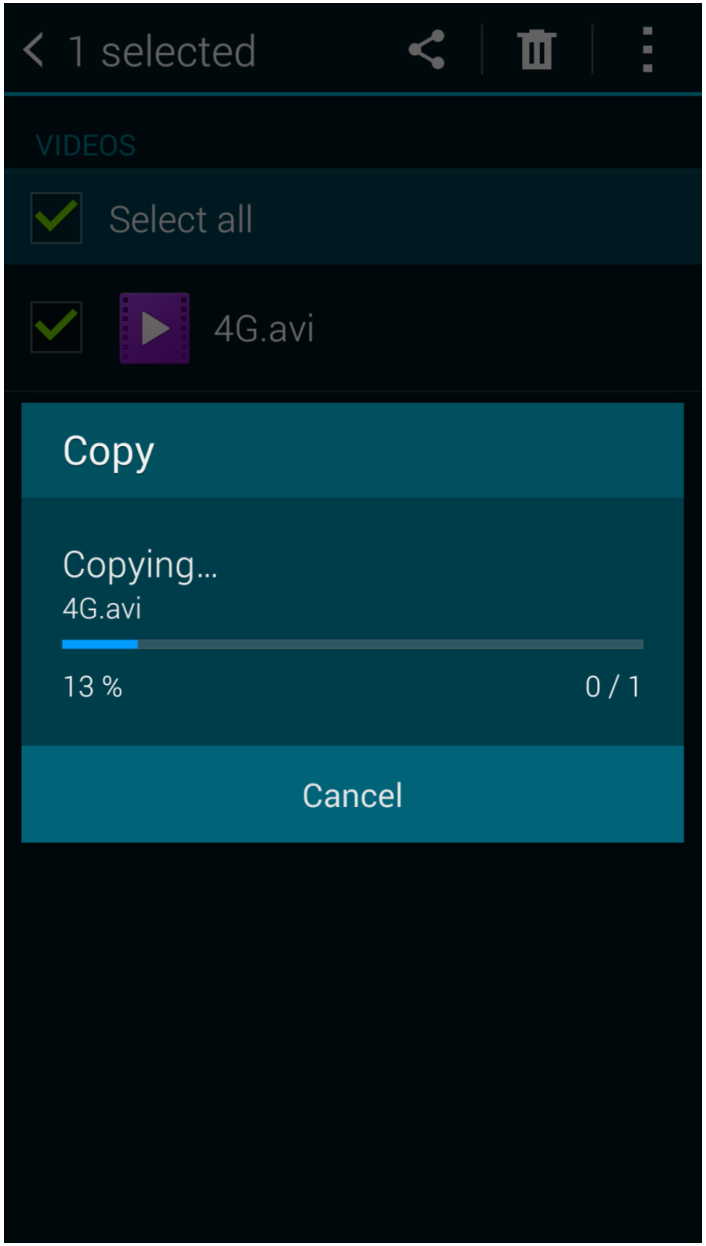
Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices

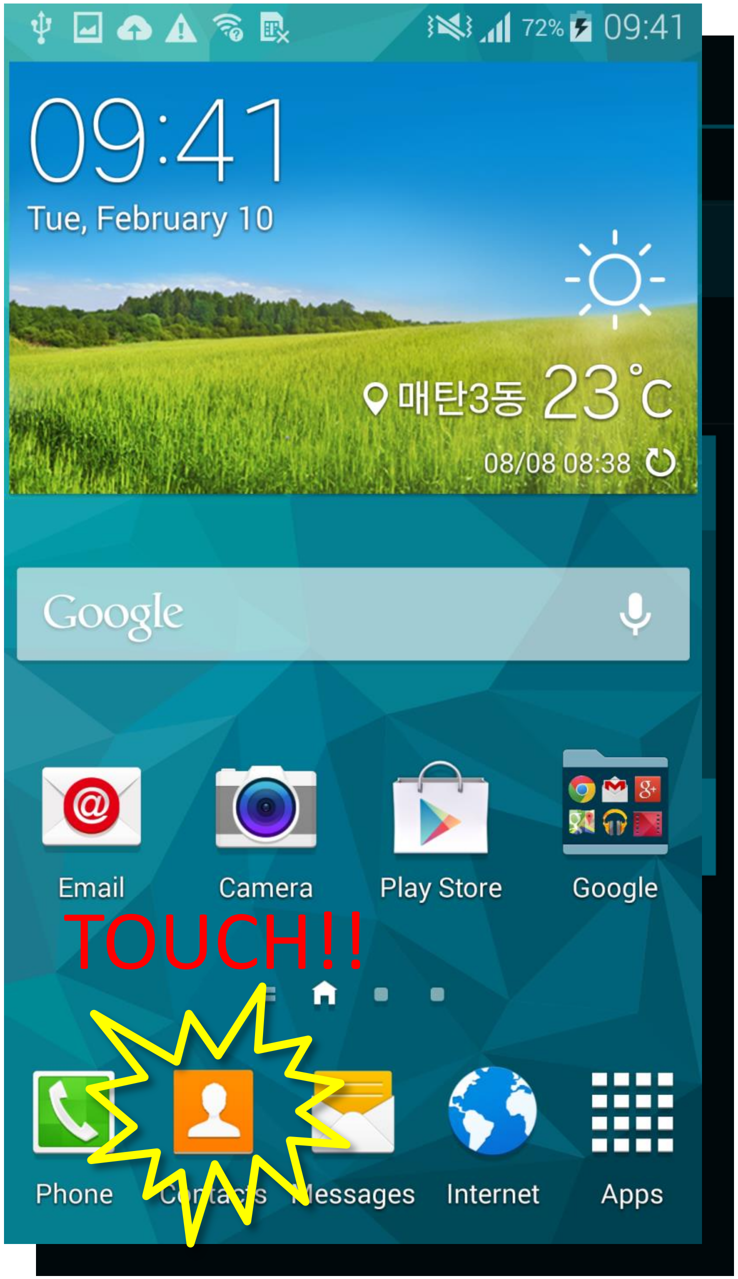
Daeho Jeong[*+], Youngjae Lee[+], Jin-Soo Kim[+]

[*]Samsung Co., Suwon, Korea

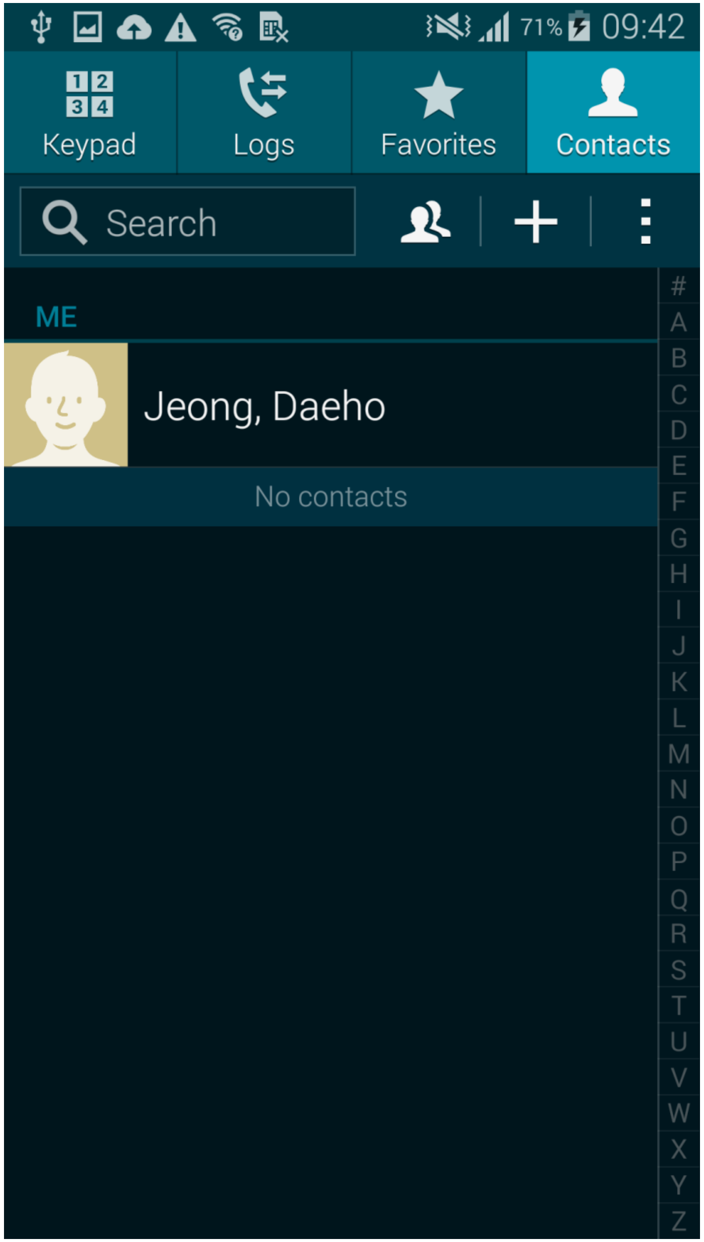
[+]Sungkyunkwan University

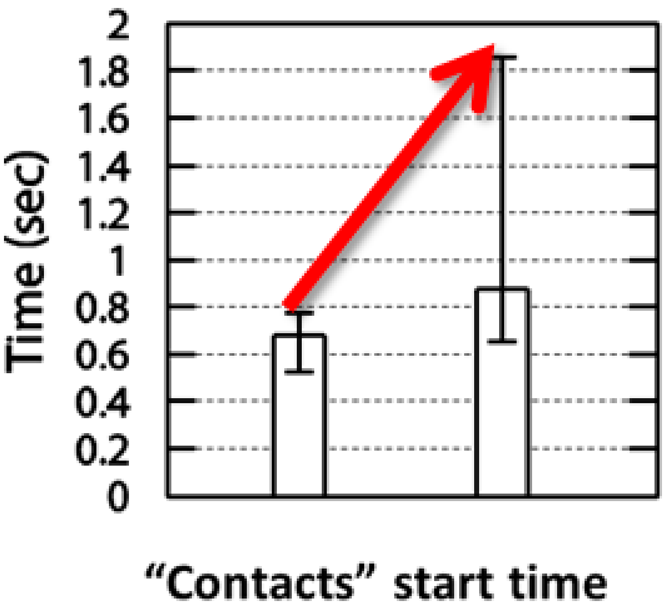

1 selected
VIDEOS
Select all
4G.avi
Copy
Copying...
4G.avi
13 %
0 / 1
Cancel

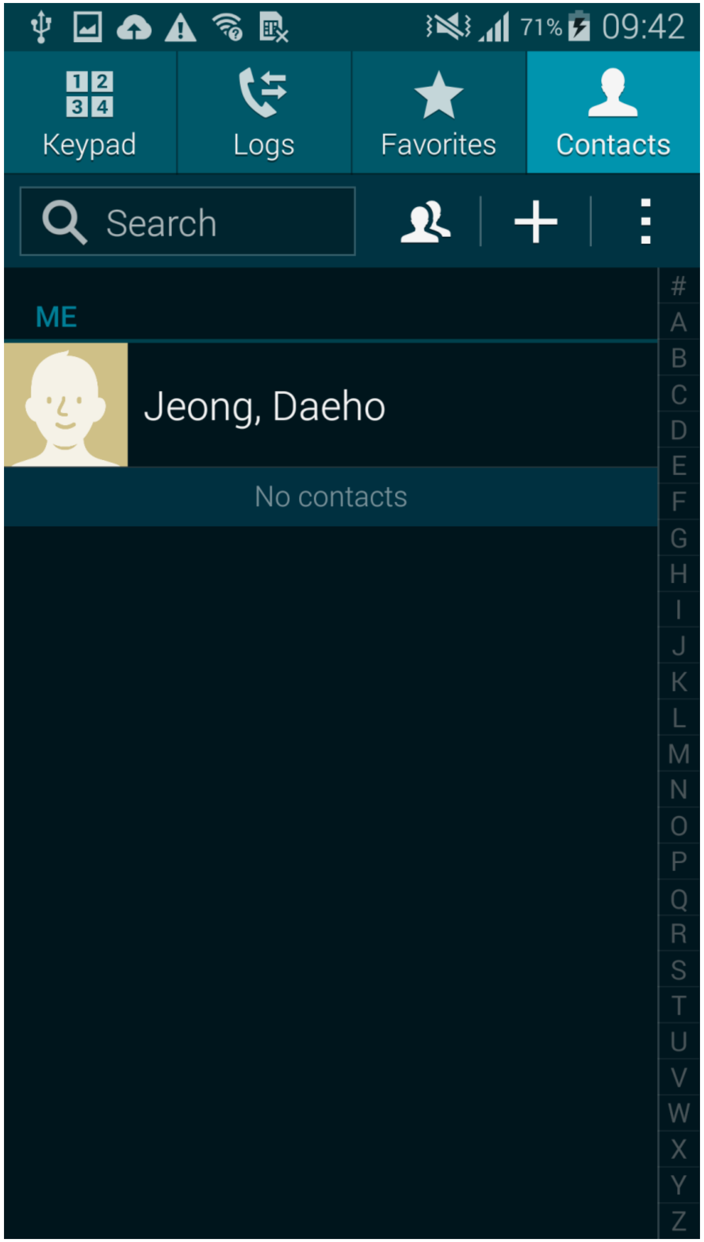

72%
09:41
09:41
Tue, February 10
매탄3동 23°C
08/08 08:38
Google
Email
Camera
Play Store
Google
Phone
Contacts
Messages
Internet
Apps

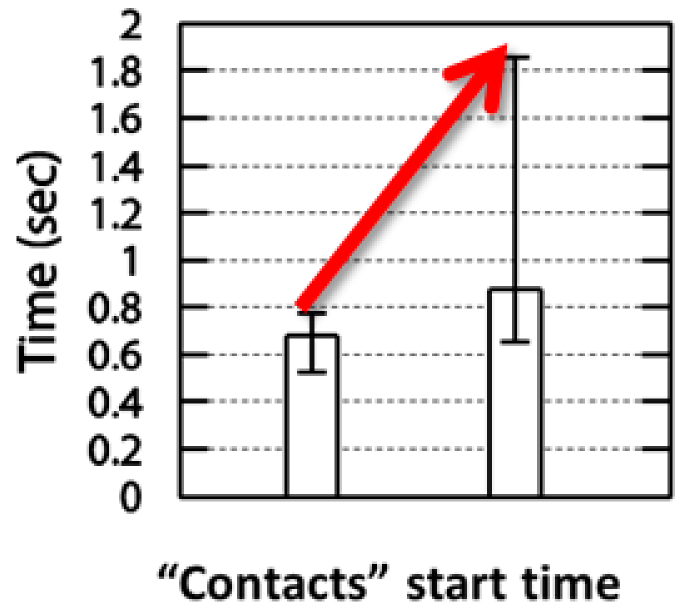

72% 09:41
09:41
Tue, February 10
매탄3동 23°C
08/08 08:38
Google
Email
Camera
Play Store
Google
TOUCH!!
Phone
Internet
Apps


72% 09:41
09:41
Tue, February 10
매탄3동 23°C
08/08 08:38
Google
Email
Camera
Play Store
Google
Phone
Contacts
Messages
Internet
Apps


71% 09:42
Keypad
Logs
Favorites
Contacts
Search
ME
Jeong, Daeho
No contacts


Time (sec)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
"Contacts" start time



