

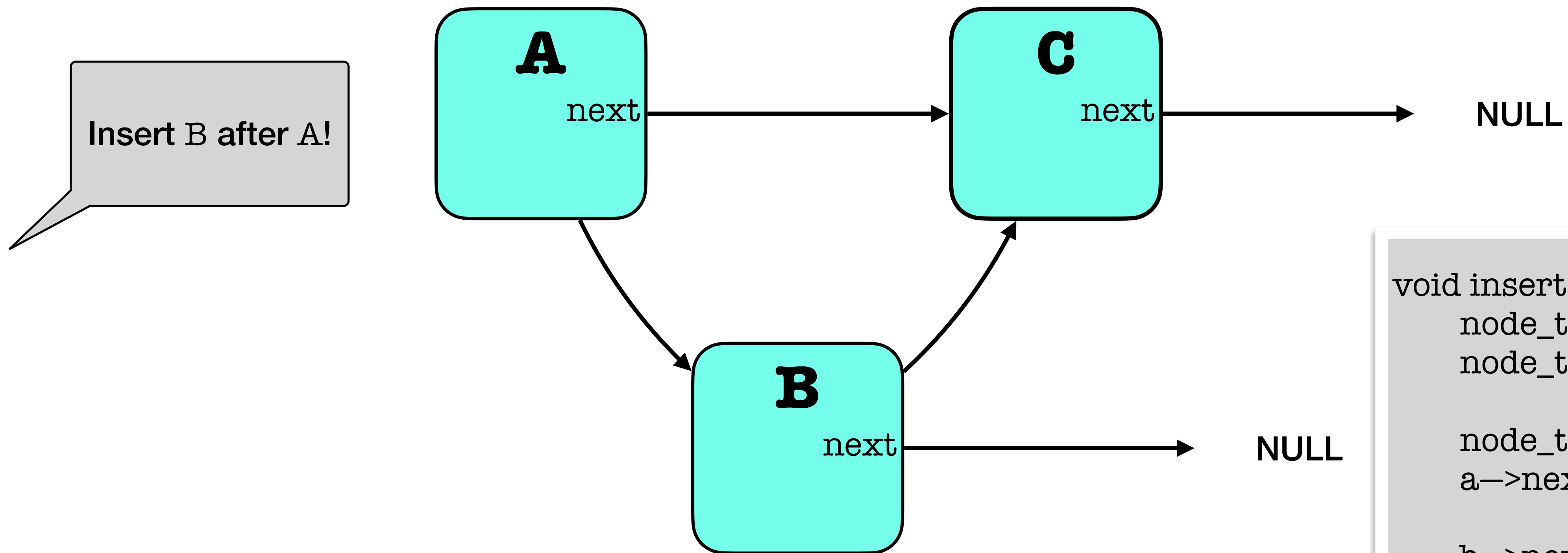
Crash Consistency

**Suggested FS Maker
deadline, Friday!**

Outline

- What can go wrong
- Dealing with crashes via soft updates
- Journaled file systems

