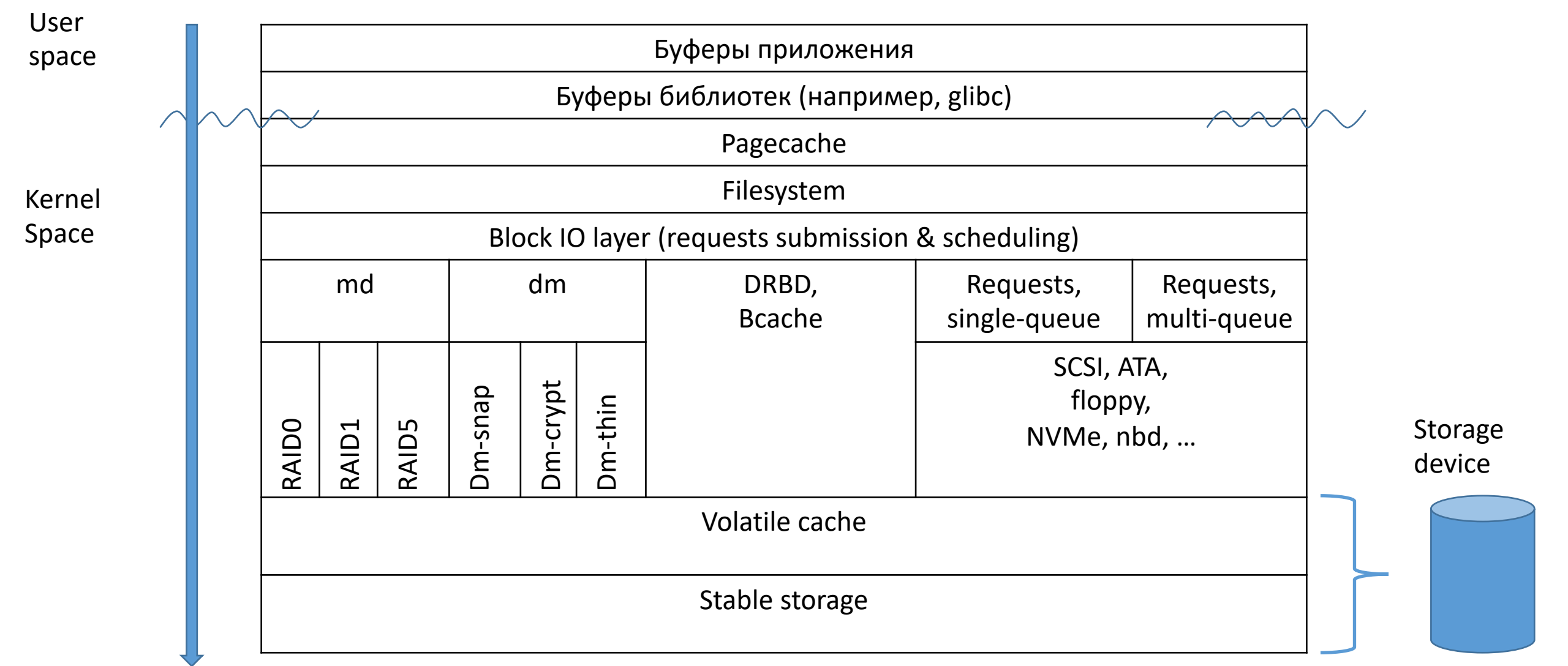


# Основы построения файловых систем

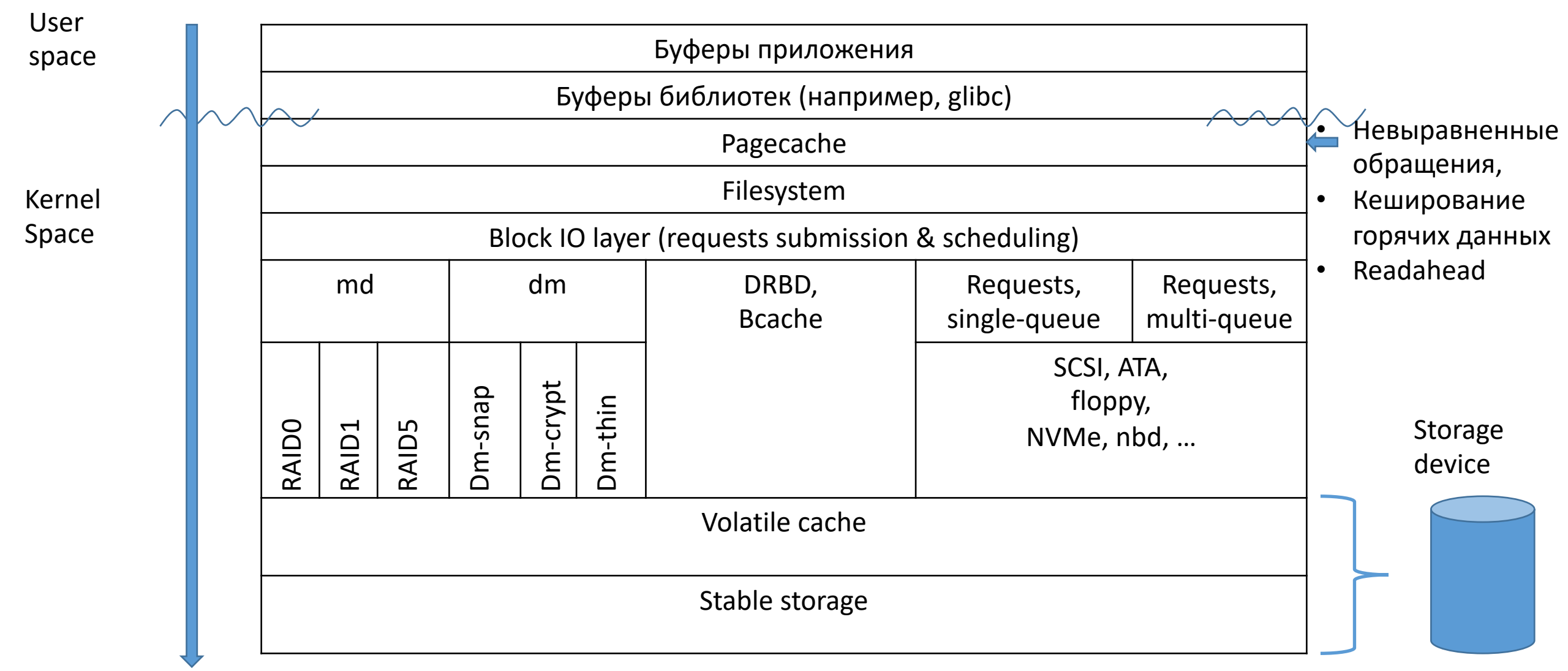


# Путь данных от приложения до диска (обзорно)

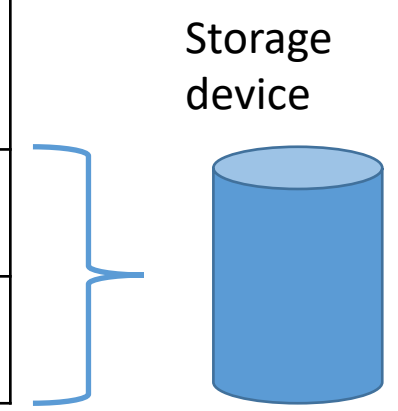


\* Ensuring data reaches disk: <https://lwn.net/Articles/457667/>

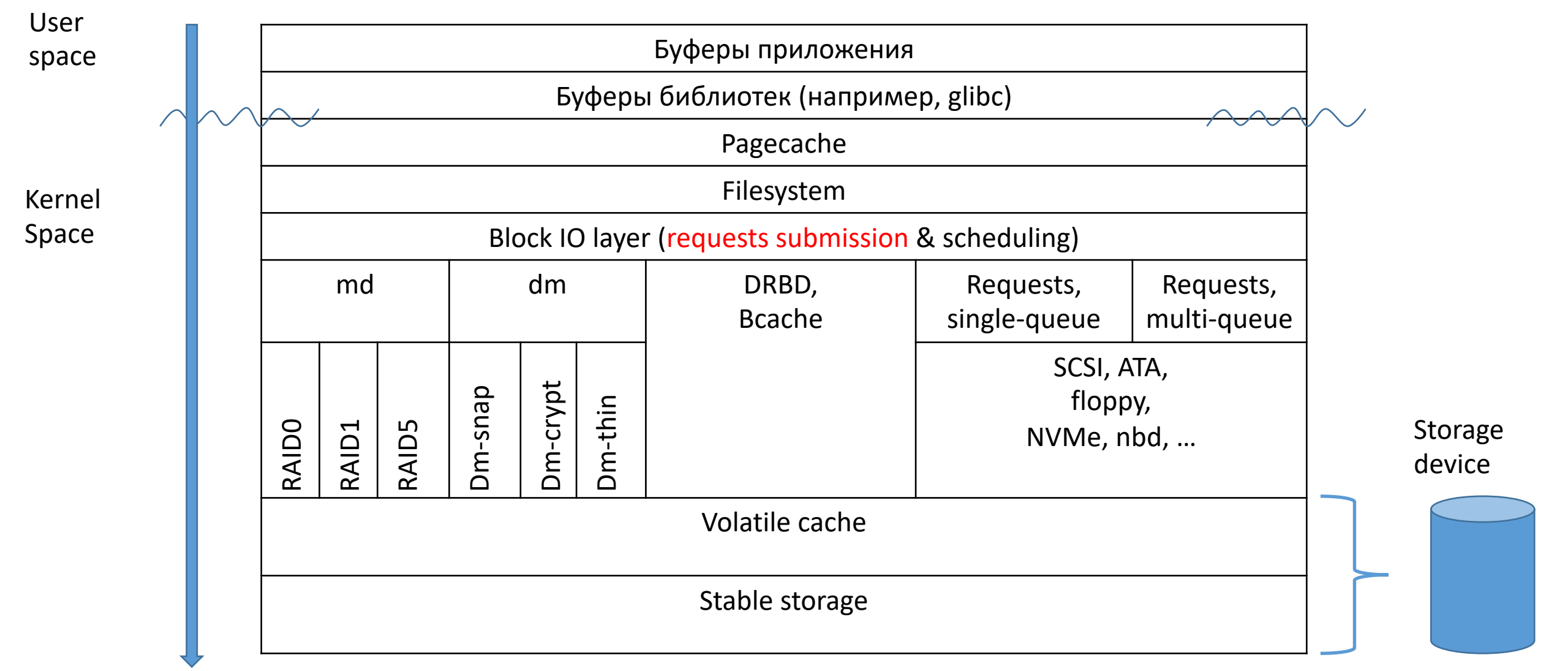
# Путь данных от приложения до диска (обзорно)



- Невыравненные обращения, (misaligned accesses)
- Кеширование горячих данных (caching of hot data)
- Readahead



# Путь данных от приложения до диска (обзорно)

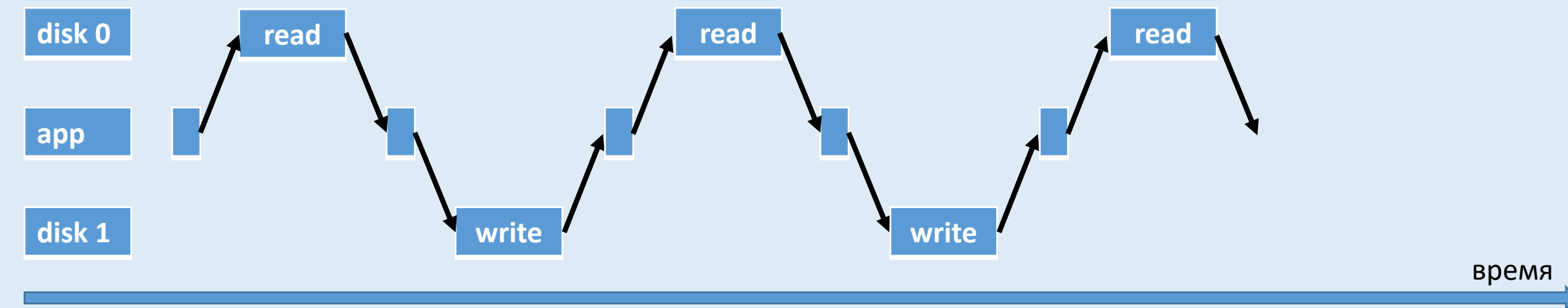


