KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект >3 урок > Процессы. Демонизация. Веб-серверы

> Оглавление

> Процессы в Linux: сироты, зомби, демоны

Как правило, наши приложения — это процессы. В ML время работы модели сильно зависит от вычислительных мощностей, поэтому в этой лекции поговорим про процессы в Linux и Python.

> Процессы в Linux. Введение

Процесс — исполняемый код со своим адресным пространством. Процессы не делят адресное пространство, поэтому они более изолированные друг от друга. Например, мы можем на каждый процесс отдельно выставлять переменные окружения, можем менять приоритет процессов, а также ограничивать их в ресурсах.

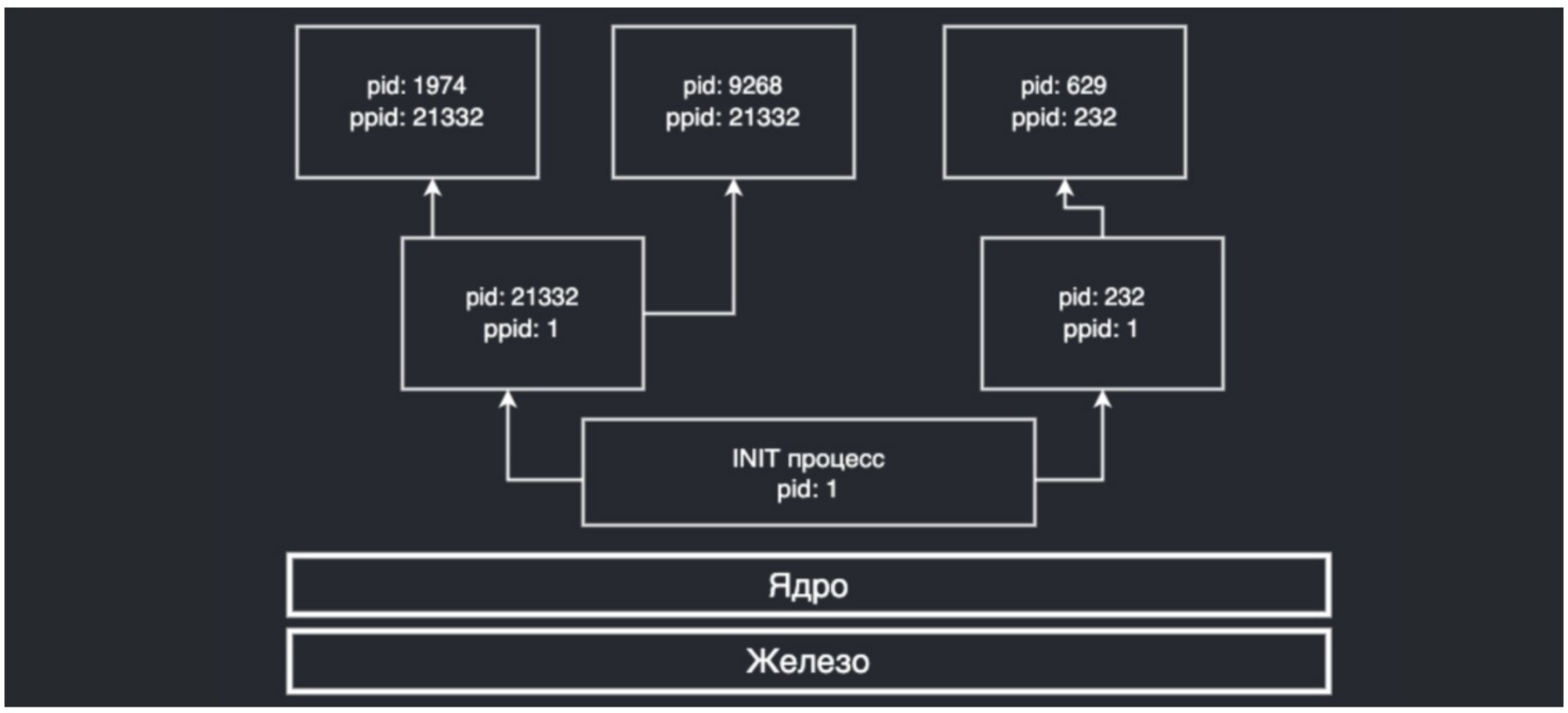
Для тредов внутри процесса это можно сделать только вручную на уровне логики кода. Почти вся информация, которую нам нужно понимать о процессах, отображается в таких утилитах, как htop или glances. Давайте откроем htop и увидим всё своими глазами.

```
1  [|||                                                          4.5%]   Tasks: 73; 1 running
2  [||||||                                                       9.0%]   Load average: 0.02 0.09 0.13
Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||531M/985M]   Uptime: 368 days(!), 23:54:28
Swp[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||112M/112M]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
    1 root       20   0  220M  5832  3924 S  0.0  0.6 24:53.30 /lib/systemd/systemd --system --deserialize 35
  508 vlad       20   0  225M 22136  5220 S  0.0  2.2  2:50.33 python3 /home/vlad/flask/app.py
  548 root       20   0  107M  1160  1012 S  0.0  0.1 29:24.21 /usr/sbin/irqbalance --foreground
  575 root       20   0 71680  2708  1936 S  0.0  0.3  0:35.89 /lib/systemd/systemd-logind
  598 root       20   0 31320  1116   992 S  0.0  0.1  0:45.99 /usr/sbin/cron -f
  600 messagebu  20   0 50548  2300  1436 S  0.0  0.2  1h10:12 /usr/bin/dbus-daemon --system --address=systemd: --nofo
  681 root       20   0  182M  1448  1356 S  0.0  0.1  0:00.08 /usr/bin/python3 /usr/share/unattended-upgrades/unatten
  686 root       20   0  166M  3848  1476 S  0.0  0.4  0:00.41 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-sta
  719 root       20   0 15956   660   592 S  0.0  0.1  0:00.00 /sbin/agetty -o -p -- \u --keep-baud 115200,38400,9600
  733 root       20   0 16180   440   312 S  0.0  0.0  0:00.00 /sbin/agetty -o -p -- \u --noclear tty1 linux
  977 x5_u_1     20   0 12884  2540  2392 S  0.0  0.3  2h02:45 bash /home/x5_u_1/log_generator.sh
 2463 evgen      20   0 76700  2860  2588 S  0.0  0.3  0:00.01 /lib/systemd/systemd --user
 2464 evgen      20   0  253M  1740     0 S  0.0  0.2  0:00.00 (sd-pam)
 2651 syslog     20   0  256M  3824  2528 S  0.0  0.4  7:07.37 /usr/sbin/rsyslogd -n
 4267 narek      20   0  4628  1480  1480 S  0.0  0.1  0:00.07 -sh
```

Интерфейс htop

Итак, в первых двух колонках видим уникальный ID процесса (`PID`) и пользователя, запустившего процесс. Процесс будет иметь те же права доступа, что и этот пользователь. Столбик `RES` показывает фактически занимаемую оперативную память, а `VIRT` — виртуальную занимаемую оперативную память (два столбика, показывающие занимаемые ресурсы относительно имеющихся). Колонка Command отображает команду, которая работает в процессе, а столбики NICENESS `NI` и PRIORITY `PRI` относятся к приоритетам процесса. Есть и другие более важные значения, отображение которых можно включить в настройках.



Как мы знаем, все процессы создаются другими процессами, кроме первого INIT-процесса (с PID 1). И у каждого процесса есть ссылка на родительский ID

процесса (PPID). В результате получается древовидная структура. INIT-процесс — первый процесс в системе с PID 1, запускающий остальные необходимые системные процессы, и у него нет родителей.

```
 PPID   PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
    0     1 root       20   0  220M  5832  3924 S  0.0  0.6 24:54.70 /lib/systemd/systemd --system --deseria
17344   508 vlad       20   0  225M 22200  5284 S  0.0  2.2  2:52.63 python3 /home/vlad/flask/app.py
    1   548 root       20   0  107M  1160  1012 S  0.0  0.1 29:25.11 /usr/sbin/irqbalance --foreground
    1   575 root       20   0 71680  2708  1936 S  0.0  0.3  0:35.92 /lib/systemd/systemd-logind
    1   598 root       20   0 31320  1116   992 S  0.0  0.1  0:46.01 /usr/sbin/cron -f
    1   600 messagebu  20   0 50548  2300  1436 S  0.6  0.2  1h10:17 /usr/bin/dbus-daemon --system --address=
    1   681 root       20   0  182M  1448  1356 S  0.0  0.1  0:00.08 /usr/bin/python3 /usr/share/unattended-u
    1   686 root       20   0  166M  3848  1476 S  0.0  0.4  0:00.41 /usr/bin/python3 /usr/bin/networkd-disp
    1   719 root       20   0 15956   660   592 S  0.0  0.1  0:00.00 /sbin/agetty -o -p -- \u --keep-baud 11
    1   733 root       20   0 16180   440   312 S  0.0  0.0  0:00.00 /sbin/agetty -o -p -- \u --noclear tty1
30117   977 x5_u_1     20   0 12884  2540  2392 S  0.0  0.3  2h02:53 bash /home/x5_u_1/log_generator.sh
    1  2463 evgen      20   0 76700  2860  2588 S  0.0  0.3  0:00.01 /lib/systemd/systemd --user
```

Иллюстрация процессов, запущенных INIT (PPID = 1)

Из базовых аспектов нам ещё пригодится понимание того, как эти процессы размножаются.