Life Without Crash Consistency



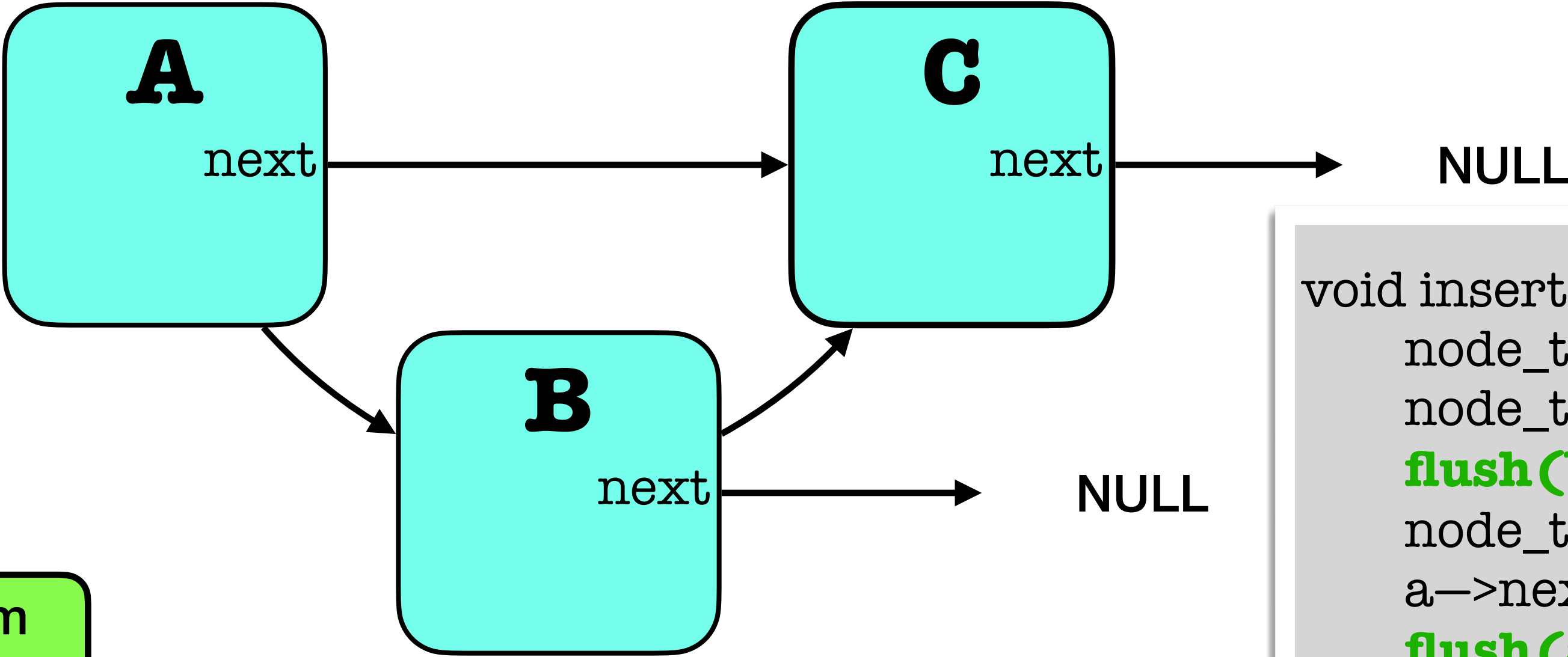
```
void insert(T key) {  
    node_t a = find_prev(key);  
    node_t b = new Node(key);  
  
    node_t *temp = a->next;  
    a->next = &b;  
  
    b->next = temp;  
}
```

Life With Crash Consistency

Very inefficient to flush to disk after every update to our structures in memory!

On Intel machines, flush operations are implemented as asynchronous function calls!

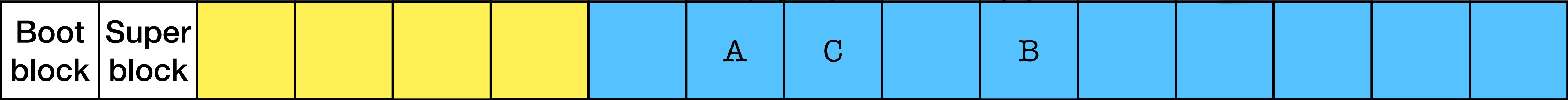
Changes propagate from memory to disk when they are *flushed* from the associated mobj



```
void insert(T key) {  
    node_t a = find_prev(key);  
    node_t b = new Node(key);  
    flush(b);  
    node_t *temp = a->next;  
    a->next = &b;  
    flush(a);  
    b->next = temp;  
    flush(b);  
}
```

Main Memory (RAM)

