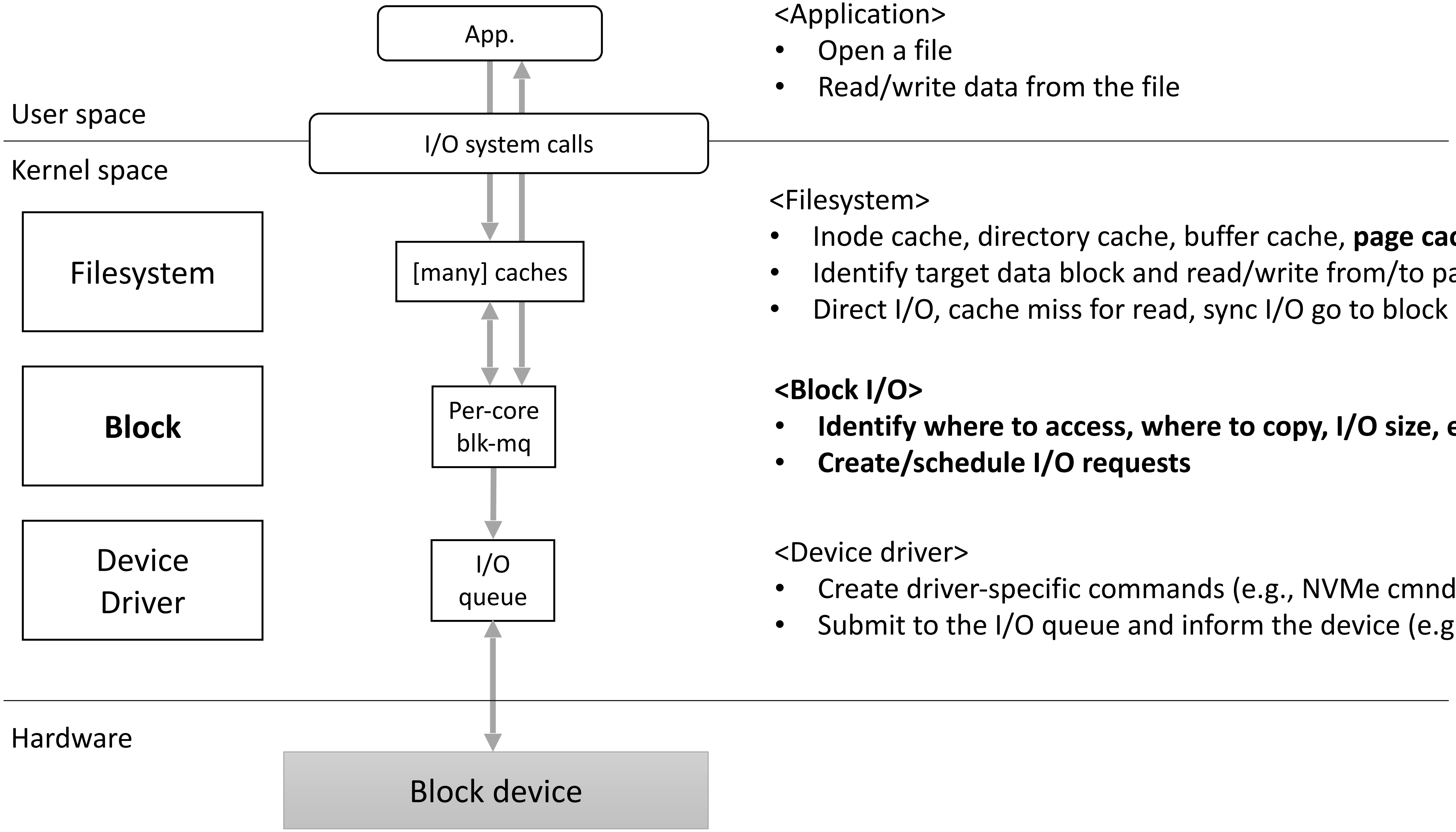
Linux kernel
Block I/O Layer

Overview: Accessing block devices

- **Block devices (e.g., HDD, SSD)**
  - Allow accessing fixed-size chunks of data
  - The fixed size chucks of data are called blocks
    - Block is the smallest logically addressable unit defined by filesystems (mostly 4KB)

- **Linux kernel has block I/O layer for accessing block devices**
  - Manage block devices
  - Create/schedule I/O requests
  - Interface with two layers
    - Upper layer: File System
    - Lower layer: Device Driver (such as NVMe)

Overview: Accessing block devices



<Application>

- Open a file
- Read/write data from the file

<Filesystem>

- Inode cache, directory cache, buffer cache, **page cache**
- Identify target data block and read/write from/to page cache
- Direct I/O, cache miss for read, sync I/O go to block layer

<Block I/O>

- **Identify where to access, where to copy, I/O size, etc.**
- **Create/schedule I/O requests**

<Device driver>

- Create driver-specific commands (e.g., NVMe cmnd)
- Submit to the I/O queue and inform the device (e.g., ring doorbells)

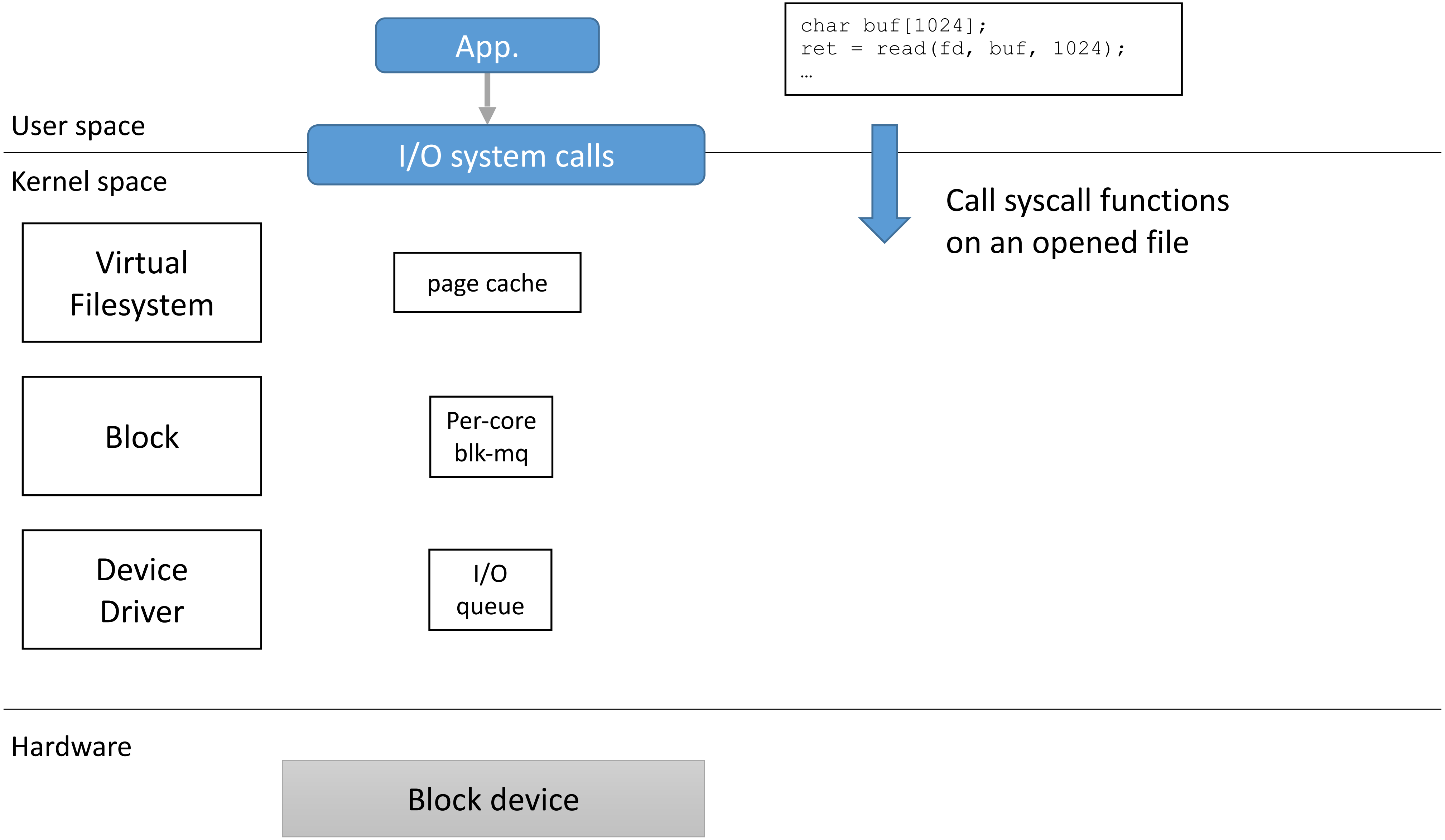
All these caches

- **inode cache and directory cache**
  - Enabling functionalities discussed in file systems
  - Faster access to information in inode and directory
    - Separated from “data” cache

