

VanillaCore Walkthrough

Part 3

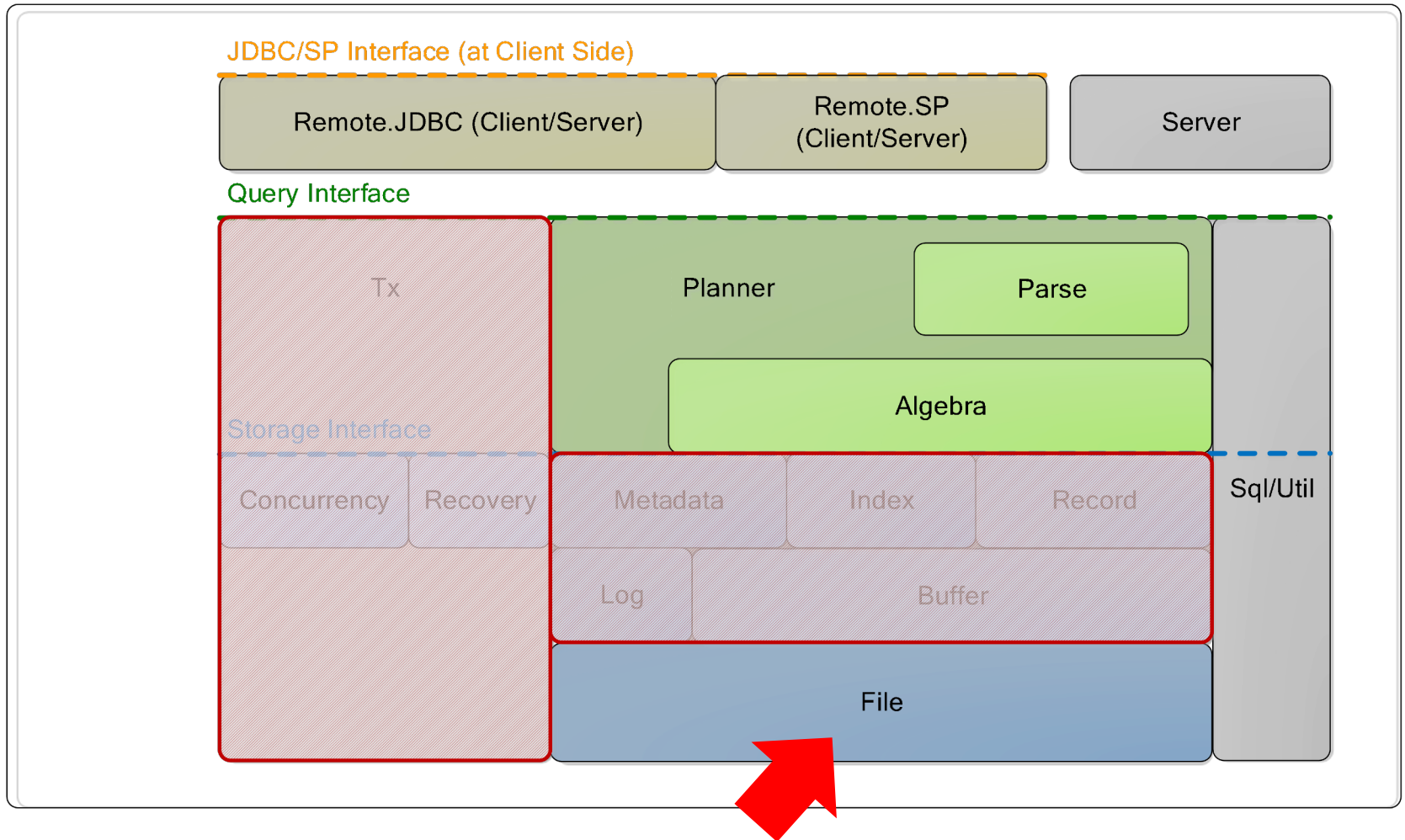
Introduction to Databases

DataLab

CS, NTHU

Today's Focus

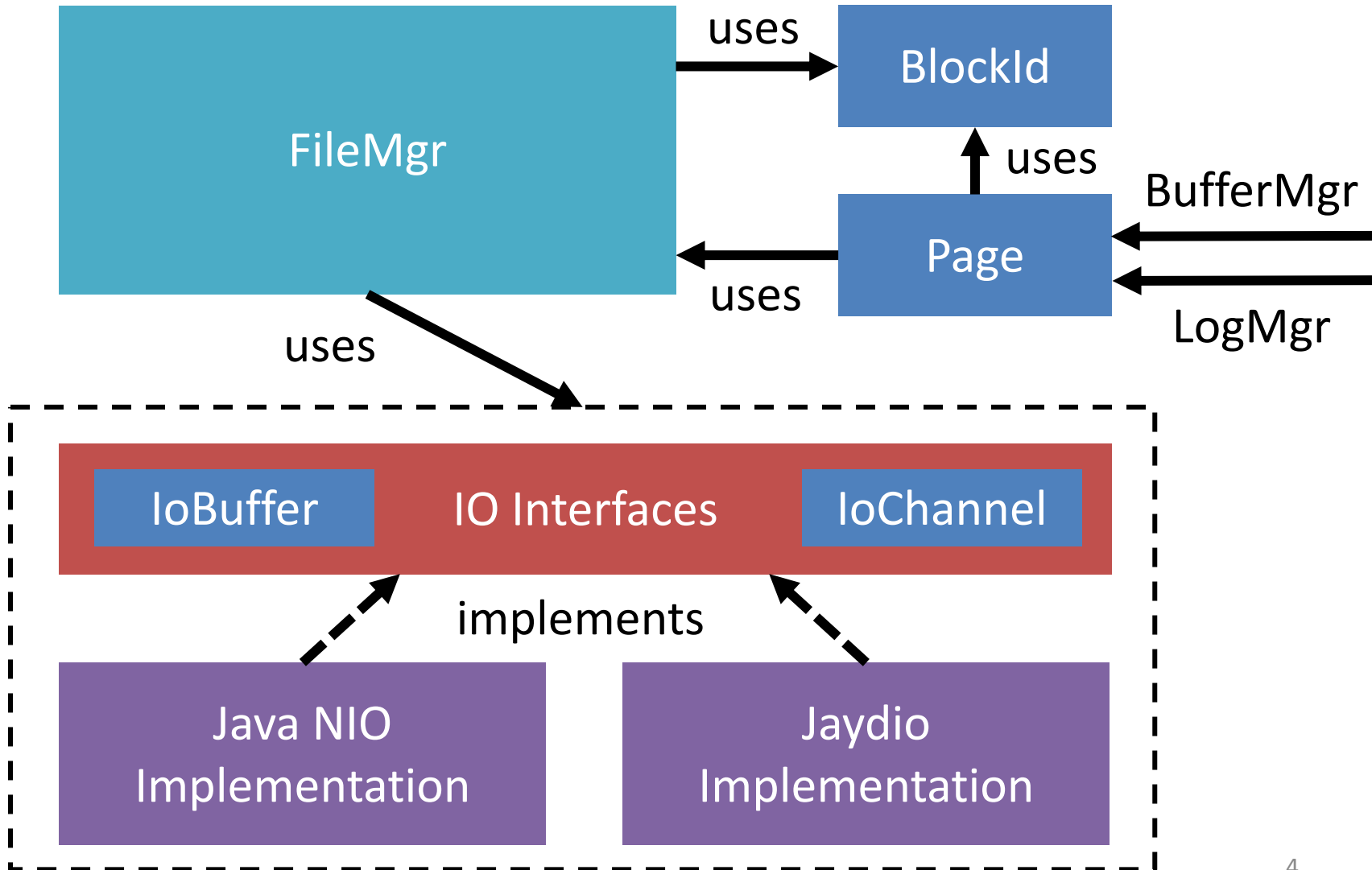
VanillaDB



The Mission

- The `file` package processes all file access requests in VanillaCore
 - Reading a block from a file
 - Writing a block to a file
 - Appending a block to a file
 - Deleting a file

file Package



Functionality

- Main Components
 - BlockId: represents the physical position of a block
 - Page: represents a memory region to hold a block
 - FileMgr: manages file access
- Low-level APIs to operating systems
 - IoBuffer: provides the API to manage a memory region
 - IoChannel: provides the API of access a file

