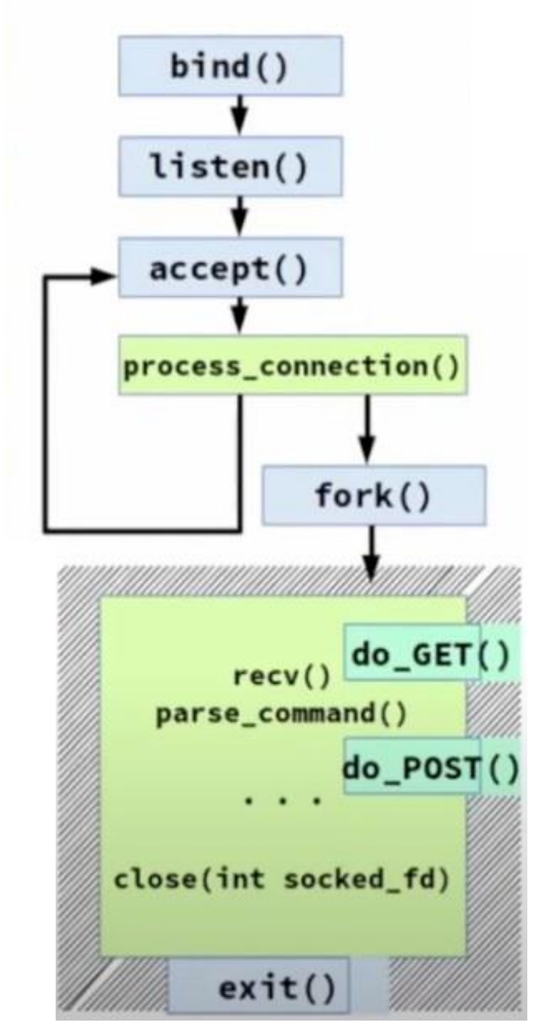
АКОС 11

Сети 2. Мультиплексирование

Проблема

- Хотим обрабатывать множество клиентов, но ждём на
  - accept
  - read
  - write
  - send
  - recv

Процессы



- Нет блокировок
- Но есть оверхед:
  - Создание процесса
  - Выделение памяти
  - Межпроцессное взаимодействие