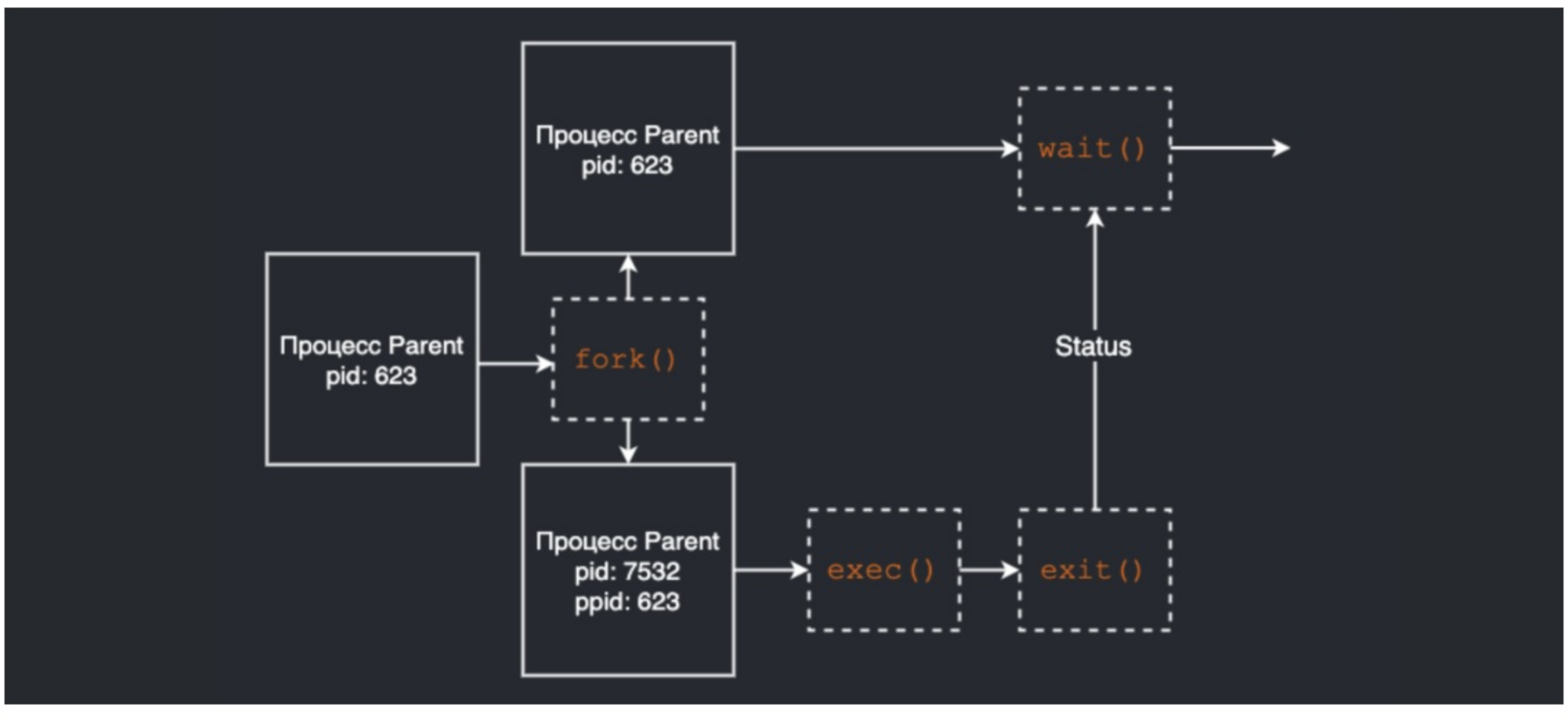


Схема размножения процессов через fork()

Для создания процесса используется системный вызов `fork()`. Эта схема приведена здесь больше для полноты повествования — нам нужно знать только про `fork()`, который как раз и создаёт полную копию процесса, вызывающего этот самый `fork()`, и начинает выполнять тот код, который в вызывающем процессе находился после `fork()`. Копия создаётся путем копирования адресного пространства процесса, что дорого и вместе с инициализацией процесса занимает время. Однако копирование адресного пространства ленивое, поэтому если адресное пространство в дочернем

процессе не изменяется, то в принципе операция может пройти быстро. Также на этой схеме видно, что после выполнения дочернего процесса, предок пытается собрать с него статус выполнения.

Также, как мы помним из предыдущей лекции, одним из способов общения между процессами являются сигналы — `SIGTERM`, `SIGHUP`, `SIGINT` и т.д.

> Процессы-сироты, процессы-зомби и процессы-демоны

На одном из слайдов мы увидели большое количество процессов, у которых родитель — INIT-процесс с `PID` 1.

Такие процессы есть из-под пользователя `root`, и команды этих процессов похожи на что-то системное, и весьма вероятно, что они и были запущены INIT-процессом. Также мы видим процессы с предком INIT не от `root` пользователя, и команда процесса даже понятная. Этому есть объяснение: дело в том, что если у процесса родитель — INIT-процесс, это не всегда означает, что эти процессы были созданы INIT-процессом и он был их первоначальным родителем. Некоторые процессы приобретают INIT-процесс как родителя, хотя изначально у них были другие родительские процессы.

Это процессы-сироты и процессы демоны — нам нужно знать и про первые, и про вторые. Не стоит путать их с процессами-зомби, которые получили своё название не из-за того, что они потеряли родителя, а потому, что они уже закончили своё выполнение, т.е. они уже мертвы, но система всё ещё считает их существующими, т.к. они по какой-то причине продолжают числиться в таблице процессов.

Кстати, зомби рождаются, потому что по какой-то причине родитель не получил статус после `exit()` потомка. Они могут стать большой проблемой, потому что таблица процессов конечная и ограничивается десятками тысяч записей, что не так уж много, и если её забить, то новые процессы не создадутся даже несмотря на то, что фактические ресурсы свободны. С зомби процессами вы будете сталкиваться крайне редко.

Процессы-сироты и демоны в отличие от зомби не завершили свою работу. Просто по какой-то причине они отвязываются от родителя, и родитель уже в общем случае не имеет над ними контроля. Процессы-сироты — это, очевидно, тоже плохо, потому что они занимают ресурсы и избавляться от них придётся вручную. С ними вы будете иметь дело время от времени. И те, и другие могут

появляться при использовании забагованного ПО — в основном C/C++, но в Python это тоже реально.

Самая частая причина появления сирот — неосторожное использование сигнала `SIGKILL`, который часто используют на подвисших пайплайнах.

Процессы-демоны — это процессы, работающие в фоновом режиме и не зависящие от пользователя. Если процесс запущен из управляющей консоли, то демон-процесс от этой консоли зависеть не должен. Поэтому в некоторых случаях единственное отличие демона от сироты в том, что демона мы намеренно сделали демоном. В общем случае предком демона не обязательно должен быть INIT-процесс, как у осиротевшего процесса. Например, если вы используете какой-нибудь демонизирующий оркестратор, то родителем для демонов может выступать процесс оркестратора. Давайте рассмотрим распространённые способы демонизации:

1. Tmux-сессии. Если вы не сталкивались с этим, то предлагаем ознакомиться самостоятельно. Технически процессы в tmux не полноценные демоны, потому что процессы до сих пор присоединены к управляющей сессии — просто у вас появляется возможность отключаться от неё. Практика показывает, что tmux-сессии не очень стабильны — они иногда слетают, и все процессы, запущенные в них, соответственно тоже слетают. Если окружение сменилось, то сессию нужно пересоздавать, чтобы новые настройки применились.
2. Постановка знака `&` в конце команды. С помощью этого можно перевести процесс в фоновый режим, и он будет доступен с помощью команд `jobs` и `fg`, а его предком будет наша консоль. После отключения от управляющей сессии скрипт продолжит исправно выполняться, но мы уже не будем иметь доступ к процессу через команды `jobs` или `fg`, и предок сменится на INIT-процесс.

Значит ли это, что здесь не полноценная демонизация? Нет, ведь наш процесс работает в фоновом режиме без участия пользователя, и предок закономерным образом сменился на INIT-процесс после смерти исходного предка, т.е. нашей консоли, чего мы и добивались. Тот факт, что `jobs` и `fg` после переподключения не находят наш процесс, означает лишь их примитивность — если бы эти утилиты каким-то образом сохраняли хотя бы PID запущенных на фоне процессов, то после переподключения они бы имели возможность найти наши процессы и управлять ими. Вообще функционал у них минимальный,

поэтому мы рассмотрим зрелые демонизаторы, которые на самом деле уже можно использовать в каком-нибудь проде.

> Менеджеры процессов PM2 и systemd

> PM2

PM2 — это довольно зрелый менеджер процессов: он может демонизировать, масштабировать, мягко обновлять ваши workload'ы. Предлагаем сразу рассмотреть пример.

Допустим, процесс запускается в демон-режиме с помощью команды `start`. Расширение `.py` парсится, и pm2 запускает скрипт интерпретатором питона. Если нужно изменить интерпретатор, то можно использовать аргумент `--interpreter=python3`.