Functionality

- **Main Components**
 - **BlockId**: represents the physical position of a block
 - Page: represents a memory region to hold a block
 - FileMgr: manages file access
- **Low-level APIs to operating systems**
 - IoBuffer: provides the API to manage a memory region
 - IoChannel: provides the API of access a file

BlockId

```
public class BlockId {  
    private String fileName;  
    private long blkNum;  
  
    public BlockId(String fileName, long blkNum) {  
        this.fileName = fileName;  
        this.blkNum = blkNum;  
    }  
  
    public String fileName() {  
        return fileName;  
    }  
  
    public long number() {  
        return blkNum;  
    }  
    ...  
}
```

BlockId
+ BlockId(filename : String, blknum : long) + fileName() : String + number() : long + equals(Object : obj) : boolean + toString() : String + hashCode() : int

Functionality

- **Main Components**
 - `BlockId`: represents the physical position of a block
 - `Page`: represents a memory region to hold a block
 - `FileMgr`: manages file access
- **Low-level APIs to operating systems**
 - `IoBuffer`: provides the API to manage a memory region
 - `IoChannel`: provides the API of access a file

Page

Page
<u><<final>> + BLOCK_SIZE : int</u>
<u>+ maxSize(type : Type) : int</u> <u>+ size(val : Constant) : int</u> + Page() <<synchronized>> + read(blk : BlockId) <<synchronized>> + write(blk : BlockId) <<synchronized>> + append(filename : String) : BlockId <<synchronized>> + getVal(offset : int, type : Type) : Constant <<synchronized>> + setVal(offset : int, val : Constant) + close()

Page

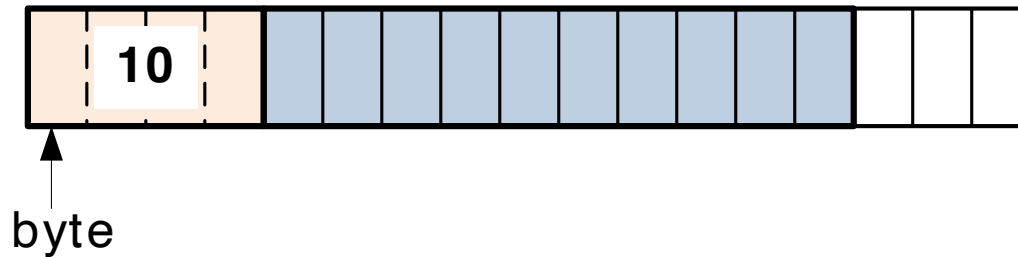
- Backed by `IoBuffer`

```
private IoBuffer contents = IoAllocator.newIoBuffer(BLOCK_SIZE);
```

- Translate constants using `Constant.asBytes()`
 - Fixed length for numeric type constants (e.g., 4 bytes for `IntegerConstant`)
 - Variable length for `VarcharConstant`
- How to reconstruct a varchar constant in getter?