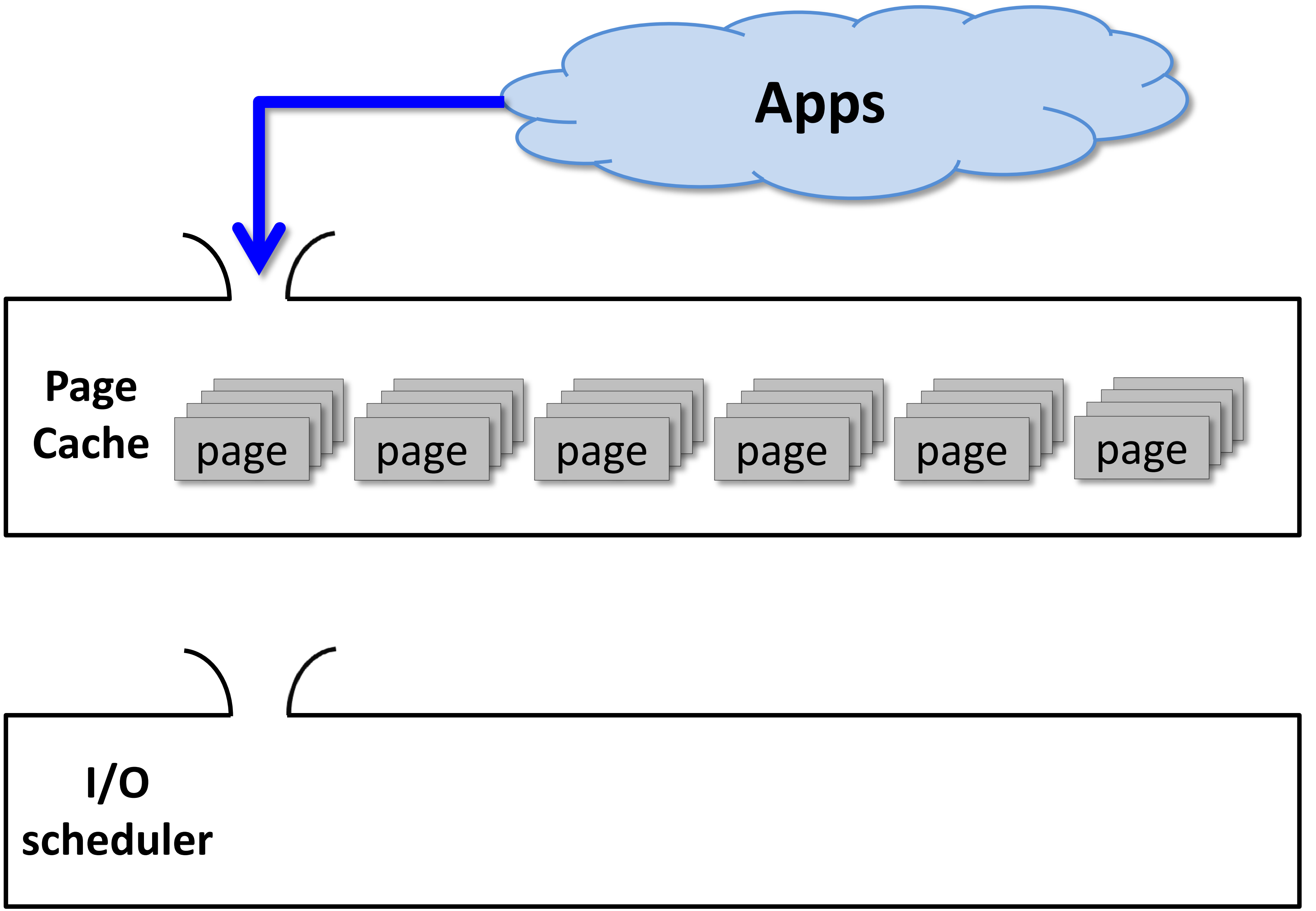


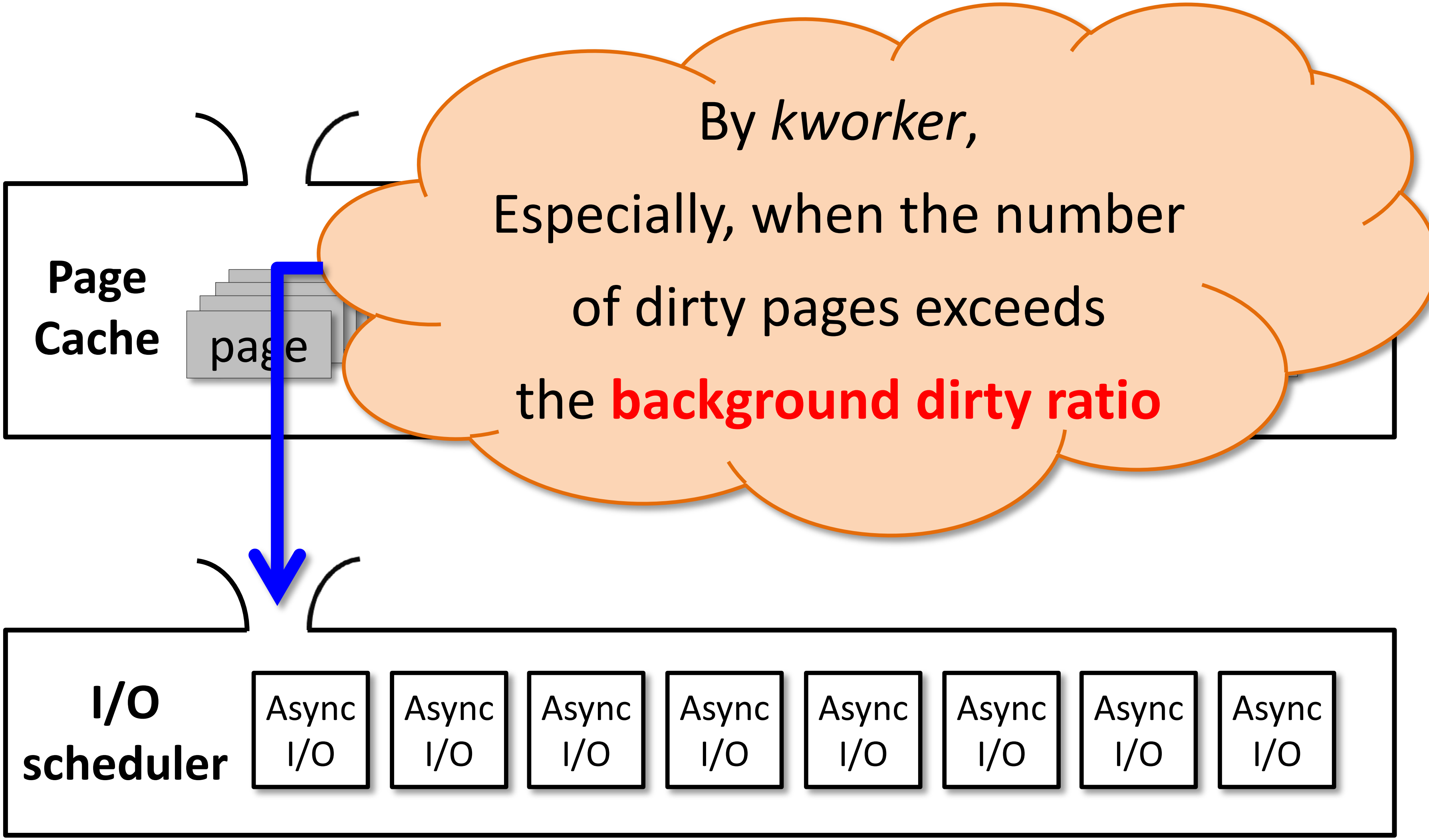
- CPU?
- Memory?
- I/O?

Linux Approaches in I/O Scheduling

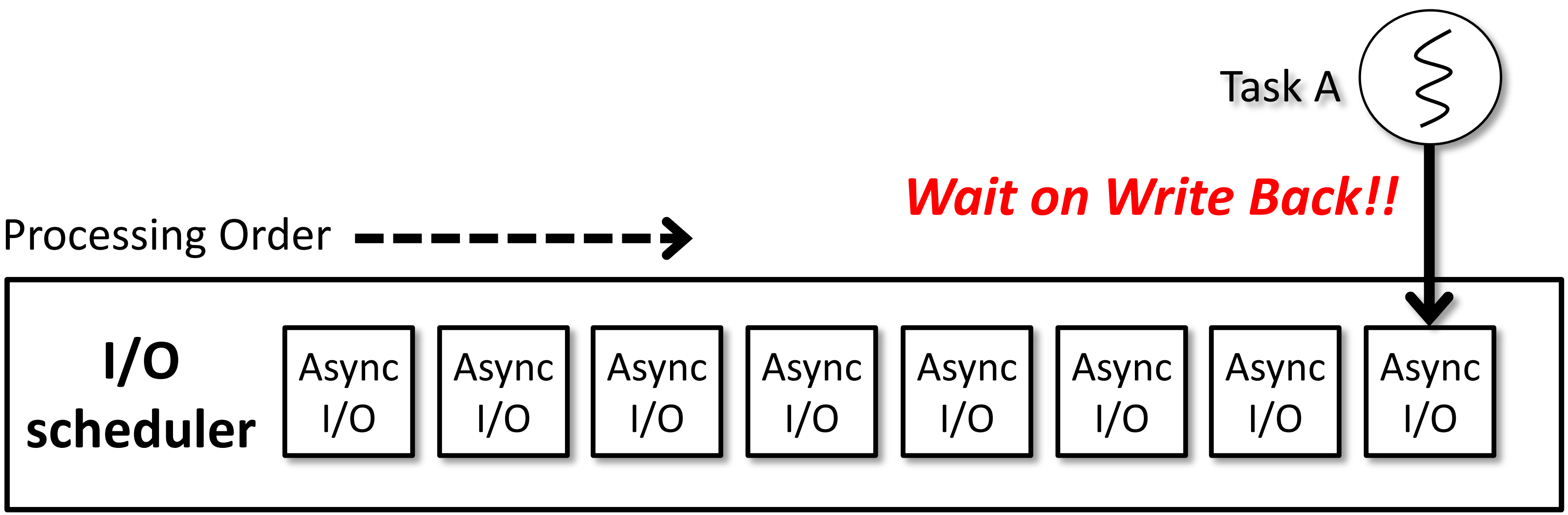
- Block layer
  - Classify I/O into {**SYNC** || **ASYNC**}

- CFQ I/O scheduler
  - **SYNC** queues have larger time slices than **ASYNC**
  - A **SYNC** queue per a process
    (vs. An **ASYNC** queue is shared)
  - Set a limit for **ASYNC** requests that can be dispatched in a single time slice
  - A new **SYNC** req. preempts other **ASYNC** req.


Apps
Page
Cache
page
page
page
page
page
page
I/O
scheduler

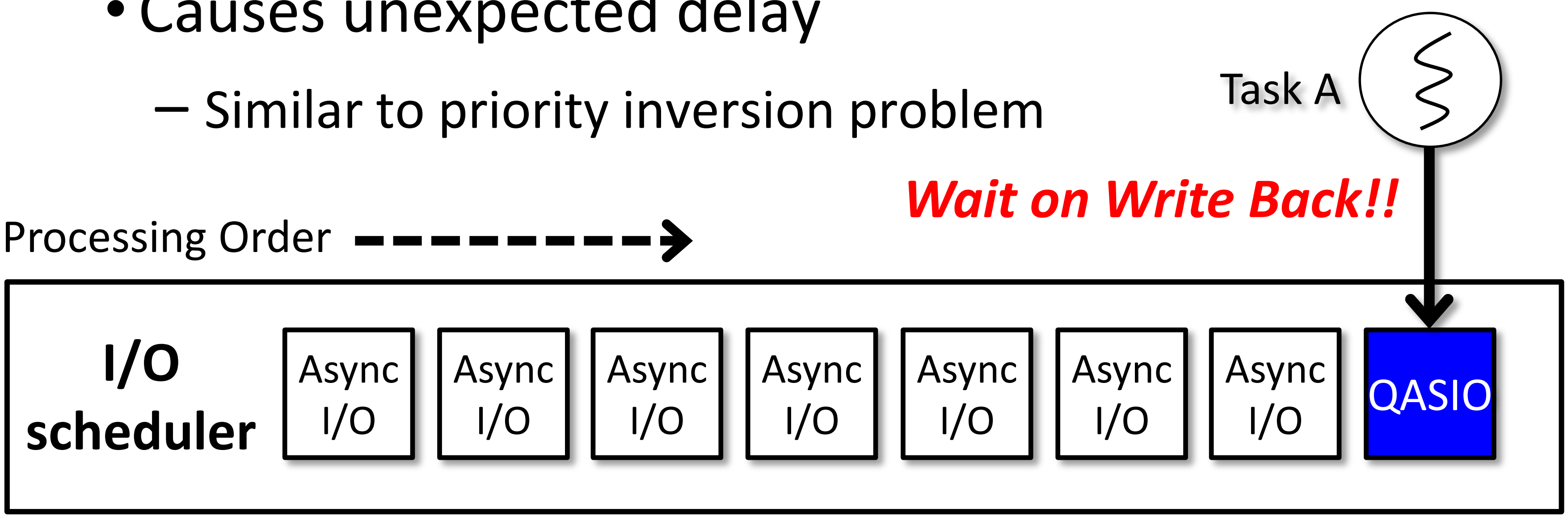

By *kworker*,
Especially, when the number
of dirty pages exceeds
the **background dirty ratio**
Page Cache
page
I/O scheduler
Async I/O
Async I/O
Async I/O
Async I/O
Async I/O
Async I/O
Async I/O
Async I/O

What if a process waits for
the completion of *asynchronous I/O*?



Quasi-Asynchronous I/O (QASIO)

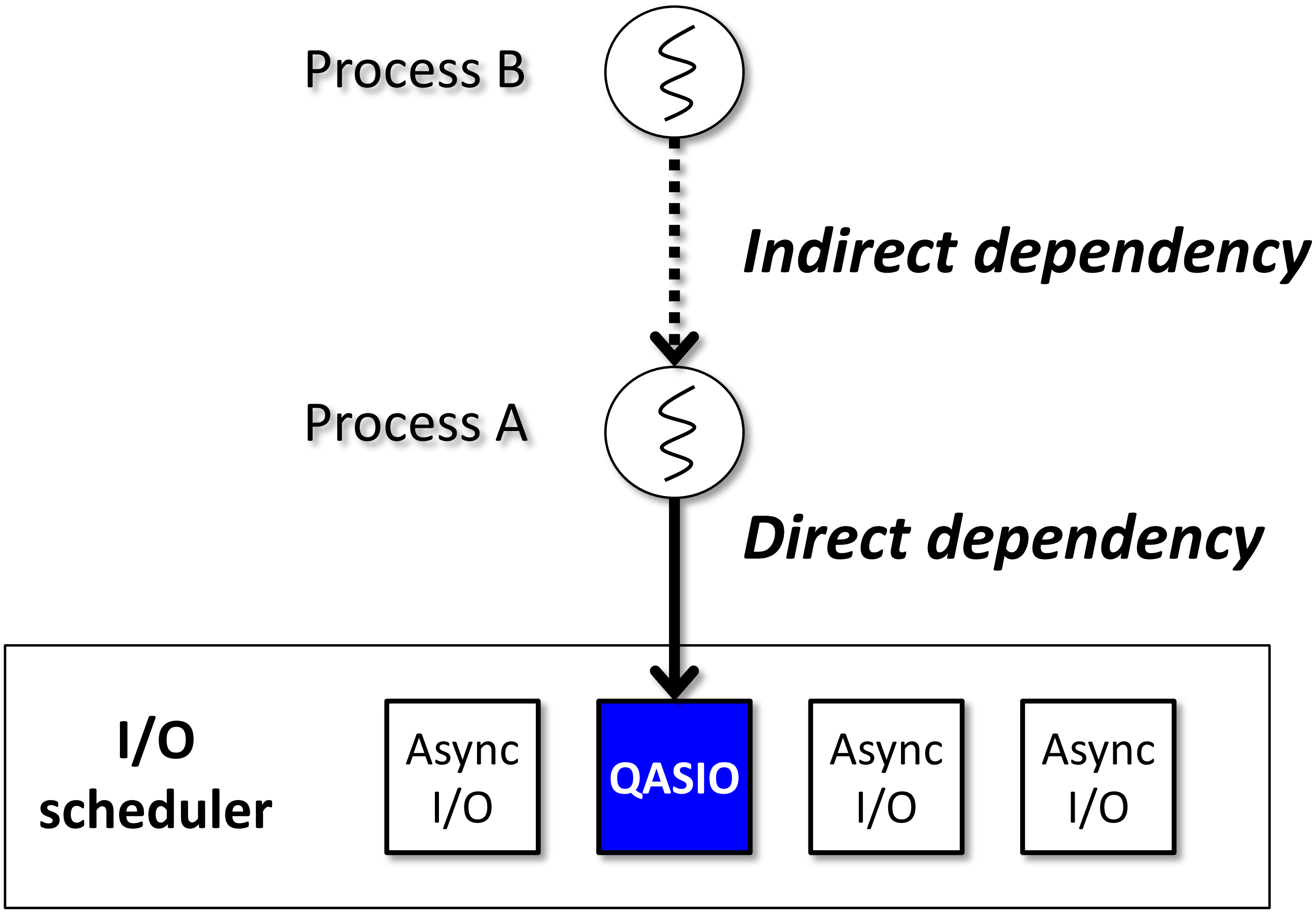
- Quasi-Asynchronous I/O
  - Issued *asynchronously*, but should be treated as *synchronous* I/O
  - Detected *at run time*
  - Causes a problem such as *priority inversion problem*
    - Causes unexpected delay
      - Similar to priority inversion problem



Outline

- Definition of QASIO
- Dependencies on QASIO
- Impact by QASIO
- Implementation Overview
- Evaluation
- Conclusions

Dependencies on QASIO



Types of Dependencies on QASIO

- Direct Dependencies
  - $\mathbf{D}_{meta}$: *When modifying a metadata page*
  - $\mathbf{D}_{data}$: *When modifying a data page*
  - $\mathbf{D}_{sync}$: *When guaranteeing data to be written back*
  - $\mathbf{D}_{discard}$: *When completing discard commands*

- Indirect Dependencies
  - $\mathbf{I}_{jhandle}$: *When unable to obtain a journal handle*
  - $\mathbf{I}_{jcommit}$: *When unable to complete* `fsync()`

Types ... QASIO

- Direct D
  - $\mathbf{D}_{meta}$: *W*
  - $\mathbf{D}_{data}$: *W*
  - $\mathbf{D}_{sync}$: *W* ... *en back*
  - $\mathbf{D}_{discard}$: ... *ds*
- Indirect
  - $\mathbf{I}_{jhandle}$: *W* ... *ndle*
  - $\mathbf{I}_{jcommit}$: ...)