# Очереди IO

Рассмотрим наивное копирование файла с одного диска на другой:

```
while (!done) {  
    r = read(fd_in, buf, sizeof(buf));  
    r0 = write(fd_out, buf, r);  
    ...  
}
```

Как расположены во времени обращения к дискам?

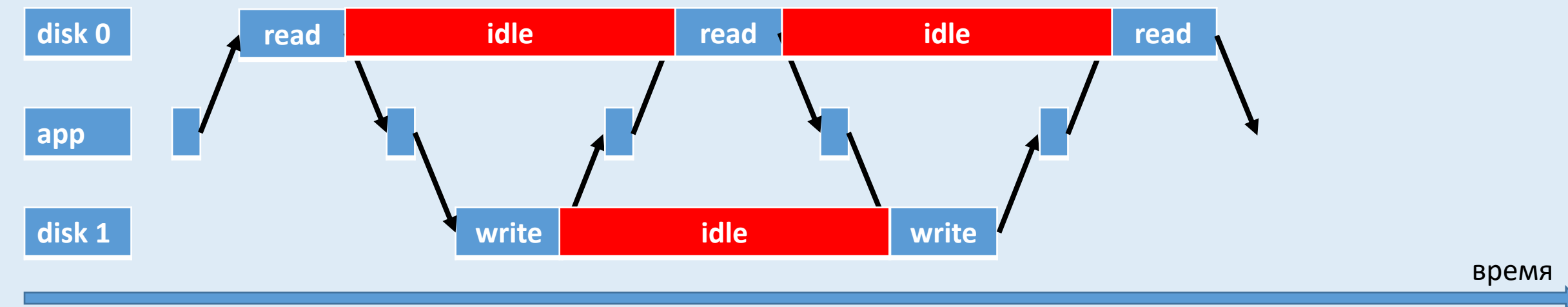


# Очереди IO

Рассмотрим наивное копирование файла с одного диска на другой:

```
while (!done) {  
    r = read(fd_in, buf, sizeof(buf));  
    r0 = write(fd_out, buf, r);  
    ...  
}
```

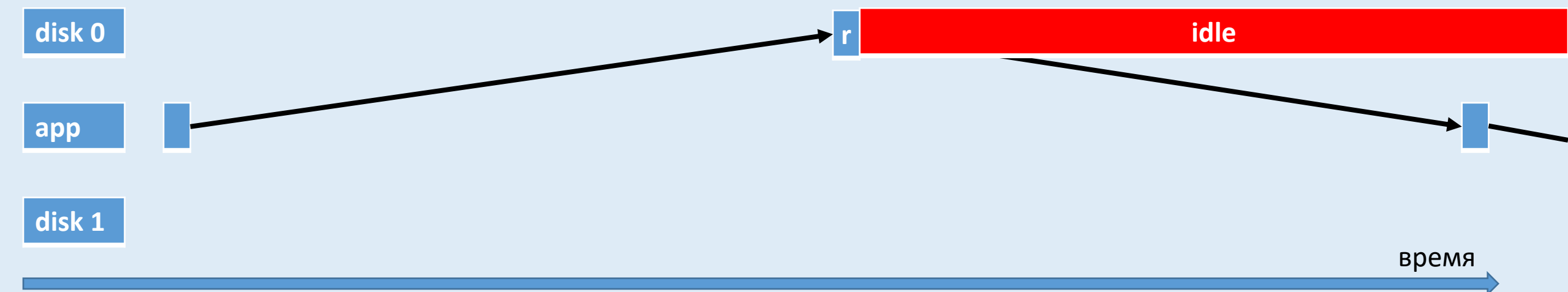
Как расположены во времени обращения к дискам?