`pm2 ls` позволит просмотреть все демонизированные процессы и их статус, а `pm2 logs 0` подключит нас к stdout (потоку вывода) процесса.

Если приложение пишет много логов, то появляется один нюанс, связанный с хранением логов и актуальный почти для многих инструментов. Поток вывода stdout процессов записывается в файлы в папке `~/pm2/logs`, и если они разрастаются, то работа PM2 с ними замедляется, поэтому в таких случаях необходимо позаботиться о ротации логов — чтобы старые записи из файла удалялись или архивировались, а файл оставался постоянного размера. Ротировать можно как bash-скриптом, поставленным на cron (этот инструмент легко гуглится, поэтому оставим его на самостоятельное изучение) или же плагином для PM2, который занимается ротацией файлов.

Перезапускать приложение можно как "жёстко" с помощью команды `pm2 restart`, так и "мягко" с нулевым временем простоя с помощью `pm2 reload`. Довольно удобно масштабировать с помощью `pm2 scale <app_name> <replics>`, который просто создаст несколько одинаковых процессов. Но такой однострочный вариант не подходит для веб-сервисов, потому что веб-сервисы занимают порт и при таком масштабировании каждый из N отмасштабированных процессов попытаются занять один и тот же порт, из-за чего возникнет конфликт. К сожалению, PM2 не предоставляет никаких сетевых абстракций, поэтому если вы хотите с помощью него балансировать трафик между приложениями на Python, то вам придётся сделать следующее:

1. Сделать так, чтобы приложения запускались на разных портах. Можно либо заранее прописать порты в файлах конфигурации, либо реализовать логику выбора порта внутри приложения.
2. Реализовать балансировку по разным портам: как на клиенте, так и поставив балансировщик перед репликами.

Если приложение не является веб-сервисом (допустим, оно читает из Kafka, предсказывает и обратно пишет в Kafka), то всё в порядке, и можно масштабировать через PM2 — конфликтовать в таком приложении нечему.

Когда мы говорили про файл конфигурации или "конфиг", в котором нужно заранее прописать порты, имелся в виду `json`-файл, который можно использовать для развёртки. Он ещё часто называется "манифестом" (manifest). В этом манифесте мы указываем процессы, переменные окружения и какие-либо другие параметры, которые мы хотим запустить с помощью PM2. Такой способ довольно прост и удобен, поскольку вы сразу видите то, что получите, а подход называется декларативным.

PM2 довольно прост и приятен в использовании. Декларативный подход к описанию итогового состояния с помощью `json`-манифестов, который мы только что рассмотрели, реально удобен и будет встречаться в других инструментах. Некоторые вещи в нём даже удобнее, чем где-либо ещё: например, можно быстро перезапустить процесс с добавлением забытой переменной окружения с помощью аргумента `--update-env`. Но такие удобства исходят из того, что мы демонизируем голые процессы, не используя контейнеризацию.

PM2 достаточно стабилен, чтобы какое-то время использовать его в проде, но из-за того, что он работает напрямую с процессами, довольно быстро возникают неудобства:

- PM2 не предоставляет никаких сетевых абстракций, и мы не можем отмасштабировать наше приложение, не "попрыгав через обручи".
- Как только дело дойдёт до деплоя на несколько хостов, вам придётся уже использовать команду `pm2 deploy`, что уже не так приятно, как командный интерфейс. Но главное — вам придётся держать окружения на всех целевых машинах одинаковыми, что добавляет работы и всё равно иногда приводит к ошибкам.
- Есть ещё одни момент, который в некоторых случаях будет плюсом PM2. Из-за того, что PM2 работает с процессами без всяких докеров, в нём

остается разграничение пространства: один пользователь не может оркестрировать чужие процессы. Сюда же можно отнести все плюсы оркестрации контейнеризированных систем.

Тут следует сказать, что в PM2 есть плагин для docker, но в таком контексте PM2 стал уже больше походить на Ansible (о нём мы будем говорить далее в лекциях). В остальном PM2 инструмент интересный, и, наверное, не будет лишним в него углубиться, особенно если вы пишете на Node.

PM2 — это всего лишь пример. На GitHub есть много других менеджеров процессов (например supervisord для Python), которые делают примерно то же самое.

> systemd

Есть ещё один способ демонизации и управления процессами, который был включён в дистрибутивы Linux относительно недавно по меркам того же Linux. Речь идёт о systemd. Мы задеплоим приложение с помощью systemd на одной из следующих лекций, а сейчас просто немного обсудим его.

Systemd — это несколько разных бинарников, собранных под одним проектом, которые фактически дублируют функционал уже существующих инструментов (что некоторым как раз очень не нравится), но делают его использование более удобным. С помощью systemd можно не только запускать и контролировать службы, но и делать для них расписание, управлять журналированием и решать далёкие от нас задачи вроде автоматического монтирования, активации службы по сокету и т.д. Из всего это нам понадобится только функционал, управляющий демонами, и функционал, запускающий демона по таймеру.

После перечисления того, что умеет systemd, складывается впечатление, что это скорее инструмент для администраторов системы и он не создан для "выкатки" приложений. На самом деле так оно и есть. В ML комьюнити deploy через systemd — большая редкость, ведь есть Docker, Kubernetes и другие инструменты. Возможно, его редко используют потому, что неудобно масштабировать процессы: нет сетевых абстракций, а когда дело доходит до развёртывания на нескольких хостов, то приходится вставлять свои инфраструктурные костыли.

Возможно, самым частым случаем использования systemd является поднятие Nginx. Но у этого инструмента есть некоторая гибкость и небедный функционал. Также у него есть отличительная особенность — он даёт возможность развёртывать нативно, т.к. распространяется с дистрибутивом

системы, и вам не нужно ничего доустанавливать: вы только описываете конфигурационный файл, и всё готово. Кстати, это может быть единственным решением, если в вашей компании всё очень строго с установкой ПО на машины.

> Процессы и потоки в Linux

> Потоки в Linux

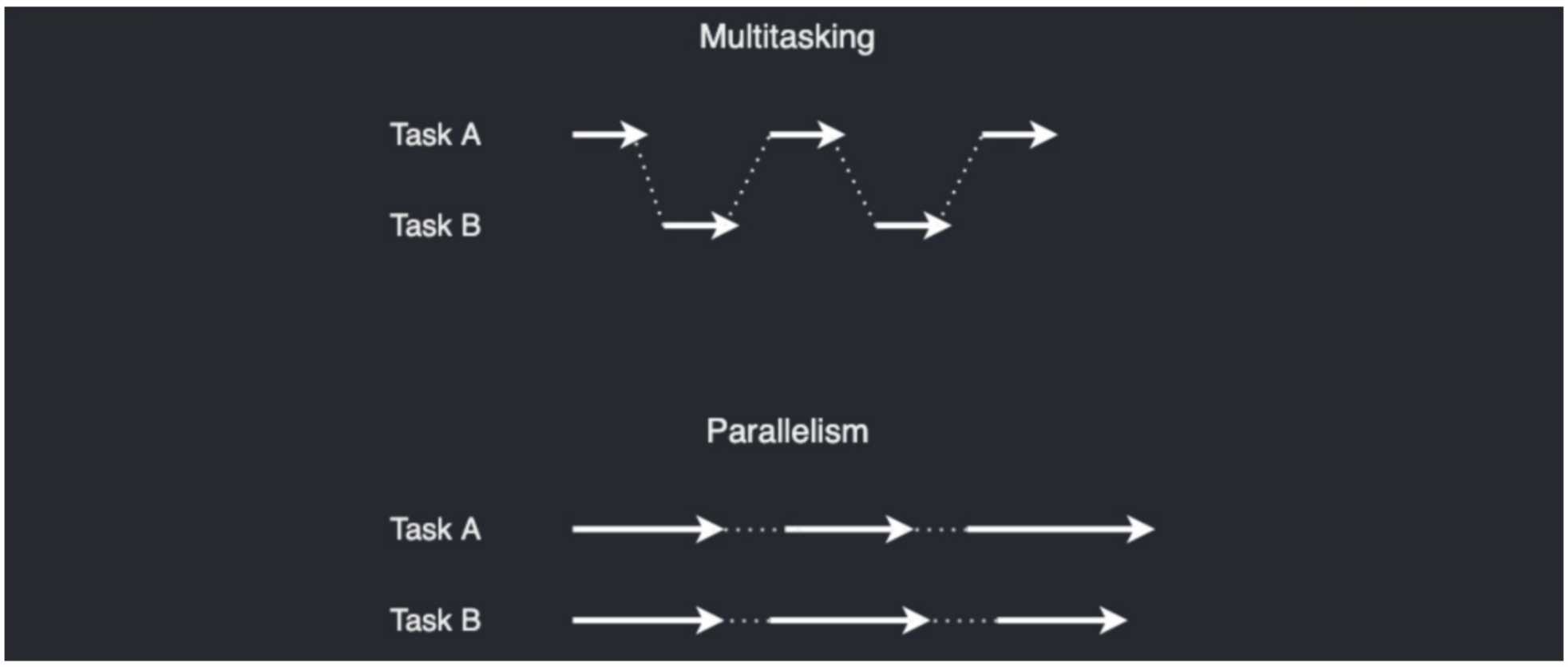
Мы немного разобрались с процессами в Linux, теперь поговорим о тредах.