Refer to our paper

to a QASIO *at run time* when a task gets blocked due to the asynchronous I/O. For better responsiveness, QASIOs should be given the higher priority than other (true) asynchronous I/Os.

4.2 Types of Dependencies on QASIO

Each task can have a *direct* or an *indirect* dependency on QASIOs. The direct dependency occurs when the execution of a task is blocked due to (quasi-) asynchronous I/Os. Figure 3 illustrates the situation where task A has a direct dependency on a QASIO. To identify when such a dependency exists, we have conducted an extensive analysis of the Linux kernel and the dynamic I/O patterns generated by file system calls. According to our analysis, we have identified the following four types of direct dependencies on QASIOs:

- *When modifying a metadata page* ($\mathbf{D}_{meta}$): This type of dependency can occur when a task invokes a file system call which modifies a metadata page (such as inodes, group descriptors, block bitmaps, inode bitmaps, and directory entries in Ext4). The target metadata page, made dirty by itself or the other tasks, may be already submitted as an asynchronous I/O by the *kworker* thread.
- *When modifying a data page* ($\mathbf{D}_{data}$): When a task appends data partially within a data page, it can be blocked since the target data page may be already flushed out asynchronously by the *kworker* thread. The task cannot proceed its execution until the data page hits the storage.
- *When guaranteeing data to be written back* ($\mathbf{D}_{sync}$): A task needs to wait for the completion of asynchronous I/Os when synchronizing or truncating the previously-issued file data in `fsync()` or `truncate()`. When performing `fsync()`, all the previous buffered writes are issued synchronously *as long as* they are still in the page cache. If calling `fsync()` is late or there are too many dirty pages in the page cache, some of them can be already flushed out as asynchronous I/Os. In this case, `fsync()` should wait until those asynchronously-issued I/Os are done.
- *When completing discard commands* ($\mathbf{D}_{discard}$): Currently, the *jbd2* kernel thread issues discard commands *asynchronously* for deallocated blocks, unlike other journal blocks which are issued synchronously. Hence, its execution is blocked on every journal commit until all the discard commands are completed. This delay in turn can affect the responsiveness of the foreground task (cf. $\mathbf{I}_{jcommit}$).

Sometimes, it is also possible that the execution of a task is being delayed due to another task that has a direct dependency on QASIOs. For example, Figure 3 shows

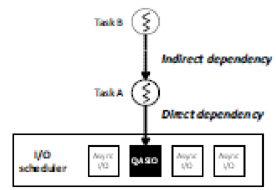


Figure 3: Direct and indirect dependency on QASIO

that task B is blocked because task A cannot make any progress due to the direct dependency on a QASIO. In this case, we call that task B has an indirect dependency on a QASIO. Typically, this situation arises when task A is blocked holding a resource that task B requires. Unlike the direct dependency, it is difficult to list all the possible types of indirect dependencies since the delay due to QASIOs can be propagated to other tasks in diverse and complicated ways. However, we found the following two types of indirect dependencies related to the JBD2 journaling which has a significant impact on the performance.

- *When unable to obtain a journal handle due to* $\mathbf{D}_{meta}$ *or* $\mathbf{D}_{data}$ ($\mathbf{I}_{jhandle}$): In Ext4, a task should obtain a journal handle to modify a metadata page or a data page. As mentioned before, the task can be blocked if the target page is already issued asynchronously, creating the $\mathbf{D}_{meta}$ or $\mathbf{D}_{data}$ type of dependency on QASIOs. Sooner or later, the transaction including the journal handle is started to be committed but the transaction is locked because the blocked task holds the journal handle. In this case, another task which attempts to perform any file operation is blocked since it fails to obtain a new journal handle.
- *When unable to complete* `fsync()` *due to* $\mathbf{D}_{discard}$ ($\mathbf{I}_{jcommit}$): This type of indirect dependency is observed only for the task that invokes `fsync()`. The `fsync()` system call needs to wait until the journal commit is completely done to ensure that the metadata of the corresponding file is written into the storage device. However, the processing time of the journal commit can be significantly prolonged since the *jbd2* kernel thread usually has a direct dependency of $\mathbf{D}_{discard}$ due to asynchronously-issued discard commands.

Whenever a foreground task interacting with a user has a direct or an indirect dependency on QASIOs, its execution has nondeterministic *hiccups* and the user can encounter sluggish responsiveness. Despite that there is room for additional I/Os in memory and request queues, the processing of system calls is blocked by the stacked

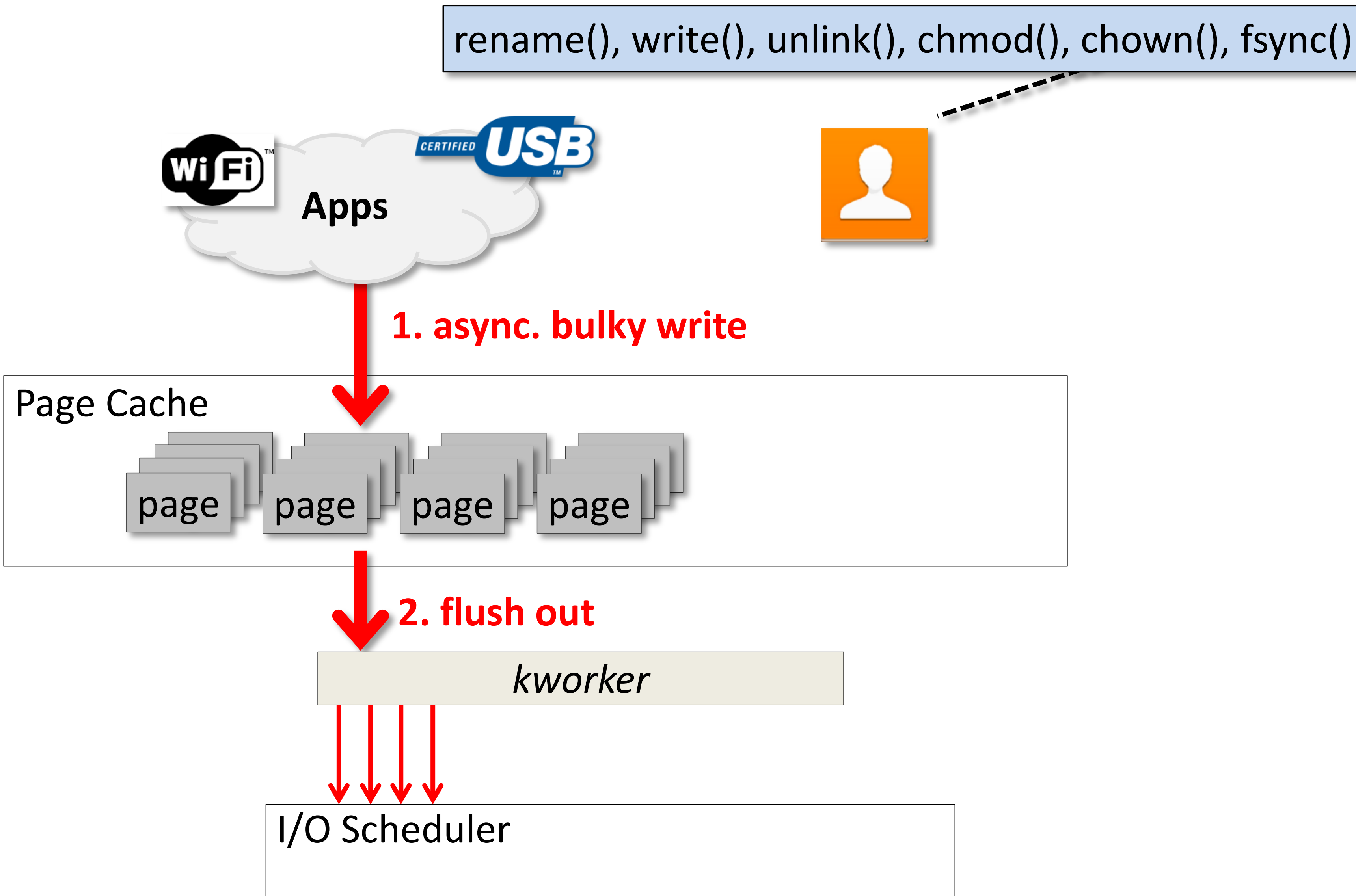
5

Types of Dependencies on QASIO

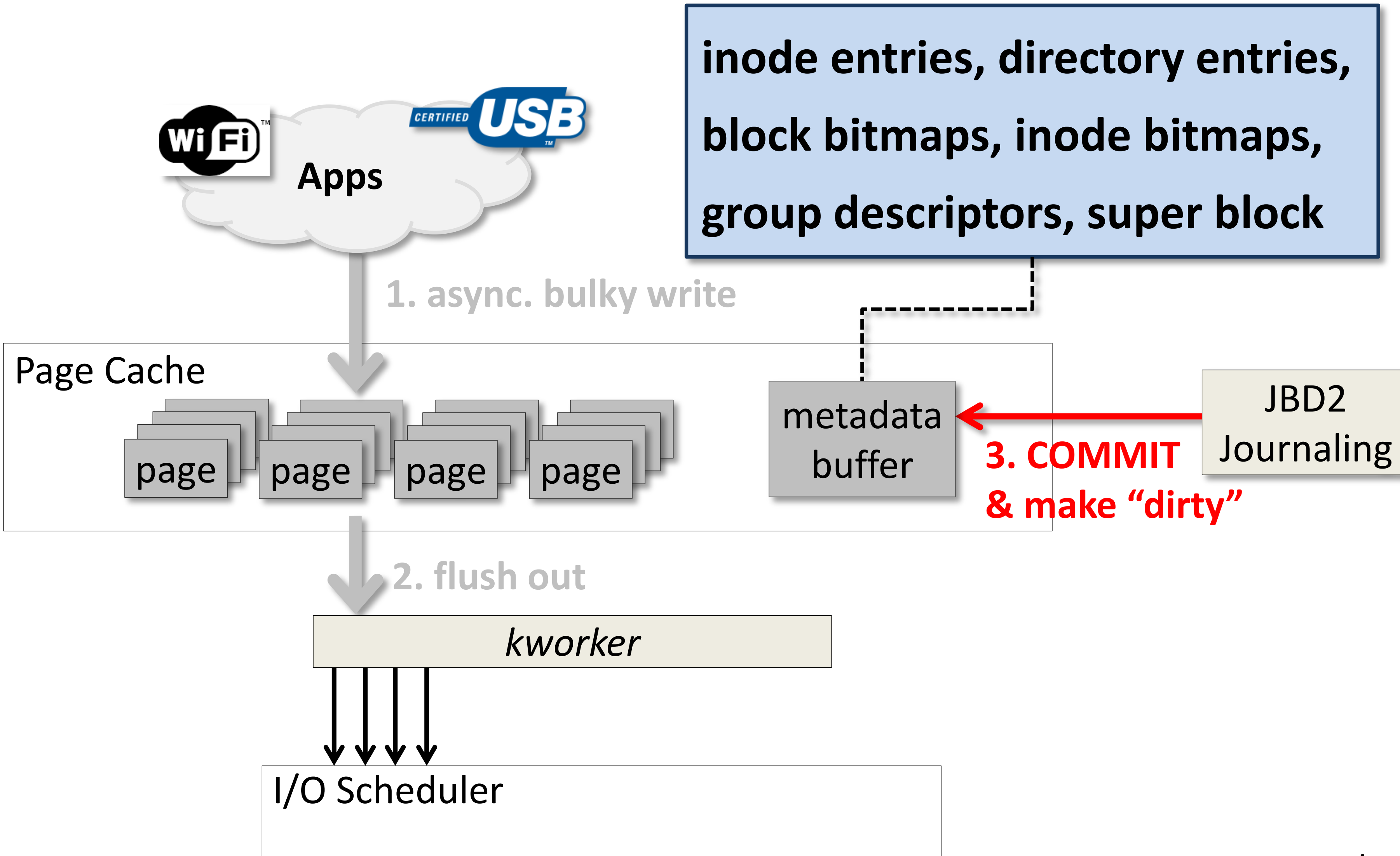
- Direct Dependencies
  - $\mathbf{D}_{meta}$: *When modifying a metadata page*
  - $\mathbf{D}_{data}$: *When modifying a data page*
  - $\mathbf{D}_{sync}$: *When guaranteeing data to be written back*
  - $\mathbf{D}_{discard}$: *When completing discard commands*

- Indirect Dependencies
  - $\mathbf{I}_{jhandle}$: *When unable to obtain a journal handle*
  - $\mathbf{I}_{jcommit}$: *When unable to complete* `fsync()`

