging.exception("Error closing client writer, ignoring.",
exc_info=e)

    async def _listen_for_messages(self, username: str, reader:
StreamReader):
        try:
            while (data := await asyncio.wait_for(reader.readline(), 60)) !=
b"":
                await self._notify_all(f"{username}: {data.decode()}")
            await self._notify_all(f"{username} has left the chat\n")
        except Exception as e:
            logging.exception("Error reading from client.", exc_info=e)
            await self._remove_user(username)

    async def _notify_all(self, message: str):
        inactive_users = []
        for username, writer in self._username_to_writer.items():
            try:
                writer.write(message.encode())
                await writer.drain()
            except ConnectionError as e:
                logging.exception("Could not write to client.", exc_info=e)
                inactive_users.append(username)

        [await self._remove_user(username) for username in inactive_users]
```

```
69.
70.
71. async def main():
72.     chat_server = ChatServer()
73.     await chat_server.start_chat_server("localhost", 8000)
74.
75.
76. asyncio.run(main())
```

Client:

```
1.  import asyncio
2.  import os
3.  import logging
4.  import tty
5.  from asyncio import StreamReader, StreamWriter
6.  from collections import deque
7.  from chapter_08.listing_8_5 import create_stdin_reader
8.  from chapter_08.listing_8_7 import *
9.  from chapter_08.listing_8_8 import read_line
10. from chapter_08.listing_8_9 import MessageStore
11.
12.
13. async def send_message(message: str, writer: StreamWriter):
14.     writer.write((message + "\n").encode()) # \n is critical!!!
15.     await writer.drain()
16.
17.
18. async def listen_for_messages(reader: StreamReader, message_store:
    MessageStore):
19.     while (message := await reader.readline()) != b"":
20.         await message_store.append(message.decode())
21.     await message_store.append("Server closed connection.")
22.
23.
24. async def read_and_send(stdin_reader: StreamReader, writer: StreamWriter):
25.     while True:
26.         message = await read_line(stdin_reader)
27.         await send_message(message, writer)
28.
29.
30. async def main():
31.     async def redraw_output(items: deque):
32.         save_cursor_position()
33.         move_to_top_of_screen()
34.         for item in items:
35.             delete_line()
```

```
            sys.stdout.write(item)
        restore_cursor_position()

    tty.setcbreak(0)
    os.system("clear")
    rows = move_to_bottom_of_screen()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()
    sys.stdout.write("Enter username: ")
    username = await read_line(stdin_reader)

    reader, writer = await asyncio.open_connection("localhost", 8000)

    writer.write(f"CONNECT {username}\n".encode())
    await writer.drain()

    message_listener = asyncio.create_task(listen_for_messages(reader,
messages))
    input_listener = asyncio.create_task(read_and_send(stdin_reader, writer))

    try:
        await asyncio.wait(
            [message_listener, input_listener],
return_when=asyncio.FIRST_COMPLETED
        )
    except Exception as e:
        logging.exception(e)
        writer.close()
        await writer.wait_closed()


asyncio.run(main())
```

Глава 9. Web applications

REST – representational state transfer. Парадигма, которая подразумевает *stateless* путь проектирование приложения, который не зависел бы от клиента. Ключевой идея – *resource*, что-то, что может быть существительным, может быть многочисленным и единственным, как продукт и продукты, реализуются с помощью *endpoints* (или ручек). В aiohttp есть средства создать сервер, который отвечал бы этим требованиям. Это модуль aiohttp.*web*. Работает это всё быстрее или не хуже, чем решения на Flask или Django, при равных ресурсах. T.e. aiohttp будет использовать классный асинхронный подход, тогда как чтобы добиться такой же производительности в Flask или Django требуется создавать несколько инстансов приложения (workers у gunicorn или uvicorn). Есть классная утилита замерять производительность – *wrk*. WSGI (Web Server Gateway Interface) – стандарт общения веб-сервера на python или же фреймворка, он даже прописан в PEP. Но для асинхронной работы придумали ASGI (Asynchronous Server Gateway Interface):

```
async def application(scope, receive, send):
      await send({
            'type': 'http.response.start',
            'status': 200,
            'headers': [[b'content-type', b'text/html']]
      })
      await    send({'type':    'http.response.body',    'body':    b'ASGI
hello!'})
```

Популярная имплементация, например, uvicorn (сделанный поверх uvloop и httptools), его можно использовать вместе с Gunicorn, чтобы ещё повысить производительность и отвечать за перезапуск workers. Есть замечательный небольшой фреймворк *Starlette*.

В какой-то период создали концепцию *websocket* – технология, позволяющая поддерживать постоянное соединение клиента с сервером, без постоянного отправки новых запросов. У сервера есть метод *.send()*, и клиент умеет это обрабатывать. Таким образом, можно сделать, например, чат в браузере (хотя не факт, что это лучшая идея). Для создания WebSocket в Starlette

есть класс *WebSocketEndpoint*, от которого можно отнаследоваться и переопределить методы on_connect, on_receive и on_disconnect.