## Очереди IO

Обычно проблема хуже. Если ФС, между которыми делается копирование, сетевые, или расположены на быстрых NVMe-устройствах, то картина будет выглядеть так:

```
while (!done) {  
    r = read(fd_in, buf, sizeof(buf));  
    r0 = write(fd_out, buf, r);  
    ...  
}
```



## Очереди IO: пример

Обычно проблема хуже. Если ФС, между которыми делается копирование, сетевые, или расположены на быстрых NVMe-устройствах, то картина будет выглядеть так:

```
while (!done) {  
    r = read(fd_in, buf, sizeof(buf));  
    r0 = write(fd_out, buf, r);  
    ...  
}
```

```
21-02-18 23:40:38.936 s#1412709.r#6998305: readfile = {offset = 0x4c44d78350, length = 16}  
21-02-18 23:40:39.191 s#1412709.r#6998305: send 16 at offset 0x4c44d78350  
21-02-18 23:40:39.191 s#1412709.r#6998305: completed
```

```
21-02-18 23:40:39.757 s#1412709.r#6998344: readfile = {offset = 0x4c44d78360, length = 944}  
21-02-18 23:40:39.757 s#1412709.r#6998344: send 944 at offset 0x4c44d78360  
21-02-18 23:40:39.757 s#1412709.r#6998344: completed
```

```
21-02-18 23:40:40.242 s#1412709.r#6998358: readfile = {offset = 0x4c44d7e360, length = 16}  
21-02-18 23:40:40.361 s#1412709.r#6998358: send 16 at offset 0x4c44d7e360  
21-02-18 23:40:40.361 s#1412709.r#6998358: completed
```



## Очереди IO: пример

Обычно проблема хуже. Если ФС, между которыми делается копирование, сетевые, или расположены на быстрых NVMe-устройствах, то картина будет выглядеть так:

```
while (!done) {  
    r = read(fd_in, buf, sizeof(buf));  
    r0 = write(fd_out, buf, r);  
    ...  
}
```

```
21-02-18 23:40:38.936 s#1412709.r#6998305: readfile = {offset = 0x4c44d78350, length = 16}  
21-02-18 23:40:39.191 s#1412709.r#6998305: send 16 at offset 0x4c44d78350  
21-02-18 23:40:39.191 s#1412709.r#6998305: completed
```

```
21-02-18 23:40:39.757 s#1412709.r#6998344: readfile = {offset = 0x4c44d78360, length = 944}  
21-02-18 23:40:39.757 s#1412709.r#6998344: send 944 at offset 0x4c44d78360  
21-02-18 23:40:39.757 s#1412709.r#6998344: completed
```


```
21-02-18 23:40:40.242 s#1412709.r#6998358: readfile = {offset = 0x4c44d7e360, length = 16}  
21-02-18 23:40:40.361 s#1412709.r#6998358: send 16 at offset 0x4c44d7e360  
21-02-18 23:40:40.361 s#1412709.r#6998358: completed
```

## Очереди IO: пример

Обычно проблема хуже. Если ФС, между которыми делается копирование, сетевые, или расположены на быстрых NVMe-устройствах, то картина будет выглядеть так:

```
while (!done) {  
    r = read(fd_in, buf, sizeof(buf));  
    r0 = write(fd_out, buf, r);  
    ...  
}
```

```
21-02-18 23:40:38.936 s#1412709.r#6998305: readfile = {offset = 0x4c44d78350, length = 16}  
21-02-18 23:40:39.191 s#1412709.r#6998305: send 16 at offset 0x4c44d78350  
21-02-18 23:40:39.191 s#1412709.r#6998305: completed  
21-02-18 23:40:39.757 s#1412709.r#6998344: readfile = {offset = 0x4c44d78360, length = 944}  
21-02-18 23:40:39.757 s#1412709.r#6998344: send 944 at offset 0x4c44d78360  
21-02-18 23:40:39.757 s#1412709.r#6998344: completed  
21-02-18 23:40:40.242 s#1412709.r#6998358: readfile = {offset = 0x4c44d7e360, length = 16}  
21-02-18 23:40:40.361 s#1412709.r#6998358: send 16 at offset 0x4c44d7e360  
21-02-18 23:40:40.361 s#1412709.r#6998358: completed
```



570ms потрачены впустую

# Очереди IO: пример

Обычно проблема хуже. Если ФС, между которыми делается копирование, сетевые, или расположены на быстрых NVMe-устройствах, то картина будет выглядеть так:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

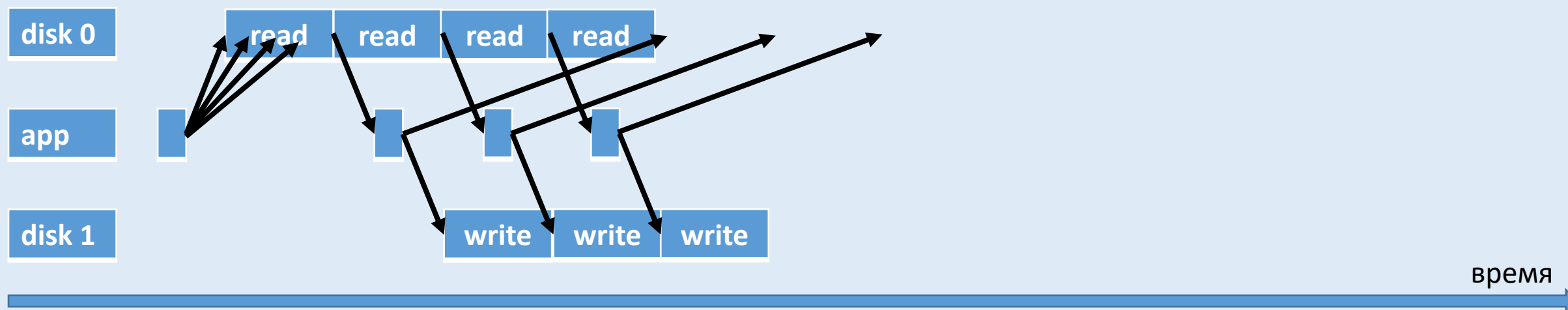
21-02-18 23:40:38.936 s#141270  
21-02-18 23:40:39.191 s#14127  
21-02-18 23:40:39.191 s#14127  
  
21-02-18 23:40:39.757 s#14127  
21-02-18 23:40:39.757 s#14127  
21-02-18 23:40:39.757 s#14127  
  
21-02-18 23:40:40.242 s#14127  
21-02-18 23:40:40.361 s#14127  
21-02-18 23:40:40.361 s#141270

976 байт мы скачали  
примерно за 1.4с.  
  
С какой скоростью  
скачиваются архивы  
из Acronis Data Cloud?

0x4c44d78350, length = 16}  
4c44d78350  
  
0x4c44d78360, length = 944}  
4c44d78360  
  
0x4c44d7e360, length = 16}  
4c44d7e360

## Очереди IO: pipelining

В случае, когда последовательные IO-запросы независимы, простым решением будет **pipelining** запросов: приложение испускает запросы наперёд, чтобы очередь IO у устройств никогда не была пустой. Таким образом мы можем скрыть время, требуемое на отправку запросов.



## Очереди IO: head-of-line blocking

Pipelining имеет следующую проблему: если первый запрос в очереди выполняется неожиданно долго, то все последующие должны его подождать. Даже если последующие запросы могут быть выполнены быстро, сервер должен отправлять ответы в порядке поступления запросов.

## Очереди IO: head-of-line blocking

Pipelining имеет следующую проблему: если первый запрос в очереди исполняется неожиданно долго, то все последующие должны его подождать. Даже если последующие запросы могут быть выполнены быстро, сервер должен отправлять ответы в порядке поступления запросов.

```
06-09-18 14:12:23.567 s#164034.r#66643120: readfile = {offset = 0x39f0d000, length = 524288}
06-09-18 14:12:23.577 s#164034.r#66643125: readfile = {offset = 0x39f8d000, length = 524288}
06-09-18 14:12:23.593 s#164034.r#66643145: readfile = {offset = 0x3a00d000, length = 524288}
06-09-18 14:12:23.604 s#164034.r#66643147: readfile = {offset = 0x3a08d000, length = 524288}
06-09-18 14:12:23.612 s#164034.r#66643147: send 0x3a08d000:524288
06-09-18 14:12:23.612 s#164034.r#66643147: completed
06-09-18 14:12:23.618 s#164034.r#66643154: readfile = {offset = 0x3a10d000, length = 524288}
06-09-18 14:12:23.627 s#164034.r#66643158: readfile = {offset = 0x3a18d000, length = 524288}
06-09-18 14:12:23.632 s#164034.r#66643154: send 0x3a10d000:524288
06-09-18 14:12:23.632 s#164034.r#66643154: completed
06-09-18 14:12:23.634 s#164034.r#66643166: readfile = {offset = 0x3a20d000, length = 524288}
06-09-18 14:12:23.636 s#164034.r#66643158: send 0x3a18d000:524288
06-09-18 14:12:23.636 s#164034.r#66643158: completed
06-09-18 14:12:23.641 s#164034.r#66643168: readfile = {offset = 0x3a28d000, length = 524288}
06-09-18 14:12:23.643 s#164034.r#66643166: send 0x3a20d000:524288
06-09-18 14:12:23.643 s#164034.r#66643166: completed
06-09-18 14:12:23.649 s#164034.r#66643168: send 0x3a28d000:524288
06-09-18 14:12:23.649 s#164034.r#66643168: completed
06-09-18 14:12:23.783 s#164034.r#66643120: send 0x39f0d000:524288
06-09-18 14:12:23.783 s#164034.r#66643120: completed
```