- **Page cache**
  - Combines virtual memory and file data
  - Caches recently read data on persistent storage at the granularity of pages
  - Has a notion of “file”

- **Buffer cache**
  - Interfaces with block devices (hardware)
  - Caches recently read data blocks on persistent storage
  - Has no notion of “files”—just blocks

Read I/O data path: Application

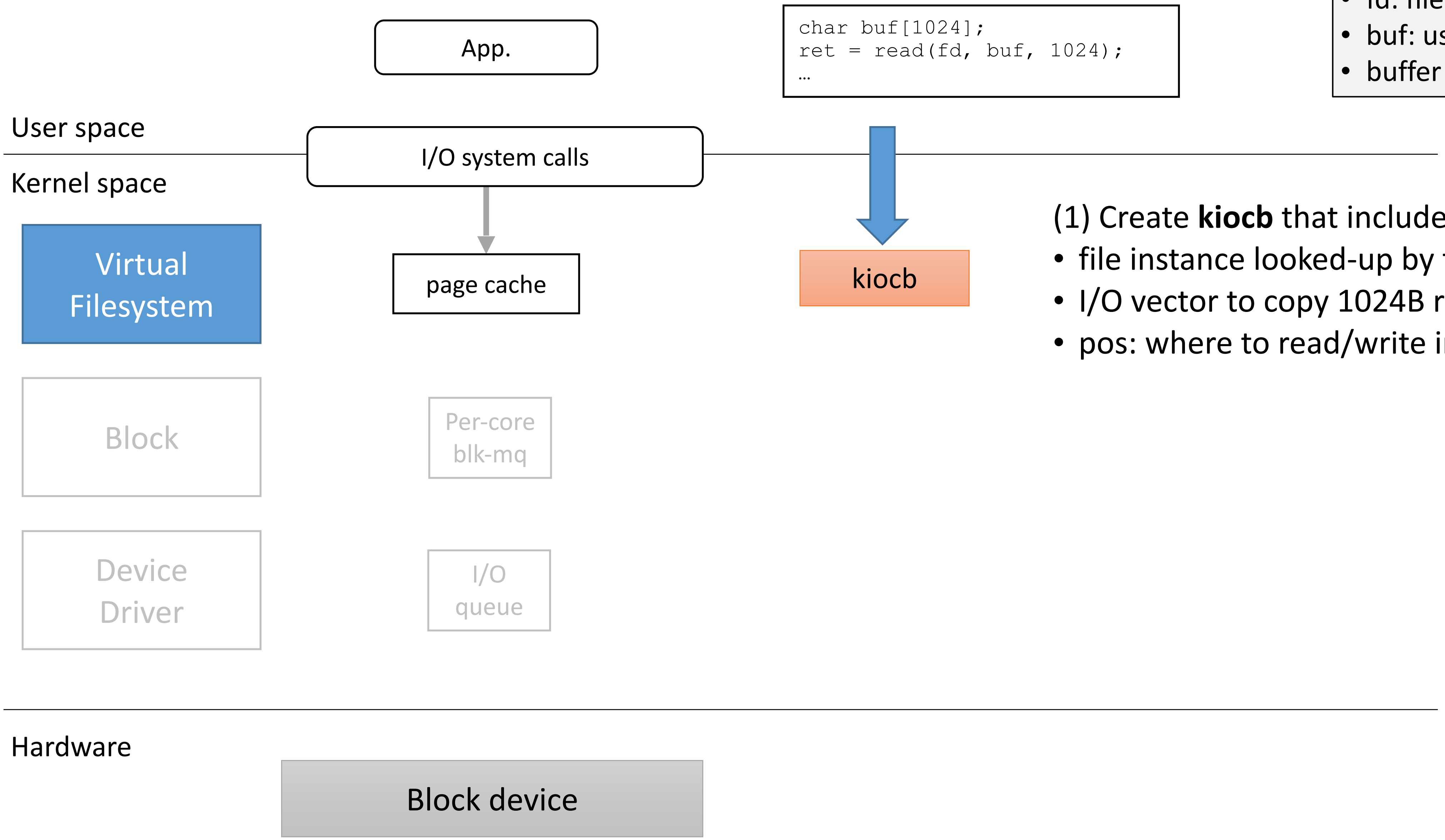


```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

Call syscall functions
on an opened file

Read I/O data path: Virtual Filesystem

[Input]
- fd: file descriptor
- buf: user buffer
- buffer length: 1024B

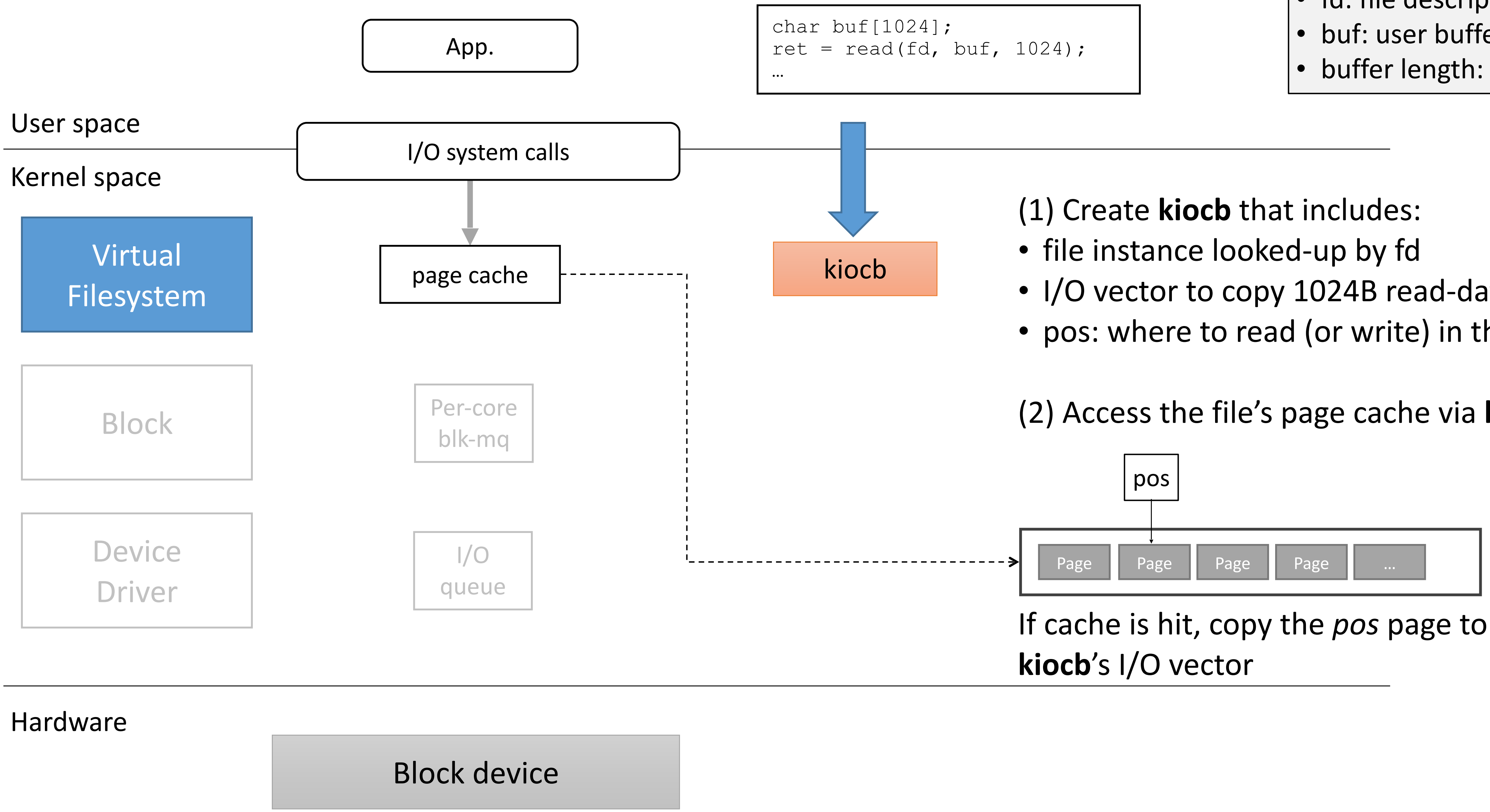


