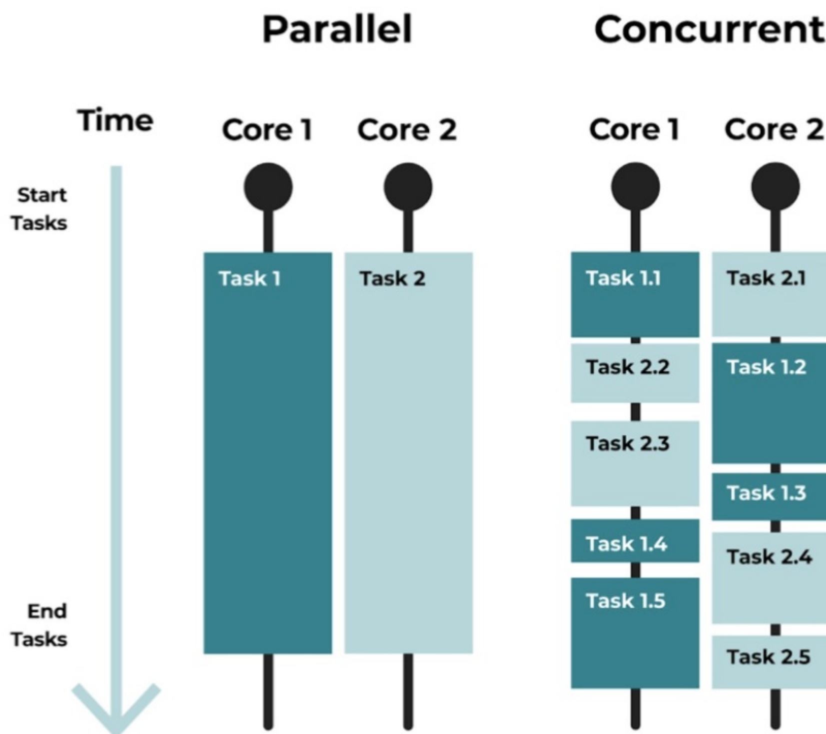
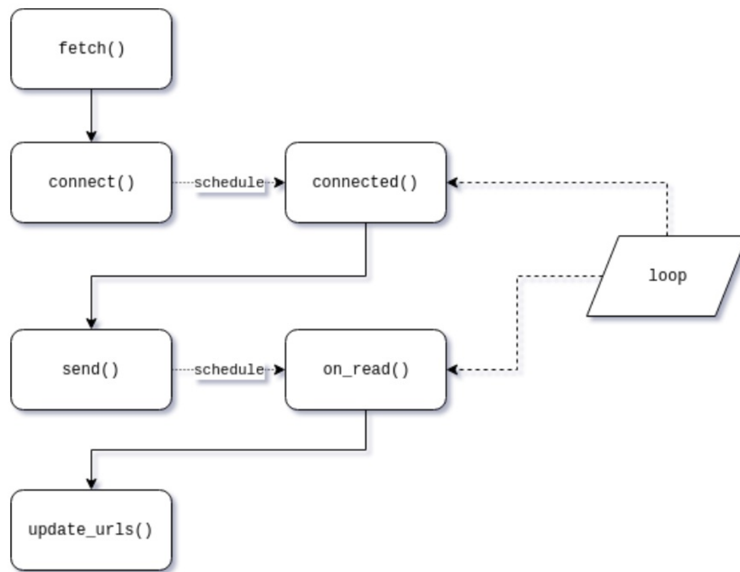
Конкурентность и параллельность



- В *параллельности* у нас есть задача 1 и задача 2, и они могут быть распределены на 2 разных ядра
- В *конкурентности* каждая из этих задач может быть поделена на какие-то небольшие подзадачи. Они могут перемещаться с одного ядра на другое

Один из подходов к асинхронной парадигме, который мы уже немного использовали — применение callback-функций. То есть при наступлении какого-то события вызывается связанная с ним callback-функция.

Пример:



Разберем, что происходит в схеме выше:

У нас есть функция fetch, которая скачивает html-документ.

Она выполняет вызов connect. После connect должен был вызваться *callback connected*, в тот момент, когда мы уверены, что у нас действительно установлено соединение. Именно этот момент означает, что мы можем отправить запрос в сокет. До наступления connected мы не можем отправить запрос — он упадет с ошибкой.

Как только произошел connected, внутри его callback-функции (*connected()*) мы можем сделать send. Но send тоже потенциально блокирующая операция, которая вернет исключение, если мы будем пытаться заблокироваться. Поэтому нужно так же подписаться на событие доступности сокета на чтение и так же сделать callback-функцию.

И так конечная функция update_urls может превратиться в псевдокод:

```
def update_urls():
  def connected():
     def can_read():
       def can_write():
         sock.send()
       sock.recv()
       selector.register(sock, can_write)
    selector.register(sock, can_read)

  sock.connect()
  selector.register(sock, connected)
```

В конечном счет все это может превратиться в CallbackHell, когда за одним callback следуют еще и еще.

Пример из js:

```
1   function hell(win) {
2    // for listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                 loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                   loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                     loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                       async.eachSeries(SCRIPTS, function(src, callback) {
14                         loadScript(win, BASE_URL+src, callback);
15                       });
16                     });
17                   });
18                 });
19               });
20             });
21           });
22         });
23       });
24     });
25   };
26 }
```

Для борьбы с CallbackHell нам могут помочь корутины.

Корутины

Принцип кооперативной многозадачности (cooperative multitasking) таков: мы в одном процессе пытаемся выполнить много задач. Антоним кооперативной — вытесняющая многозадачность, preemptive multitasking.

Вытесняющая многозадачность — это то, как работают потоки и процессы. Работал один поток, пришла ОС, сняла его с ядра и поставила другой поток на какой-то квант времени.

Корутины — функции, которые выполняются в рамках одного процесса.

Когда мы говорим, что мы поработаем в рамках одного процесса, и пытаемся что-то делать с множеством открытых соединений, корутин, которые должны *как бы* параллельно что-то запустить, мы говорим о корпоративной многозадачности. Смысл в том, что эти корутины и функции должны *сами* между собой договориться: когда будет выполняться одна, а когда — другая.

Корутины есть во многих языках. Самый яркий пример — Go. Там есть *горутины*, которые еще называют подпрограммы или сопрограммы. По факту это то же самое. Разница в том, что горутины могут работать на разных потоках: Go запускает несколько рабочих потоков и может сам управлять графиком корутин между потоками, а в каждом потоке работает свой событийный цикл.

Мы сейчас рассматриваем Python, и у нас 1 поток, 1 процесс, и в рамках этого процесса и потока у нас должен работать набор корутин.

Самое большое достоинство корутин относительно потоков в том, что мы можем создать их невероятное количество — они очень легковесны. Корутина — просто функция, а функций может быть десятки и сотни тысяч одновременно. Если ваша программа написана таким образом, что она не позволяет на долгое время заблокироваться, она будет работать эффективно — управление будет передаваться между корутинами, которые дождались выполнения нужного им события, а остальные корутины будут просто спать.

Корутины в Python 3 представлены библиотекой asyncio и синтаксисом async await. Но не подумайте, что если вы напишете async рядом с декларацией функции и await рядом с вызовом какой-то другой функции, код сразу станет асинхронным, быстрым и красивым. Все гораздо сложнее.

В следующем уроке мы напишем свой *Event Loop* и попробуем наконец-то понять, как избежать CallbackHell и писать красивый, эффективный и простой асинхронный код.