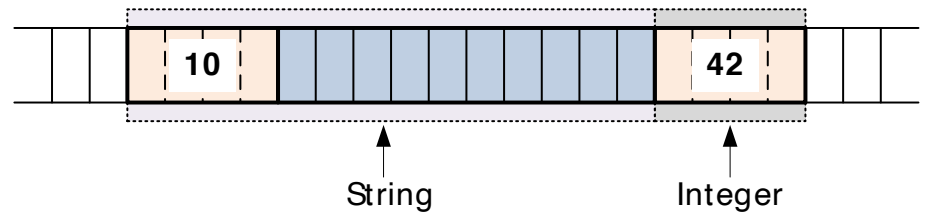
Storing A Varchar

- Page stores a Varchar in two parts
 - The first is the length of those bytes
 - The second is the bytes from `asByte()`



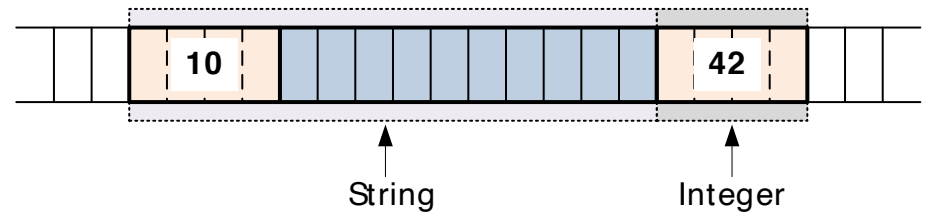
setVal

```
public synchronized void setVal(int offset, Constant val) {  
    byte[] byteval = val.asBytes();  
  
    // Append the size of value if it is not fixed size  
    if (!val.getType().isFixedSize()) {  
        // check the field capacity and value size  
        if (offset + ByteHelper.INT_SIZE + byteval.length > BLOCK_SIZE)  
            throw new BufferOverflowException();  
  
        byte[] sizeBytes = ByteHelper.toBytes(byteval.length);  
        contents.put(offset, sizeBytes);  
        offset += sizeBytes.length;  
    }  
  
    // Put bytes  
    contents.put(offset, byteval);  
}
```



getVal

```
public synchronized Constant getVal(int offset, Type type) {  
    int size;  
    byte[] byteVal = null;  
  
    // Check the length of bytes  
    if (type.isFixedSize()) {  
        size = type.maxSize();  
    } else {  
        byteVal = new byte[ByteHelper.INT_SIZE];  
        contents.get(offset, byteVal);  
        size = ByteHelper.toInteger(byteVal);  
        offset += ByteHelper.INT_SIZE;  
    }  
  
    // Get bytes and translate it to Constant  
    byteVal = new byte[size];  
    contents.get(offset, byteVal);  
    return Constant.newInstance(type, byteVal);  
}
```



Sizing Information

- There are static APIs providing sizing information in Page

```
public static int maxSize(Type type) {  
    return type.isFixedSize() ? type.maxSize() : ByteHelper.INT_SIZE  
        + type.maxSize();  
}  
  
public static int size(Constant val) {  
    return val.getType().isFixedSize() ? val.size() : ByteHelper.INT_SIZE  
        + val.size();  
}
```