```
import asyncpg
from asyncpg import Record
from asyncpg.pool import Pool
from starlette.applications import Starlette
from starlette.requests import Request
from starlette.responses import JSONResponse, Response
from starlette.routing import Route


async def create_database_pool():
    pool: Pool = await asyncpg.create_pool(
        host="127.0.0.1",
        port=5432,
        user="postgres",
        password="password",
        database="products",
        min_size=6,
        max_size=6,
    )

    app.state.DB = pool

async def destroy_database_pool():
    pool = app.state.DB
    await pool.close()


async def brands(request: Request) -> Response:
    connection: Pool = request.app.state.DB
    brand_query = "SELECT brand_id, brand_name FROM brand"
    results: list[Record] = await connection.fetch(brand_query)
    result_as_dict: list[dict] = [dict(brand) for brand in results]
    return JSONResponse(result_as_dict)

app = Starlette(routes=[Route("/brands", brands)],
                on_startup=[create_database_pool],
                on_shutdown=[destroy_database_pool])
```

```
import asyncio
from starlette.applications import Starlette
from starlette.endpoints import WebSocketEndpoint
from starlette.routing import WebSocketRoute
```

```
5.
6.
7.  class UserCounter(WebSocketEndpoint):
8.      encoding = "text"
9.      sockets = []
10.
11.     async def on_connect(self, websocket):
12.         await websocket.accept()
13.         UserCounter.sockets.append(websocket)
14.         await self._send_count()
15.
16.     async def on_disconnect(self, websocket, close_code):
17.         UserCounter.sockets.remove(websocket)
18.         await self._send_count()
19.
20.     async def on_receive(self, websocket, data):
21.         pass
22.
23.     async def _send_count(self):
24.         if len(UserCounter.sockets) > 0:
25.             count_str = str(len(UserCounter.sockets))
26.             task_to_socket = {
27.                 asyncio.create_task(websocket.send_text(count_str)):
    websocket
28.                 for websocket in UserCounter.sockets
29.             }
30.             done, pending = await asyncio.wait(task_to_socket)
31.
32.             for task in done:
33.                 if task.exception() is not None:
34.                     if task_to_socket[task] in UserCounter.sockets:
35.                         UserCounter.sockets.remove(task_to_socket[task])
36.
37. app = Starlette(routes=[WebSocketRoute("/counter", UserCounter)])
```

Глава 10. Microservices

Микросервисы как альтернатива **монолитам**. В общем случае монолиты крайне удобная штука, но бывают ситуации, когда микросервисы могут быть удобнее и эффективнее. А вообще, что такое микросервисы в общем случае, какие его характеристики:

1. Слабо связаны, независимо развёртываются
2. Имеют независимые технологии и инструменты, включая модель данных
3. Они общаются по одному из протоколов REST или gRPC
4. Следуют принципу “единственной ответственности”

Backend-for-frontend паттерн – реализует сервис, который делают все необходимые обращения, вместо того, чтобы клиентские приложения самостоятельно делали запросы ко множеству сервисов. В таком случае, все запросы клиентов идут на этот сервис. Это позволяет избежать перегрузки остальных сервисов, соблюдать согласованность данных, позволяет устанавливать проверки, вроде healthcheck, и отвергать запросы сразу, если какой-то сервис не работает, без обращения к нему. А ещё есть паттерн *“предохранитель” (circuit breaker)*. Например, чтобы в случае поломки сервиса не делать к нему запросы много раз, а сразу отправлять ответ с ошибкой и периодически проверять заработал ли сервис. Если заработал, то начинать делать туда запросы, иначе отдавать ошибку. Т.е. “когда у нас появляется определенное количество ошибок за каждый период времени, мы можем использовать это для обхода медленного обслуживания до тех пор, пока проблемы с ним не устранятся, гарантируя, что наш ответ нашим пользователям остается максимально быстрым”.