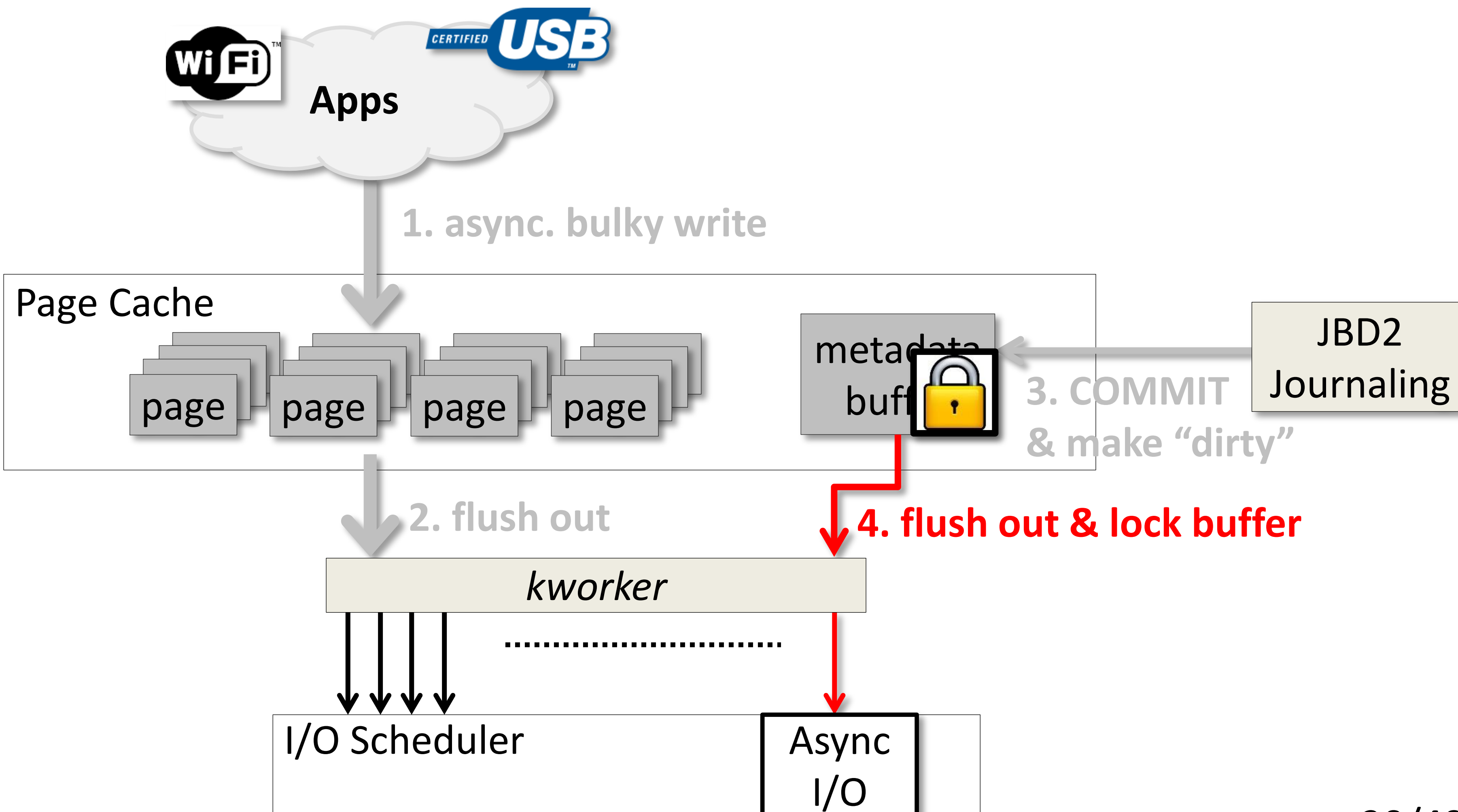
Case Study of D_{meta}



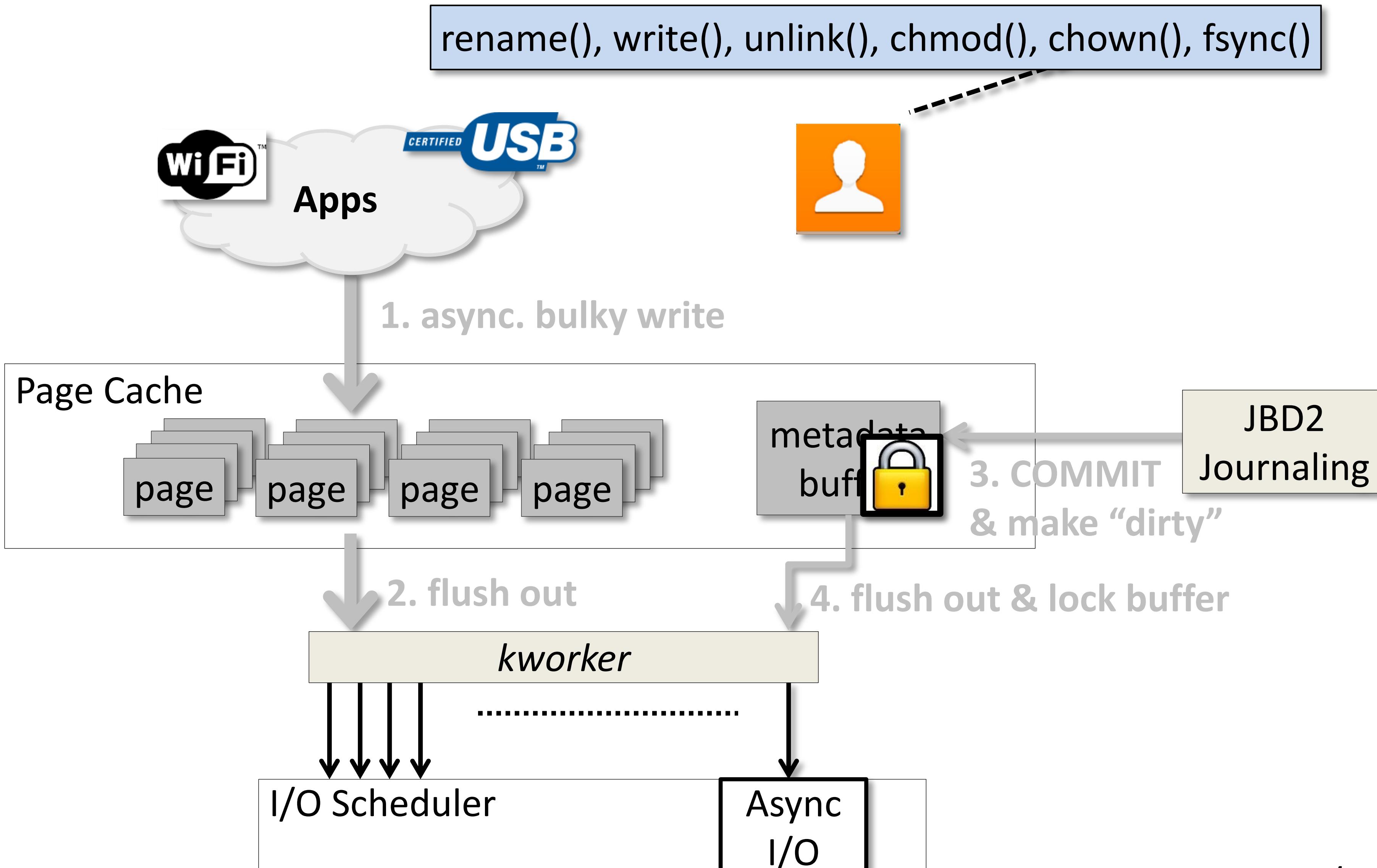
Case Study of D_{meta}



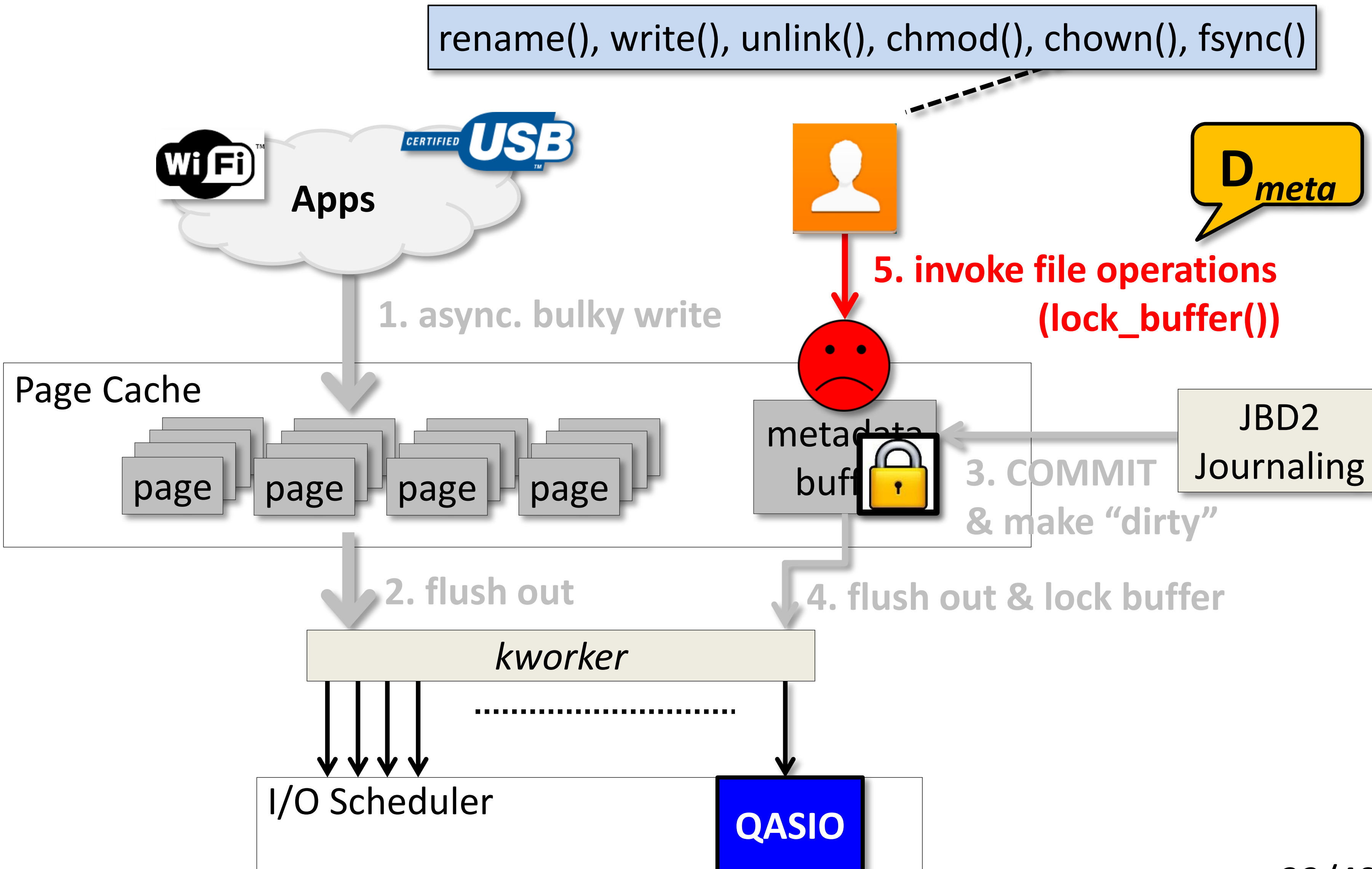
Case Study of D_{meta}



Case Study of D_{meta}

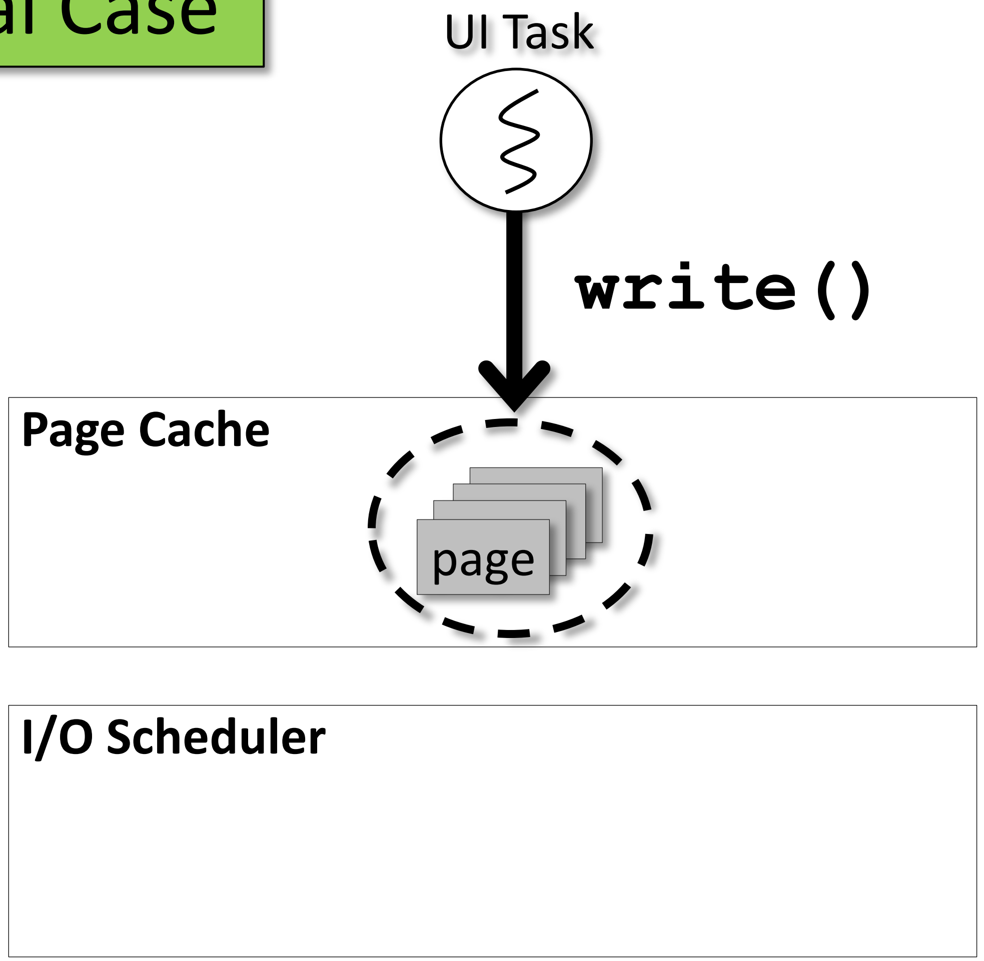


Case Study of D_{meta}



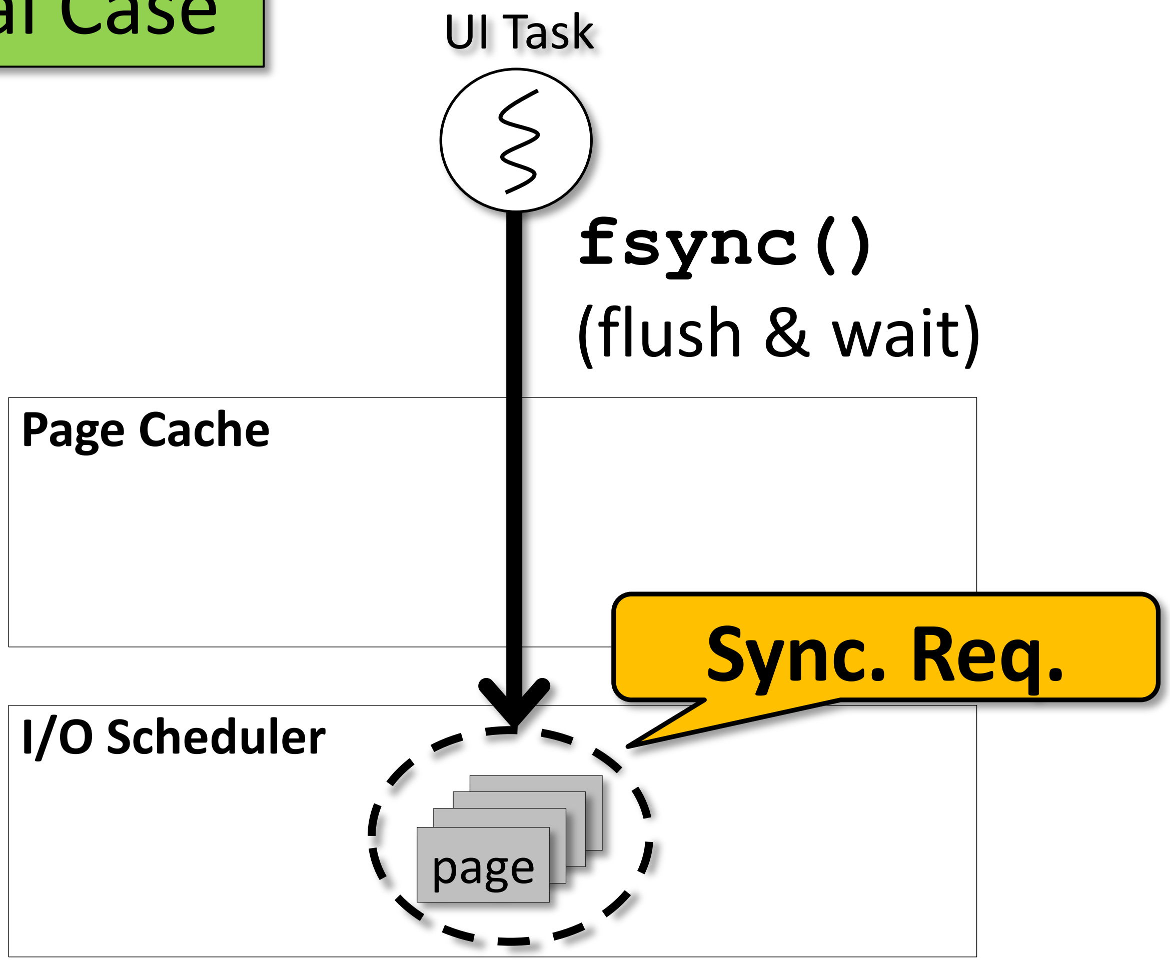
Case Study of D_{sync}

Normal Case

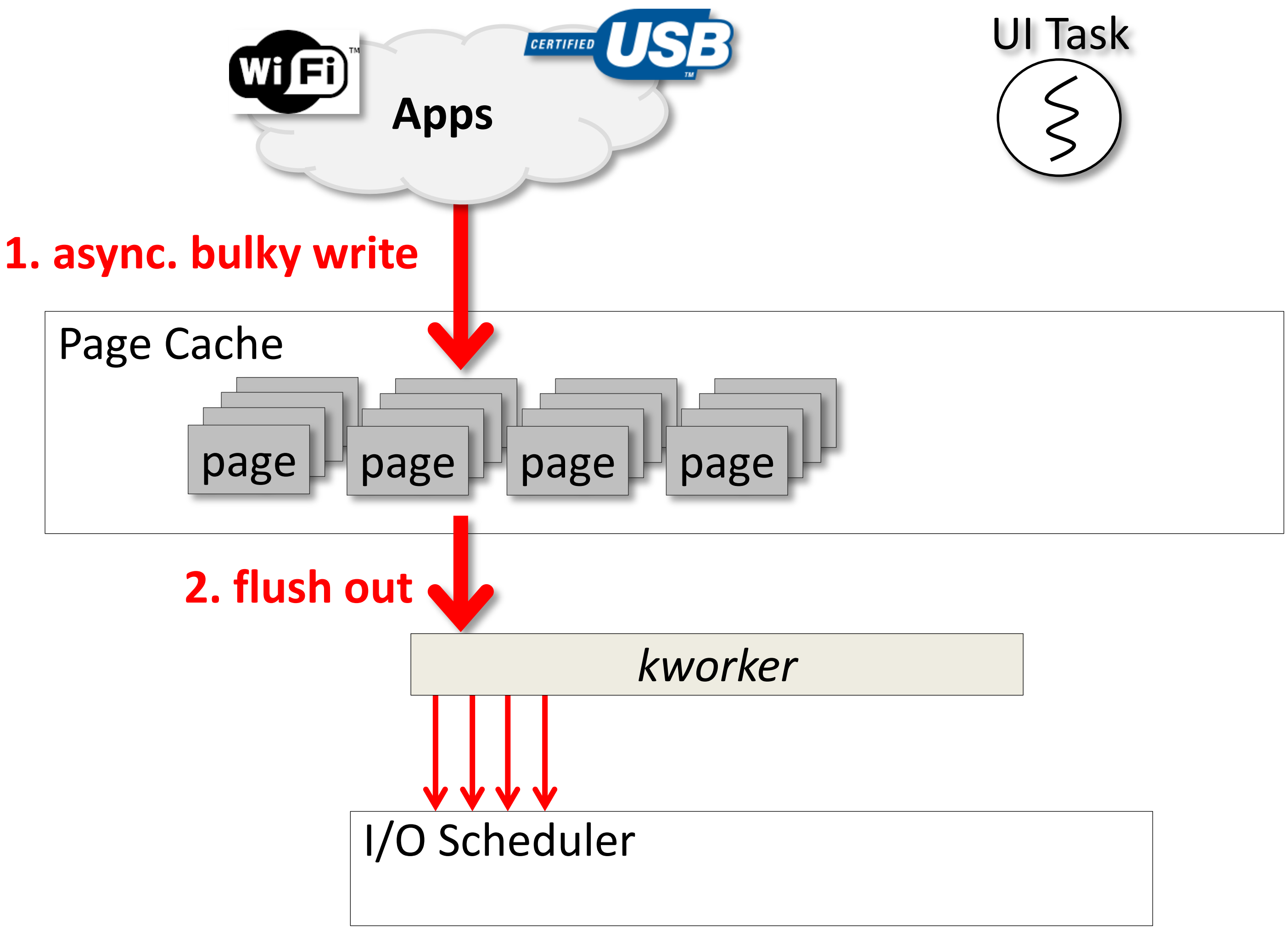


Case Study of D_{sync}

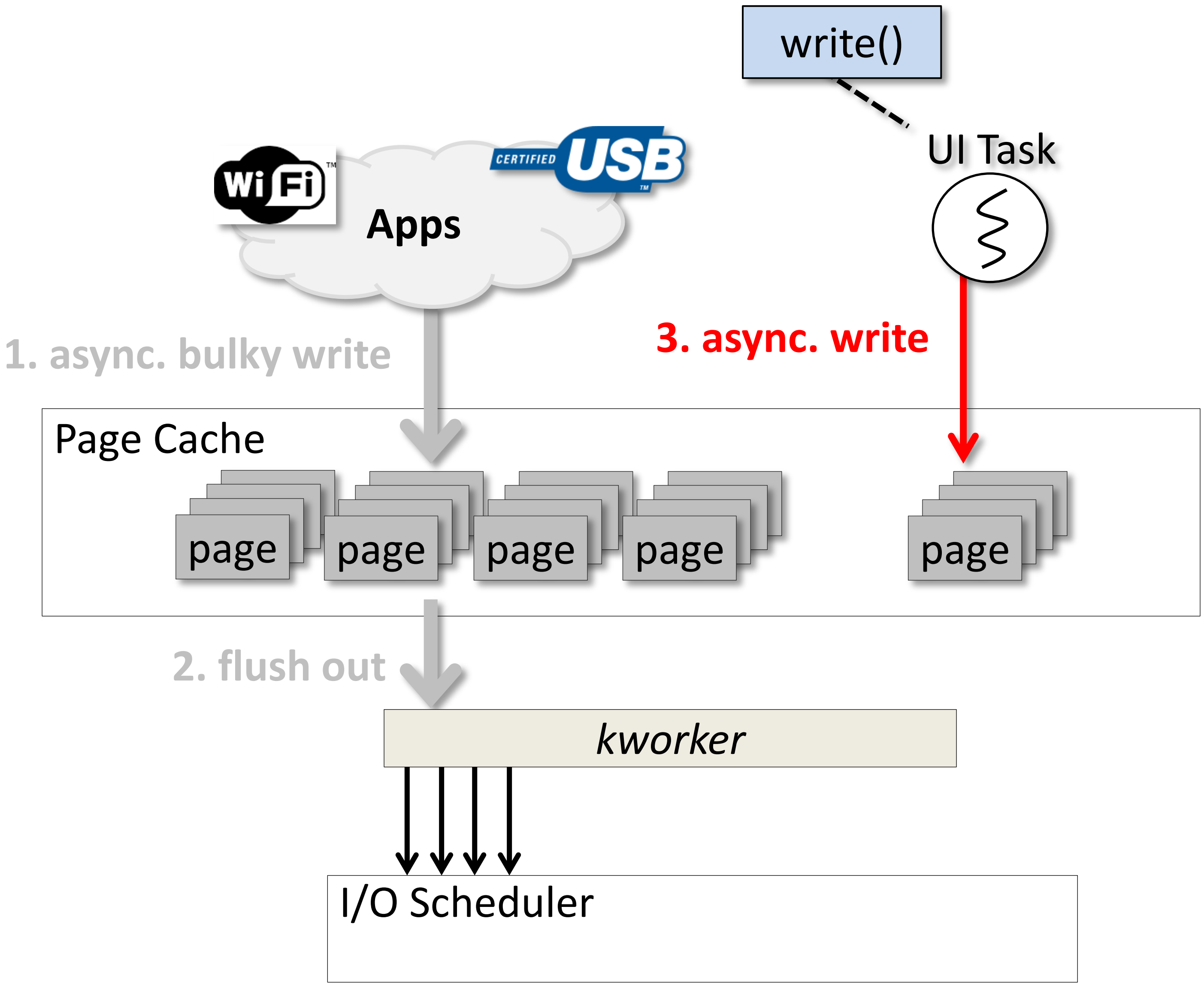
Normal Case



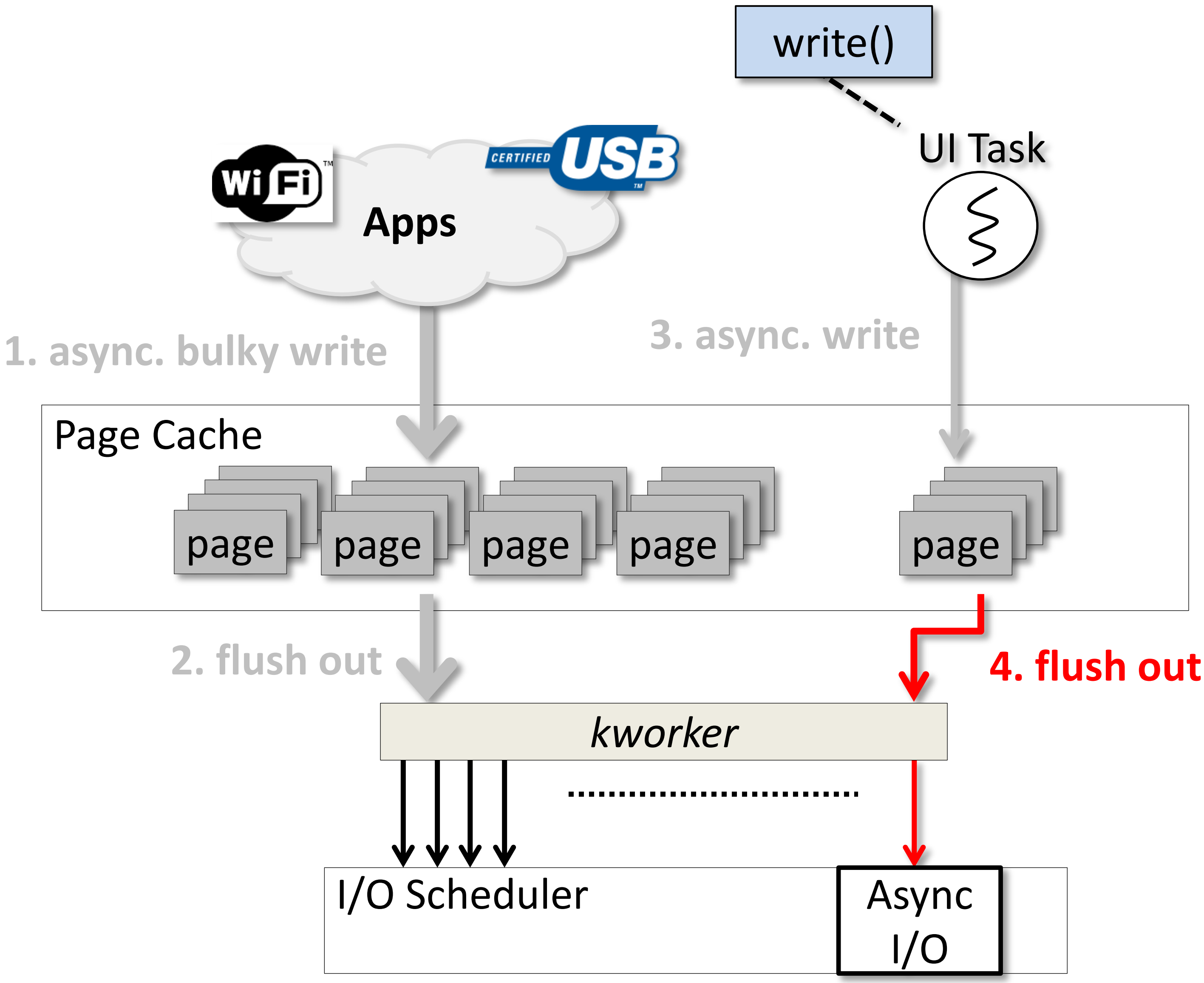
Case Study of D_{sync}



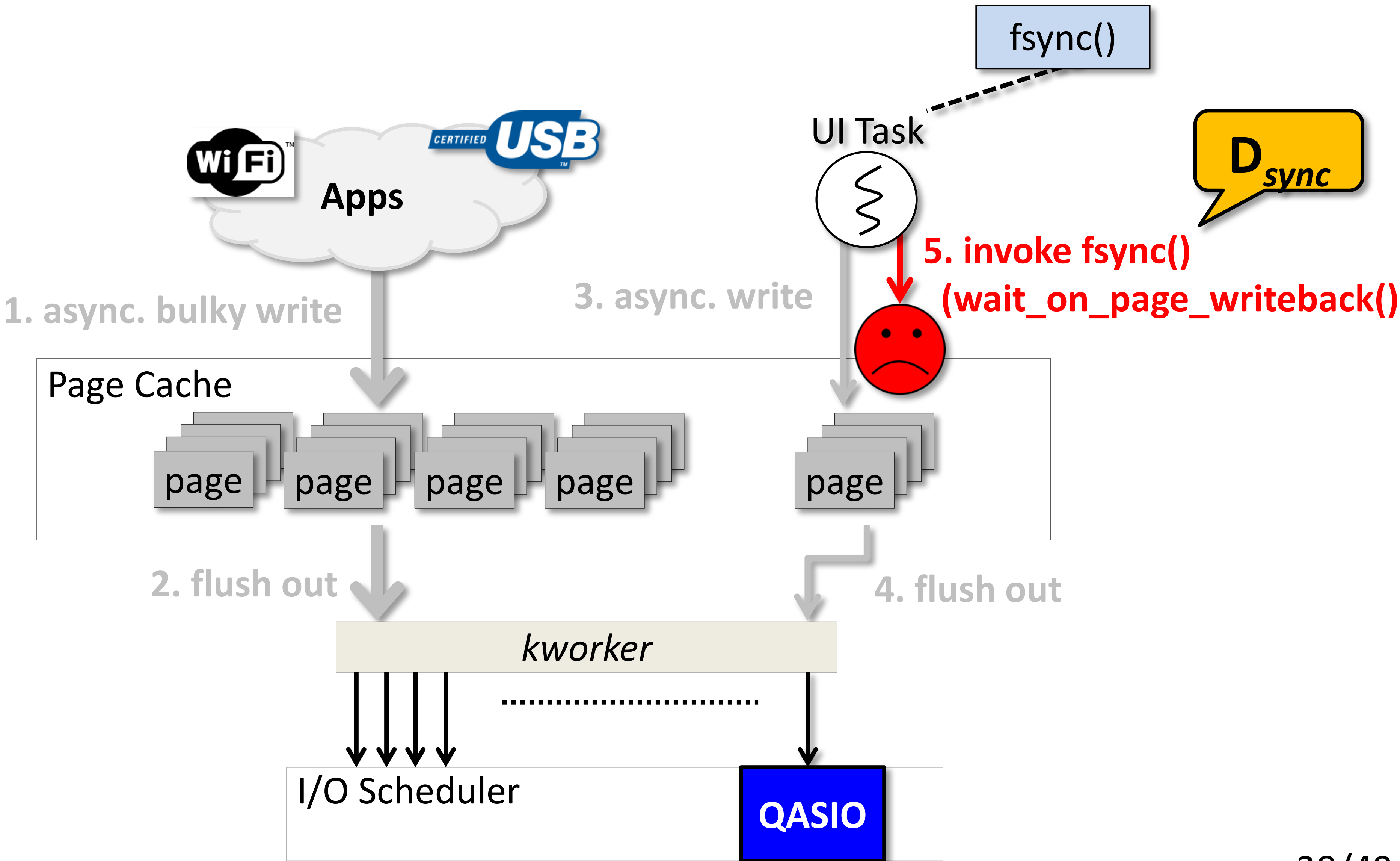
Case Study of D_{sync}



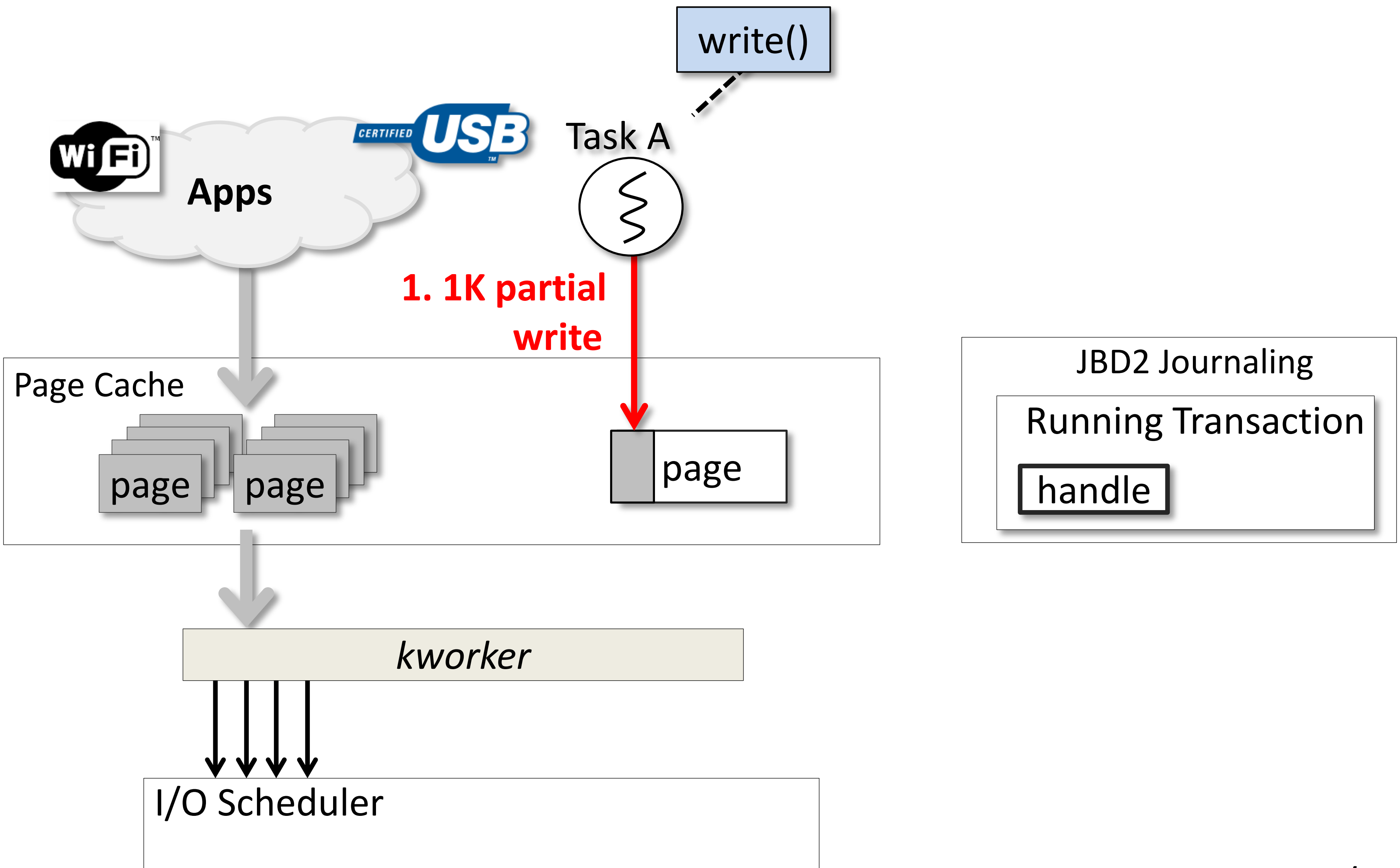