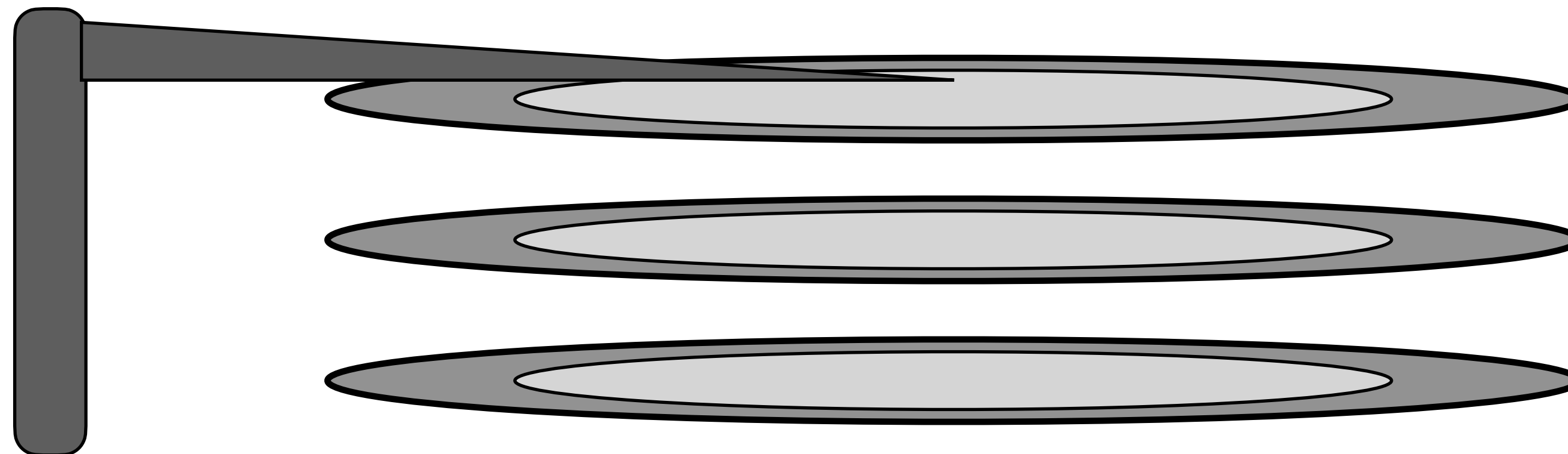
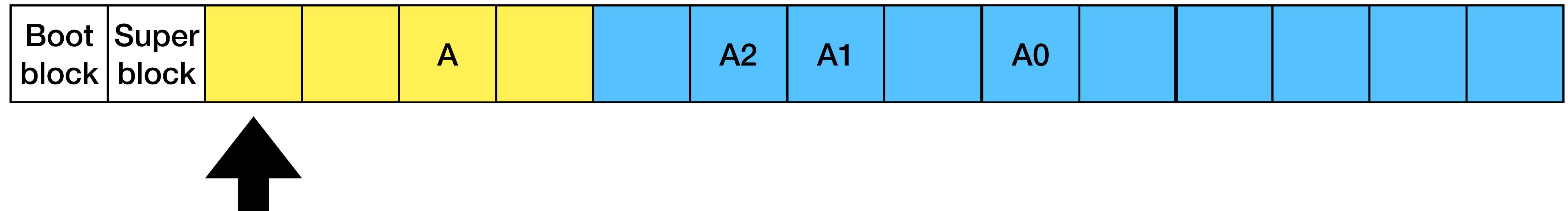
Disk



Implementing Crash Consistency

- Recall that we use `mobj` to implement a mechanism to cache data (and metadata) relating to files in on disk in our file system
- To ensure that data is consistent between the cached copy of data in memory and the version of the data that lives on disk, it must be explicitly flushed (at a frame granularity) to disk
- In hardware, flushes are implemented as sending a buffer of data to the disk device via the driver ➡ the disk device will raise an interrupt when the flush operation completes
- Because of this, flush operations are performed asynchronously and the disk hardware *does not guarantee* ordering between flush operations on distinct objects!

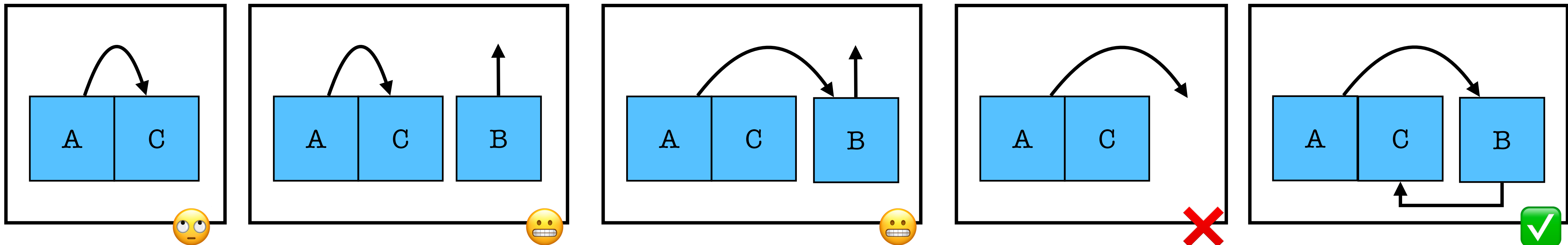
No Guarantee of Order on Disk



Chat with your neighbor(s)!

```
void insert(T key) {  
    node_t a = find_prev(key);  
    node_t b = new Node(key);  
    flush(b);  
    node_t *temp = a->next;  
    a->next = &b;  
    flush(a);  
    b->next = temp;  
    flush(b);  
}
```

