В этом модуле достаточно много примеров, которые надо смотреть, трогать и разбираться, поэтому заранее даем ссылку на Mercury — по мере чтения в материале будут отсылки к коду в нем.

Если какие-то примеры у вас не запускаются с ошибкой вроде такой: *module examples not found,* то напишите в консоли команду, которая поможет Python определиться, откуда надо начинать искать модули.

Обратите внимание, что эту команду нужно вызывать в каждой новой консоли:

```
export PYTHONPATH=.
# теперь пример можно запустить из корня так:
python3 examples/coroutines/event_loop.py
```

Кооперативная многозадачность

Мы уже говорили про потоки и процессы — средства, которые позволяют одновременно с течением программы выполнить какую-то дополнительную работу.

Минус процессов и тредов в том, что это безумно тяжелые объекты. Поэтому нельзя создать много процессов или тредов одновременно — банально не хватит памяти.

Сейчас мы разберемся, как происходит создание нового процесса на уровне операционной системы.

Старт процессов

В Linux и MacOS старт процесса осуществляется с помощью двух системных вызовов. В Windows свои интересные штуки, но все это близко и похоже.

Всегда есть родительский процесс, который запускает другой процесс. Если мы запускаем Python, родительским процессом является bash, то есть наша командная строка.

При старте системы существует один-единственный процесс, который имеет id 1. Он и является родительским процессом для всех остальных.

fork и exec

Задача fork в том, чтобы отсоединиться от текущего процесса. Отсоединение означает, что вся память процесса копируется в новую сущность, которая затем может продолжить существовать.

Вызов exec заменяет текущий контекст выполнения новой сущности другой программой.

Например, в терминале мы вызывали функцию find:

1. произошел вызов fork — вся память процесса терминала скопировалась в новый созданный процесс
2. произошел вызов exec — текущий контекст выполнения (то есть терминал) сменился на контекст выполнения программы find
3. find теперь выполняется в другом дочернем для терминала процессе

Создание нового процесса — это очень долгое и очень дорогое по памяти явление.

Потоки

Потоки как будто решают проблему с памятью и временем создания. Их дешевле создавать, и они находятся в рамках одного процесса. Но это не лучшее решение проблемы: запуск одного потока означает копирование состояния процесса, потому что ему все равно нужно создать какие-то структуры.

Это гораздо быстрее, но нельзя создать, например, 1000 полноценно-работающих потоков, если процессор не состоит из 1000 ядер.

Для работы потока в полную силы ему нужно отдельное ядро процессора. Чаще всего в одном процессе запускают не так много потоков. Например, если работает какой-то веб-сервер, там запускают число потоков, равное числу ядер — либо чуть больше.

Асинхронный подход

Допустим, мы хотим скачать весь сайт. Например, всю «Википедию».

Мы получили список ссылок, который состоит из 1 000 000 документов. Если мы хотим сделать это как можно быстрее, то попробуем создать 1 000 000 потоков и быстро все скачать.

Но так не получится: во-первых, мы упремся в память, потому что потоки много весят в оперативной памяти. Во-вторых, даже если бы все они влезли, процессор не смог бы обеспечить нормальный квант времени каждому из потоков. Кому-то придется ждать, и достаточно долго. Скорее всего, на 4-ядерной системе будут работать 4-8 потоков, и все остальные будут просто по очереди ждать и завершаться.

Тут нам на помощь приходит асинхронный подход: идея в том, чтобы не ждать какой-то операции, а доверить это ожидание кому-то другому.

Давайте разберемся, как вообще работать с сетью без библиотек requests и aiohttp, чтобы подойти к идее асинхронного программирования.
Переходим к следующей карточке.