## Очереди IO: head-of-line blocking

Pipelining имеет следующую проблему: если первый запрос в очереди выполняется неожиданно долго, то все последующие должны его подождать. Даже если последующие запросы могут быть выполнены быстро, сервер должен отсылать ответы в порядке поступления запросов.

```
06-09-18 14:12:23.567 s#164034.r#66643120: readfile = {offset = 0x39f0d000, length = 524288}
06-09-18 14:12:23.577 s#164034.r#66643125: readfile = {offset = 0x39f8d000, length = 524288}
06-09-18 14:12:23.593 s#164034.r#66643145: readfile = {offset = 0x3a00d000, length = 524288}
06-09-18 14:12:23.604 s#164034.r#66643147: readfile = {offset = 0x3a08d000, length = 524288}
06-09-18 14:12:23.612 s#164034.r#66643147: send 0x3a08d000:524288
06-09-18 14:12:23.612 s#164034.r#66643147: completed
06-09-18 14:12:23.618 s#164034.r#66643154: readfile = {offset = 0x3a10d000, length = 524288}
06-09-18 14:12:23.627 s#164034.r#66643158: readfile = {offset = 0x3a18d000, length = 524288}
06-09-18 14:12:23.632 s#164034.r#66643154: send 0x3a10d000:524288
06-09-18 14:12:23.632 s#164034.r#66643154: completed
06-09-18 14:12:23.634 s#164034.r#66643166: readfile = {offset = 0x3a20d000, length = 524288}
06-09-18 14:12:23.636 s#164034.r#66643158: send 0x3a18d000:524288
06-09-18 14:12:23.636 s#164034.r#66643158: completed
06-09-18 14:12:23.641 s#164034.r#66643168: readfile = {offset = 0x3a28d000, length = 524288}
06-09-18 14:12:23.643 s#164034.r#66643166: send 0x3a20d000:524288
06-09-18 14:12:23.643 s#164034.r#66643166: completed
06-09-18 14:12:23.649 s#164034.r#66643168: send 0x3a28d000:524288
06-09-18 14:12:23.649 s#164034.r#66643168: completed
06-09-18 14:12:23.783 s#164034.r#66643120: send 0x39f0d000:524288
06-09-18 14:12:23.783 s#164034.r#66643120: completed
```

# Очереди IO: head-of-line blocking

Pipelining имеет следующую проблему: если первый запрос в очереди выполняется неожиданно долго, то все последующие должны его подождать. Даже если последующие запросы могут быть выполнены быстро, сервер должен отправлять ответы в порядке поступления запросов.

```
06-09-18 14:12:23.567 s#164034.r#66643120: readfile = {offset = 0x39f0d000, length = 524288}
06-09-18 14:12:23.577 s#164034.r#66643125: readfile = {offset = 0x39f8d000, length = 524288}
06-09-18 14:12:23.593 s#164034.r#66643145: readfile = {offset = 0x3a00d000, length = 524288}
06-09-18 14:12:23.604 s#164034.r#66643147: readfile = {offset = 0x3a08d000, length = 524288}
06-09-18 14:12:23.612 s#164034.r#66643147: send 0x3a08d000:524288
06-09-18 14:12:23.612 s#164034.r#66643147: completed
06-09-18 14:12:23.618 s#164034.r#66643154: readfile = {offset = 0x3a10d000, length = 524288}
06-09-18 14:12:23.627 s#164034.r#66643158: readfile = {offset = 0x3a18d000, length = 524288}
06-09-18 14:12:23.632 s#164034.r#66643154: send 0x3a10d000:524288
06-09-18 14:12:23.632 s#164034.r#66643154: completed
06-09-18 14:12:23.634 s#164034.r#66643166: readfile = {offset = 0x3a20d000, length = 524288}
06-09-18 14:12:23.636 s#164034.r#66643158: send 0x3a18d000:524288
06-09-18 14:12:23.636 s#164034.r#66643158: completed
06-09-18 14:12:23.641 s#164034.r#66643168: readfile = {offset = 0x3a28d000, length = 524288}
06-09-18 14:12:23.643 s#164034.r#66643166: send 0x3a20d000:524288
06-09-18 14:12:23.643 s#164034.r#66643166: completed
06-09-18 14:12:23.649 s#164034.r#66643168: send 0x3a28d000:524288
06-09-18 14:12:23.649 s#164034.r#66643168: completed
06-09-18 14:12:23.783 s#164034.r#66643120: send 0x39f0d000:524288
06-09-18 14:12:23.783 s#164034.r#66643120: completed
```

Запрос r#66643120 попал на диск, занятый другим клиентом.

Запрос r#66643147 исполнился с незанятого диска, но не имеет права отослать готовый ответ вперёд r#66643120.

Из-за описанного поведения скорость Acronis Disaster Recovery Service отличается на порядок на нагруженном и ненагруженном сторедж-кластере.



## Очереди IO: multiplexing

Pipelining имеет следующую проблему: если первый запрос в очереди выполняется неожиданно долго, то все последующие должны его подождать. Даже если последующие запросы могут быть выполнены быстро, сервер должен отправлять ответы в порядке поступления запросов.

Избегать head-of-line blocking можно путём **мультиплексирования** запросов:

- добавим к каждому IO-запросу уникальный номер,
- сервер теперь может посылать ответы в любом порядке, снабдив их номерами.

## Очереди IO: multiplexing

Pipelining имеет следующую проблему: если первый запрос в очереди выполняется неожиданно долго, то все последующие должны его подождать. Даже если последующие запросы могут быть выполнены быстро, сервер должен отправлять ответы в порядке поступления запросов.

Избегать head-of-line blocking можно путём **мультиплексирования** запросов:

- добавим к каждому IO-запросу уникальный номер,
- сервер теперь может посылать ответы в любом порядке, снабдив их номерами.

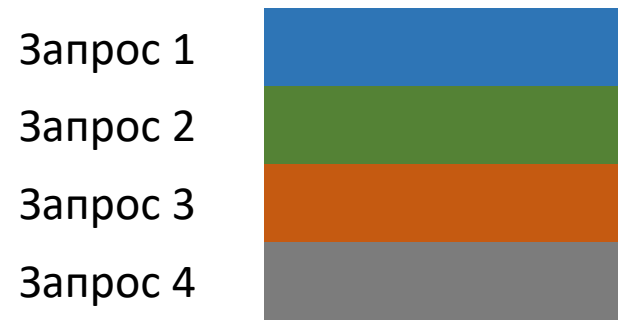
Эта идея используется в:

- SCTP,
- HTTP/2,
- QUIC\*.