Сервер

Apache

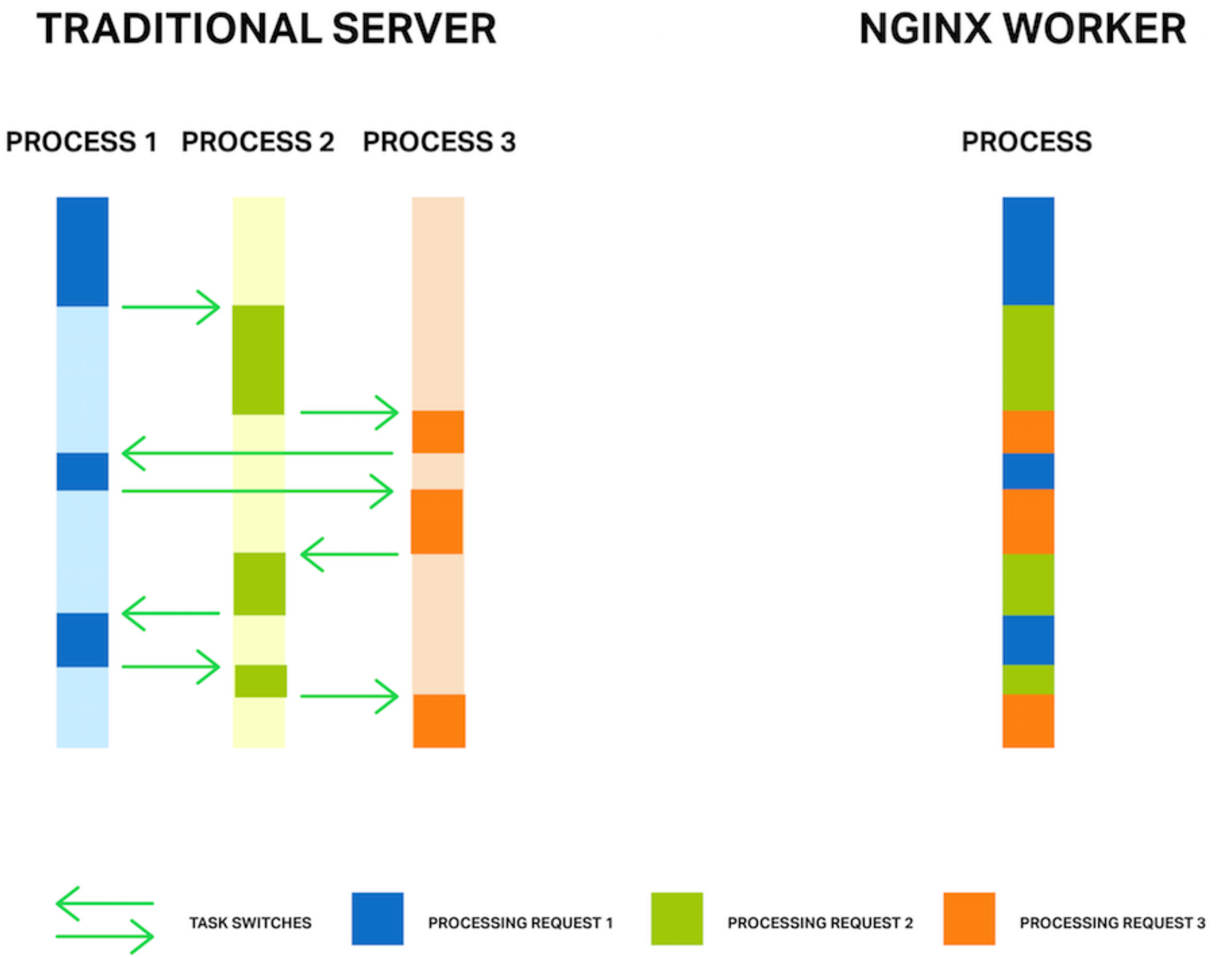
- На каждое соединение создаётся процесс / поток
- Ведёт к трате времени и CPU на создание кучи, стека, на switch-context
- Плохо маштабируется