```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

(1) Create **kiocb** that includes:
- file instance looked-up by fd
- I/O vector to copy 1024B read-data to buf
- pos: where to read/write in the file

Read I/O data path: Virtual Filesystem

[Input]
- fd: file descriptor
- buf: user buffer
- buffer length: 1024B



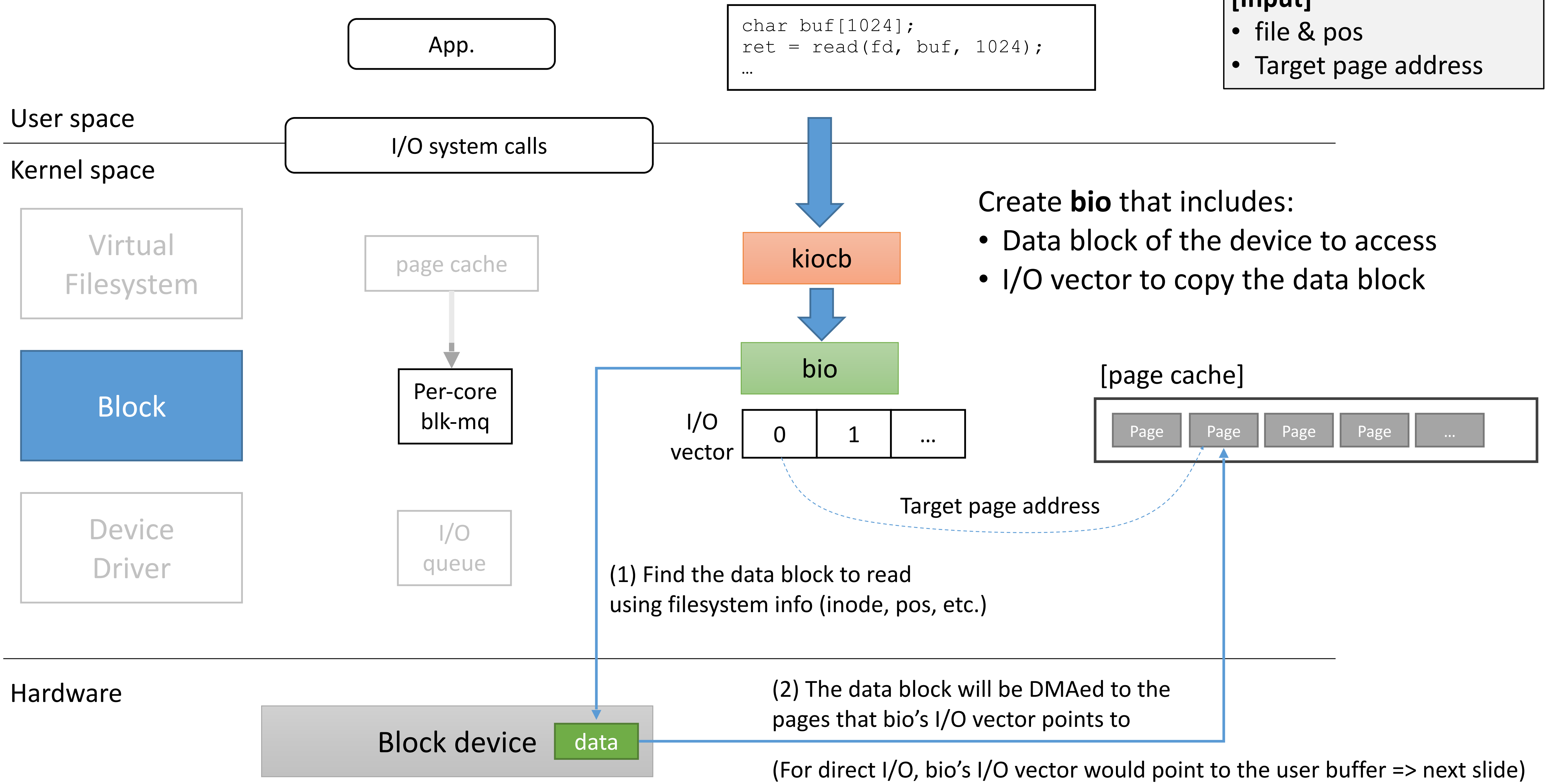
```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

(1) Create **kiocb** that includes:
- file instance looked-up by fd
- I/O vector to copy 1024B read-data to buf
- pos: where to read (or write) in the file

(2) Access the file's page cache via **kiocb**

If cache is hit, copy the *pos* page to *buf* using **kiocb**'s I/O vector

Read I/O data path: Block (init. bio)



```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

[Input]

- file & pos
- Target page address

Create **bio** that includes:

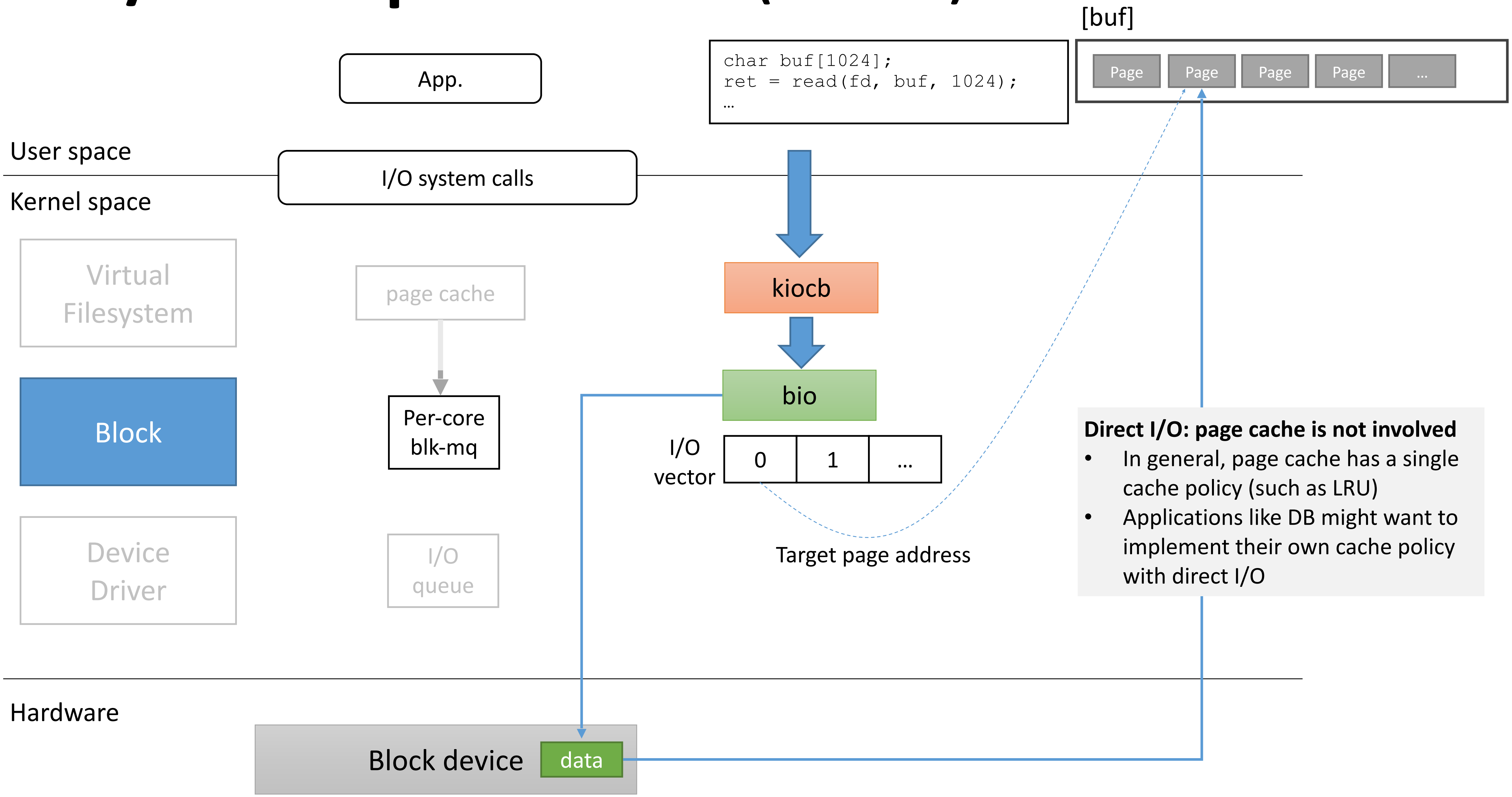
- Data block of the device to access
- I/O vector to copy the data block

(1) Find the data block to read using filesystem info (inode, pos, etc.)