File I/Os

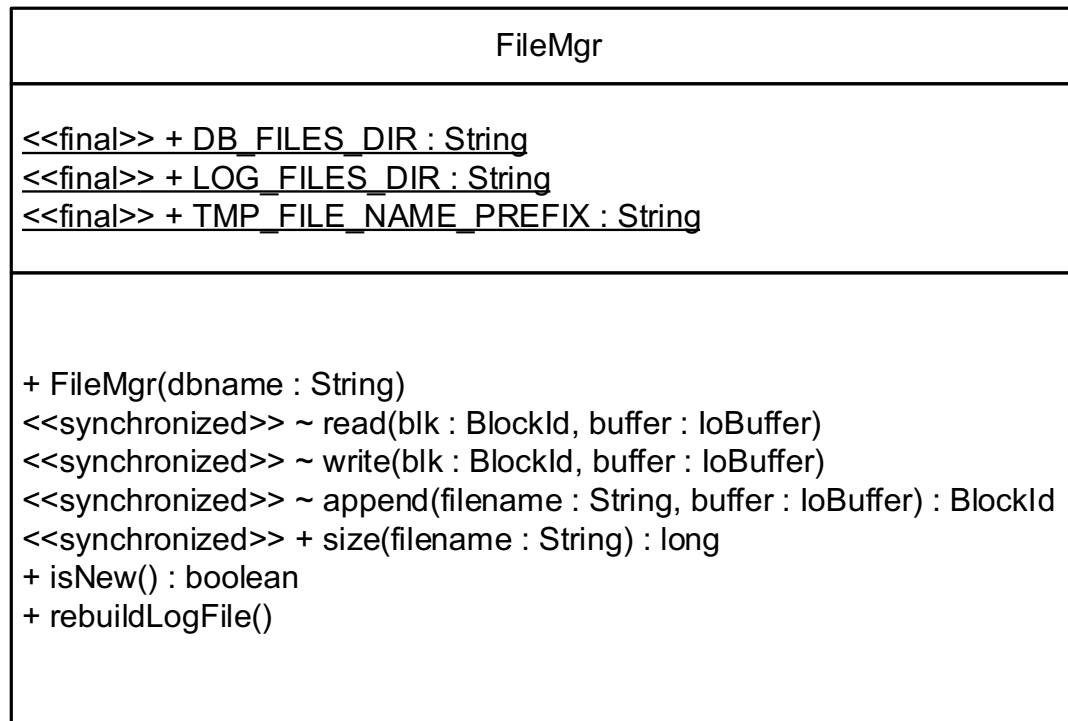
```
public Page() {  
}  
  
public synchronized void read(BlockId blk) {  
    fileMgr.read(blk, contents);  
}  
  
public synchronized void write(BlockId blk) {  
    fileMgr.write(blk, contents);  
}  
  
public synchronized BlockId append(String fileName) {  
    return fileMgr.append(fileName, contents);  
}
```

Functionality

- **Main Components**
 - `BlockId`: represents the physical position of a block
 - `Page`: represents a memory region to hold a block
 - `FileMgr`: manages file access
- **Low-level APIs to operating systems**
 - `IoBuffer`: provides the API to manage a memory region
 - `IoChannel`: provides the API of access a file