Мы знаем, что потоки (треды, нити) отличаются от процессов тем, что делят оперативную память между собой. То есть треды создаются довольно быстро, поскольку адресное пространство для них уже готово, а процессам часто приходится копировать это адресное пространство к себе, что делает их создание не таким быстрым. Взаимодействие между тредами также довольно быстрое за счёт общей памяти, тогда как у процессов оно происходит медленнее, потому что у них свои механизмы коммуникации.

В остальном процессы и треды не сильно отличаются — для ОС это всё задачи (tasks). Разработчики ядра утверждают, что упомянутые небыстрые механизмы, специфичные для процессов, оптимизируются и по скорости приближаются к тредам.

В htop, кстати, треды отображаются так же, как и процессы по умолчанию. Поэтому если вы решаете на C++ CPU-bound задачу под Linux, то вполне возможно не будет особой разницы в производительности между реализацией на тредах и реализацией на процессах (конечно, всё зависит от алгоритма). Так может происходить потому, что треды, как и процессы, могут выполняться

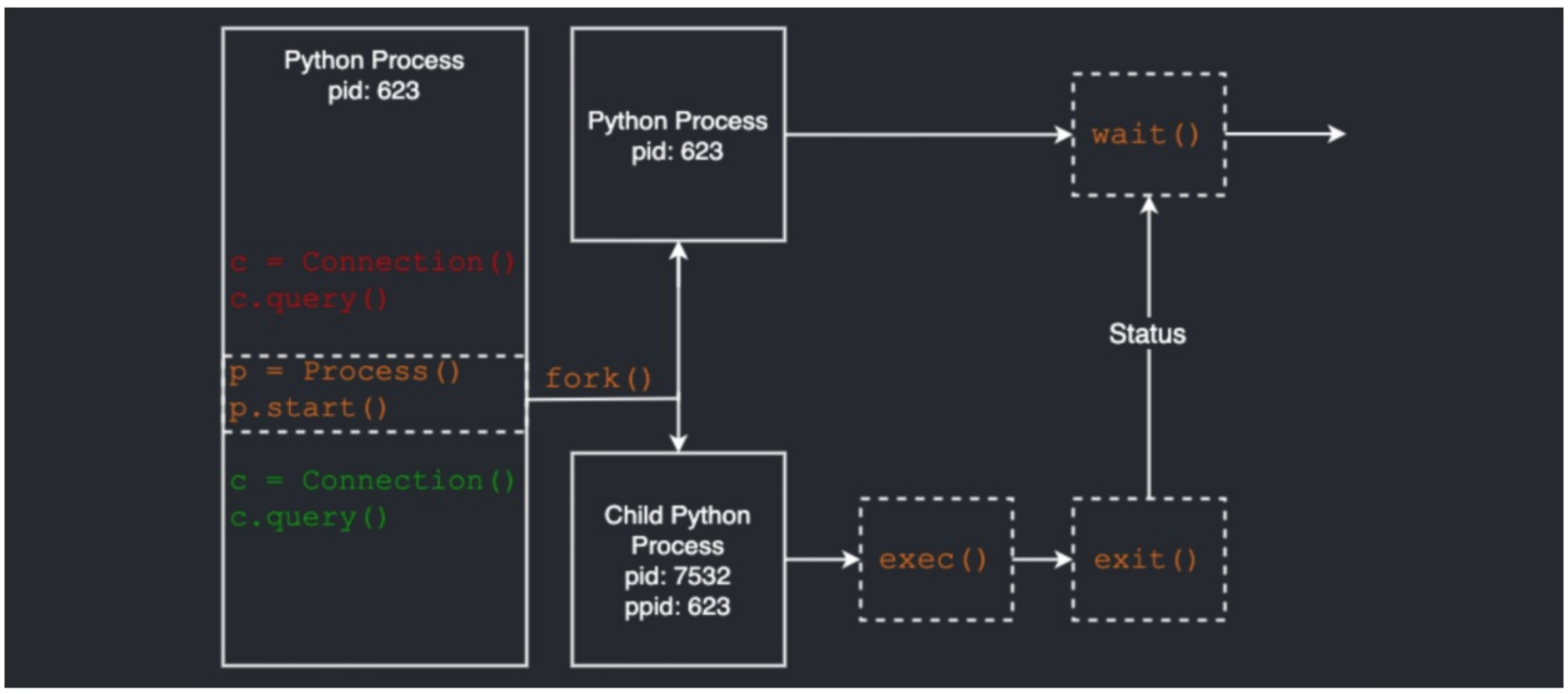
параллельно и одновременно, используя разные ядра, и, в отличие от использования процессов, издержки тут будут минимальны. На самом деле треды могут выполняться не параллельно, а последовательно.



На этом моменте перейдём от Linux к Python.

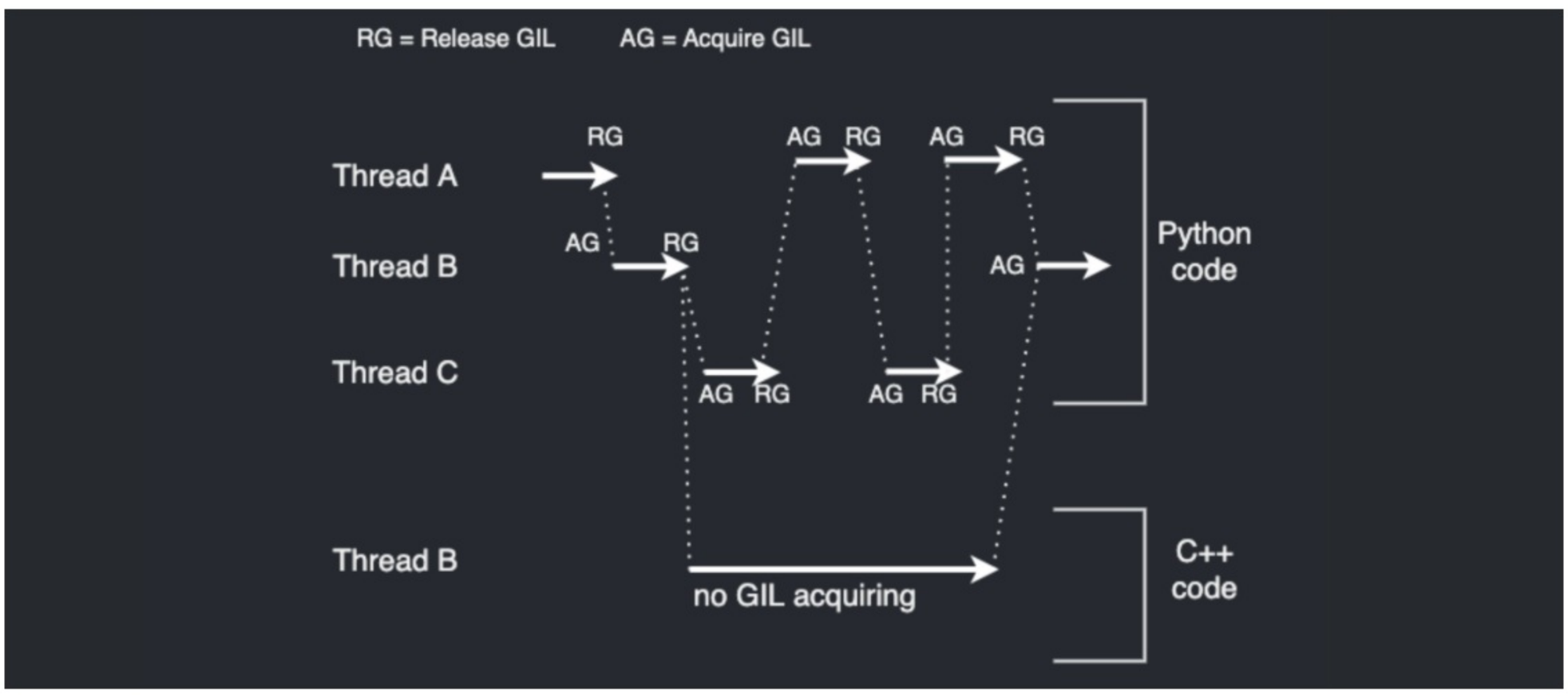
> Процессы и потоки в Python

На питоновские процессы мы уже посмотрели — это обычные процессы, которые можно демонизировать, склонировать и превратить в сироту. Однако тут необходимо дать рекомендацию: поскольку мультипроцессинг "форкает" (`fork()`) процессы, старайтесь делать программы с ним как можно проще, чтобы при создании дочерних процессов в них не копировались из родителя такие сложные объекты, как `Connection()` и различные объекты синхронизации. Также старайтесь избегать использования тредов.