(2) The data block will be DMAed to the pages that bio’s I/O vector points to

(For direct I/O, bio’s I/O vector would point to the user buffer => next slide)

Read I/O data path: Block (init. bio)

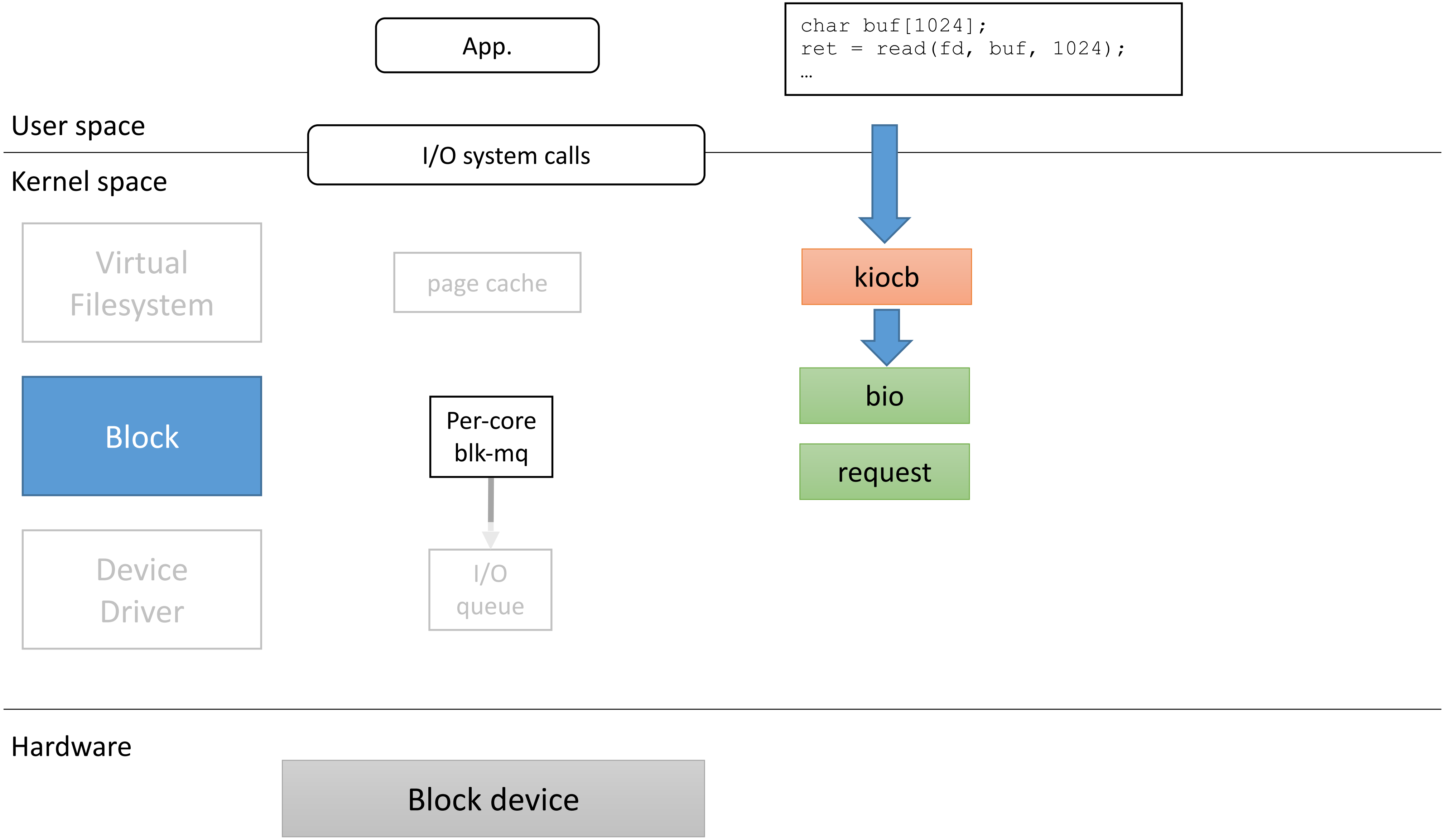


```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

Direct I/O: page cache is not involved

- In general, page cache has a single cache policy (such as LRU)
- Applications like DB might want to implement their own cache policy with direct I/O

Read I/O data path: Block (init. request)



App.

```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

User space

I/O system calls

Kernel space

Virtual Filesystem

page cache

kiocb

bio

request

Block

Per-core blk-mq

Device Driver

I/O queue

Hardware

Block device

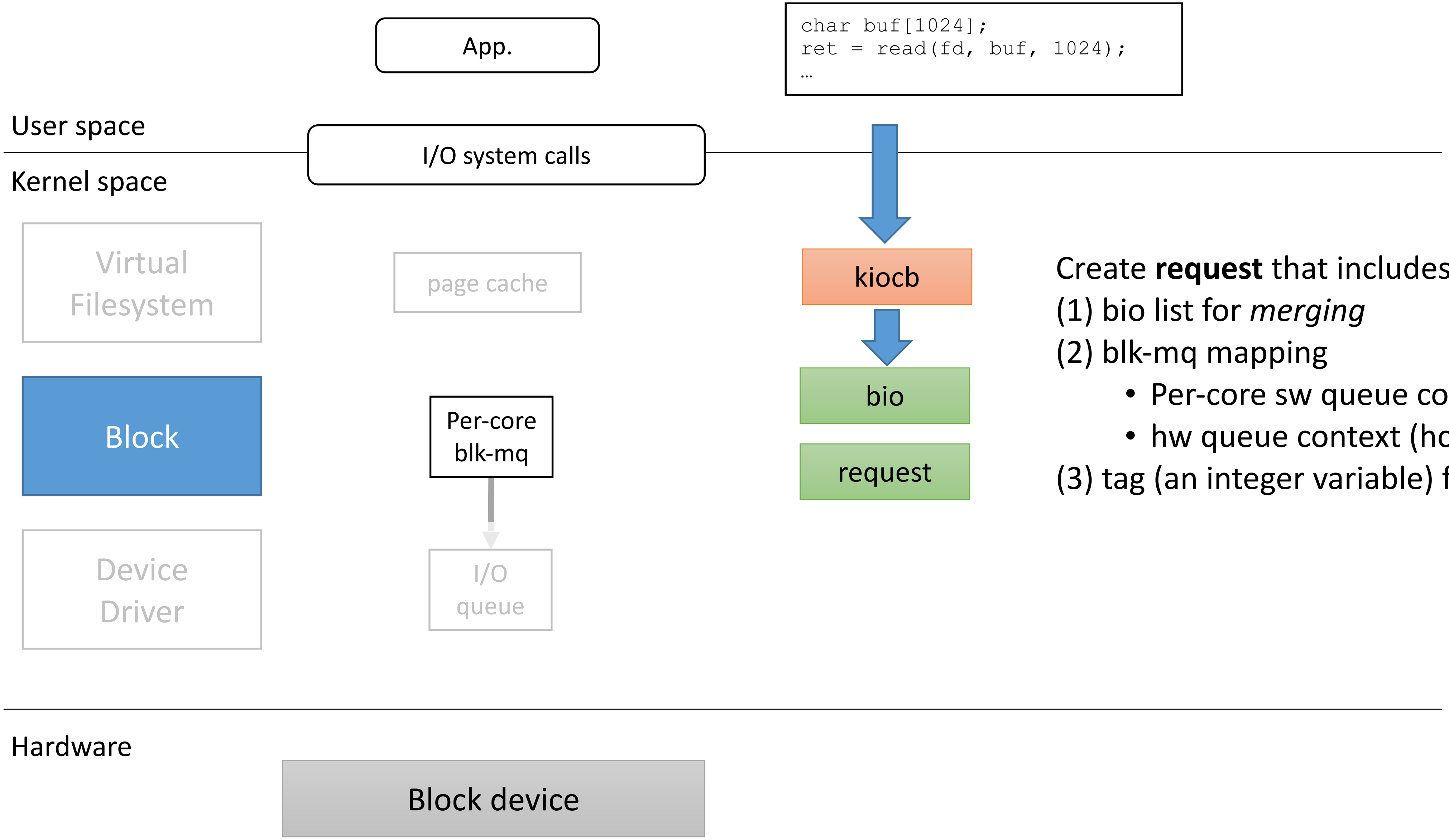
What should a “request” contain?