FileMgr

- Handles the actual I/Os
- Keeps the `IoChannel` instances of all opened files



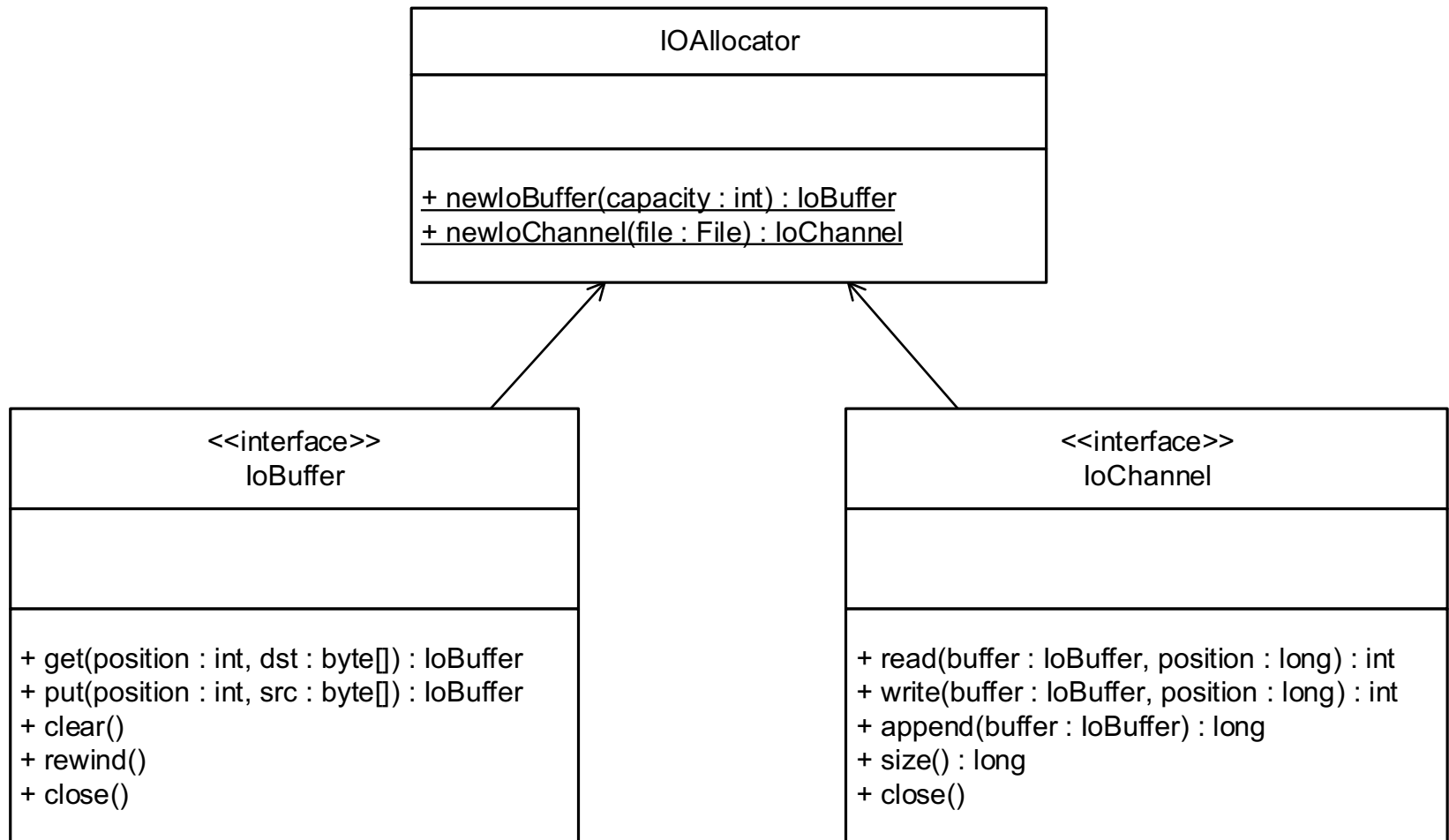
FileMgr

- A page delegates `read`, `write` and, `append` to `FileMgr`
- Note that the file manager always reads/writes/appends a ***block-sized*** number of bytes from/to a file
 - Exactly one disk access per call

Functionality

- Main Components
 - BlockId: represents the physical position of a block
 - Page: represents a memory region to hold a block
 - FileMgr: manages file access
- Low-level APIs to operating systems
 - IoBuffer: provides the API to manage a memory region
 - IoChannel: provides the API of access a file

file.io



Two Implementations

- Java NIO
 - Part of Java Standard Library
 - Provides high performance memory and file I/O
- Jaydio
 - A third-party library that provides finer controls over file I/O
 - Supports `O_DIRECT`
 - Only supported on Linux distributions