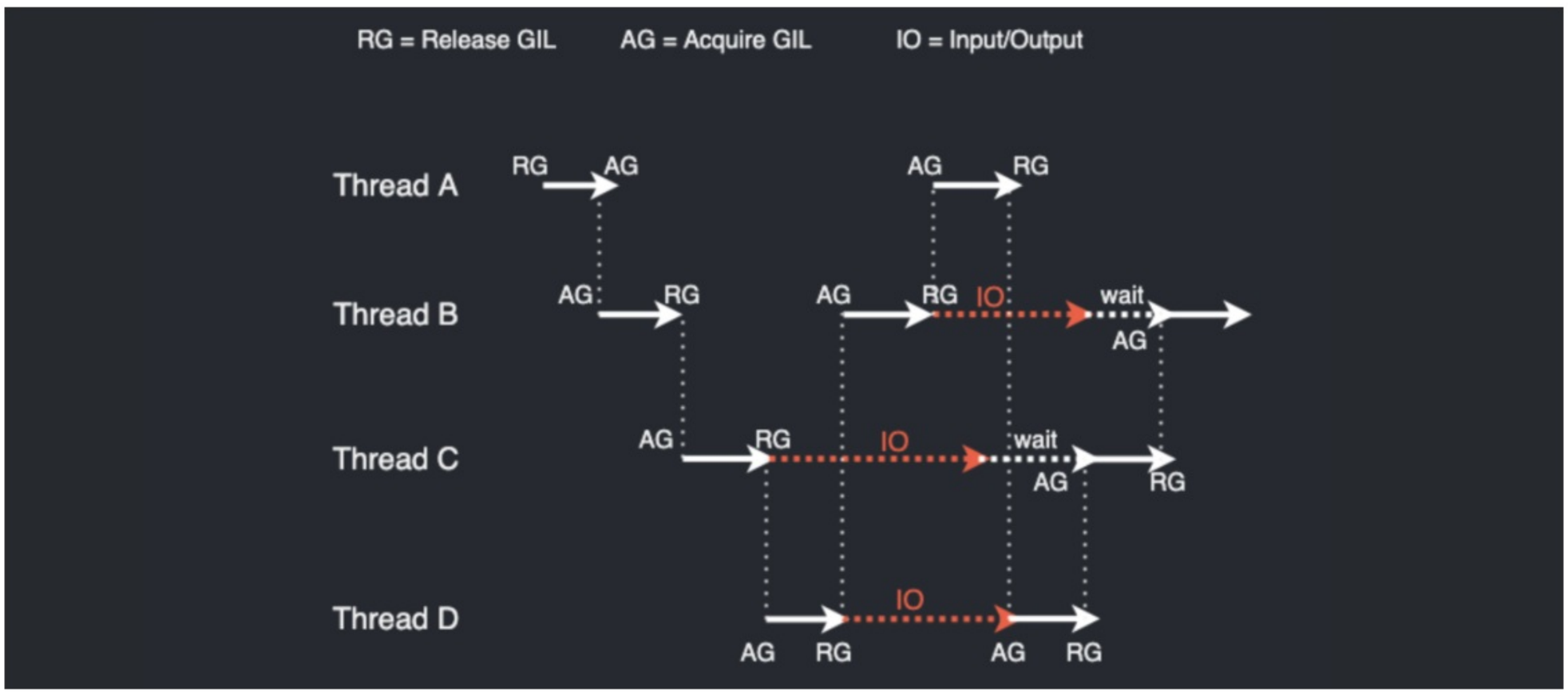
В Python также есть многопоточность, а вместе с ней и печально известный GIL, который запрещает параллелизм тредов, делая их выполнение последовательным.

Последовательное выполнение означает, что в каждый момент времени выполняется только один тред, следовательно, в каждый момент времени процесс Python может использовать только 1 ядро. Очевидно, это существенное ограничение, ведь пока выполняется один тред, другие ждут. В результате все они выполняются с большой задержкой.



В идеале нам хотелось бы, чтобы треды в процессе выполнялись параллельно, и поэтому интерпретатор выделяет каждому потоку (треду) какое-то небольшое время исполнения и быстро переключается между потоками. В результате создаётся ощущение полноценной параллельности работы потоков.

Эти постоянные переключения проходят совсем не незаметно для системы, но всё не так плохо. Например, когда в потоке выполняется IO-команда, то поток отпускает GIL, тем самым допуская настоящее параллельное выполнение нескольких IO-команд разных потоков. Также необходимо сказать, что питоновский поток — это обёртка над Linux-тредом, которая как раз позволяет работать с GIL и делать некоторые другие вещи, а как известно, операции с Linux-тредами хоть и не бесплатные, но всё же более дешёвые, чем операции с процессами.



> Синхронные веб-сервисы

Итак, теперь мы знаем и про процессы, и про треды в Python. Нужно разобраться, что из этого используют наши Python-приложения, чтобы подобрать лучшие параметры для их запуска.

Для офлайн-задач, требующих вычислений, multiprocessing решает проблему — там не важна скорость, а потребность в вычислениях, как правило, высокая. Поэтому нужны несколько процессов для использования нескольких ядер, а сколько процесс будет создаваться, уже не так важно.

А что насчёт онлайн-задач и нашего Flask-сервиса? Было бы хорошей идеей создавать новый процесс на каждый запрос? Для сервиса, работающего на лету, создание процесса всё-таки долговато, и при весьма реальном количестве запросов на такой сервис, много ресурсов начнёт уходить именно на `fork()` операцию, что понизит производительность. Как мы помним, количество процессов ограничено в системе, что в свою очередь ограничивает количество принимаемых запросов. Вы, вероятно, могли подумать, что модель процессов хороша тем, что на Input/Output (I/O) задачах треды/потоки будут блокироваться и другие треды будут ожидать эту бессмысленную блокировку, хотя полезной нагрузки на процессорах не будет. Но ведь GIL тоже отпускается на I/O задачах, так что это не является преимуществом. Получается, что потоки наиболее оптимальны с точки зрения производительности, ведь создаются они быстрее, чем процессы.

Вообще говоря, переключение между тредами всё-таки стоит каких-то ресурсов и при большом количестве тредов времени на их переключение требуется всё больше. Поэтому если у вас бизнес-кейс, при котором требуется отдавать

inference, а результат запросов, выполняющийся дольше N миллисекунд, отбрасывать с каким-то fallback, то было бы здорово иметь возможность ограничивать количество активных тредов на сервере, чтобы успевшие пройти запросы отрабатывались за нужное время. Об этом мы поговорим далее.

Мы узнали, что наш Flask-сервис работает по мультипоточной схеме. Также мы выяснили, что есть и мультипроцессорная модель, которая, кстати, использовалась на заре веба. И обе эти модели синхронные. Те, кто не знаком с асинхронной моделью и услышал, что их сервисы синхронны, обычно сразу попадают под власть навязчивой идеи, что асинхронная модель подходит им больше. Давайте поскорее разберёмся, нужны ли нам асинхронные сервисы в ML.

> Асинхронные веб-сервисы

Давайте для начала убедимся, что мы правильно понимаем, что такое асинхронность. Если у вас есть какая-то очередь, в которую вы можете помещать свои задачи, которые уйдут выполняться на выделенных машинах, а результат потом можно будет забрать в любой момент, то такую обработку задач можно назвать асинхронной. Но это асинхронность на архитектурном, системном уровне — она не годится для real-time сервисов. Мы будем говорить про другую асинхронность — асинхронность задач на уровне интерпретатора.

Потребность в асинхронной модели возникла потому, что синхронная модель обладает некоторыми недостатками:

1. Работа с тредами не такая уж и простая, и если вы хотите использовать объекты, разделяемые между тредами, то вам всё равно понадобится работать с объектами синхронизации даже несмотря на то, что уже есть GIL.
2. Когда тредов много, переключение между ними становится дорогим.

> Event Loop

+15

Асинхронная модель полностью отказывается от тредов, фактически оставляя только один, который и будет обрабатывать все запросы. Раньше в один момент времени все равно работал только один тред, поэтому мы ничего не теряем, а если учитывать избавление от необходимости переключаться между тредами, то можно сказать, что мы даже выигрываем. Остаётся только придумать, как внутри этого одного потока переключаться между обработкой

запросов, ведь раньше эта проблема решалась интерпретатором, когда он переключал потоки.

В асинхронной однопоточной модели эта проблема решается с помощью event loop и coroutines (сопрограммами). Разумеется, погружаться в это мы не будем, но если вкратце, то происходит следующее. В нашем потоке работает бесконечный цикл, который знает о всех запросах, которые к нему пришли, и почти сразу берёт их в выполнение. Как только в обработчике запроса вызывается асинхронная I/O команда, event loop откладывает эту задачу и берёт следующую команду, которая уже выполнила I/O задачу, и ждёт завершения или выполнения синхронной команды. Таким образом, получается, что как только задаче требуется выполнить асинхронную команду, команда выполняется асинхронно в этом же потоке (а не в параллельном потоке, как это было в синхронном режиме), а по её завершении она в порядке очереди, но без лишних lock'ов и переключений, вернётся дальше на исполнение в event loop. Чтобы эта схема работала, нужно только, чтобы event loop сразу же каким-то образом мог узнать о завершившейся асинхронной команде в задаче, чтобы вернуть её на выполнение. И он умеет это понимать: как раз на каждой итерации цикла он проверяет, есть ли такие задачи, которые завершили I/O операции и готовы вернуться на выполнение в поток.