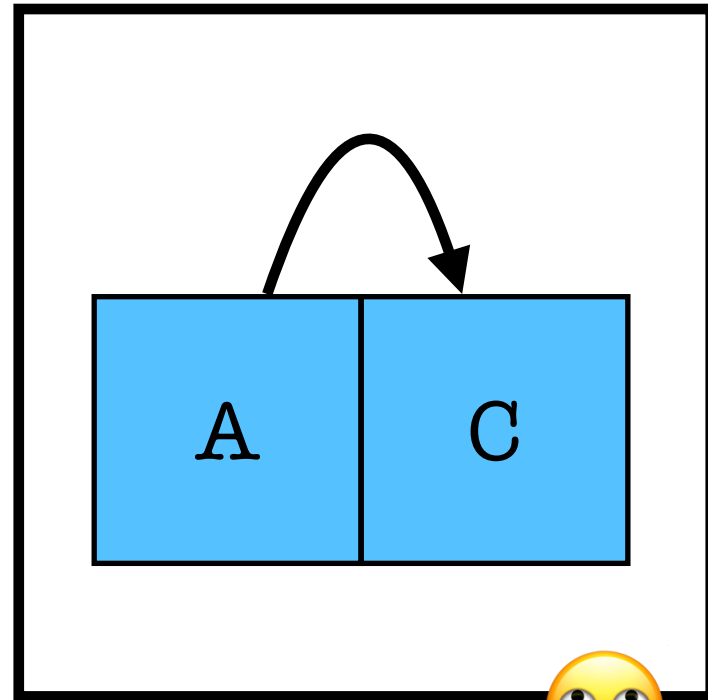
Consider the function to insert elements into a list in a crash consistent manner. If there is a crash at various points during the execution of our function, what are some of the possible outcomes of our disk?