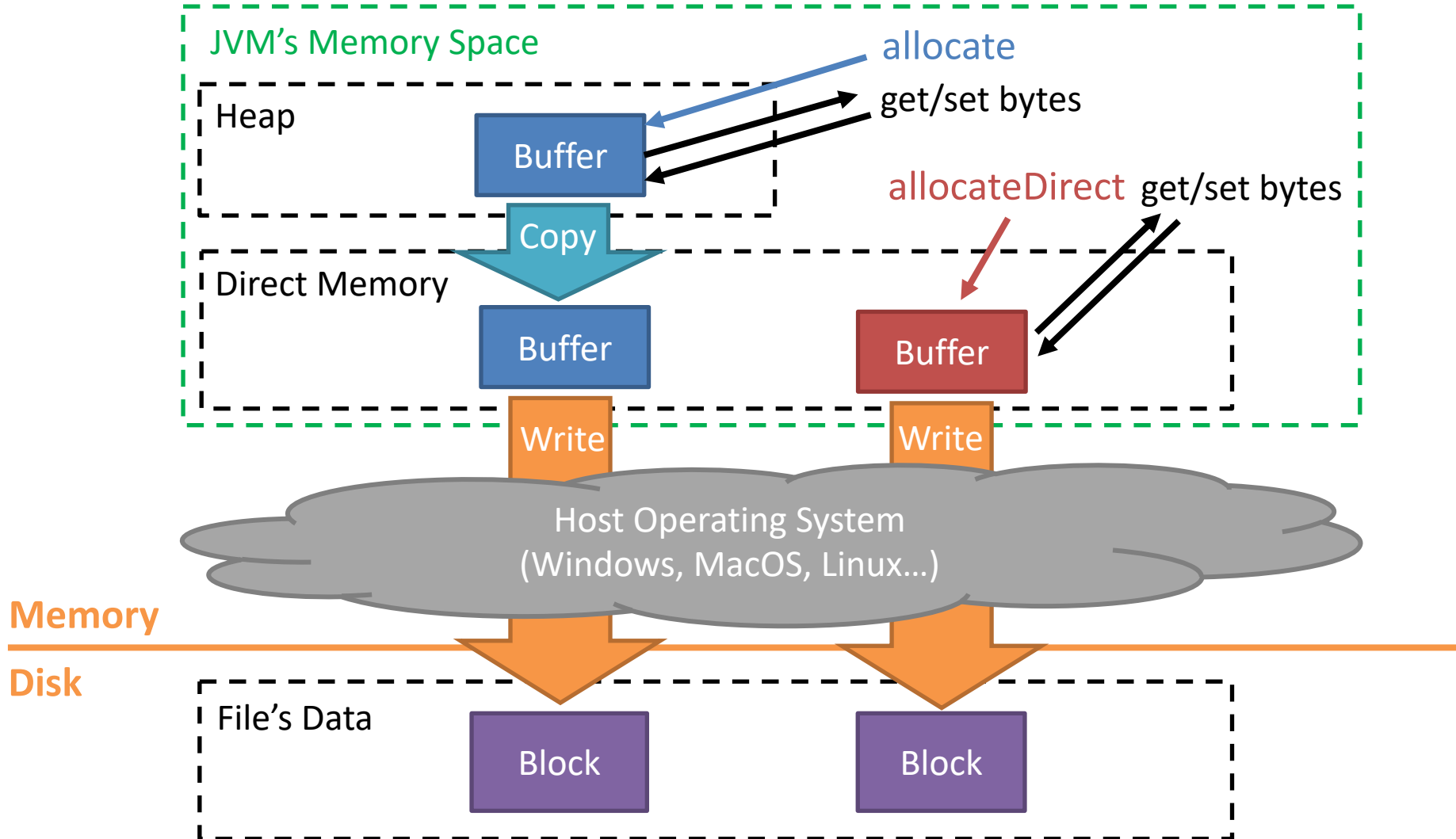
IoChannel in Java NIO

- Opens a file by creating a new `RandomAccessFile` instance and then obtain its file channel via `getChannel()`
- Files are open in “**rw**s” mode when using Java NIO
 - The “**rw**” means that the file is open for reading and writing
 - The “**s**” means that the OS should not delay disk I/O in order to optimize disk performance; instead, every *write* operation must be written immediately to the disk

IoBuffer in Java NIO

- IoBuffer in Java NIO is implemented by wrapping ByteBuffer
- ByteBuffer has two factory methods: allocate and allocateDirect
 - allocateDirect tells JVM to use one of the OS's I/O buffers to hold the bytes
 - **Not** in Java programmable buffer, no garbage collection
 - Eliminates the redundancy of **double buffering**