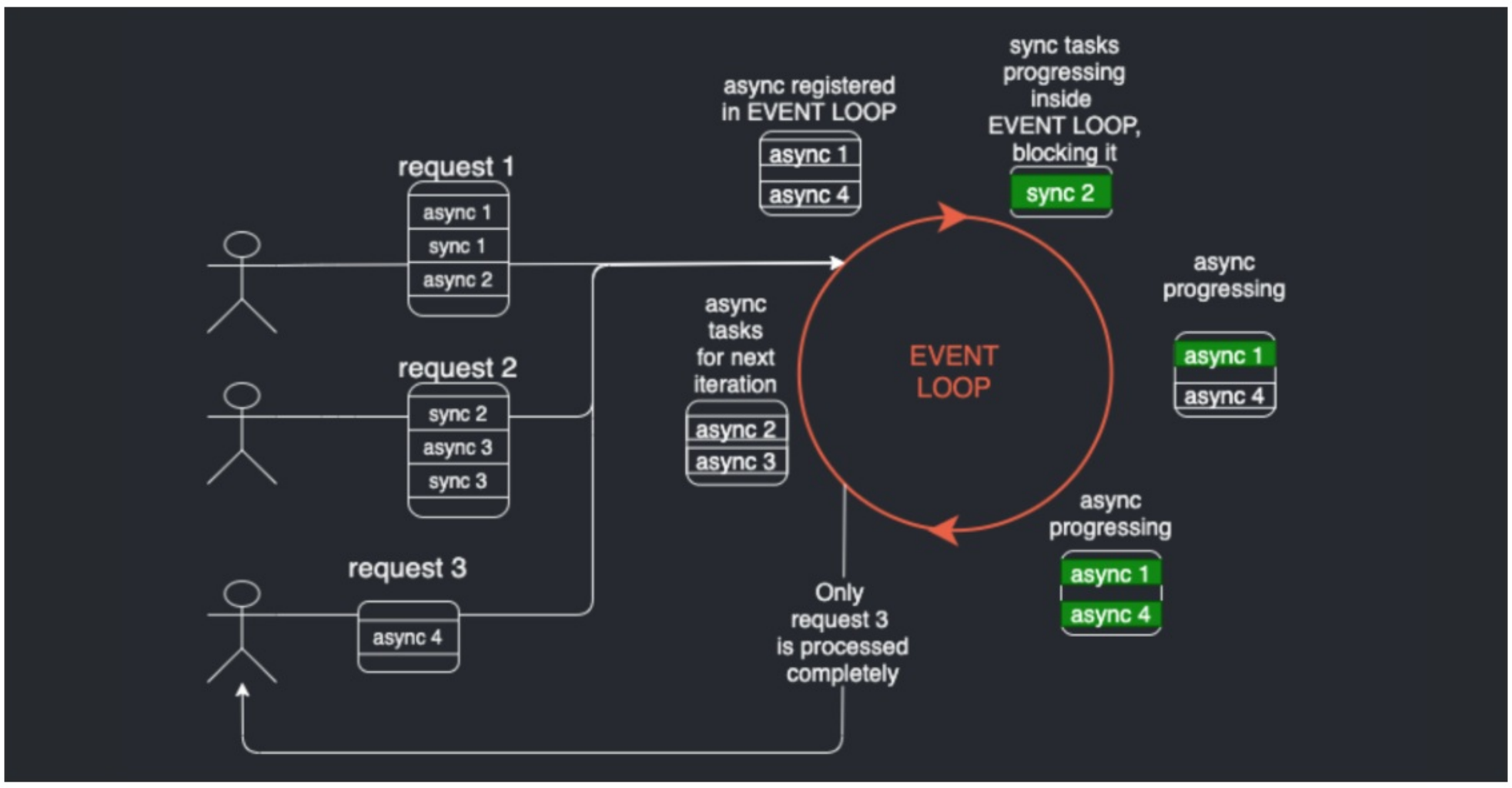
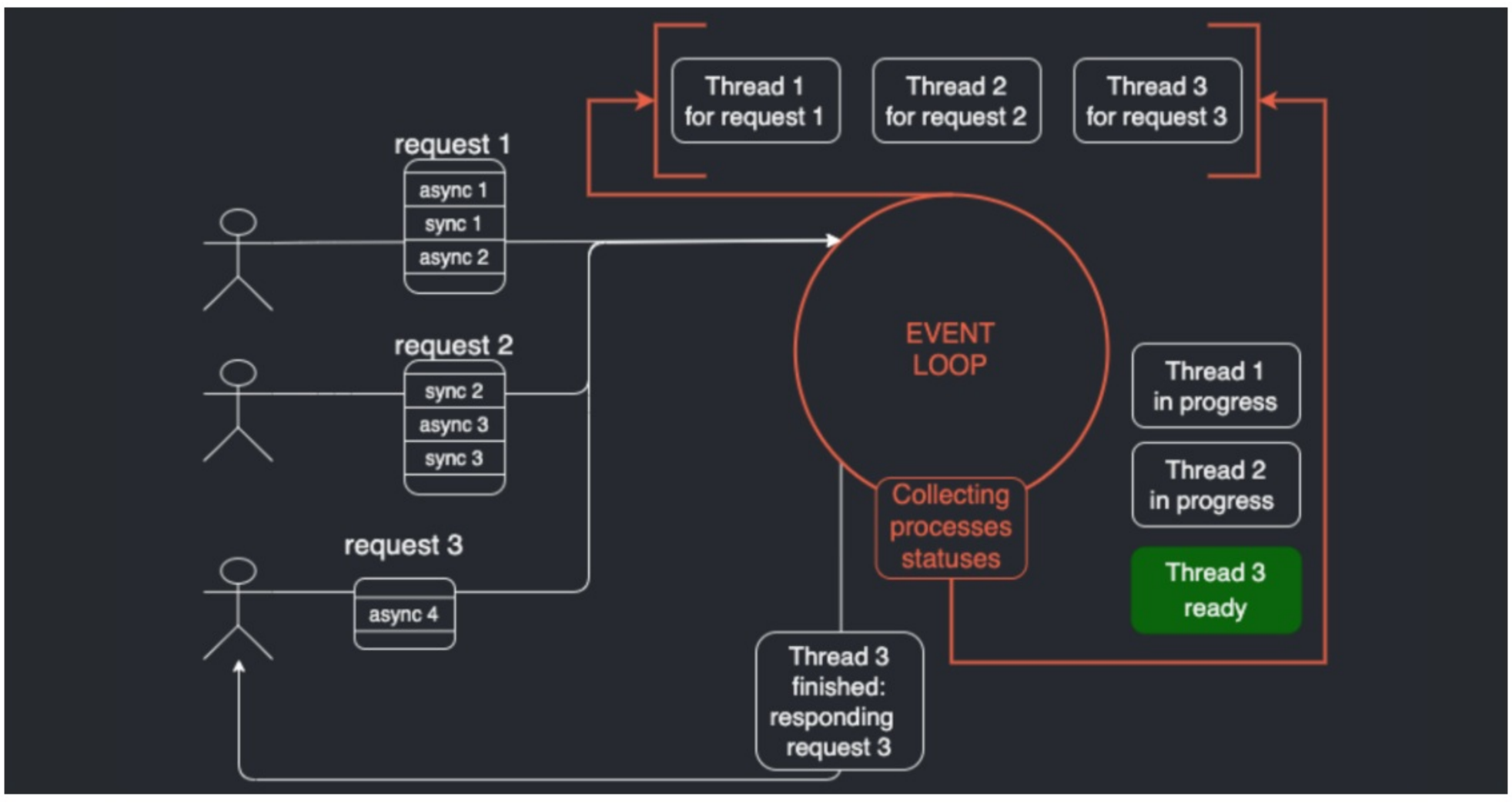


Схема работы Event Loop

Вы могли заметить, что и выполнение синхронных задач, и переключение между задачами запросов происходят в одном и том же потоке. То есть если

поток заблокируется какой-то операцией, то переключение на таску и создание новой таски не произойдет.