Case Study of D_{sync}



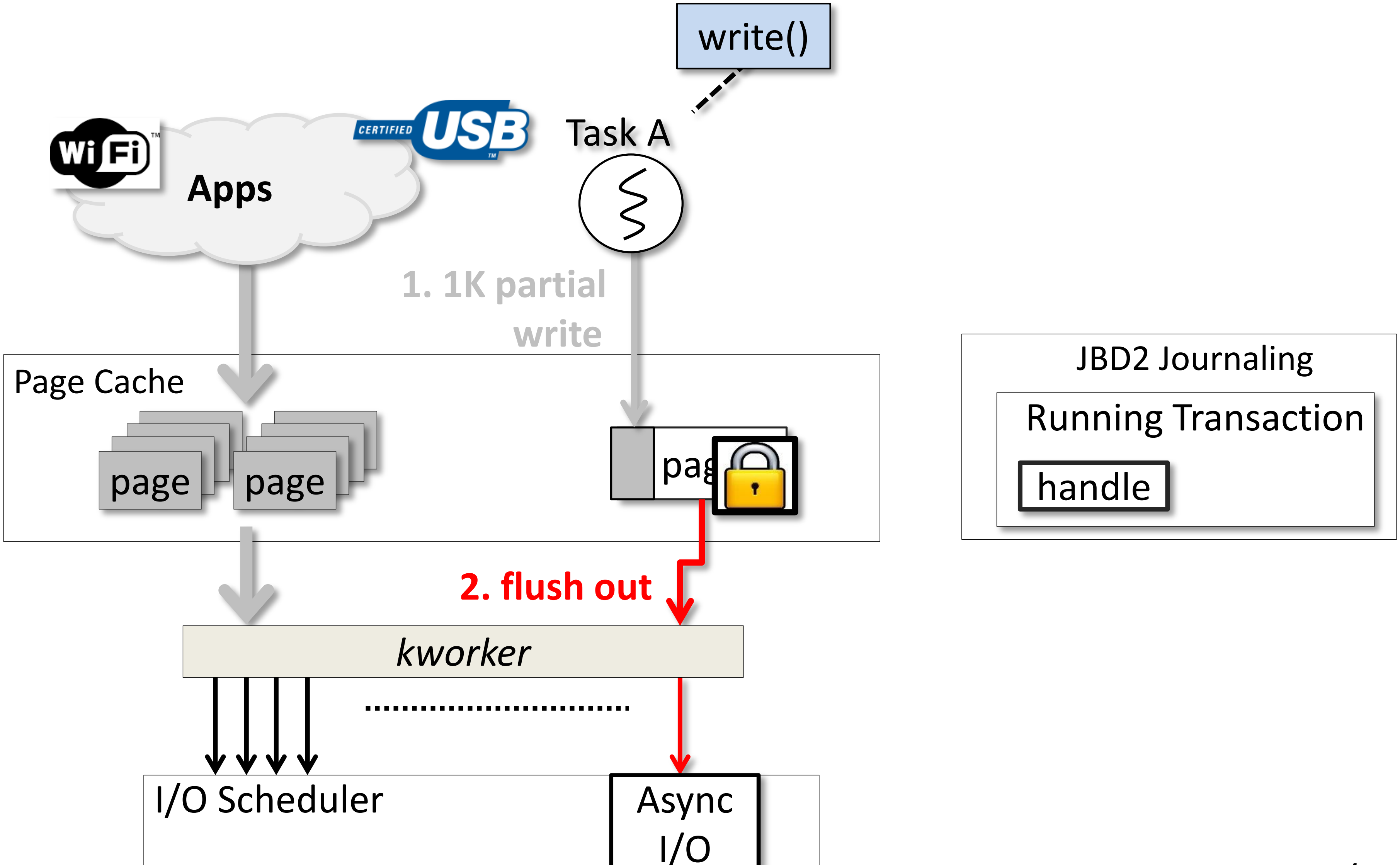
Case Study of D_{sync}



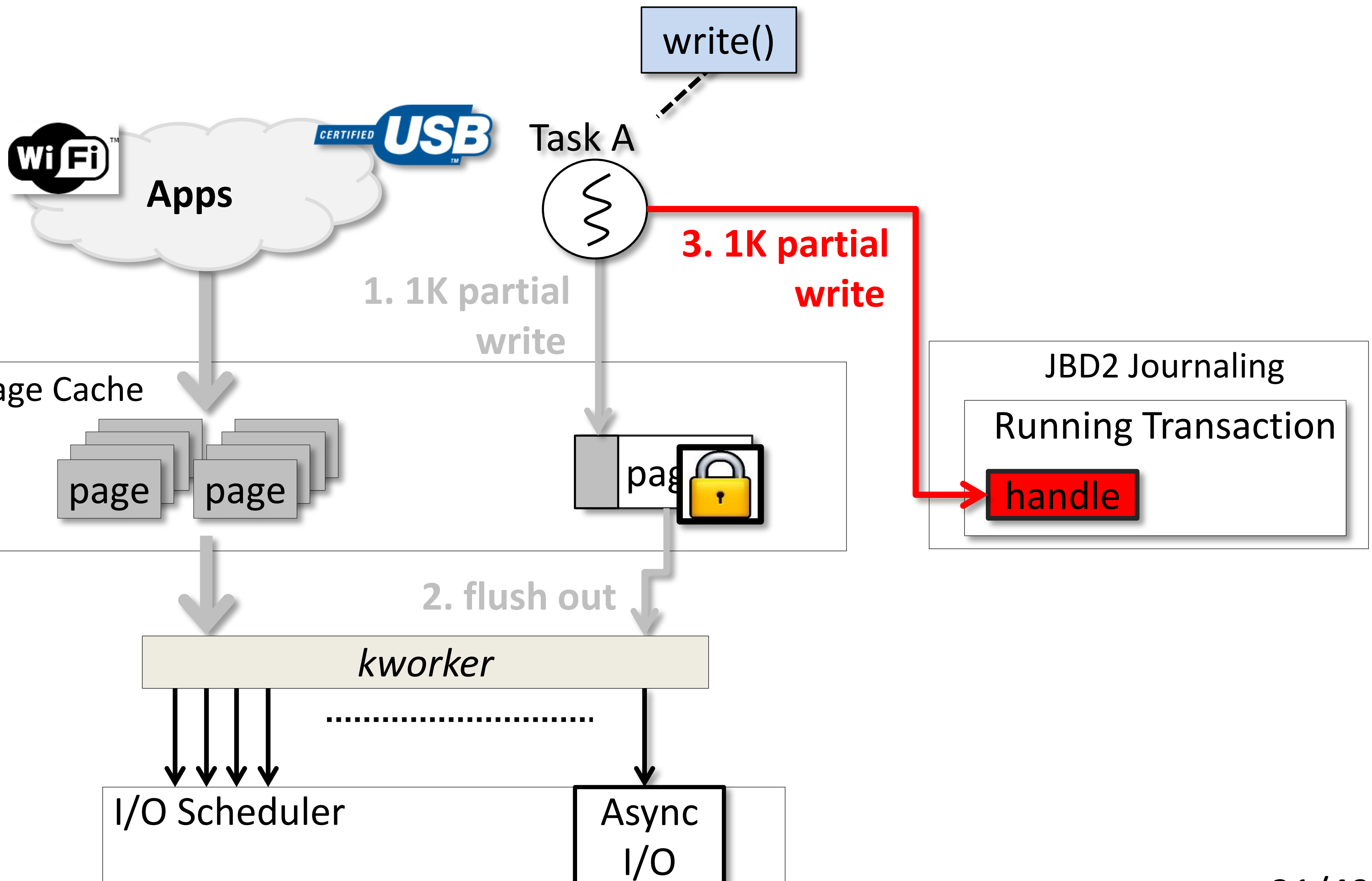
Case Study of $I_{jhandle}$



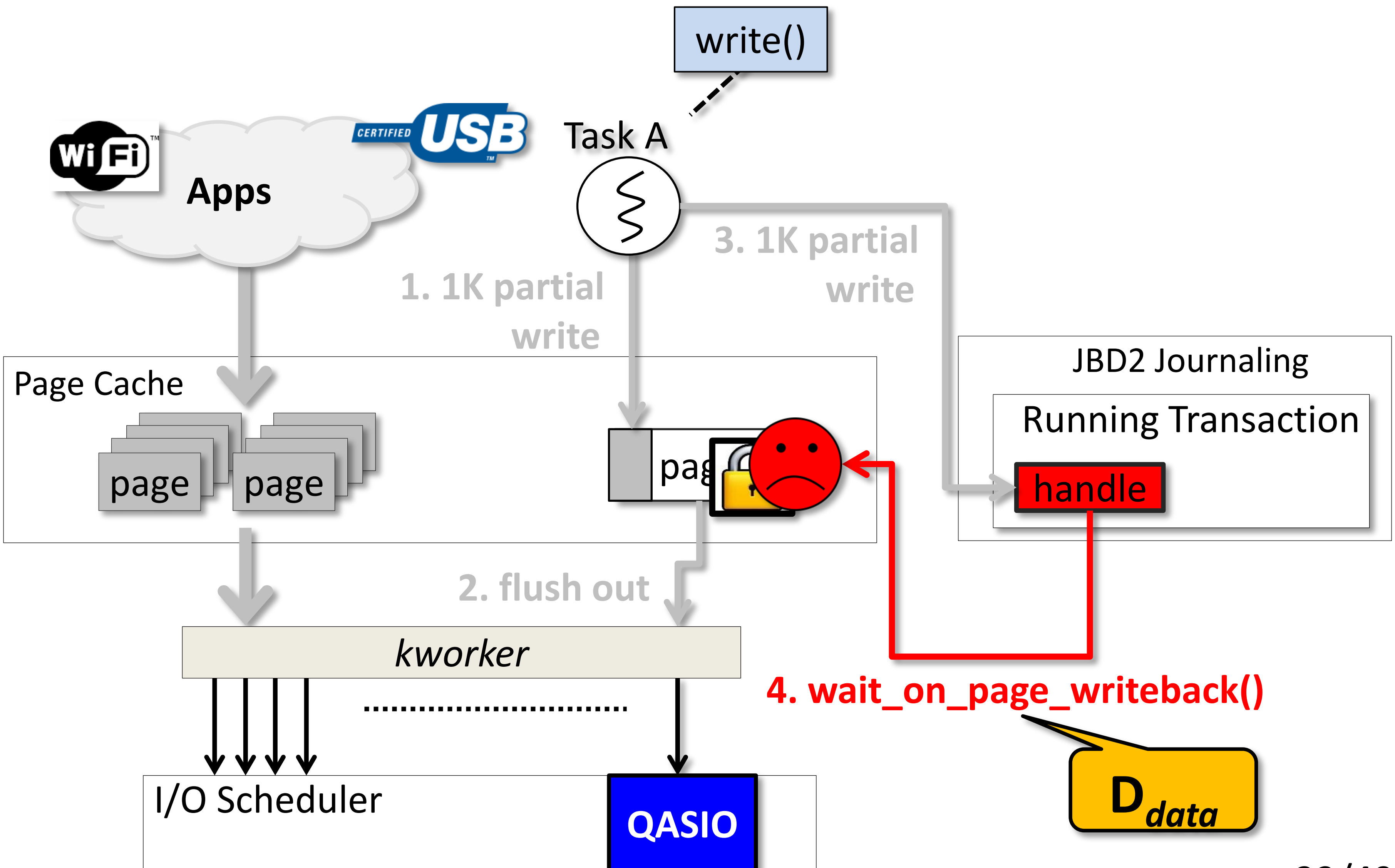
Case Study of $I_{jhandle}$



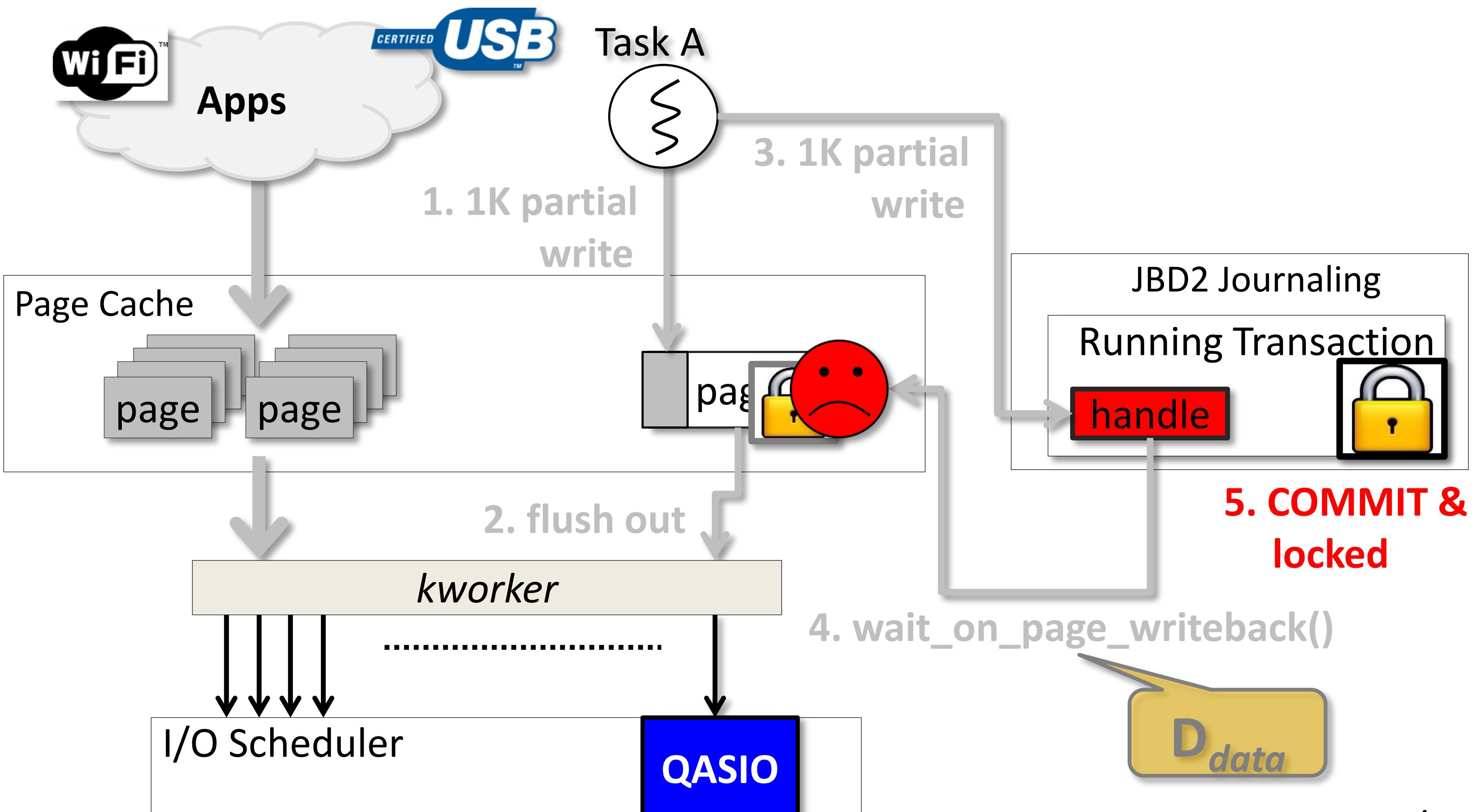
Case Study of $I_{jhandle}$



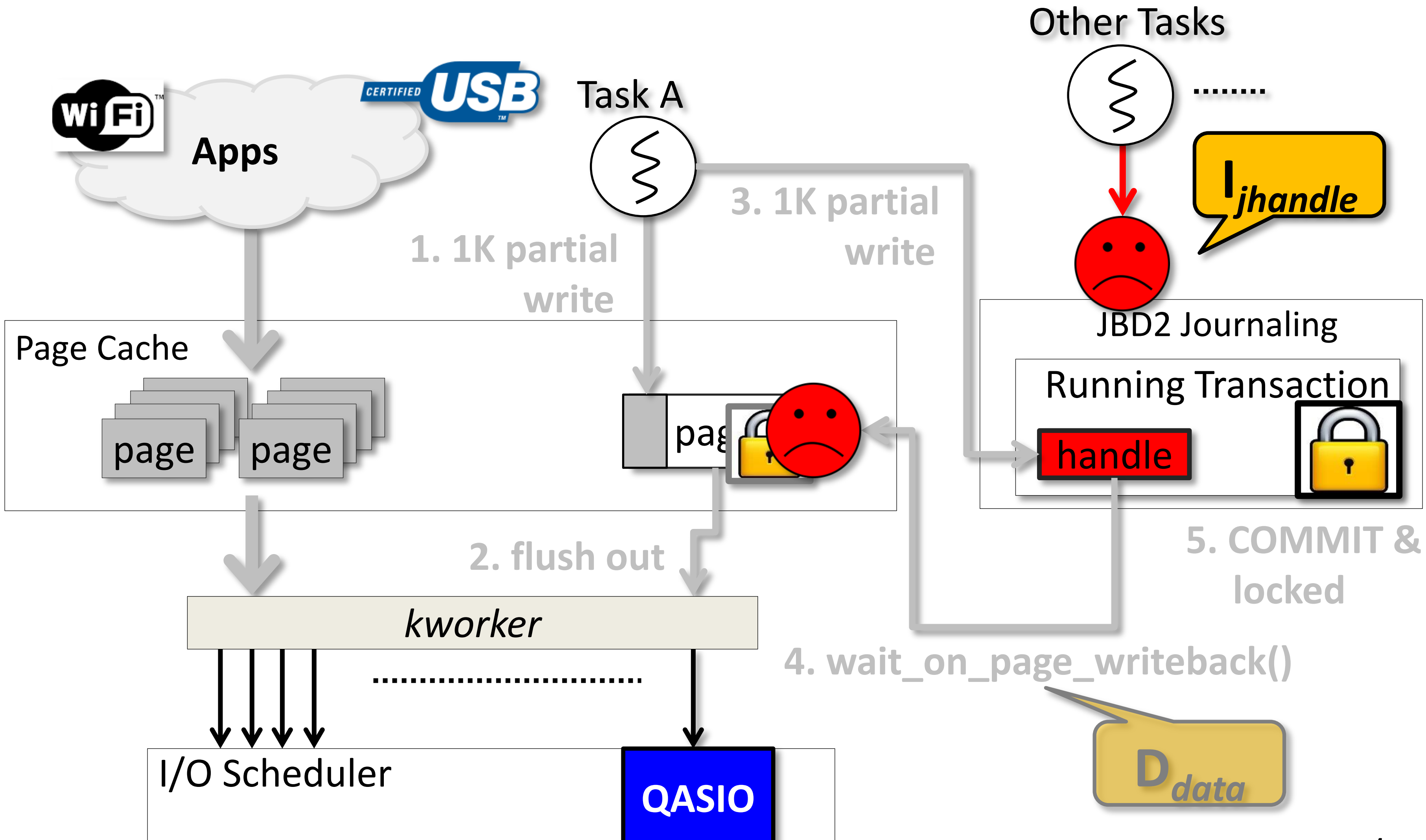
Case Study of $I_{jhandle}$



Case Study of $I_{jhandle}$



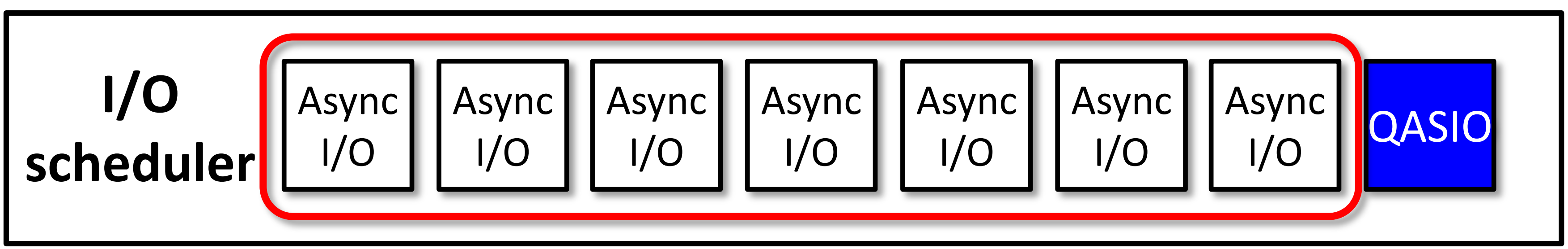
Case Study of $I_{jhandle}$



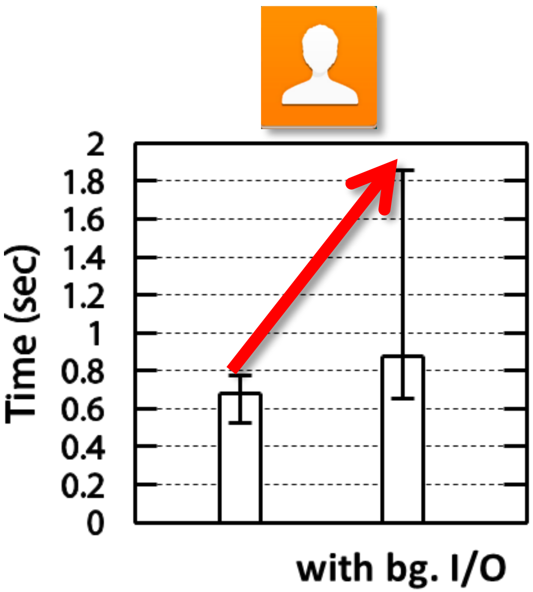
How Severe is the delay by QASIO?

- The delay by QASIOs depends on
  - The number of outstanding requests
  - The maximum number of requests