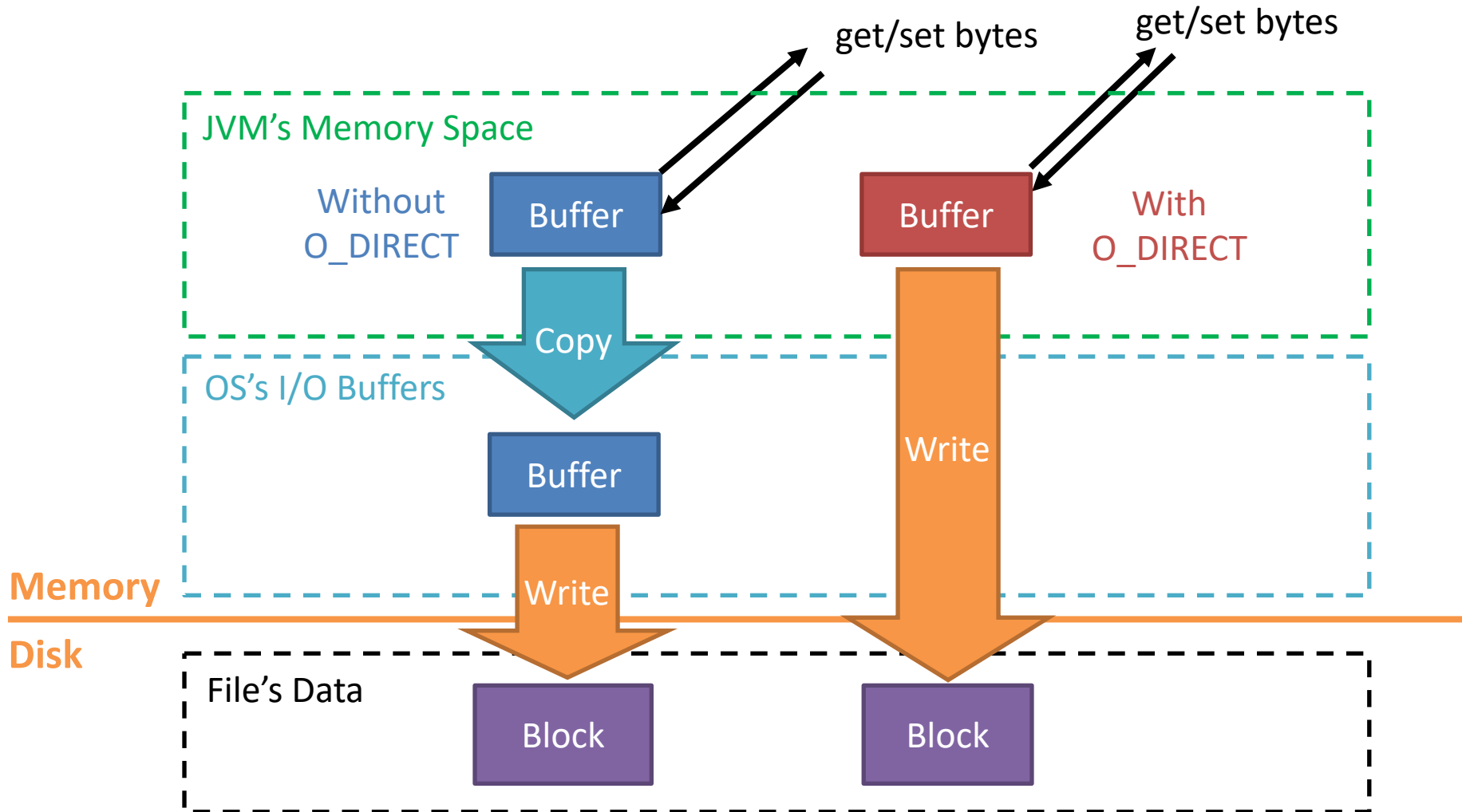
Double Buffering



IoChannel in Jaydio

- Supports `O_DIRECT` option
 - This option forces the operating system to **directly send the written bytes to the file** without buffering.
- Opens a file by calling `DirectIoByteChannel.getChannel()`

Double Buffering in OS



IoBuffer in Jaydio

- Jaydio's `AlignedDirectByteBuffer` has been ensured to be allocated in direct memory.
 - The double buffering in JVM has been avoided