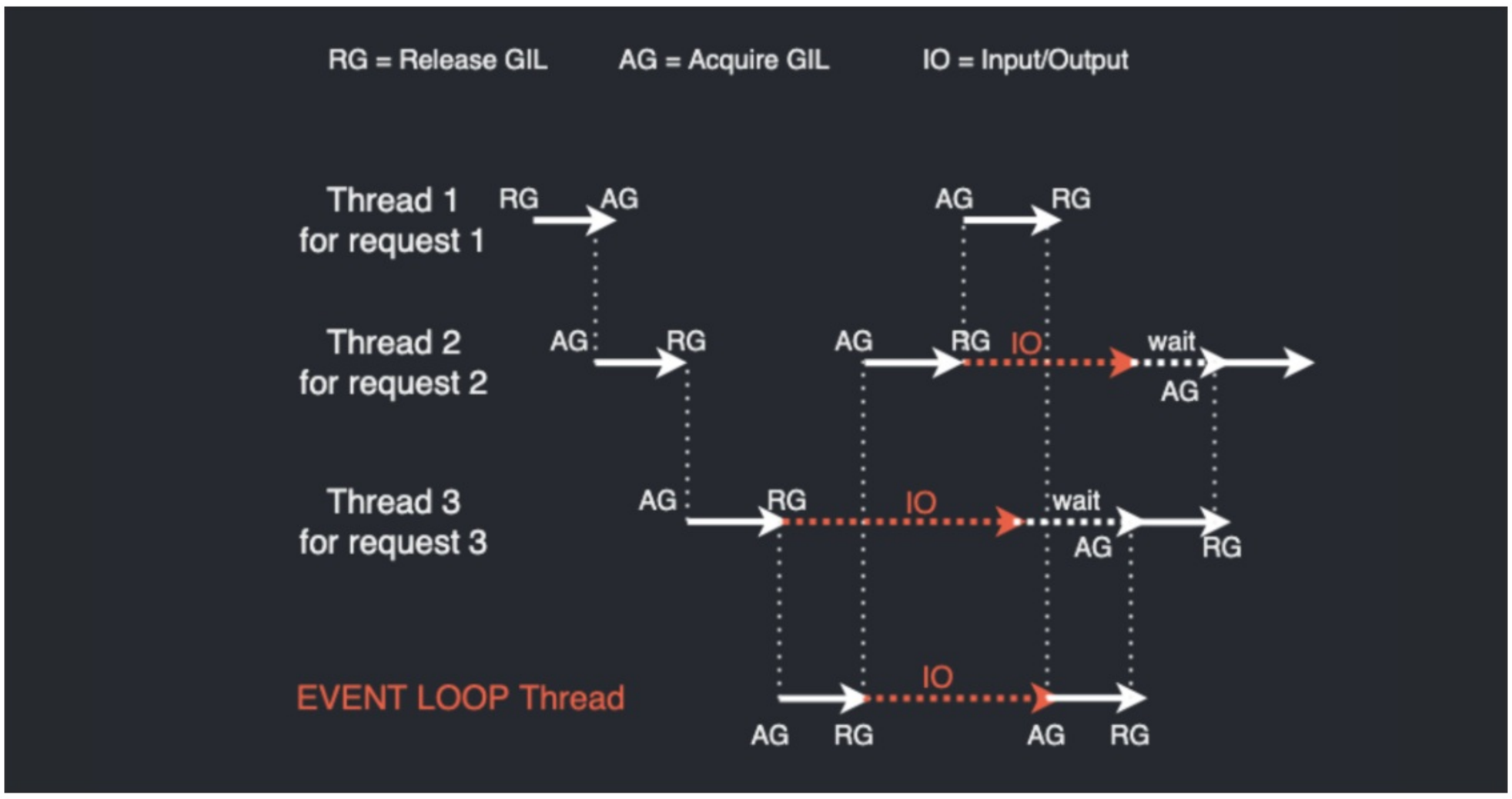
Предположим, в ваших задачах (тасках) почти все операции асинхронные, но есть на первый взгляд безобидная команда, которая проходится по листу и делает переформатирование. Однако из-за того, что это занимает какие-то миллисекунды, у нас не берутся в работу новые запросы. С какого-то момента времени выполнения этой команды, запросы начнут лавинообразно накапливаться и наш веб-сервис не справится с нагрузкой. В общем, event-loop максимально чувствителен к вычислительно-ёмким с точки зрения CPU задачам и при большой нагрузке нужно следить даже за тем, чтобы логи писались быстро и без всяких форматирований и преобразований. Поэтому в event-loop делать inference модели нельзя.



Конечно, вы можете выносить inference из event loop, создавая подпроцессы, но если каждая таска будет создавать подпроцессы, то их станет много и это заспамит систему (практически произойдёт возвращение к синхронной модели работы). Поэтому такую модель для сервинга моделей рекомендуется использовать тогда, когда нет других вариантов.

Вместо процессов можно создавать и треды, но в этом случае мы возвращаемся к тому, что event-loop'у будут мешать конкурирующие треды, и такой вариант вообще не рекомендуется использовать. Если вы когда-либо использовали TF-serving, то в логах могли видеть, что он использует event-loop.

Но TF-serving написан на C++, поэтому там на каждый запрос можно создавать тред, и он не будет конфликтовать с event-loop'ом.



Мы уже сказали, что для ML асинхронную модель в Python мы использовали бы только в последнюю очередь. Для того чтобы обслуживать модель, нужны синхронная схема и потоки. Желательно, как мы уже говорили, чтобы потоков было как можно меньше, чтобы не было оверхеда, а масштабировать приложение хорошо бы процессами. Но и процессы далеко не панацея, ведь ядер ограниченное число, и одновременных запросов скорее всего будет больше, чем ядер, т.е. система будет делить ядра между процессами. Но это точно лучше, чем 1-2 процесса и по 50 одновременных потоков/тредов в каждом.

Что же делать, если во время inference нашему сервису нужно ходить в хранилища и другие сервисы, т.е. совершать I/O задачи. Если на каждый запрос мы должны делать и I/O обращения, и inference, то стоит оставить сервис синхронным и, возможно, постараться ускорить эти I/O обращения, используя кэш. Если необходимо сделать много I/O обращений, а потом inference, то можно вынести I/O обращения в отдельный асинхронный сервис, а сам inference оставить в синхронном сервисе. Если же вашему сервису изредко нужно делать inference, и на большинство запросов его не происходит, потому что используется кэш или что-то в этом духе, то всё равно рекомендуется использовать синхронный сервис, потому что достать из кэша — это быстро. Если вы точно знаете, что вам нужен асинхронный сервис, который будет

делать inference в субпроцессе, то будьте готовы, что с ростом нагрузки, скорее всего, придётся агрессивно горизонтально масштабировать сервис, чтобы он выжил.

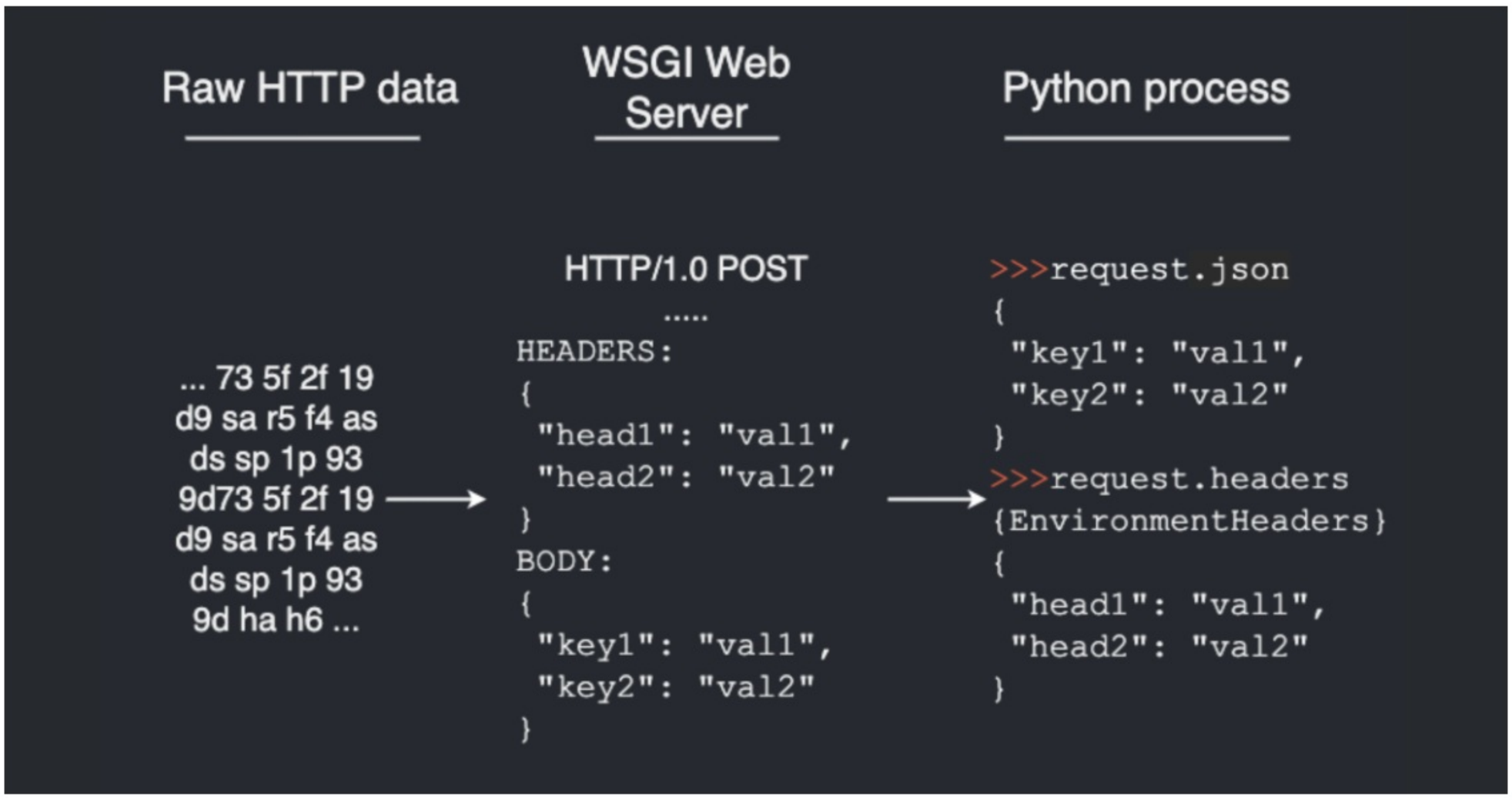
> Веб-серверы

И последняя тема на сегодня — веб-серверы.

Наше Flask-приложение — это application service, в нём содержится только логика приложения, но это приложение не предназначено для приёма запросов в проде, поэтому когда вы запускате development service Flask'a, то видите предупреждение `Flask uses development server - don't use it in production`. Предлагаем разобраться, зачем нам production web-server и какой стек веб-серверов нужно поставить перед нашим приложением.