  - I/O performance of underlying storage device
    - A file system call can be blocked for
      - Over 1 second on an MLC eMMC (S.W.: 57.4 MB/s)
      - Over 4 seconds on a TLC eMMC (S.W.: 26.0 MB/s)

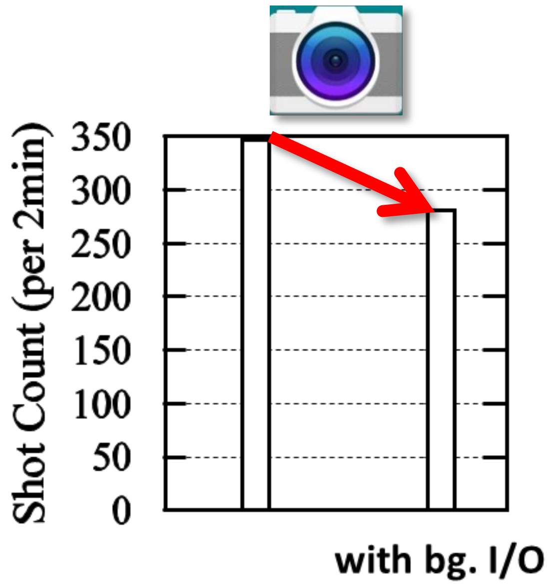
S.W. => Sequential Write Bandwidth



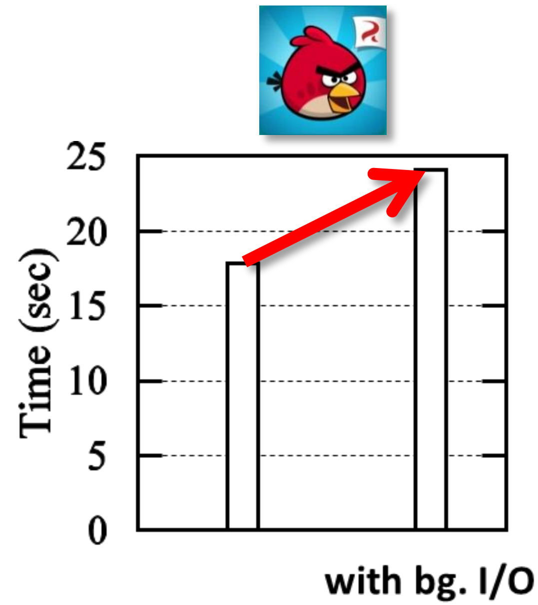
Degradations by QASIOs in Real-Life Scenarios



"Contacts" start time



Burst shot performance



"Angry Birds" install time

App start time is slowed down by **2.4x** in the worst case

Shot count is decreased by **19%**

App install time is increased by **35%**

How to Boost a QASIO

- Just focus on Direct Dependencies
- Two requirements