\* The QUIC Transport Protocol: Design and Internet-scale Deployment: <https://research.google.com/pubs/archive/46403.pdf>

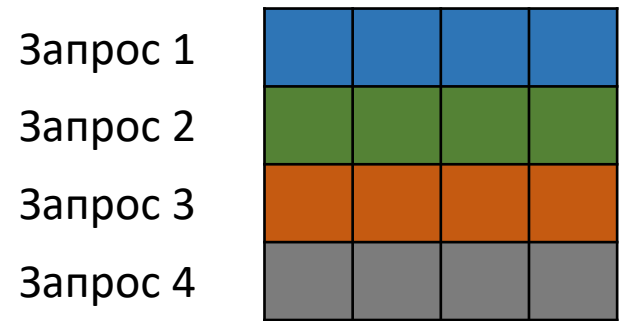
# Взаимодействие pipelining и сети

Клиент, испустивший множество запросов по разным соединениям	Сеть	Очередь запросов на сервере.
--	------	------------------------------



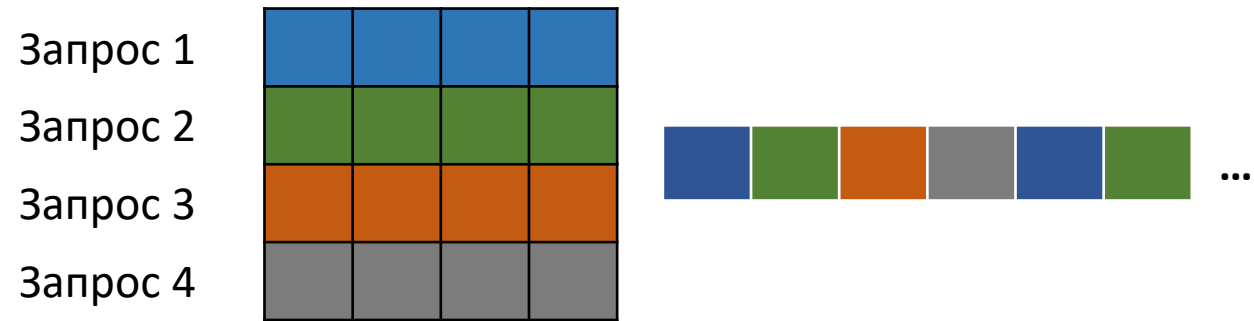
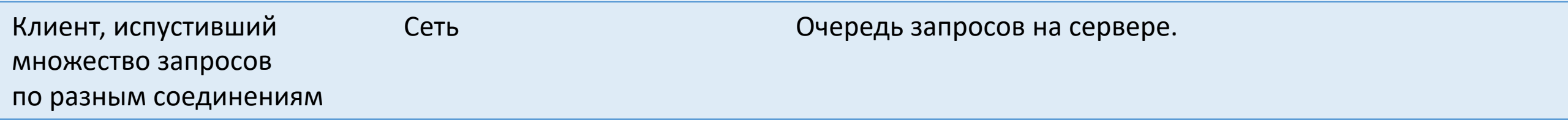
# Взаимодействие pipelining и сети

Клиент, испустивший множество запросов по разным соединениям	Сеть	Очередь запросов на сервере.
--	------	------------------------------



С точки зрения сети, соединения – это независимые потоки байтов и каждому надо дать равную долю канала.

# Взаимодействие pipelining и сети



С точки зрения сети, соединения – это независимые потоки байтов и каждому надо дать равную долю канала.

# Взаимодействие pipelining и сети

Клиент, испустивший множество запросов по разным соединениям      Сеть      Очередь запросов на сервере.

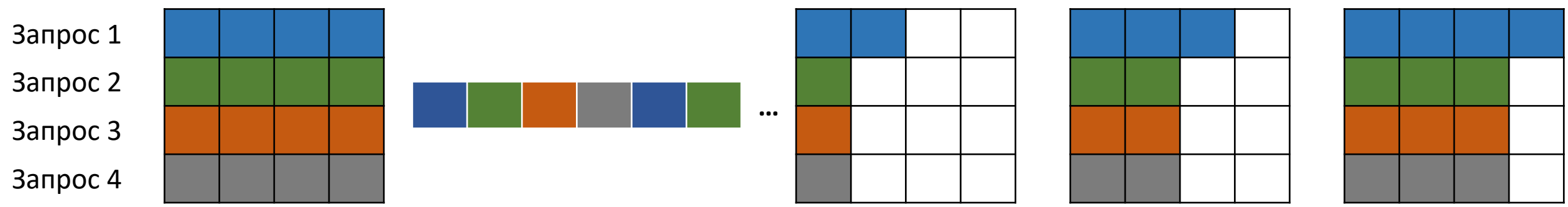


С точки зрения сети, соединения – это независимые потоки байтов и каждому надо дать равную долю канала.

Неполные запросы, например, в gRPC, ещё нельзя исполнять, так как не все их аргументы известны.

# Взаимодействие pipelining и сети

Клиент, испустивший множество запросов по разным соединениям      Сеть      Очередь запросов на сервере.

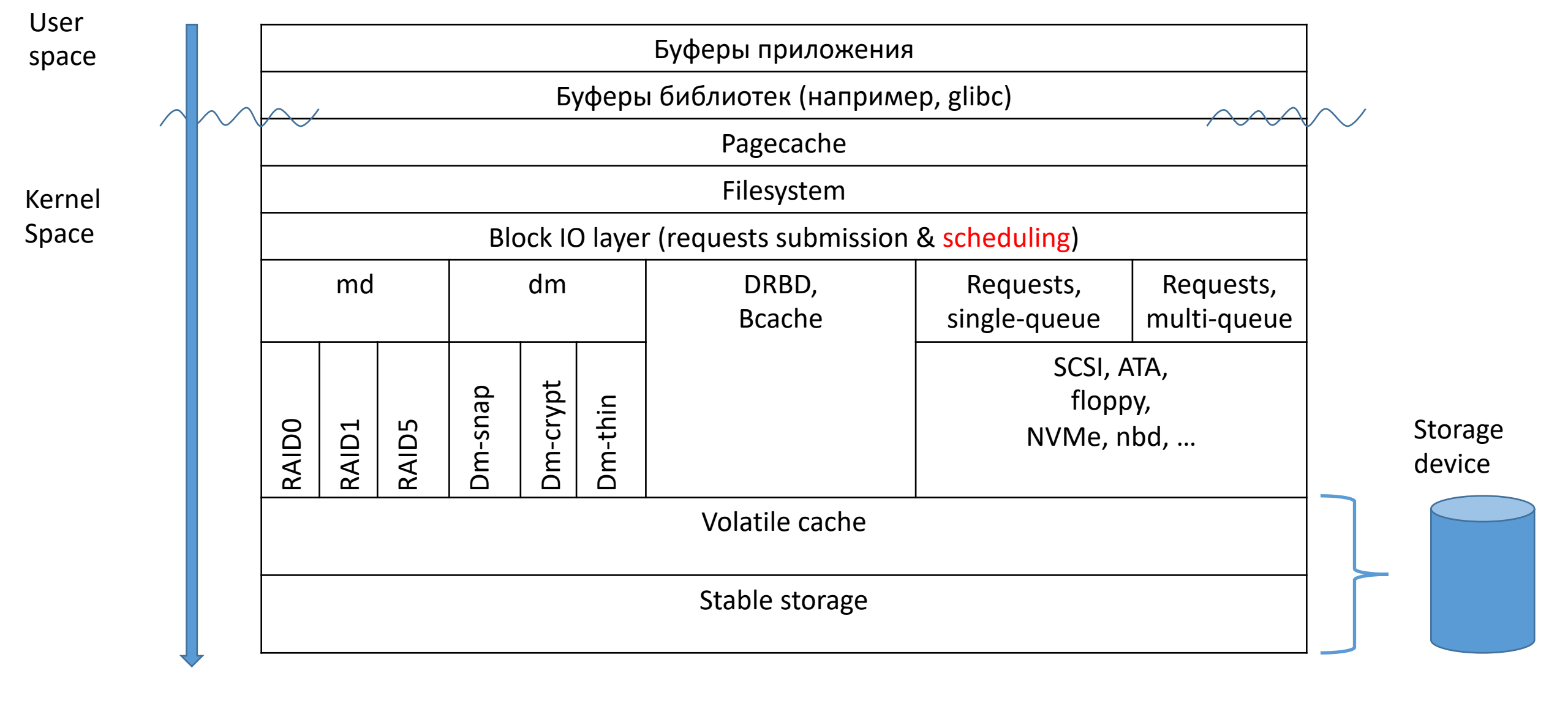


С точки зрения сети, соединения – это независимые потоки байтов и каждому надо дать равную долю канала.

Сервер "получил" запрос только сейчас. По существу, он дожидался получения всех запросов перед тем, как начать исполнять первый.

Ситуация не сильно лучше, чем если бы был один большой запрос.