Nginx

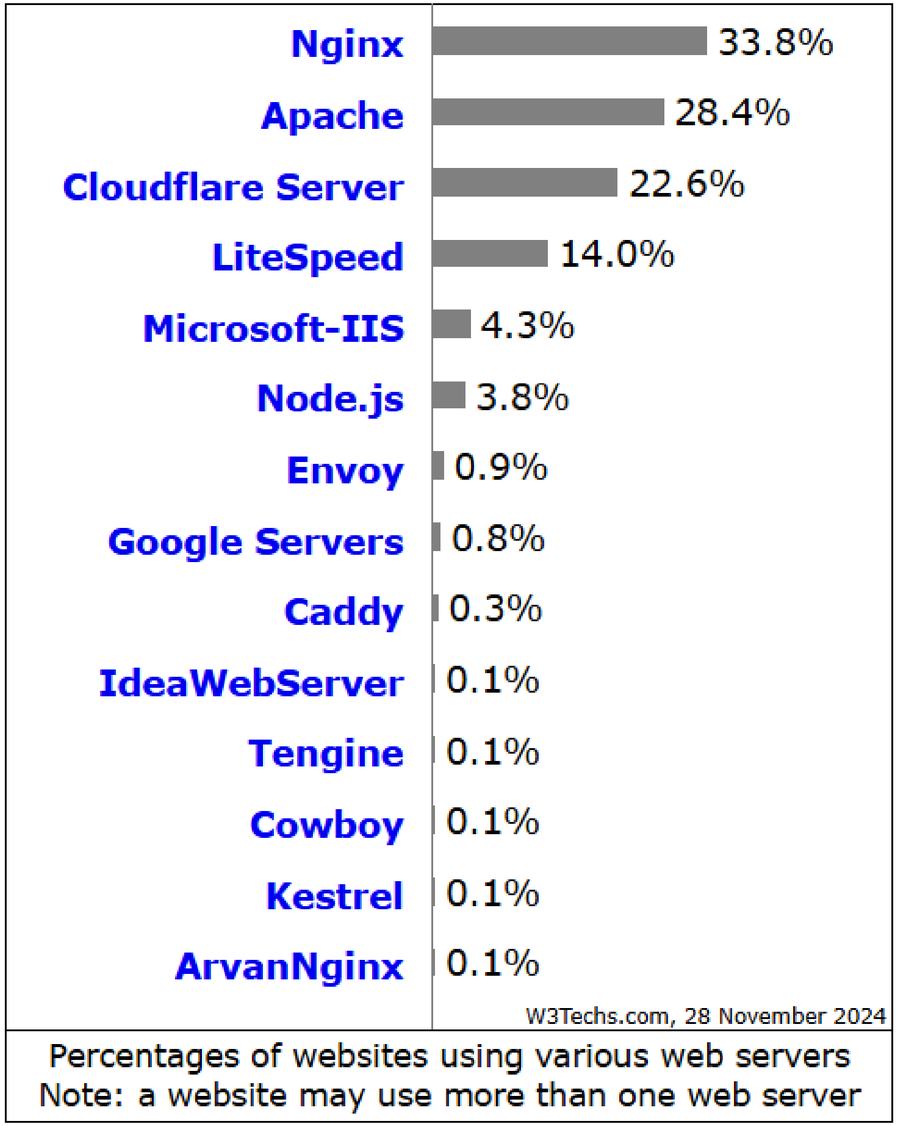
- Event-based
- Asynchronous
- Single-threaded
- Non-blocking
- Multiplexing and event-notification
- A limited number of single-threaded processes called workers. Within each worker nginx can handle many thousands of concurrent connections and requests per second

Nginx architecture

Request processing



Статистика


Nginx 33.8%
Apache 28.4%
Cloudflare Server 22.6%
LiteSpeed 14.0%
Microsoft-IIS 4.3%
Node.js 3.8%
Envoy 0.9%
Google Servers 0.8%
Caddy 0.3%
IdeaWebServer 0.1%
Tengine 0.1%
Cowboy 0.1%
Kestrel 0.1%
ArvanNginx 0.1%
W3Techs.com, 28 November 2024
Percentages of websites using various web servers
Note: a website may use more than one web server