  - *Req.(1): When a task is waiting for an asynchronous I/O's completion, the kernel gives information about QASIO to the I/O scheduler*
  - *Req.(2): The I/O scheduler should prioritize them among asynchronous I/Os based on the hint*

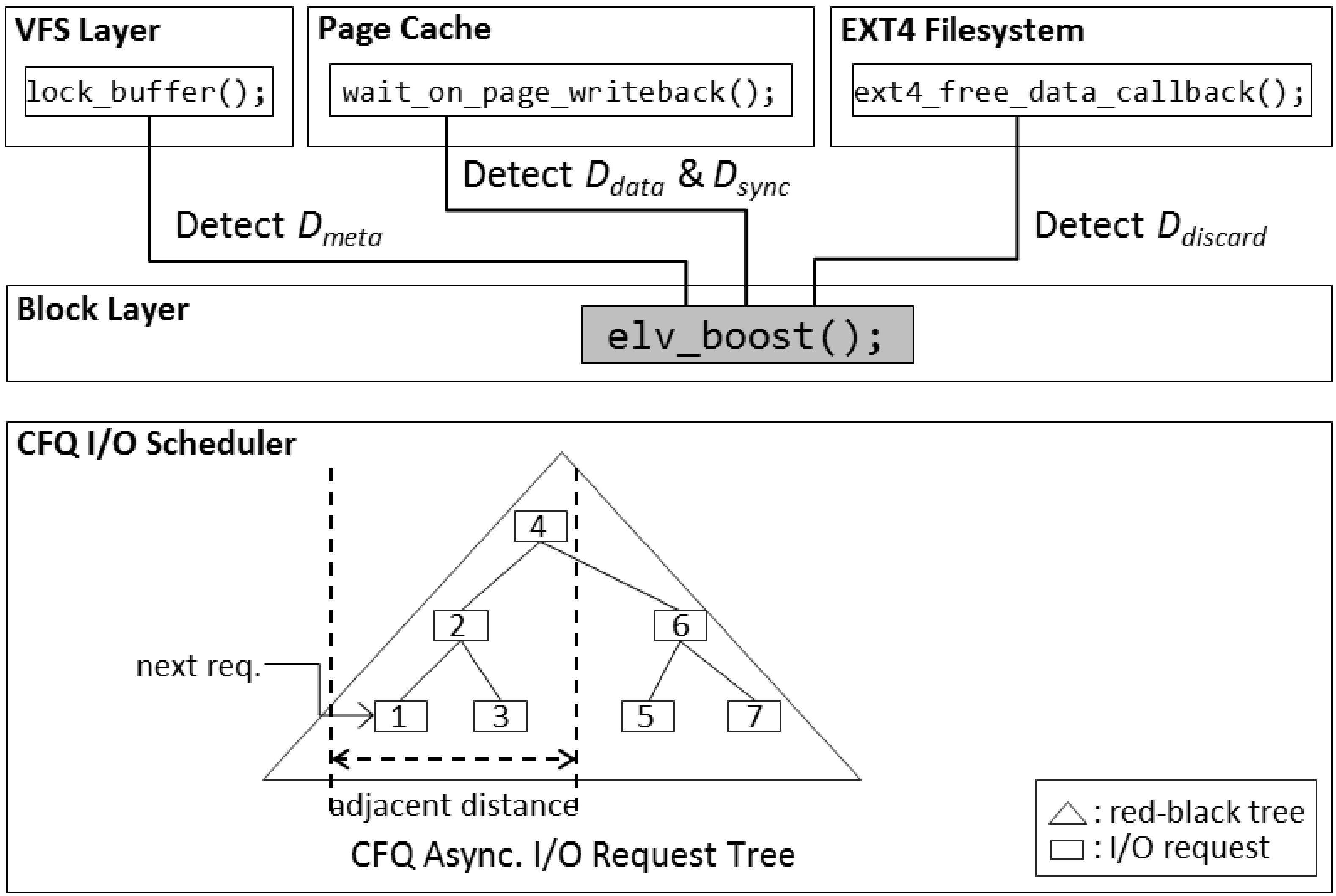
How to Boost a QASIO

- Just focus on Direct Dependencies
- Two requirements
  - *Req.(1): When a task is waiting for an asynchronous I/O's completion, the kernel gives information about QASIO to the I/O scheduler*

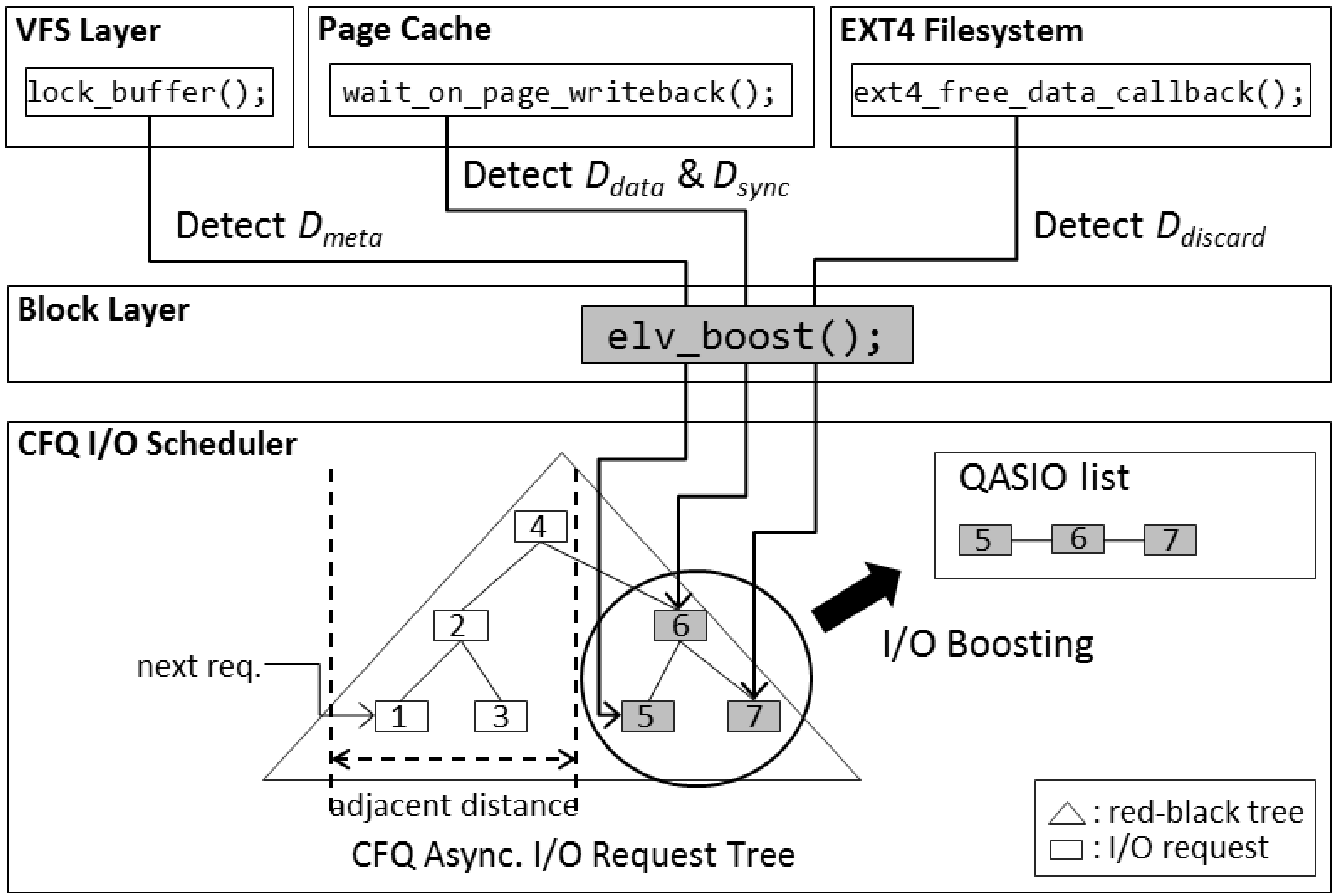
    => VFS, MM, FS, Block Layer
  - *Req.(2): The I/O scheduler should prioritize them among asynchronous I/Os based on the hint*