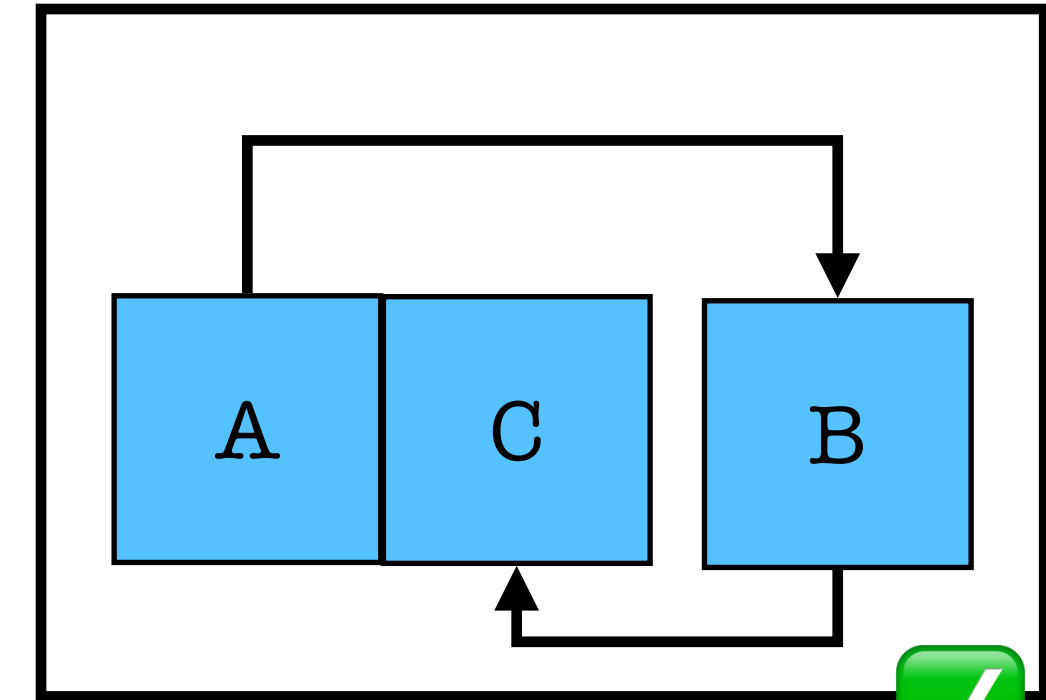
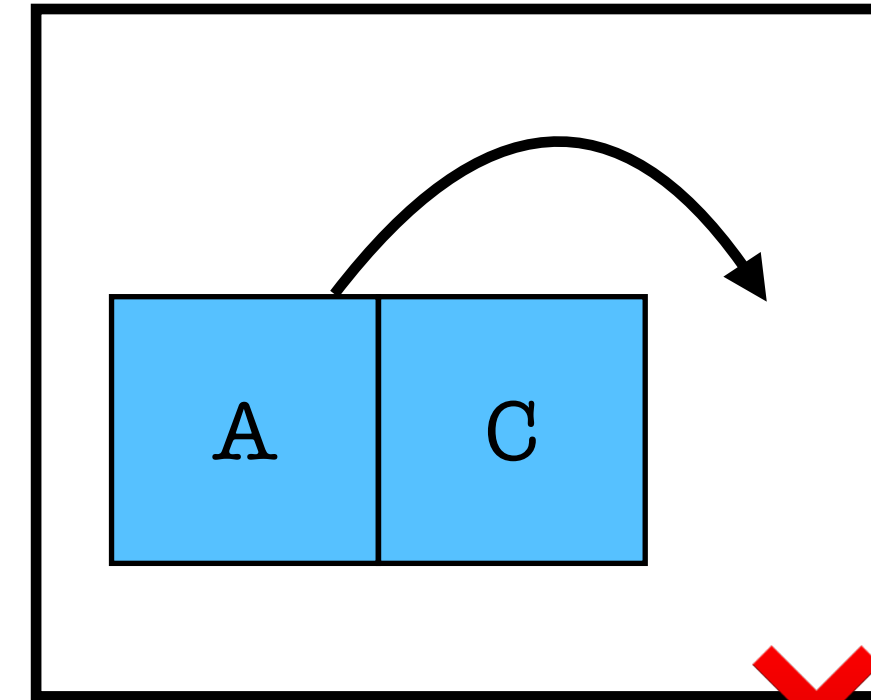