- **Where is the request going?**
  - Which device
  - But devices have multiple “queues”
    - Which queue
    - Identified by a “hardware context”

- **Where should the request response be directed?**
  - Which CPU core and which application
  - Identified by a “software context”

- **A request identifier**
  - tag

Read I/O data path: Block (init. request)



```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

Create **request** that includes:

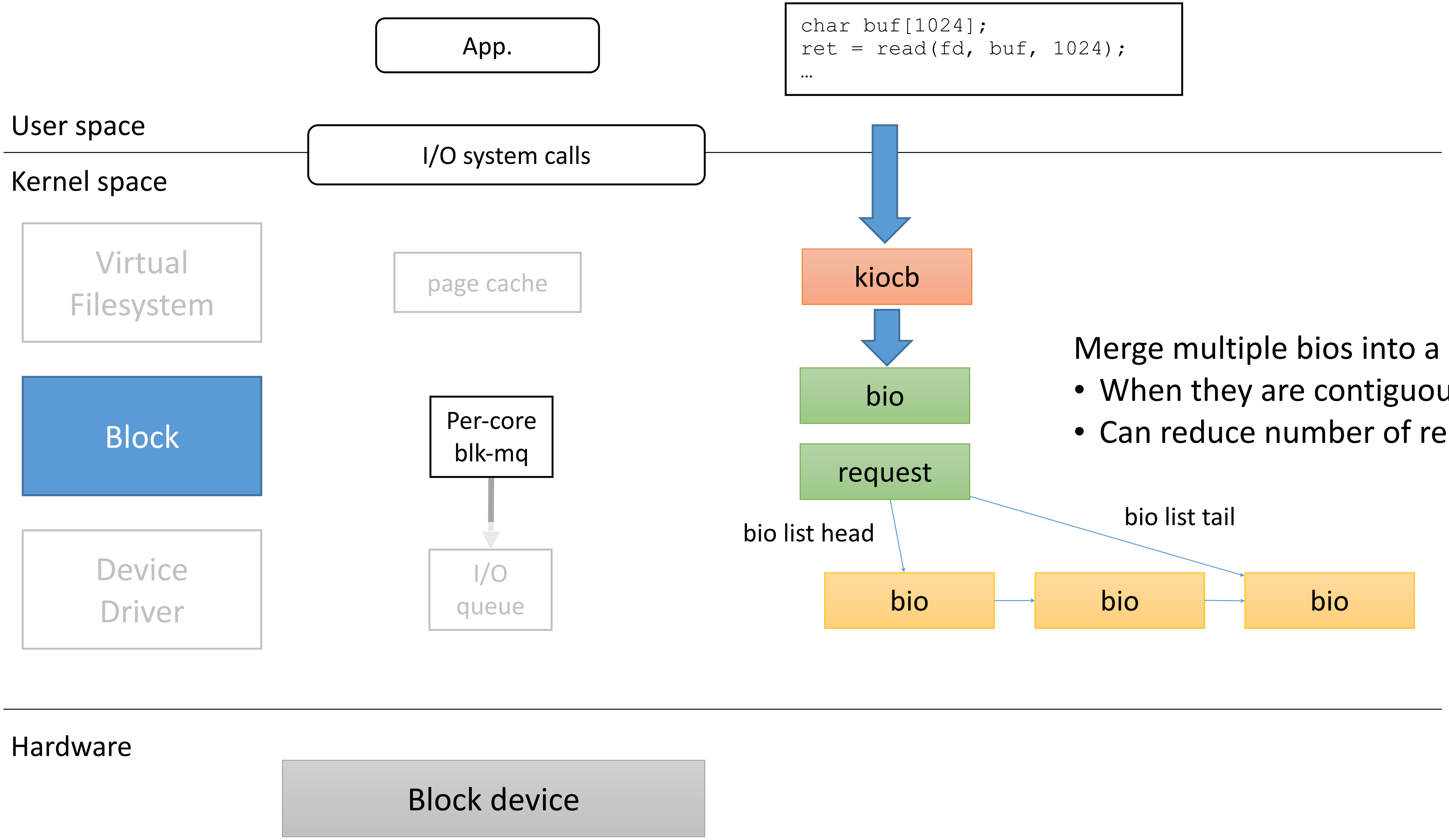
(1) bio list for *merging*

(2) blk-mq mapping

- Per-core sw queue context (ctx)
- hw queue context (hctx)

(3) tag (an integer variable) for *tagging*

Read I/O data path: Block (merging)



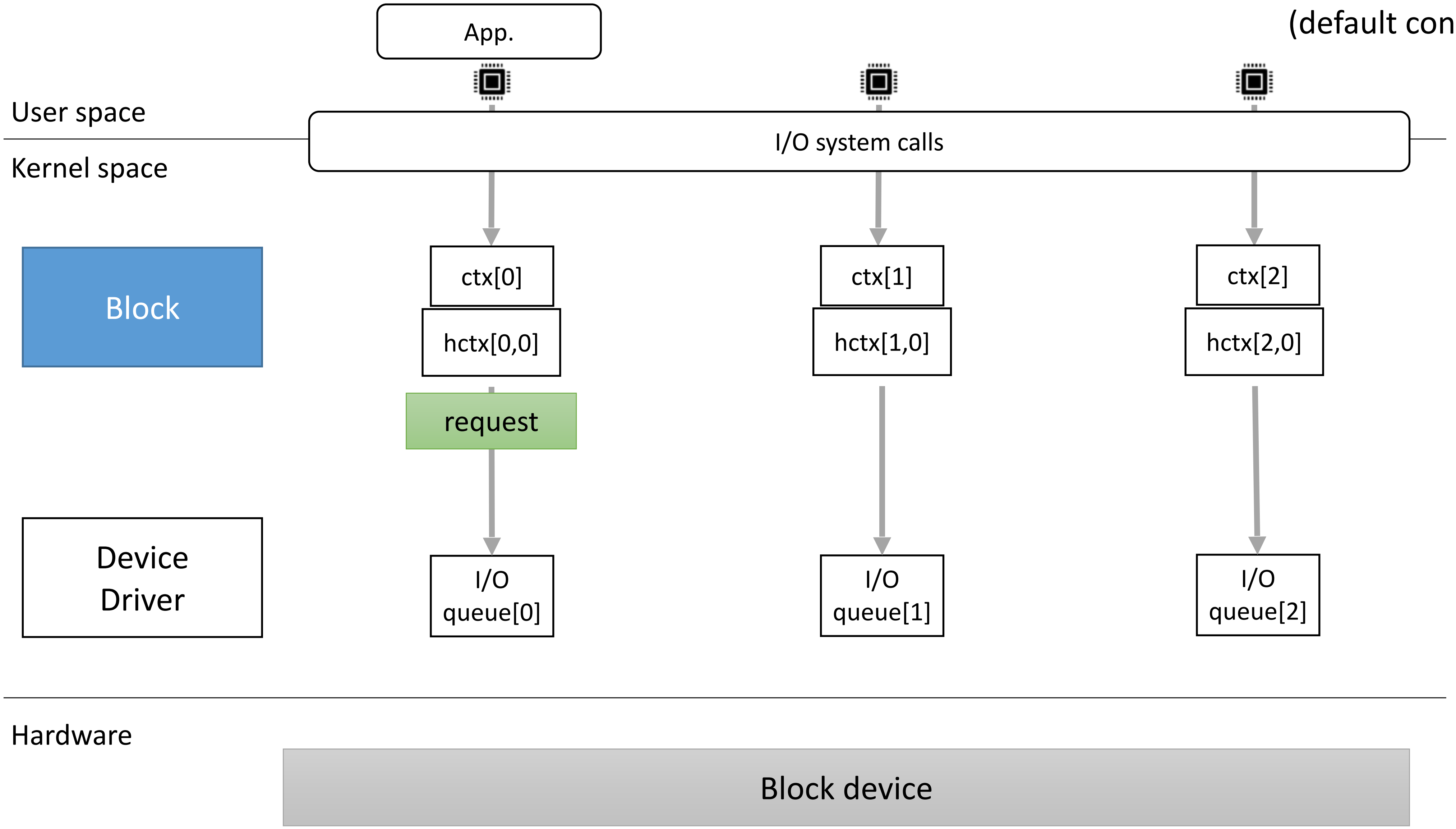
```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