    => Each I/O Scheduler

Implementation Overview



Implementation Overview



Evaluations

- *Samsung Galaxy S5*
  - Exynos 5422 (quad Cortex-A15 & quad Cortex-A7)
  - 2GB DRAM
  - 16GB eMMC storage (S.W.: 54.5MB/s, R.W.: 10.4MB/s)
  - Android platform version 4.4.2 (KitKat)
  - Linux kernel version 3.10.9

- Evaluation methodology
  - Microbenchmarks, Real-life scenarios, Android I/O benchmarks

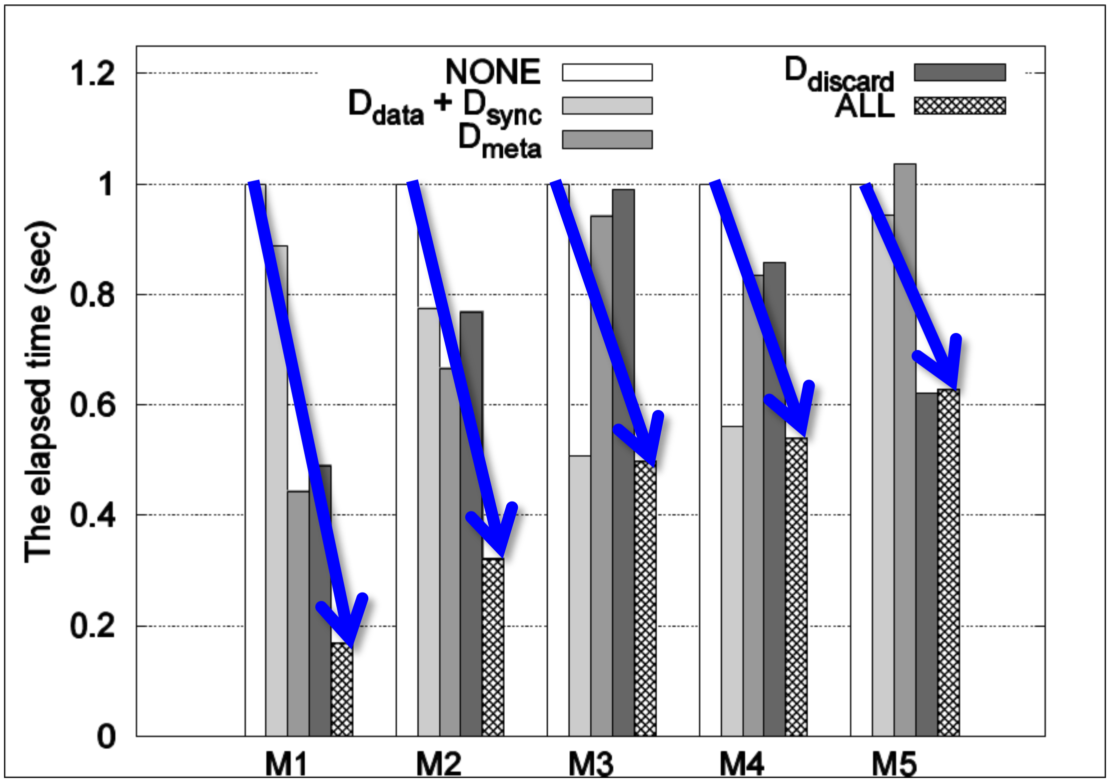
Microbenchmarks

- Five microbenchmarks (M1 ~ M5)

- M1
  - Iterates the creation of a 4KB file, 500 times
  - performs `fsync()` to each created file
  - `creat()` -> `write(`4KB`)` -> `fsync()` -> `close()`
  - Mimics the I/O pattern of DB

Microbenchmarks

- The normalized total elapsed time



The total elapsed time is reduced by up to **83.1%↓**

Microbenchmarks

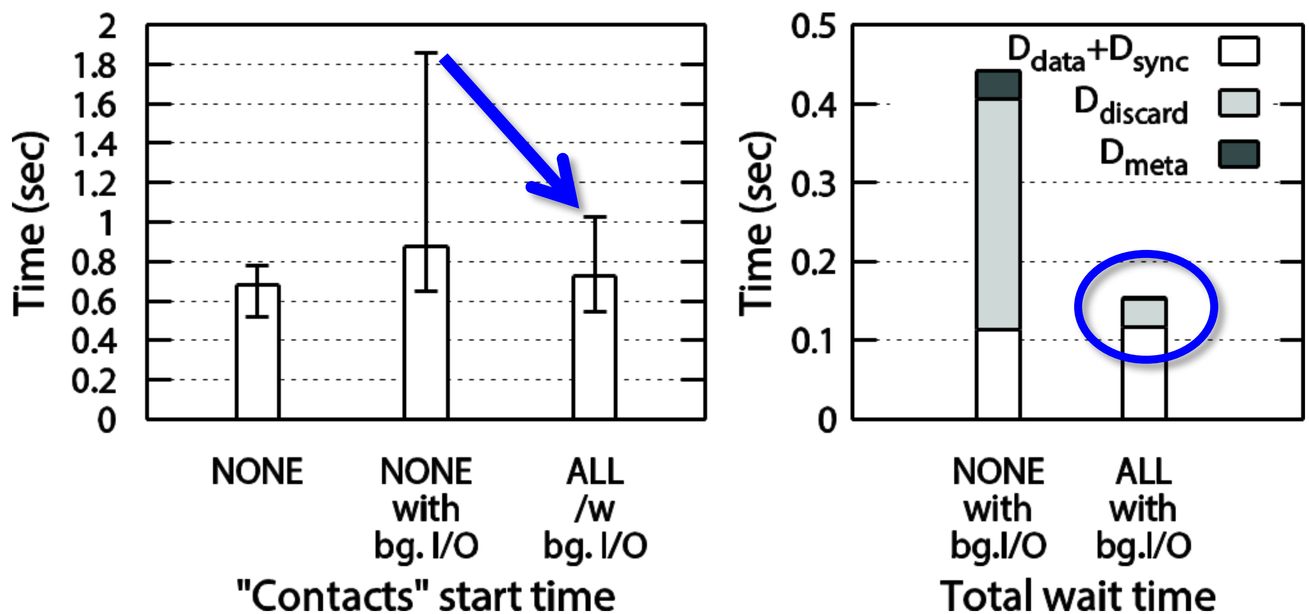
- The latency of key file system calls

Opt	M1				M2				M3		M4		M5	
	creat()		fsync()		creat()		fsync()		write()		truncate()		fsync()	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
NONE	119.57	1435.54	98.24	1119.62	109.01	1397.64	172.05	1417.82	0.19	1813.54	62.60	1632.15	13.27	13.87
ALL	**1.02**	**39.15**	**35.64**	**144.69**	**3.90**	**22.52**	**69.24**	**298.83**	**0.10**	**177.40**	**12.85**	**334.57**	**6.85**	**7.11**

97.3%↓ **98.4%↓** **90.2%↓**

Real-life scenarios

- Scenario A: Launching the “Contacts” App

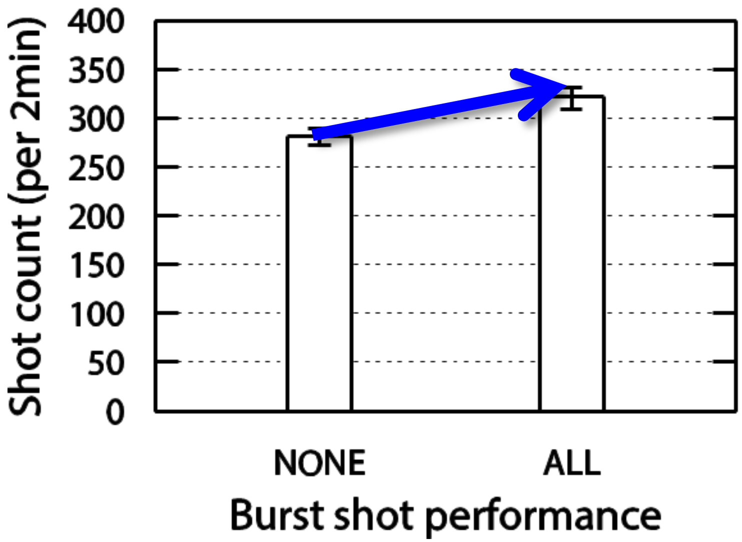


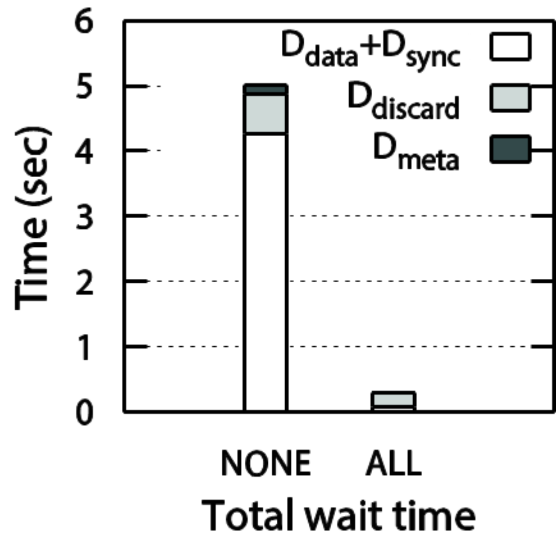
The worst case launch time is reduced by up to **44.8%↓**

The total wait time by $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$ is reduced by **96.1%↓** and **87.4%↓**

Real-life scenarios

- Scenario B: Burst Mode in the "Camera" App



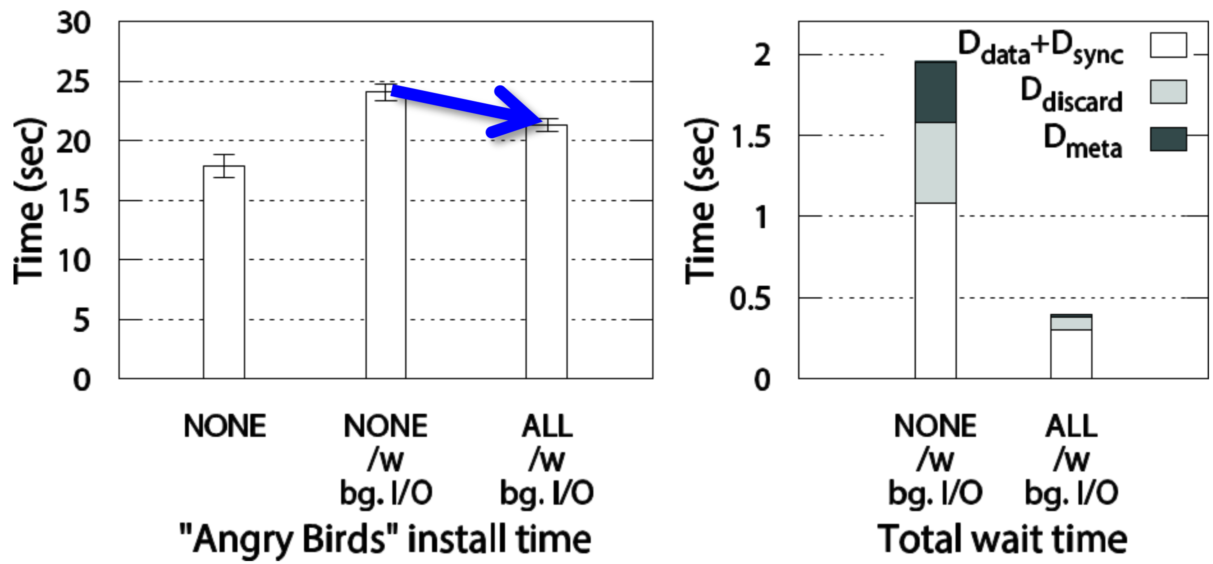


The shot count is improved by up to **14.4%↑**

The total wait time by **D**$_{data}$ is reduced by **98.4%↓**

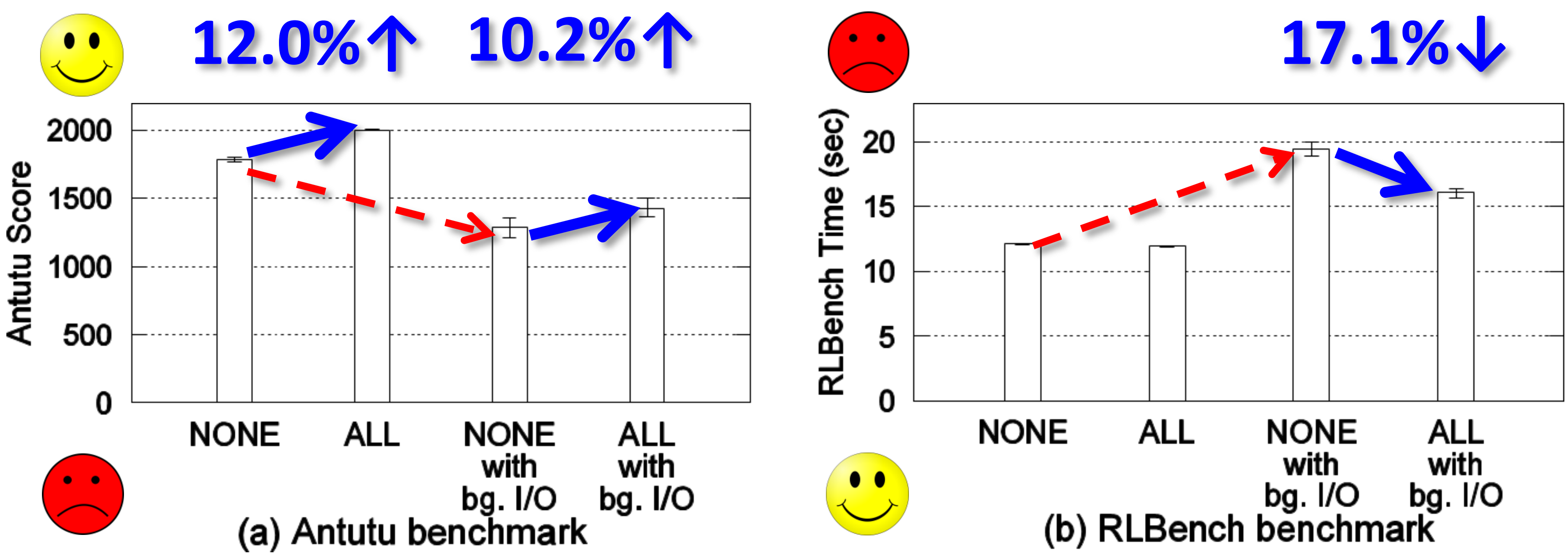
Real-life scenarios

- Scenario C: Installing the "Angry Birds" App



The app install time is improved by up to **11.5%↓**

Android I/O benchmarks



Conclusions

- A new type of I/O, QASIO
  - Seemingly asynchronous, but has the synchronous property
  - Types of dependency on QASIO
- Novel scheme to detect and boost QASIO
  - The worst case latency of a file system call, **98.4%↓**
  - The worst case “Contacts” app start time, **44.8%↓**
- Future work
  - Analyze the impact of QASIO on other types of systems
  - Devise another solutions optimized for each type of QASIO

Thank You

Daeho Jeong (daeho43@gmail.com)
Youngjae Lee (yjlee@csl.skku.edu)
Jin-Soo Kim (jinsookim@skku.edu)