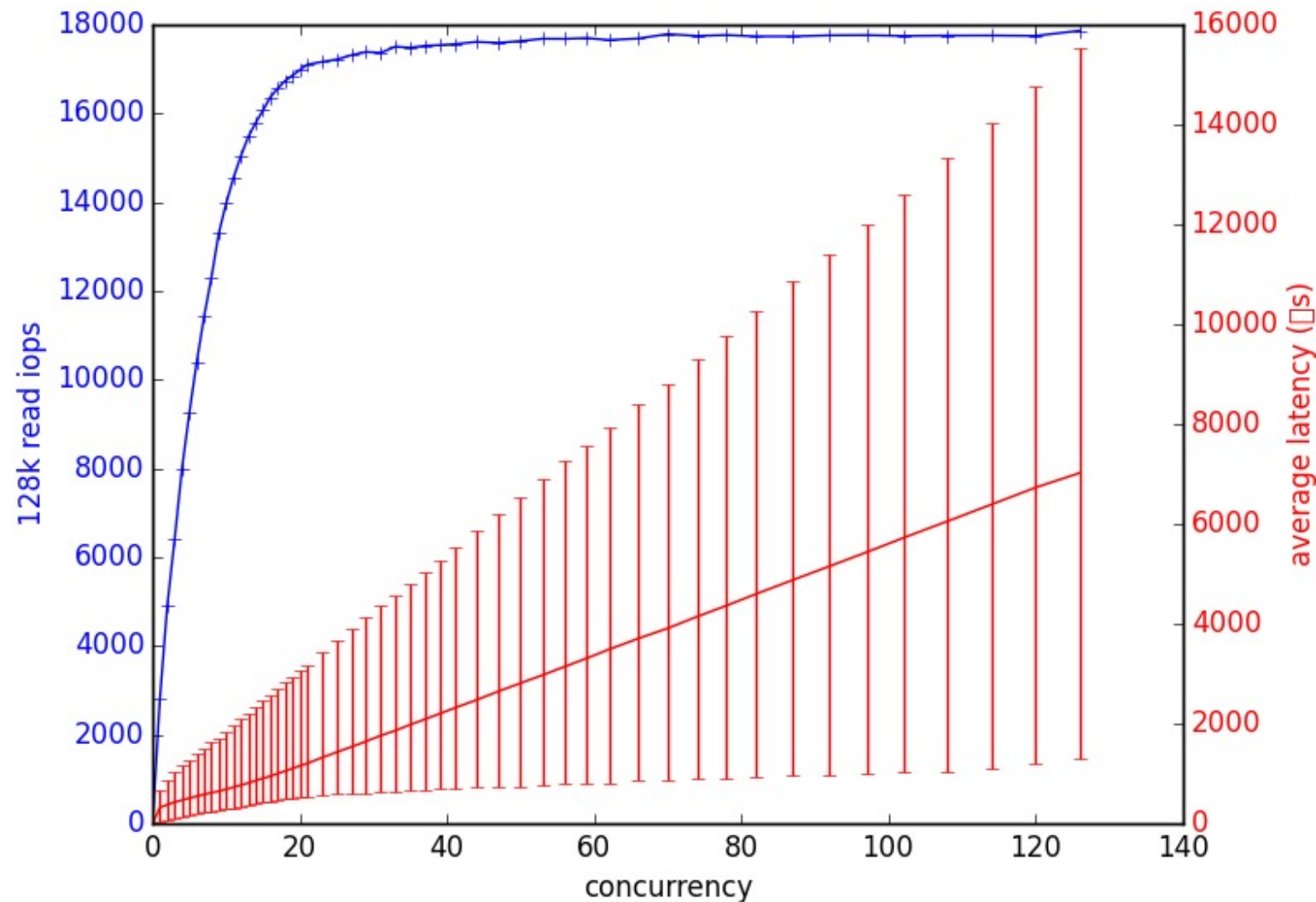
# Путь данных от приложения до диска (обзорно)





# Очереди IO: queueing и scheduling

Время отклика и количество IOPS  
в зависимости от глубины очереди  
запросов:

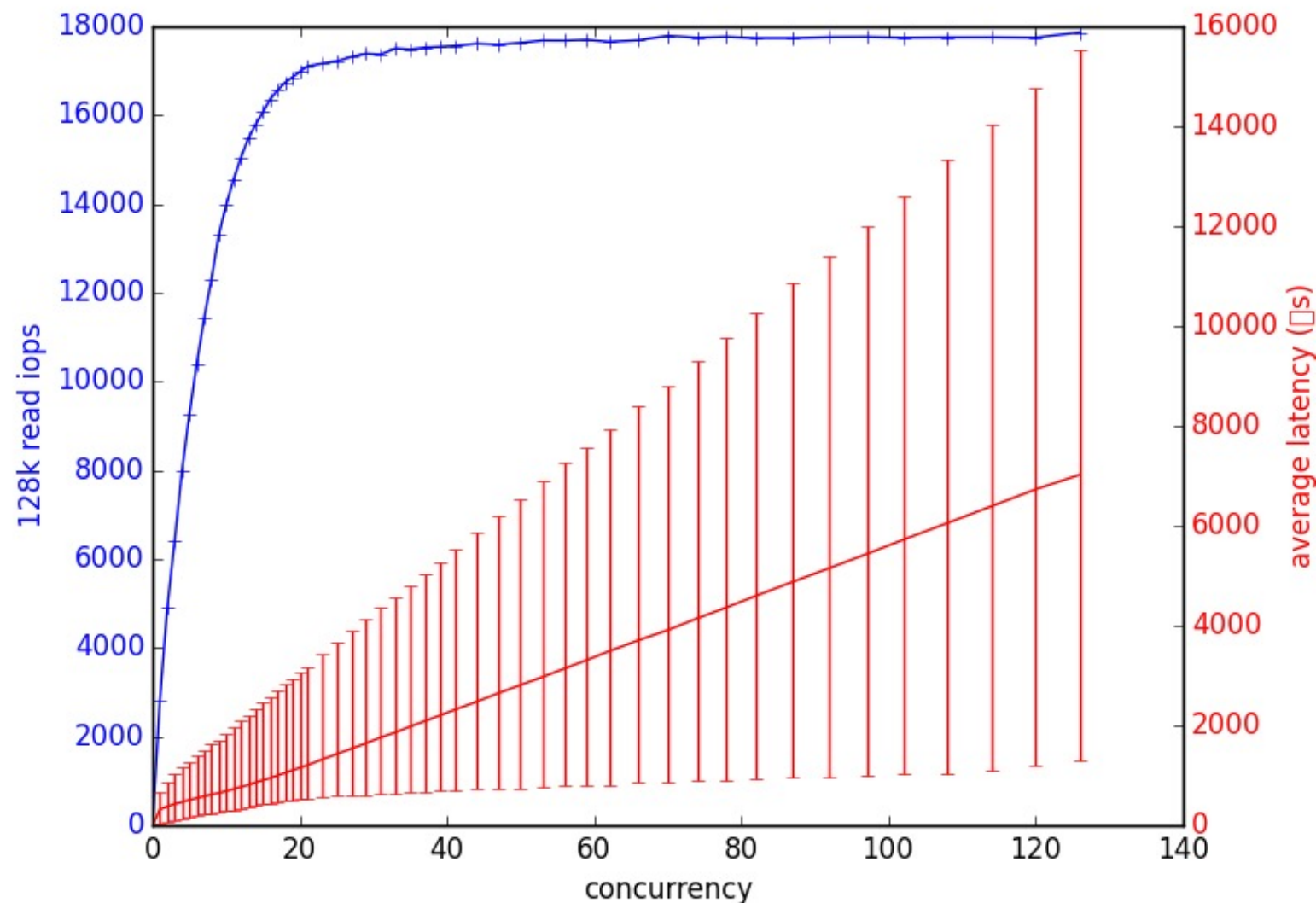


# Очереди IO: queueing и scheduling

Время отклика и количество IOPS  
в зависимости от глубины очереди  
запросов:

См. также:

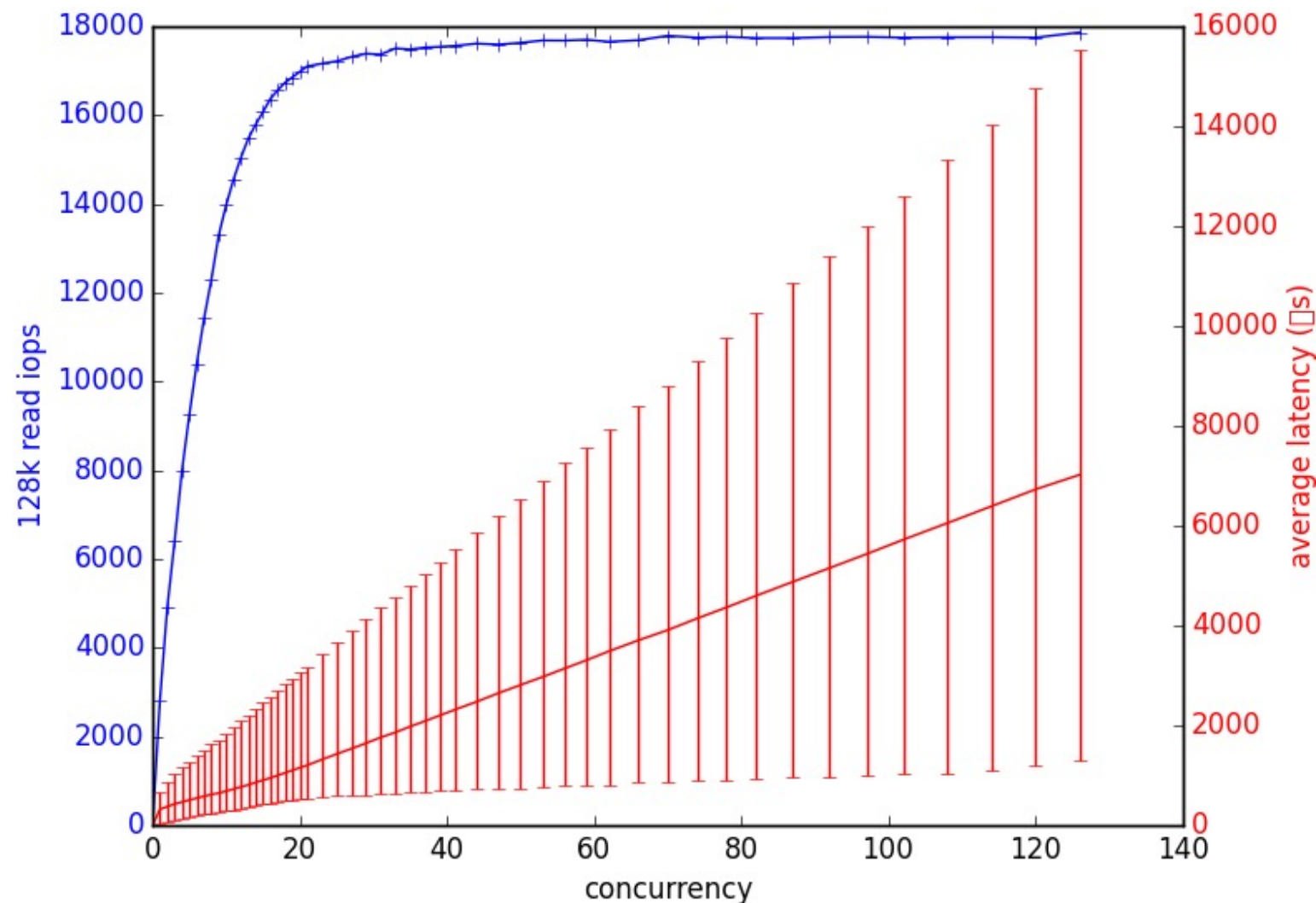
- Bufferbloat: <https://lwn.net/Articles/458625/>,
- Скорость Google Cloud Storage достигает максимума при загрузке блоками по 16М; после 64М дисперсия скорости растёт скачкообразно.



# Очереди IO: queueing и scheduling

Время отклика и количество IOPS  
в зависимости от глубины очереди  
запросов:

Вывод: нет смысла испускать слишком  
много IO-запросов одновременно, их  
можно накапливать в очереди приложения  
в userspace.



## Очереди IO: queueing и scheduling

Идея: нет смысла испускать слишком много IO-запросов одновременно, их можно накапливать в очереди приложения в userspace.

Очереди в userspace не ухудшают время обработки запросов, но позволяют делать обработку, невозможную в ядре:

- собирать статистику о том, сколько времени запрос проводит в каких очередях,
- настраивать приоритеты запросов, пользуясь знанием об их природе,
- отменять запросы, если они стали не нужны.

*См. также “ScyllaDB userspace disk IO scheduler”:*

- <https://www.scylladb.com/2016/04/14/io-scheduler-1/>
- <https://www.scylladb.com/2016/04/29/io-scheduler-2/>
- <https://www.scylladb.com/2018/04/19/scylla-i-o-scheduler-3/>

## Очереди IO: queueing и scheduling

Идея: нет смысла испускать слишком много IO-запросов одновременно, их можно накапливать в очереди приложения в userspace.

Очереди в userspace не ухудшают время обработки запросов, но позволяют делать обработку, невозможную в ядре:

- собирать статистику о том, сколько времени запрос проводит в каких очередях,
- настраивать приоритеты запросов, пользуясь знанием об их природе,
- отменять запросы, если они стали не нужны.

Какую статистику можно собирать:

- время обработки запросов,

## Очереди IO: queueing и scheduling

Идея: нет смысла испускать слишком много IO-запросов одновременно, их можно накапливать в очереди приложения в userspace.

Очереди в userspace не ухудшают время обработки запросов, но позволяют делать обработку, невозможную в ядре:

- собирать статистику о том, сколько времени запрос проводит в каких очередях,
- настраивать приоритеты запросов, пользуясь знанием об их природе,
- отменять запросы, если они стали не нужны.

Какую статистику можно собирать:

- ~~время обработки запросов~~ плохой показатель для машин, которые разбивают обработку запросов на более мелкие и отправляют их подчинённым; он показывает, насколько загружены подчинённые машины.

## Очереди IO: queueing и scheduling

Идея: нет смысла испускать слишком много IO-запросов одновременно, их можно накапливать в очереди приложения в userspace.

Очереди в userspace не ухудшают время обработки запросов, но позволяют делать обработку, невозможную в ядре:

- собирать статистику о том, сколько времени запрос проводит в каких очередях,
- настраивать приоритеты запросов, пользуясь знанием об их природе,
- отменять запросы, если они стали не нужны.

Какую статистику можно собирать:

- ~~время обработки запросов,~~
- загрузка CPU и RAM,

## Очереди IO: queueing и scheduling

Идея: нет смысла испускать слишком много IO-запросов одновременно, их можно накапливать в очереди приложения в userspace.

Очереди в userspace не ухудшают время обработки запросов, но позволяют делать обработку, невозможную в ядре:

- собирать статистику о том, сколько времени запрос проводит в каких очередях,
- настраивать приоритеты запросов, пользуясь знанием об их природе,
- отменять запросы, если они стали не нужны.

Какую статистику можно собирать:

- ~~время обработки запросов,~~
- ~~загруженность CPU и RAM~~ если мы успеваем обрабатывать запросы за требуемое время, то нет разницы, загружен CPU на 50% или на 80%; можно использоваться только для планирования масштабирования системы.



## Очереди IO: queueing и scheduling

Идея: нет смысла испускать слишком много IO-запросов одновременно, их можно накапливать в очереди приложения в userspace.

Очереди в userspace не ухудшают время обработки запросов, но позволяют делать обработку, невозможную в ядре:

- собирать статистику о том, сколько времени запрос проводит в каких очередях,
- настраивать приоритеты запросов, пользуясь знанием об их природе,
- отменять запросы, если они стали не нужны.

Какую статистику можно собирать:

- ~~время обработки запросов,~~
- ~~загруженность CPU и RAM,~~
- длина очереди в байтах или других единицах «стоимости» запроса,
- время ожидания запросов в очереди.