```
import asyncio
from datetime import datetime, timedelta


class CircuitOpenException(Exception):
    pass


class CircuitBreaker:
    def __init__(
```

```
        self,
        callback,
        timeout: float,
        time_window: float,
        max_failures: int,
        reset_interval: float,
    ):
        self.callback = callback
        self.timeout = timeout
        self.time_window = time_window
        self.max_failures = max_failures
        self.reset_interval = reset_interval
        self.last_request_time = None
        self.last_failure_time = None
        self.current_failures = 0

    async def request(self, *args, **kwargs):
        if self.current_failures >= self.max_failures:
            if datetime.now() > self.last_request_time + timedelta(
                seconds=self.reset_interval
            ):
                self._reset("Circuit is going to closed, resetting!")
                return await self._do_request(*args, **kwargs)
            else:
                print("Circuit is open, failing test!")
                raise CircuitOpenException()
        else:
            if (
                self.last_failure_time
                and datetime.now()
                > self.last_failure_time +
timedelta(seconds=self.time_window)
            ):
                self._reset("Interval since first failure elapsed,
resetting!")
                print("Circuit is closed, requesting!")
                return await self._do_request(*args, **kwargs)

    def _reset(self, msg: str):
        print(msg)
        self.last_failure_time = None
        self.current_failures = 0

    async def _do_request(self, *args, **kwargs):
        try:
```

```
            print("Making request!")
            self.last_request_time = datetime.now()
            return await asyncio.wait_for(
                self.callback(*args, **kwargs), timeout=self.timeout
            )
        except Exception as e:
            self.current_failures += 1
            if self.last_failure_time is None:
                self.last_failure_time = datetime.now()
            raise
```

Глава 11. Synchronization

Даже в однопоточном коде в asyncio может возникнуть ситуация неправильного доступа к ресурсам, когда одна корутина не атомарно отработала, и таким образом, другая корутина получила неверные данные. Для таких случаев существуют примитивы синхронизации: *Lock*, *Semaphore*, *BoundedSemaphore*, *Event*, *Condition*. Первый обычный лок, просто предоставляет доступ к ресурсам по-очереди. Используется обычная конструкция *async with*, Semaphore же это лок, который можно захватывать несколько раз, но есть опасность: его можно отпустить (release) больше раз, чем нужно и тогда дальнейшее количество попыток захвата (acquire) увеличится. Например, Semaphore(2): acquire, acquire, release, release, release. Должна быть ошибка, но на деле теперь ограничение станет у него не 2, а 3. Для таких ситуация есть BoundedSemaphore, он не позволяет отпускать Semaphore больше, чем указано и вызовет исключение. Event (*.set(), .clear(), .wait()*) же это способ отправлять “сигнал” корутинам о том, что что-то выполнилось. Например, пришло сообщение (вызвали set на объект Event): корутины получили сигнал, и wait() получило сигнал “продолжить выполнение дальше”. Если нужно убрать флаг (поставить снова Event в False) можно вызвать .clear(). Есть проблема: **корутина не увидит событие .set(), если выполняется в данный момент**. Есть ещё Condition – фактически Lock и Event в одном флаконе, позволяет предоставлять разграниченный доступ к shared resources, и срабатывать на event. Есть методы *.notify()*, *.notify_all() (многопоточная версия)*, *.wait()*, *.wait_for()*. В отличие от wait, wait_for будет ждать не просто когда позовут notify (или notify_all), но и проверит условие (предикат, должен быть callable), которые мы сами можем задать.

```
1. import asyncio
2. from enum import Enum
3.
4.
5. class ConnectionState(Enum):
6.     WAIT_INIT = 0
7.     INITIALIZING = 1
8.     INITIALIZED = 2
9.
```