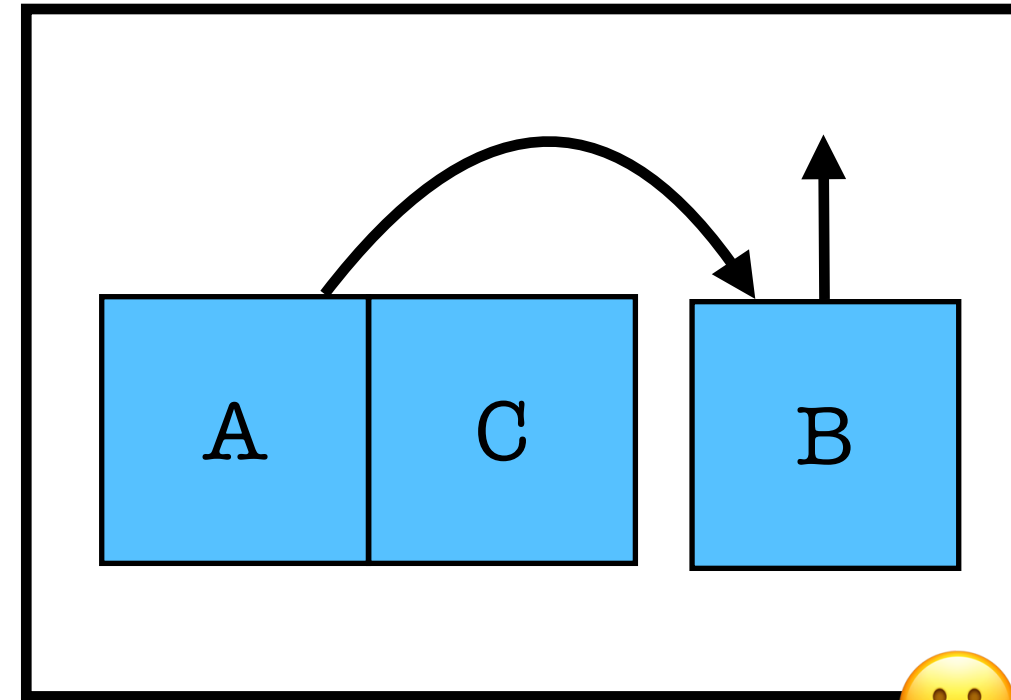
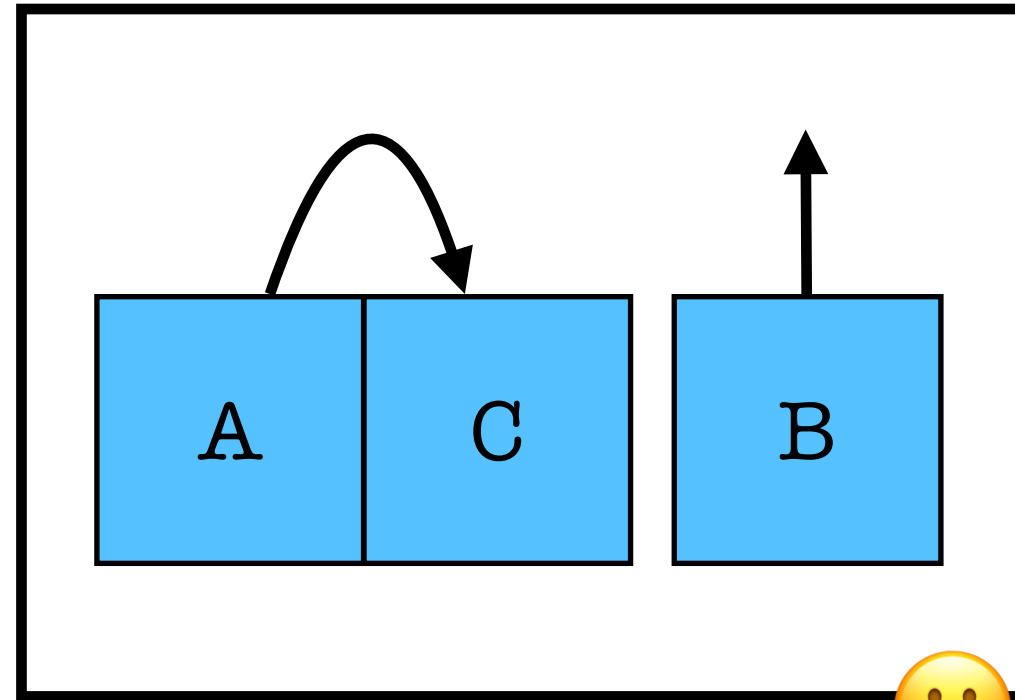
What Can Go Wrong