## Очереди IO: CoDel и RED

Для TCP оказались полезными две идеи об управлении очередями:

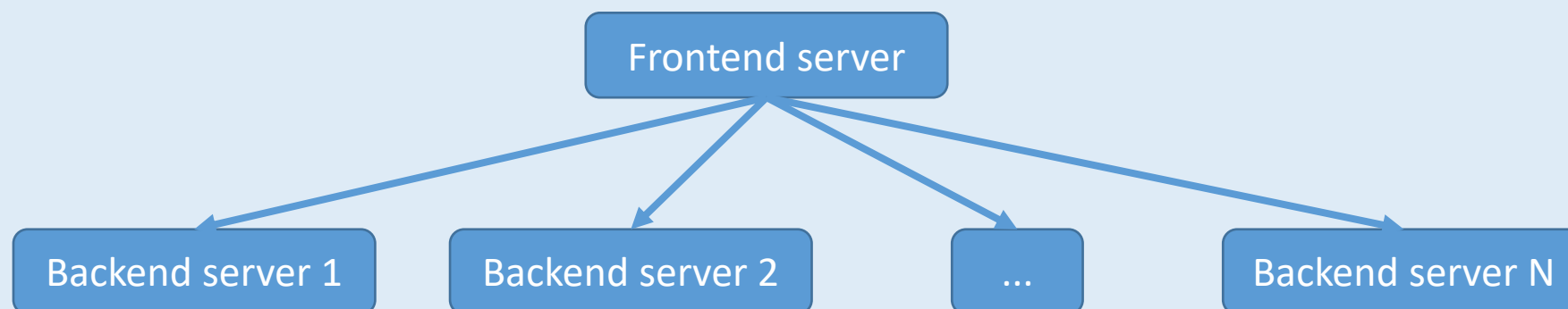
- Controlled Delay: промежуточные узлы ограничивают время ожидания пакетов в очередях; как только время превышает некоторый порог, узел сигнализирует о перегруженности линии (TCP congestion),
- Random Early Detection\*: промежуточные узлы сигнализируют о congestion незадолго **ДО** того, как истечёт время ожидания в очереди или длина очереди будет превышена; пакеты отбрасываются из случайно выбранных соединений и из случайно выбранных позиций.

\* См., однако, “RED in different light”: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.9406>

## Очереди IO: отбрасывание IO в распределённой системе

Для TCP отбрасывание пакетов на нагруженных линиях работает хорошо, заставляя обе стороны замедлять скорость отсылки пакетов. Случайность выбора отбрасываемого пакета не представляет проблемы, поскольку одновременных потоков TCP обычно много.

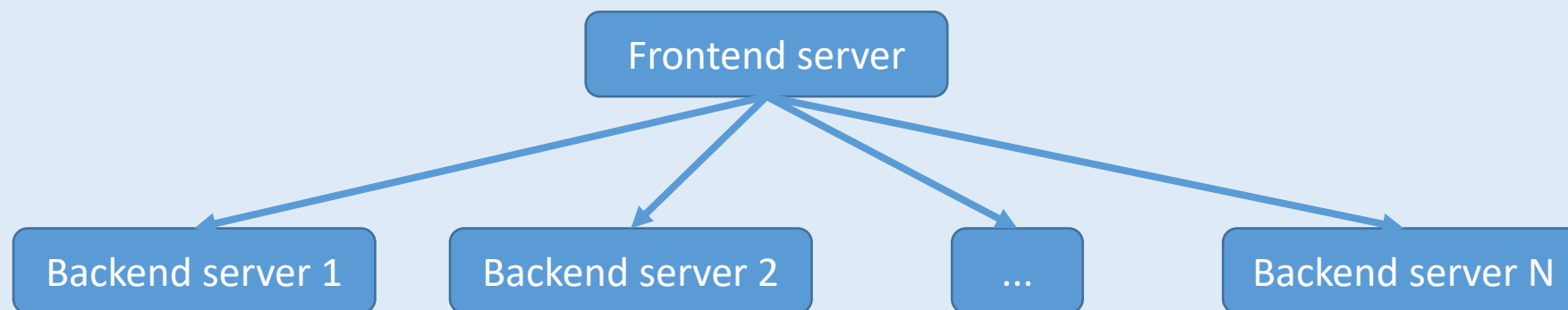
Для распределённой системы случайное отбрасывание запросов – плохая затея:



Обычно для выполнения одного запроса к системе требуется разослать запросы на несколько подчинённых систем и собрать ответ из частей.

Если сервера 1, 2, ..., N-1 выполнили свою работу, а сервер N отбросил запрос, то сервера с 1 по N-1 отработали впустую. Таким образом, случайное отбрасывание запросов в условиях перегруженности сервиса только ведёт к бОльшей перегрузке.

## Очереди IO: отбрасывание IO в распределённой системе

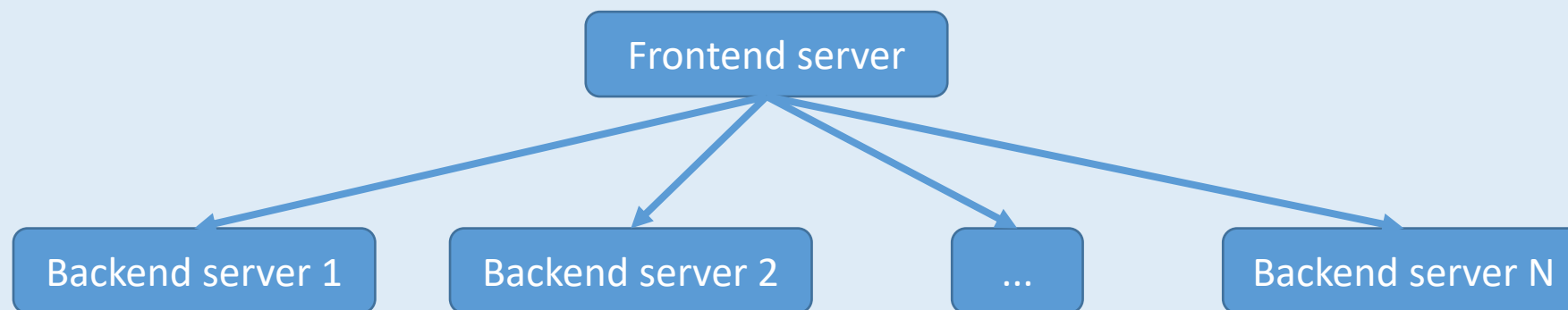


Обычно для выполнения одного запроса к системе требуется разослать запросы на несколько подчинённых систем и собрать ответ из частей.

Если сервера 1, 2, ..., N-1 выполнили свою работу, а сервер N отбросил запрос, то сервера с 1 по N-1 отработали впустую. Таким образом, случайное отбрасывание запросов в условиях перегруженности сервиса только ведёт к бОльшей перегрузке.

**Напоминание:** мы уже видели похожую проблему при обсуждении **tail latency**.

## Очереди IO: отбрасывание IO в распределённой системе



Обычно для выполнения одного запроса к системе требуется разослать запросы на несколько подчинённых систем и собрать ответ из частей.

Если сервера 1, 2, ..., N-1 выполнили свою работу, а сервер N отбросил запрос, то сервера с 1 по N-1 отработали впустую. Таким образом, случайное отбрасывание запросов в условиях перегруженности сервиса только ведёт к бОльшей перегрузке.

**Идея:** запросы надо отбрасывать не случайным образом, а выделить менее важные, и отбрасывать уже их.

## Очереди IO: отбрасывание IO в распределённой системе

**Идея:** в условиях перегруженности отбрасывать запросы с меньшими приоритетами.

Проблема (приоритеты в порядке убывания – 0, 1, ...):

- если запросов с приоритетами  $\leq N$  достаточно много, чтобы перегрузить систему, то отбрасываем все запросы с приоритетами  $\geq N$ ,
- если оказывается, что запросов с приоритетами  $\leq N-1$  недостаточно много, что перегрузить систему, то через небольшое время мы снова разрешим принимать запросы с приоритетом  $N$ ,
- в нашей системе получается колебание между перегруженным и недогруженным состояниями.

## Очереди IO: отбрасывание IO в распределённой системе

**Идея:** в условиях перегруженности отбрасывать запросы с меньшими приоритетами.

Проблема (приоритеты в порядке убывания – 0, 1, ...):

- если запросов с приоритетами  $\leq N$  достаточно много, чтобы перегрузить систему, то отбрасываем все запросы с приоритетами  $\geq N$ ,
- если оказывается, что запросов с приоритетами  $\leq N-1$  недостаточно много, что перегрузить систему, то через небольшое время мы снова разрешим принимать запросы с приоритетом  $N$ ,
- в нашей системе получается колебание между перегруженным и недогруженным состояниями.

**Идея:** сделать очень много уровней приоритета. Например, в пределах каждого уровня приоритетов типа запроса ввести уровни приоритета, соответствующие номеру пользователя.

Больше деталей см. в Overload Control for Scaling WeChat Microservices: <https://www.cs.columbia.edu/~ruigu/papers/socc18-final100.pdf>

# Путь данных от приложения до диска (обзорно)