```
10.
11. class Connection:
12.     def __init__(self):
13.         self._state = ConnectionState.WAIT_INIT
14.         self._condition = asyncio.Condition()
15.
16.     async def initilize(self):
17.         await self._change_state(ConnectionState.INITIALIZING)
18.         print("initialize: Initializing connection...")
19.         await asyncio.sleep(3)
20.         print("initialize: Finished initializing connection")
21.         await self._change_state(ConnectionState.INITIALIZED)
22.
23.     async def execute(self, query: str):
24.         async with self._condition:
25.             print("execute: Waiting for connection to initialize")
26.             await self._condition.wait_for(self._is_initialized)
27.             print(f"execute: Running {query}!!!")
28.             await asyncio.sleep(3)
29.
30.     async def _change_state(self, state: ConnectionState):
31.         async with self._condition:
32.             print(f"change_state: State changing from {self._state} to
    {state}")
33.             self._state = state
34.             self._condition.notify_all()
35.
36.     def _is_initialized(self):
37.         if self._state is not ConnectionState.INITIALIZED:
38.             print(
39.                 f"_is_initialized: Connection not finished initializing,
    state is {self._state}"
40.             )
41.             return False
42.         print("_is_initialized: Connection is initialized!")
43.         return True
44.
45.
46. async def main():
47.     connection = Connection()
48.
49.     query_one = asyncio.create_task(connection.execute("select * from
    table"))
50.     query_two = asyncio.create_task(connection.execute("select * from
    another_table"))
```

```
51.     asyncio.create_task(connection.initilize())
52.
53.     await query_one
54.     await query_two
55.
56.
57. asyncio.run(main())
```

Глава 12. Asynchronous queues

Когда нужно естественным образом ограничить количество одновременно обрабатываемых событий, можно поставить их в очередь. Например, пользователь загрузил файл и мы хотим его конвертировать в другой, тогда ставим задачу в очередь, или делает покупку, и пока транзакция обрабатывается, пользователь может дальше делать свои дела. Это же позволяет ограничить одновременное количество задач, короче, прямо как в магазине очередь: 3 кассы, десять покупателей. Для этого в asyncio есть *Queue*, самая обычная FIFO очередь. Добавлять элементы *.put()*, *.put_nowait()*, брать – *.get()*, *.get_nowait()*. nowait не блокирующая операция (первые – корутины), но если очередь пустая или наоборот, забита, то оно вызовет исключение. Есть метод *.empty()*, который говорит, есть ли элементы в очереди. Каждый раз, когда вызывается .get(), внутренний счётчик Queue увеличивается, и, чтобы сказать, что работа над элементом очереди завершена нужно вызвать *.task_done()*, на каждый вызов .get(), который уменьшит счётчик, и это позволит вызвать *.join()*, например, в asyncio.gather, что позволит завершить выполнение, а не ждать вечно. Есть *PriorityQueue*, самая настоящая, построенная на *heaps* (heaqp module). *heap* – дерево, у которого значение ребёнка всегда больше значения родителя. Это позволяет держать самый маленький элемент в корне и получать его за O(1). Чтобы задавать приоритет можно использовать *dataclasses* (или переопределить методы __eq__, __lt__ и т.д. у классов), с полем *order=True*:

```
@dataclass(order=True)
class WorkItem:
      priority: int
      order: int # with the same priority will use this field
      data: str = field(compare=False)
```

Если приоритеты совпадут, то пойдёт сравнивать второе поле и т.д. Есть LIFO очередь, собственно, стэк, *LifoQueue*.

Естественно, такие очереди не устойчивы, т.к. полностью находятся в памяти и в случае аварийного завершения все данные пропадут. В таком случае есть решения, которые работают с диском, и предоставляют расширенный набор функциональности, короче, отдельные приложения, например: Celery, RabbitMQ;

```python
import asyncio
from asyncio import Queue, Task
from random import randrange
from enum import IntEnum
from dataclasses import dataclass, field
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response


routes = web.RouteTableDef()


QUEUE_KEY = "order_queue"
TASKS_KEY = "order_tasks"


class UserType(IntEnum):
    POWER_USER = 1
    NORMAL_USER = 2


@dataclass(order=True)
class Order:
    user_type: UserType
    order_delay: int = field(compare=False)


async def process_order_worker(worker_id: int, queue: Queue):
    while True:
        print(f"Worker {worker_id}: Waiting for an order...")
        order: Order = await queue.get()
        print(f"Worker {worker_id}: Processing order {order}")
        await asyncio.sleep(order.order_delay)
        print(f"Worker {worker_id}: Processed order {order}")
        queue.task_done()


@routes.post("/order")
async def place_order(request: Request) -> Response:
    body = await request.json()
    user_type = (
        UserType.POWER_USER if body["power_user"] == "True" else
UserType.NORMAL_USER
```

```
    )
    order_queue = app[QUEUE_KEY]
    await order_queue.put(Order(user_type, randrange(5)))
    return Response(body="Order placed!")