- When we allocate blocks in memory, we are creating cached copies of the data that we are writing to without keeping the data on our disk consistent
 - ➡ if memory acts as a cache for disk, then things appear on disk in “least-recently used” order
 - ➡ even if we aggressively implement flushing of data to disk, our hardware may still reorder events
- To work around this issue, we need to ensure that updates to disk are *atomic* and that any changes to structures that will appear on disk are *well-formed* particularly as it relates to the data in our file system
- Our directory structure is implemented as a directed-acyclic graph (DAG) to build a pointer-chasing tree: the structure is *not well-formed* if it is ever the case that: (1) a disk block appears in more than one file; (2) a disk block appears as both free and in a file; (3) there is a cycle in the DAG; (4) a disk block is unaccounted for

Consistency-Preserving Approach



Consistent



Consistent

Although not implemented in System V FS, we can use a file system that (from time-to-time or on boot) scans the disk and “garbage collects” inaccessible blocks

We can refer to these inconsistencies as *innocuous inconsistencies*

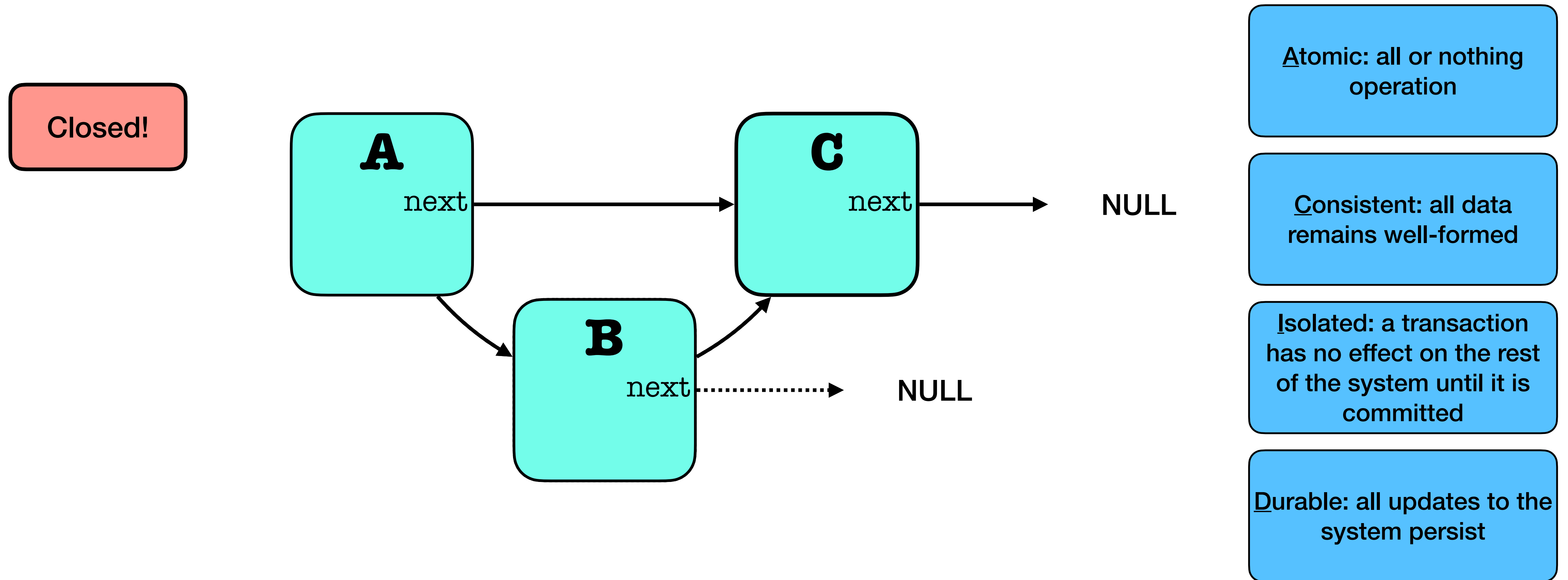
```
void insert(T key) {  
    node_t a = find_prev(key);  
    node_t b = new Node(key);  
    flush(b);  
    fence();  
    node_t *temp = a->next;  
    a->next = &b;  
    flush(a);  
    b->next = temp;  
    flush(b);  
}
```

Using fences to implement synchronous writes will incur significant overhead...

Consistency-Preserving Approach

- Our file system needs to have a mechanism to avoid creating irrecoverable inconsistencies on disk in terms of the order of write/flush operations
- To implement a consistency-preserving approach, the file system code needs to be modified throughout to ensure that writes to disk appear synchronously (e.g., performing a flush operation and a fence) ➡ this is very expensive!
- Without using fences, we need to update values in metadata using *soft updates*: saving a copy of each version of a disk block that has all dependencies embedded within it ➡ this was used to implement “Async FFS”