Merge multiple bios into a single request

- When they are contiguous data blocks
- Can reduce number of requests

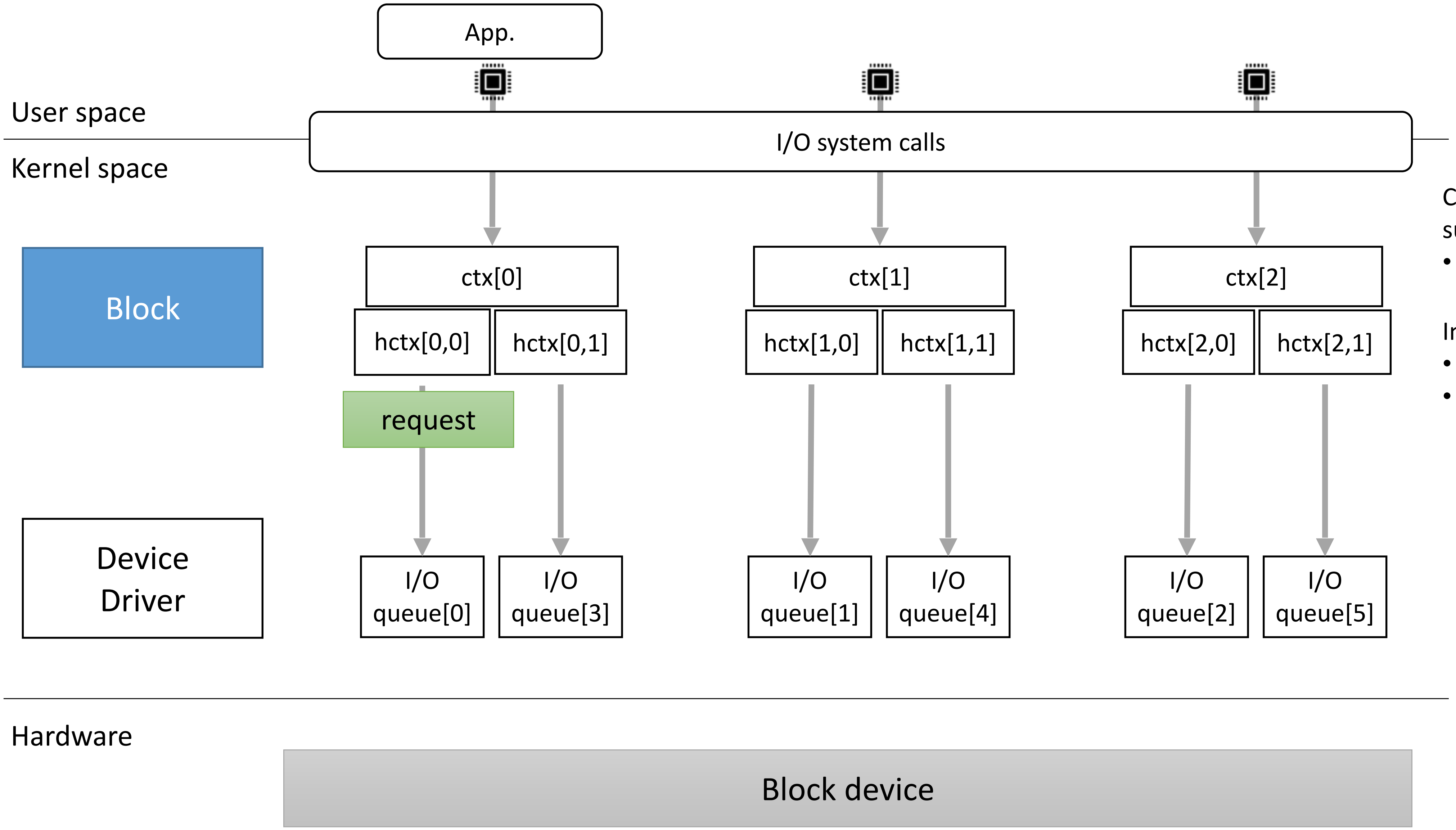
Read I/O data path: Block (blk-mq mapping case 1)

#I/O queues = #cores
(default configuration)



Read I/O data path: Block (blk-mq mapping case 2)

#I/O queues = #cores x 2



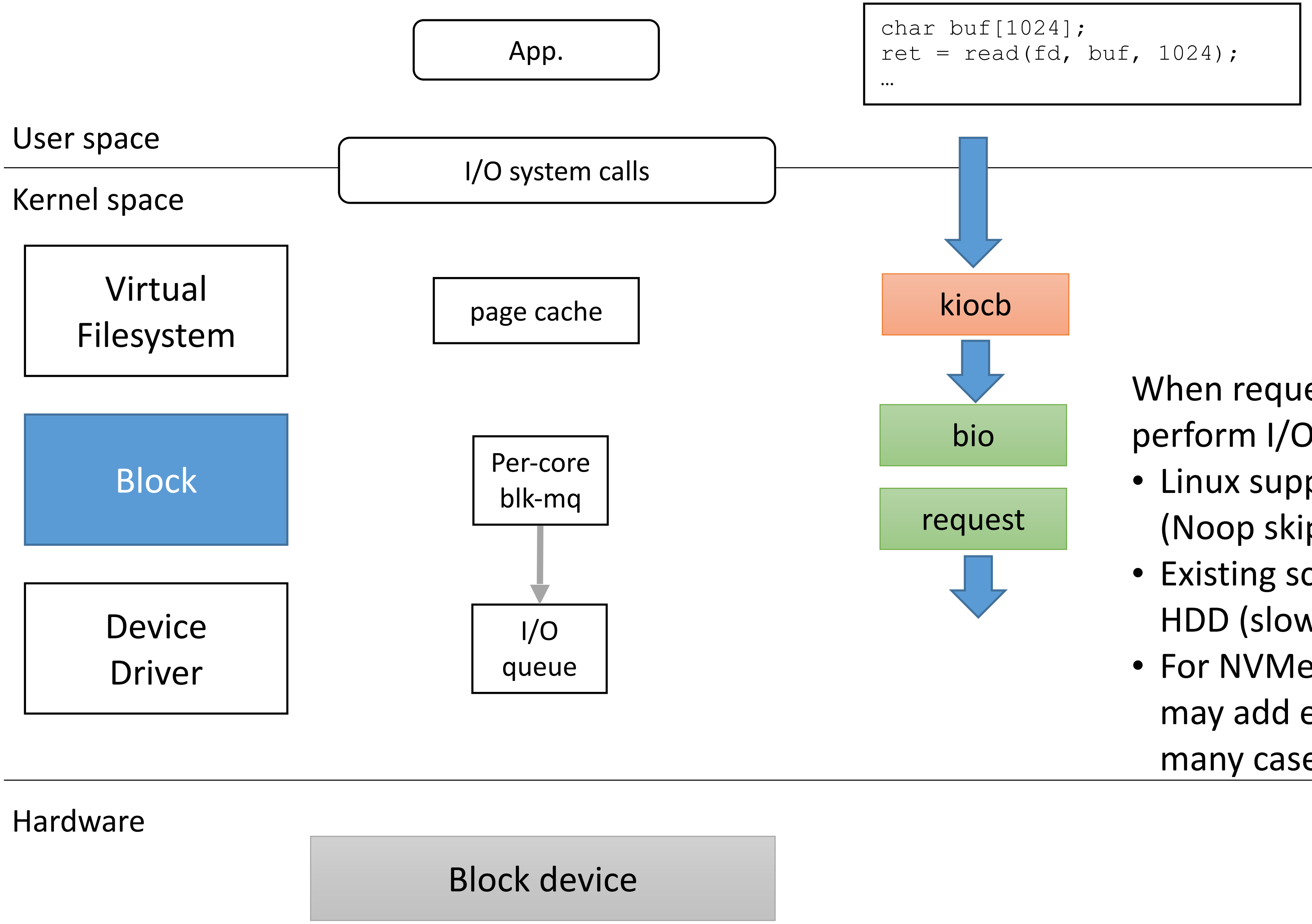
Current Linux kernel supports three hctx types:

- read/write/poll

In this example:

- hctx[*,0] for read I/O
- hctx[*,1] for write I/O

Read I/O data path: Block (scheduling)

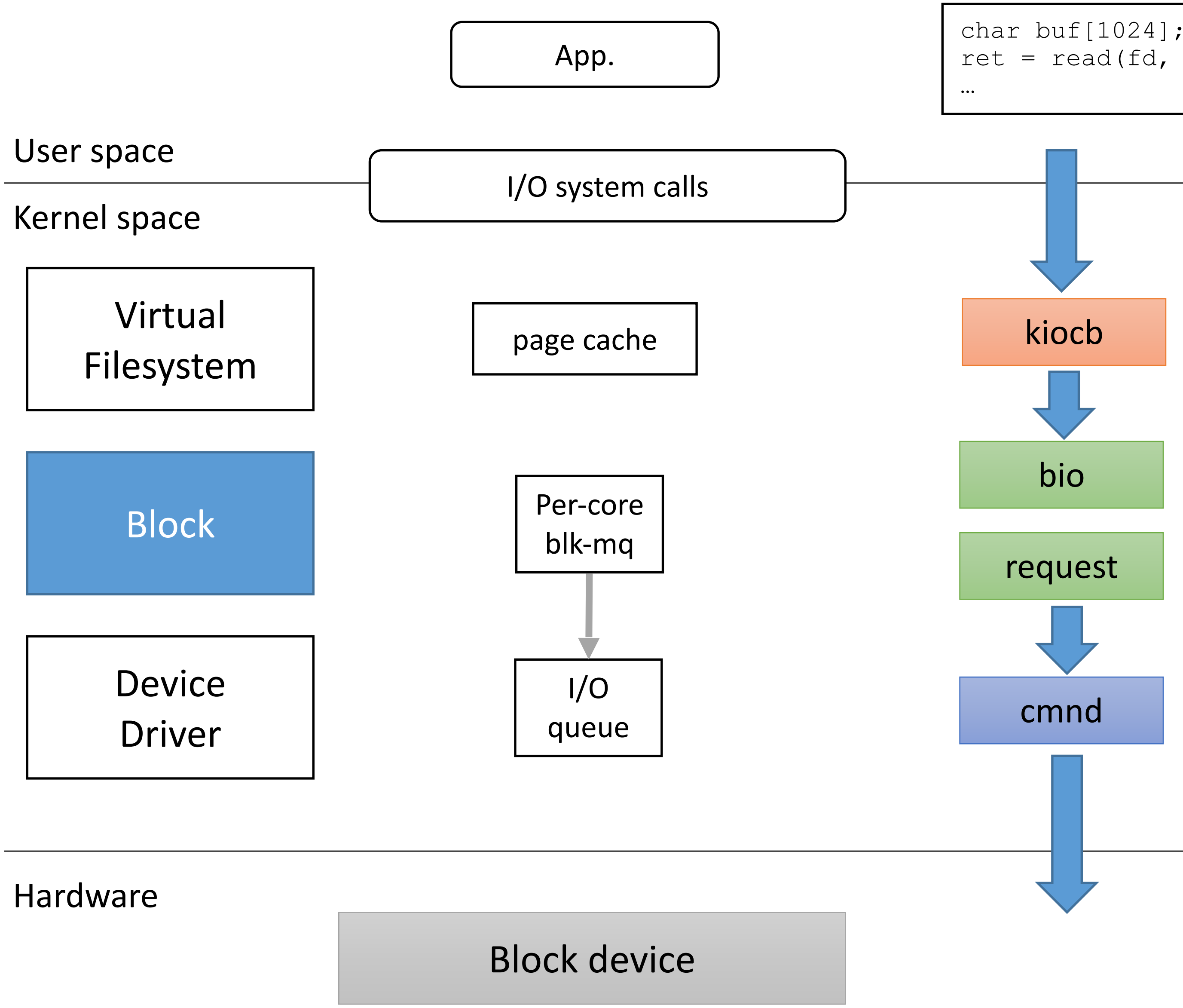


```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

When request initialization is done, perform I/O scheduling at per-req granularity

- Linux supports Noop/deadline/BFQ/Kyber (Noop skips I/O scheduling step)
- Existing schedulers were mostly designed for HDD (slow devices)
- For NVMe SSD (fast devices), I/O scheduling may add extra latency, so Noop is chosen in many cases

Read I/O data path: Device Driver (NVMe)