async def create_order_queue(app: Application):
    print("Creating order queue and tasks.")
    queue: Queue = asyncio.Queue(10)
    app[QUEUE_KEY] = queue
    app[TASKS_KEY] = [
        asyncio.create_task(process_order_worker(i, queue)) for i in range(5)
    ]


async def destroy_queue(app: Application):
    order_tasks: list[Task] = app[TASKS_KEY]
    queue: Queue = app[QUEUE_KEY]

    print("Waiting for pending queue workers to finish...")
    try:
        await asyncio.wait_for(queue.join(), timeout=10)
    finally:
        print("Finished all pending time, canceling worker tasks...")
        [task.cancel() for task in order_tasks]


app = web.Application()
app.on_startup.append(create_order_queue)
app.on_shutdown.append(destroy_queue)

app.add_routes(routes)
web.run_app(app)
```

Глава 13. Managing subprocesses

Чтобы запускать процессы вне Python, вроде execv в C, можно использовать модуль asyncio.subprocesses. Процесс будет создаваться с помощью *asyncio.create_subprocess_exec*, есть ещё *asyncio.create_subprocess_shell*, который собственно предоставляет shell, который стоит в системе по умолчанию. Будет возвращён объект типа *Process* из asyncio. Его можно принудительно завершить *.terminate*, и ожидать выполнения *.wait*. Контролировать ввод и вывод можно передавая аргументы в параметры stdout, stdin, например asyncio.subprocess.PIPE. С .wait есть важный момент: он может вступить в состояние deadlock при использовании PIPE: когда буфер StreamReader заполняется, все последующие записи в него блокируются до тех пор, пока там не будет места, а если процесс будет стараться всё ещё записать туда данные, то StreamReader не разблокируется и процесс встанет и программа наша зависнет:

```
import sys
[sys.stdout.buffer.write(b"Hello there!!\n") for _ in range(1000000)]
sys.stdout.flush()
```

```
import asyncio
from asyncio.subprocess import Process

async def main():
      program = ["python3", "chapter_13/listing_13_4.py"]

      process: Process = await asyncio.create_subprocess_exec(
      *program, stdout=asyncio.subprocess.PIPE
      )
      print(f"Process pid is: {process.pid}")

      return_code = await process.wait()
      print(f"Process returned: {return_code}")
asyncio.run(main())
```

Избежать можно, например, используя метод *.communicate*, который вернёт stdout, stderr, и получит все данные нормально, но правда, сохранит в память, что может тоже вызвать проблемы с памятью. Можно запускать много процессов

одновременно создавая create_task и в функции вызывая create_subprocess_exec и тд. Ограничение на создание количества процессов можно регулировать с помощью Semaphore.

```
from random import randrange
import time


user_input = ''

while user_input != "quit":
    user_input = input("Enter text to echo: ")
    for i in range(randrange(10)):
        time.sleep(.5)
        print(user_input)
```

```
import asyncio
from asyncio import StreamWriter, StreamReader, Event
from asyncio.subprocess import Process


async def output_consumer(input_ready_event: Event, stdout: StreamReader):
    while (data := await stdout.read(1024)) != b"":
        print(data)
        if data.decode().endswith("Enter text to echo: "):
            input_ready_event.set()

async def input_writer(text_data, input_ready_event: Event, stdin:
StreamWriter):
    for text in text_data:
        await input_ready_event.wait()
        stdin.write(text.encode())
        await stdin.drain()
        input_ready_event.clear()

async def main():
    program = ["python3", "chapter_13/listing_13_13.py"]
    process: Process = await asyncio.create_subprocess_exec(*program,

stdout=asyncio.subprocess.PIPE,

stdin=asyncio.subprocess.PIPE)
    input_ready_event = asyncio.Event()

    text_input = ["one\n", "two\n", "three\n", "four\n", "quit\n"]
```

```
38.
39.     await asyncio.gather(output_consumer(input_ready_event, process.stdout),
40.                          input_writer(text_input, input_ready_event,
    process.stdin),
41.                          process.wait())
42.
43. asyncio.run(main())
```

Из важного стоит отметить, что есть thread local переменные, а в рамках корутин ContextVar, из модуля contextvars. Первые позволяют иметь уникальные для потока значения, а вторая уникальное значение в рамках корутины. async это синтаксический сахар для @asyncio.corouitne, a await это yield from.