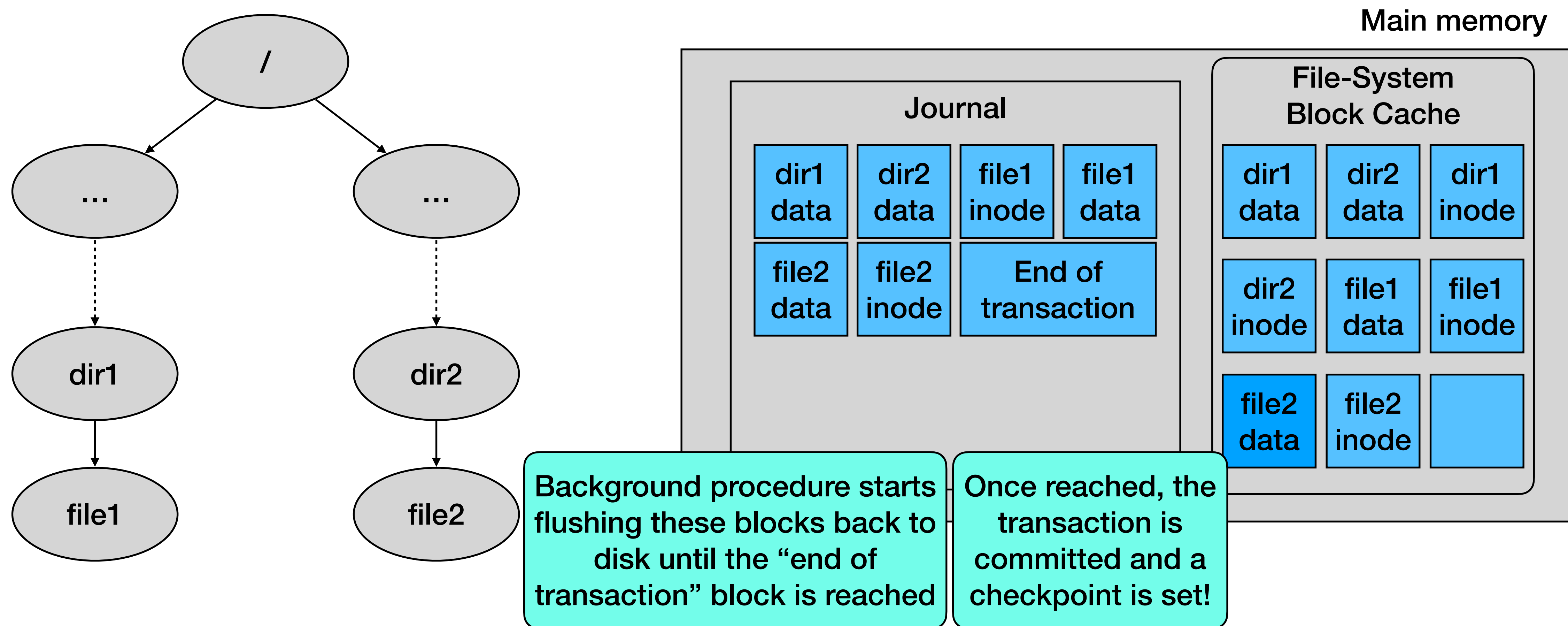
Updates Using Transactions



Updates Using Transactions

- *Transactions* are a synchronization mechanism to describe how updates may be locally and globally applied ➡ they are generally used to describe operations in databases and file systems
- In order for a transaction to be correct, it must ensure that updates to the local structure are atomic with the updates in the shared structure ➡ if we write to a node in main memory, we cannot further operate on that node until it appears persistent on disk (but we may be able to perform other operations)
- The act of “opening” a transaction describes the process of starting to perform an update to the local structure that should be consistent with the persistent copy, whereas “closing” a transaction is committing its state
- Transactions are generally implemented as “redo” operations (e.g., update the data locally with a copy to revert back to) or “undo” operations (store both copies of the data on disk and only update which one is referred to locally when the update completes)

Implementing Journaled File Systems



Chat with your neighbor(s)!

Suppose we are in the middle of flushing the data blocks that are currently stored in the journal region of memory at the time of a crash. How do we ensure that the file system is crash consistent?

Implementing Journalled File Systems (ext3)

- In addition to tracking the cached version of files, we should also maintain a region in memory with the version of files that is the consistent copy that will be flushed to disk ➡ this ensures that the disk will remain well formed
- When flushing blocks to disk, we do so by implementing “undo logging” to ensure that any changes to the data state are done atomically
- Undo logging is implemented by utilizing unused disk blocks to have “up-to-date” copies of the data, and inodes are updated to track the pointer and the “checkpoint number” to ensure that all inodes are referring to the most up-to-date version of the file system
- The most recent checkpoint number is tracked in the superblock

Summary

- To ensure that our file system is correct, we need to ensure that updates to files preserve *crash consistency*
- Crash consistency may be implemented using “soft updates,” e.g. explicit flushes to certain data, or by implementing transactions
- To implement transactions, our file system needs to implement undo or redo logging, which allows the transaction to be performed atomically and consistently