```
char buf[1024];
ret = read(fd, buf, 1024);
…
```

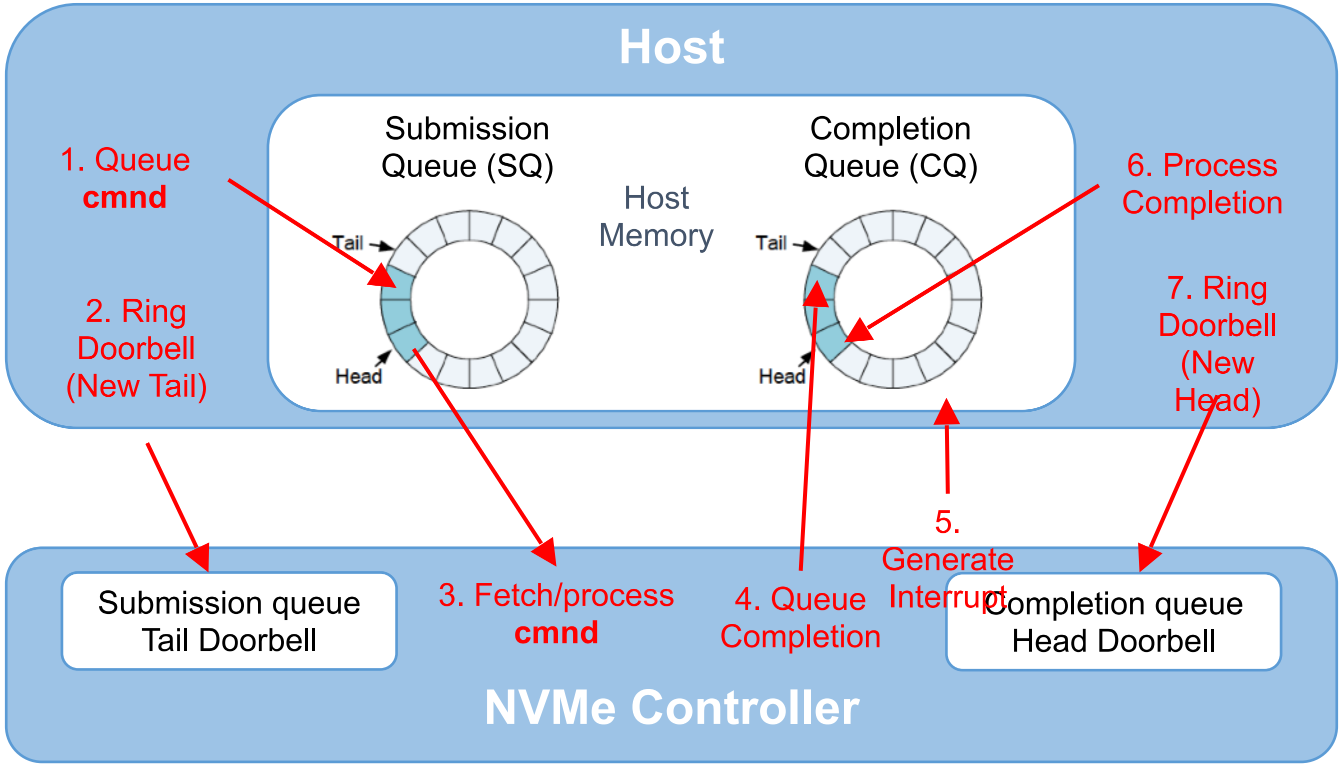
Create **cmnd** that includes:

- request instance
- NVMe-specific data structures (command header, etc.)

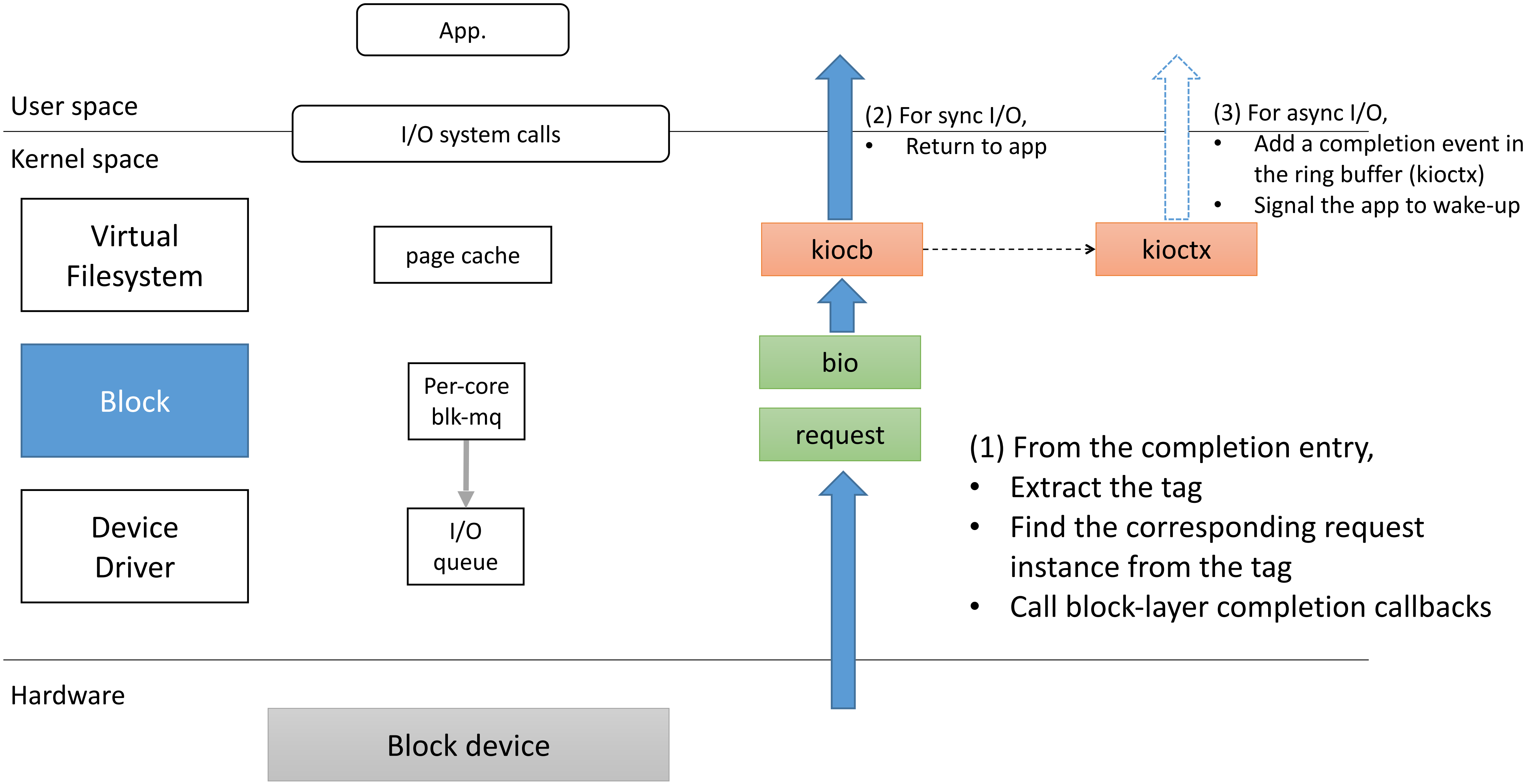
Perform NVMe operations (next slide)

- As I/O queues, NVMe defines submission queue (SQ) and completion queue (CQ) per core

Read I/O data path: Device Driver (NVMe)



Read I/O data path: Response



App.

User space

I/O system calls

Kernel space

Virtual Filesystem

page cache

Block

Per-core blk-mq

Device Driver

I/O queue

Hardware

Block device

kiocb

kioctx

bio

request

(2) For sync I/O,

- Return to app

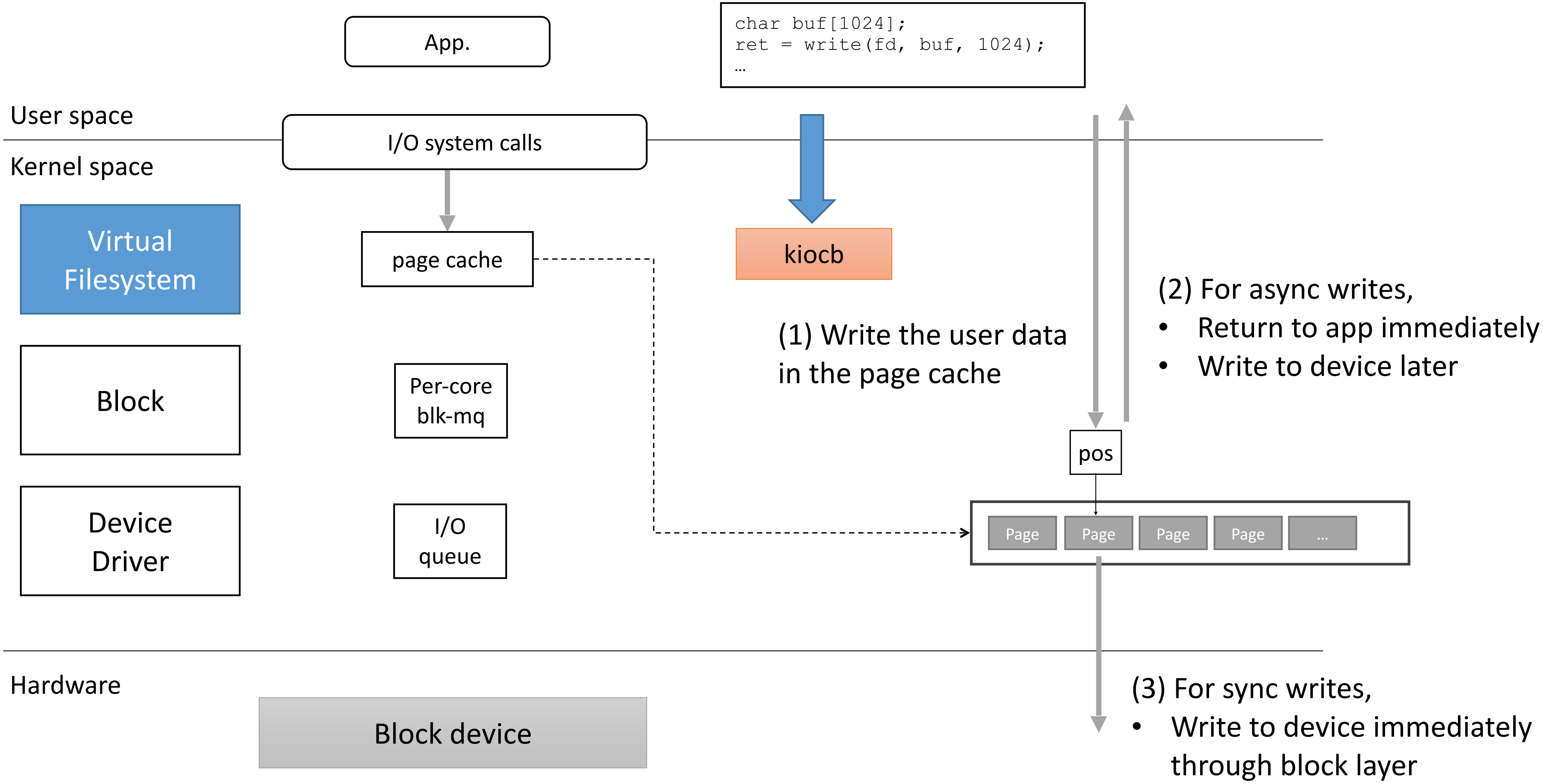
(3) For async I/O,

- Add a completion event in the ring buffer (kioctx)
- Signal the app to wake-up

(1) From the completion entry,

- Extract the tag
- Find the corresponding request instance from the tag
- Call block-layer completion callbacks

Write I/O path (difference from Read)



```
char buf[1024];
ret = write(fd, buf, 1024);
…
```

(1) Write the user data in the page cache

(2) For async writes,

- Return to app immediately
- Write to device later

(3) For sync writes,

- Write to device immediately through block layer