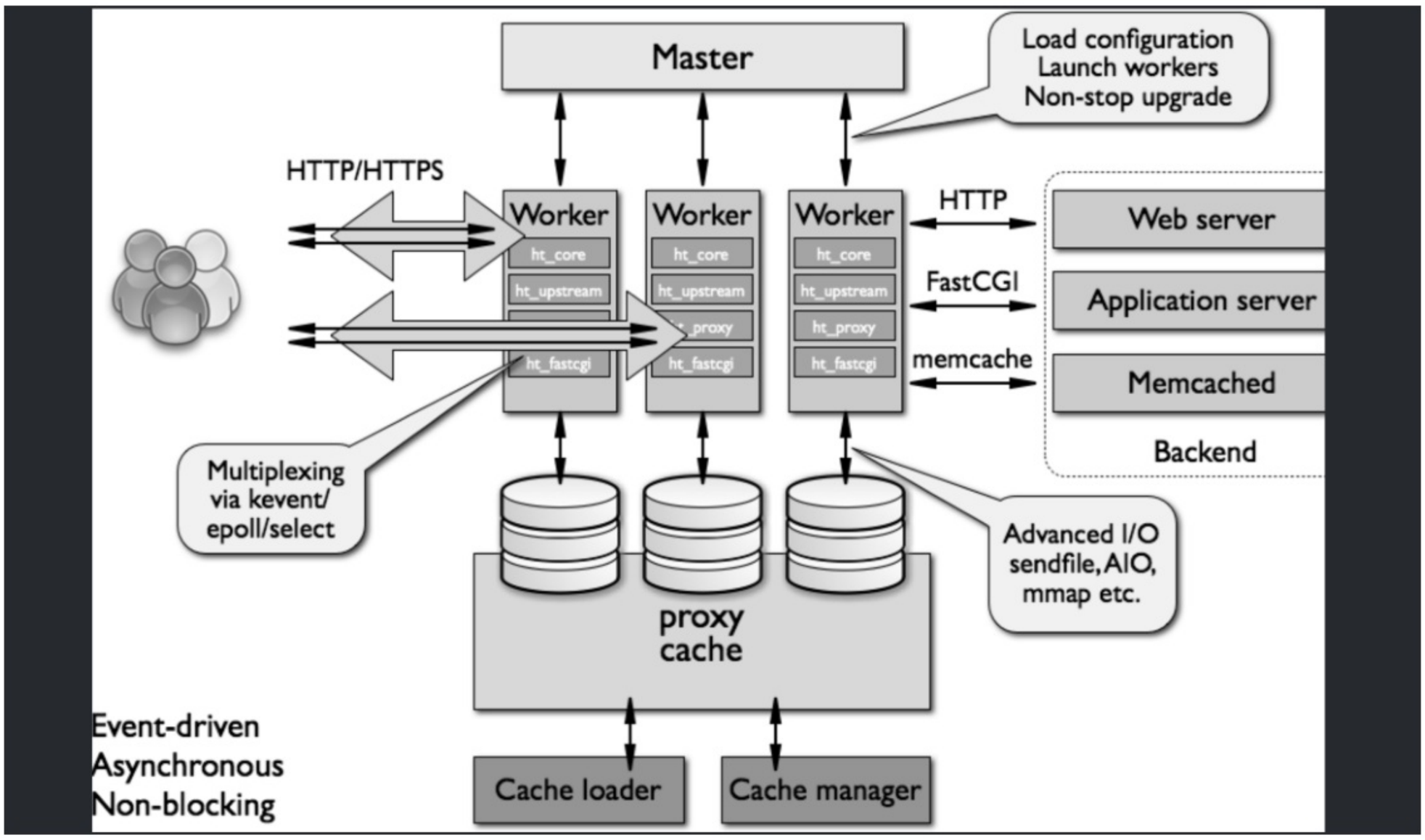
> WSGI, CGI

Итак, HTTP-запрос — это набор байтов, и чтобы в приложении получилась удобная структура request, с которой можно работать, нужно, чтобы эти байты были распарсены. После парсинга получаются заголовки, тело и всё то, с чем вы работаете в request. Development сервисы фреймворков, конечно, справляются с этим, но не так эффективно, как веб-сервера, написанные, например, на C специально для этого. При большой нагрузке обработка запросов с веб-сервером происходит быстрее, чем без него. После того, как всё распарсено, веб-сервер передаёт эти данные в процесс application сервиса. Чтобы он смог понять веб-сервер, данные должны передаваться в формате, известном и тому, и другому. Один из таких форматов и называется WSGI. Вообще формат включает в себя больше — например то, как должны выглядеть handler'ы запросов (т.е. представления или вьюхи), которые мы пишем, но мы не будем останавливаться на этом. Есть и другие стандарты, например CGI, который поддерживает NGINX, но для Python специфичен WSGI, поэтому для его фреймворков нужны WSGI-совместимые веб-сервера.



> Архитектура веб-сервера

Теперь давайте посмотрим на архитектуру веб-сервера.



Это архитектурная схема веб-сервера, взятая из документации NGINX, но архитектурно веб-серверы схожи, поэтому uWSGI или Gunicorn устроены приблизительно схожим образом. Есть мастер-процесс, который поднимается в

первую очередь. Затем с помощью мастера поднимаются worker'ы, которые и будут обрабатывать нагрузку. В конце поднимается балансировщик нагрузки.

Например, для uWSGI вы увидите при запуске что-то подобное:

```
spawned uWSGI master process (pid: 45811)
spawned uWSGI worker 1 (pid: 45813, cores: 1)
spawned uWSGI http 1 (pid: 45814)
```

Именно этот `http 1` отвечает за балансировку между worker'ами. Т.е. master-процесс ничего не знает про запросы — он просто управляет worker'ами, запускает/переподнимает их и т.д. Поднятие worker'ов при старте веб-сервиса называется prefork моделью, и сейчас многие сервера работают таким образом.

> Зачем вообще нужны веб-сервера для ML

На самом деле если говорить про простые ML кейсы с маленькой нагрузкой, то ничего страшного не случится, если приложение будет работать без веб-сервера. Если нагрузка возрастает, то быстрота парсинга запросов будет почти единственным аргументом для установки веб-сервера. Если нагрузка на сервис растет, то вырисовывается еще одна причина, которой мы касались, когда говорили про треды в питоне: если вы хотите, чтобы запросы обрабатывались максимально быстро, то, возможно, стоит ограничивать количество тредов для каждого Python процесса, и веб-сервера с этим помогут.

Количество одновременных запросов, которое должен выдерживать сервис, должно равняться общему количеству тредов на всех репликах — от этого никуда не деться, поэтому вам предстоит для себя подобрать оптимальную пару параметров (количество процессов и количество тредов в каждом процессе). Будет это 5 реплик по 60 тредов или 10 реплик по 30 тредов — лучшее значение вы подбираете в результате экспериментов с нагрузкой. Есть дефолтная рекомендация по количеству worker'ов: `n_worker = 2 * n_cores`, но она устарела с тех времён, когда worker'ы могли обрабатывать только один запрос в один момент времени.

Есть ещё одна причина, по которой вам может понадобиться веб-сервер. Если вы хотите быстро масштабироваться, то воркеры uWSGI помогут сделать это реально быстро. Архитектура веб-серверов с master'ом позволяет делать тот же graceful reload и некоторые другие фичи, что, безусловно, полезно. Но для тех развёрток, которые мы попробуем, эти функции веб-серверов нам не понадобятся, поэтому на этом останавливаться мы не будем.

Также веб-серверы занимаются кешированием, сервингом статики, шифрованием, но для нас это нерелевантно.

> Нужен ли NGINX?

Перед uWSGI часто советуют ставить NGINX, но WSGI-сервера мало чем от него отличаются (за исключением некоторых фичей NGINX). Вопрос: тогда зачем нужен NGINX?

uWSGU и Gunicorn — это по сути полноценные WSGI: у NGINX CGI-протокол, поэтому он не может управлять Python напрямую, как этот делают Gunicorn и uWSGU. Можно сделать почти всё что угодно и превратить питоновский uWSGI в CGI, чтобы можно было использовать NGINX, но так не делают.

Некоторые WSGI-сервера поддерживают CGI-протокол, что в принципе делает возможным запуск uWSGI-воркеров NGINX'ом и позволяет worker'ам принимать запросы CGI-протоколом. Но это вопрос целесообразности.

Если WSGI-сервер справляется со своими функциями веб-сервера, то NGINX нам нужен разве что в качестве балансировщика над вашими несколькими WSGI-экземплярами, а балансировка трафика будет проходить по HTTP-, а не CGI- или WSGI-протоколу. Возможно, поэтому и нет серьёзной причины ставить NGINX перед WSGI-сервером. В некоторых сценариях развёртки NGINX может стоять перед несколькими uWSGI в качестве балансировщика трафика, но

функции прокси веб-сервера, а именно функции веб-сервера, использоваться не будут.

> Дополнительные материалы

- Про HTOP
- Способ создать в питоне процессы-сироты: stackoverflow, документация
- Создание зомби-процессов, убивая потомка: stackoverflow
- Немного про процессы в Linux
- Видео про Systemd: видео 1, видео 2, видео 3
- Про GIL
- Про Application-серверы
- Про советы и трюки с uWSGI