Управление файловыми дескрипторами

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ ); // File CoNTroL
```

- **fd** – кайловый дескриптор
- **cmd** – команда
- **arg** – опциональные аргументы для конкретной команды

Установка флагов

- fcntl(int fd, **F_GETFL**) -> flags
- fcntl(int fd, **F_SETFL**, **flags**)
- Полезные флаги:
  - O_CREAT
  - O_APPEND
  - O_RDONLY
  - **O_NONBLOCK**
- Флаги можно установить при создании (open)
- Можно установить флаги уже готовому файловому дескриптору (от вызова accept)

Проблемы работы с блокирующей записью

- Если недостаточно RAM, то произойдёт блокировка
- pipe: ограничение на размер (по умолчанию 64KB)
- TCP-socket: тоже ограничение на размер
  $ cat /proc/sys/net/ipv4/tcp_wmem (min/default/max)

Проблемы с блокирующимся чтением

- Нет данных в pipe или socket
- Процесс переходит в режим ожидания

O_NONBLOCK

- Процесс не переходит в режим ожидания
- Сразу возвращается -1
- errno = EAGAIN, что не является ошибкой

- Можем повторить read/write позже
- accept - нет текущих подключений
- Для файлов не работает, но есть IO_URING

Non-Blocking Serving

- Работаем с одним клиентом, иногда переключаемся на другого если нужно
- Постоянно крутимся => потребляем много ресурсов
- Можно сделать sleep (но сколько ждать?)

Kernel-Side Events

- Дескриптор готов к записи:
  - Есть свободное место в буфере
- Дескриптор готов к чтению:
  - Появились данные в пайпе/сокете
  - Новое подключение
  - EOF / разорвано соединение
- Достижение тайм-аута (если он выставлен)
- Необработанный сигнал

select (BSD) / poll (UNIX)

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
        fd_set *exceptfds, struct timeval *timeout);

#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

- Ожидание хотя бы одного события
- Возвращает количество дескрипторов для проверки

Select vs Poll

Select:

- Bitmaps of fixed size (FD_SETSIZE, usually 1,024)
- user applications should refill the interest sets for every call
- Kernel scans the entire bitmaps for every call to find descriptors of interest
- Scan the entire bitmap for every call to find ready descriptors

Poll:

- Uses an array of fd instead of bitmaps
- Separate fields for ready and of interest
- Kernel still scans all fd

10k problem (C10k)

- C10k - 10k connections - проблема 10 тысяч соединений
- Уже актуально C10M
- select / poll работают за **O(N)**
- **N** - количество отлеживаемых, но не готовых дескрипторов

FreeBSD Kernel Queue / UNIX epoll

- #include <sys/event.h>
- Новые события кладутся в очередь
- Позволяет процессу брать события из очереди
- **O(N)**, где **N** - количество произошедших событий

EPOLL_CREATE

Шаг 1

```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_create1(int flags);
```

- Создаёт очередь
- Параметр size - deprecated. Раньше указывал на размер очереди, сейчас не используется, но для обратной совместимости должен быть не 0
- flags - обычно 0, тогда вызов аналогичен epoll_create

EPOLL_EVENT

Шаг 2

```
typedef union epoll_data {
    void        *ptr;
    int         fd;
    uint32_t    u32;
    uint64_t    u64;
} epoll_data_t;

struct epoll_event {
    uint32_t    events;      /* Epoll events */
    epoll_data_t data;       /* User data variable */
};
```

- **epoll_data_t -** то, что вернёт ядро, например, event.data.fd = fd
- **events -** битовая маска из:
  - **EPOLLIN**
  - **EPOLLOUT**
  - **...**

EPOLL_CTL

Шаг 3

```
#include <sys/epoll.h>
```

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

- Действие с epoll
- **epfd** - дескриптор очереди
- **op** - действие над очередью
  - `EPOLL_CTL_ADD - добавить дескриптор в interest list`
  - `EPOLL_CTL_MOD - изменить настройки`
  - `EPOLL_CTL_DEL - удалить из interest list`
- **event** - уже создали раньше

EPOLL_WAIT

Шаг 4

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

- Дождаться наступления хотя бы одного события
- **epfd** - дескриптор очереди
- **maxevents** - максимально количество произошедших событий
- **timeout** - время ожидания в миллисекундах

Epoll vs Kqueue

Epoll:

- Linux
- Epoll_ctl() работает только с одним дексриптором за раз
- Только файловые дескрипторы
- Не работает со всеми дескрипторами, а именно с “обычными” файловыми дескрипторами

Kqueue:

- FreeBSD, macOS
- Может работать с большим количеством файловых дескрипторов за раз
- Работает с любыми объектами, даже с сигналами и таймерами
- Умеет работать с диском

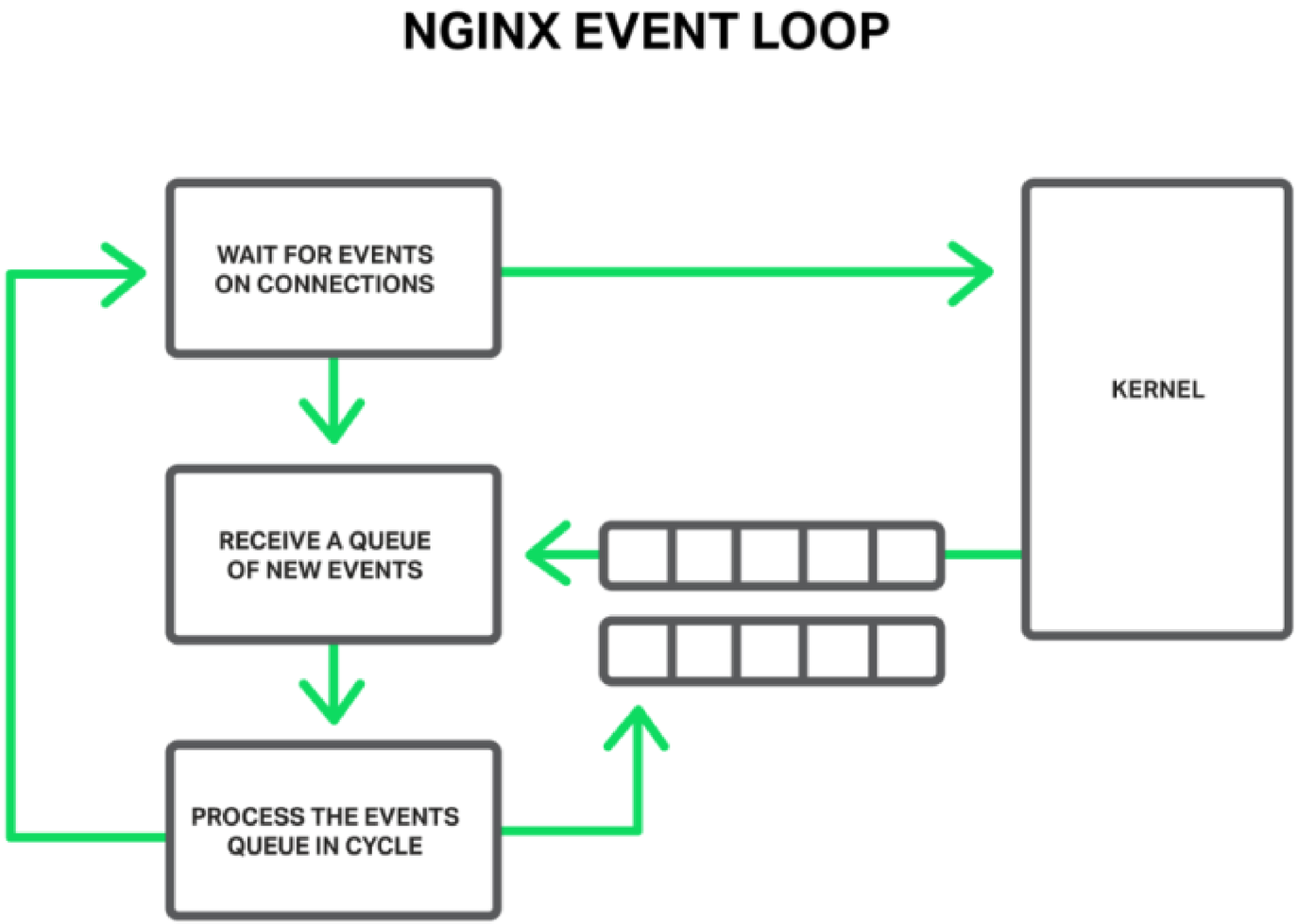
Статья

Nginx run-loop

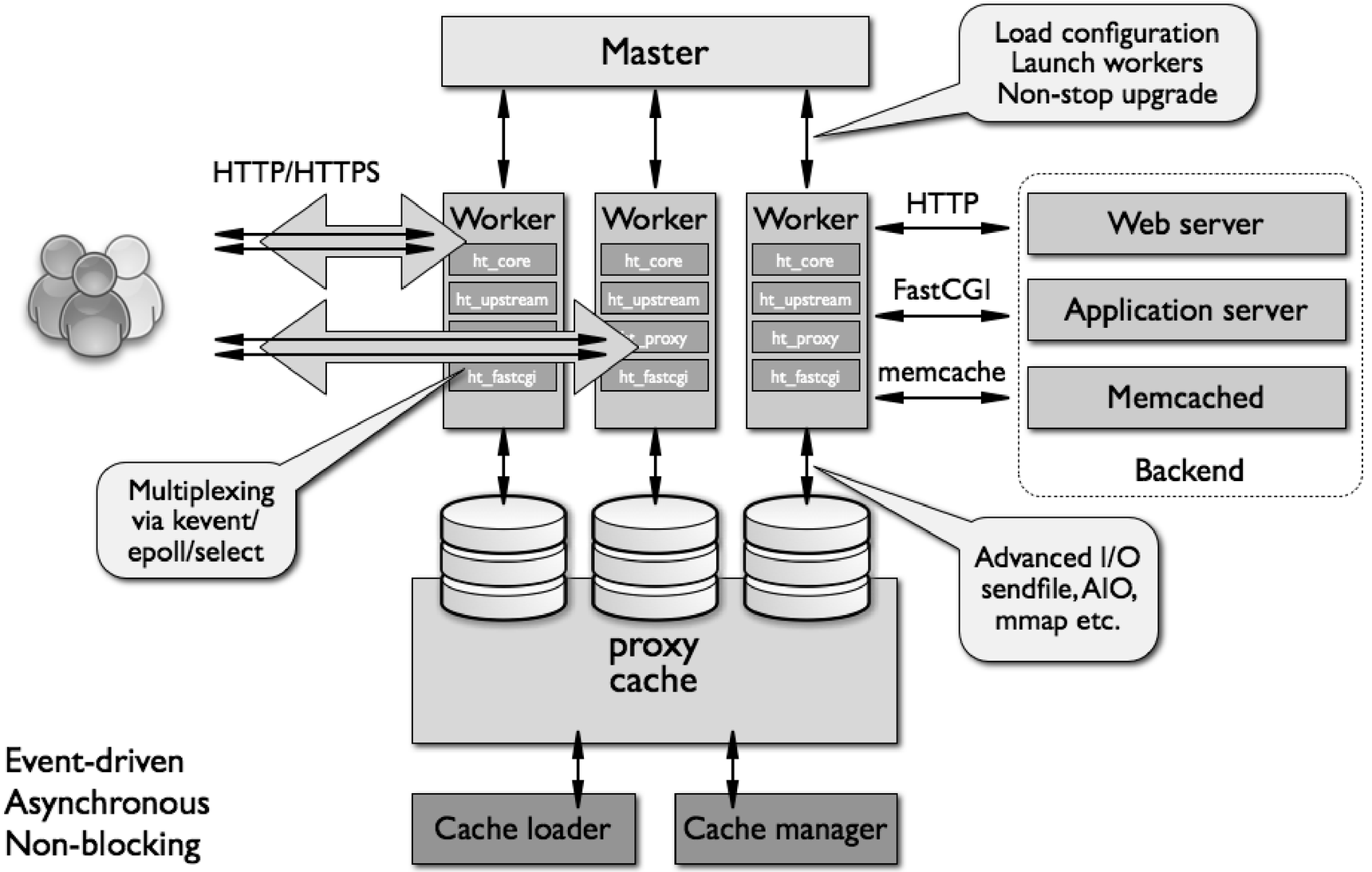
Inside a worker, the sequence of actions leading to the run-loop where the response is generated looks like the following:

1. Begin ngx_worker_process_cycle().
2. Process events with OS specific mechanisms (such as epoll or kqueue).
3. Accept events and dispatch the relevant actions.
4. Process/proxy request header and body.
5. Generate response content (header, body) and stream it to the client.
6. Finalize request.
7. Re-initialize timers and events.

Nginx event-loop



Nginx architecture



- Event-driven
- Asynchronous
- Non-blocking

ASIO

Асинхронное апи поверх epoll. Устанавливаются коллбэки на события.

Установка:
$ sudo apt-get install -y libasio-dev

Пример

Что почитать

Про Epoll с картинками и объяснениями