CI583: Data Structures and Operating Systems
File systems: Resiliency

Resiliency

Early file systems such as `S5FS` performed badly in the face of crashes and other unexpected shutdowns.

Not only could the files which were open at the time be corrupted, but other completely unrelated files could also be damaged.

It is easy to understand how unwritten modifications to a file could be lost, but how were other files being affected?

Resiliency

This happened because data structures which describe the whole file system, such as the superblock and the I-list, could easily become corrupted.

This is even more serious than losing the latest version of a user's file.

In the worst case, two separate inodes could point to the same data, or system files could be marked as free space, etc!

Resiliency

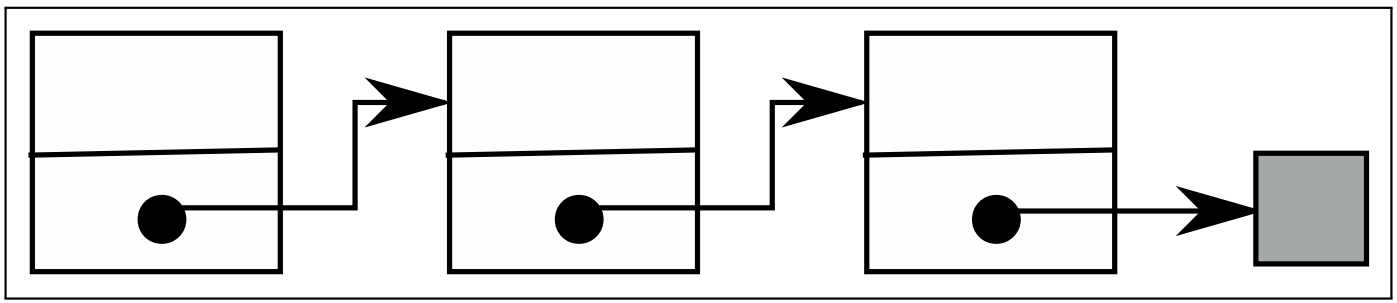
The problem stems from the fact that many operations that we think of as a "semantic unit", such as the system call `write`, actually comprise many operations.

Consider the task of adding an element to back of a queue.

In combination with caching, an interruption here can lead to the "last" element in the queue being some random memory location.

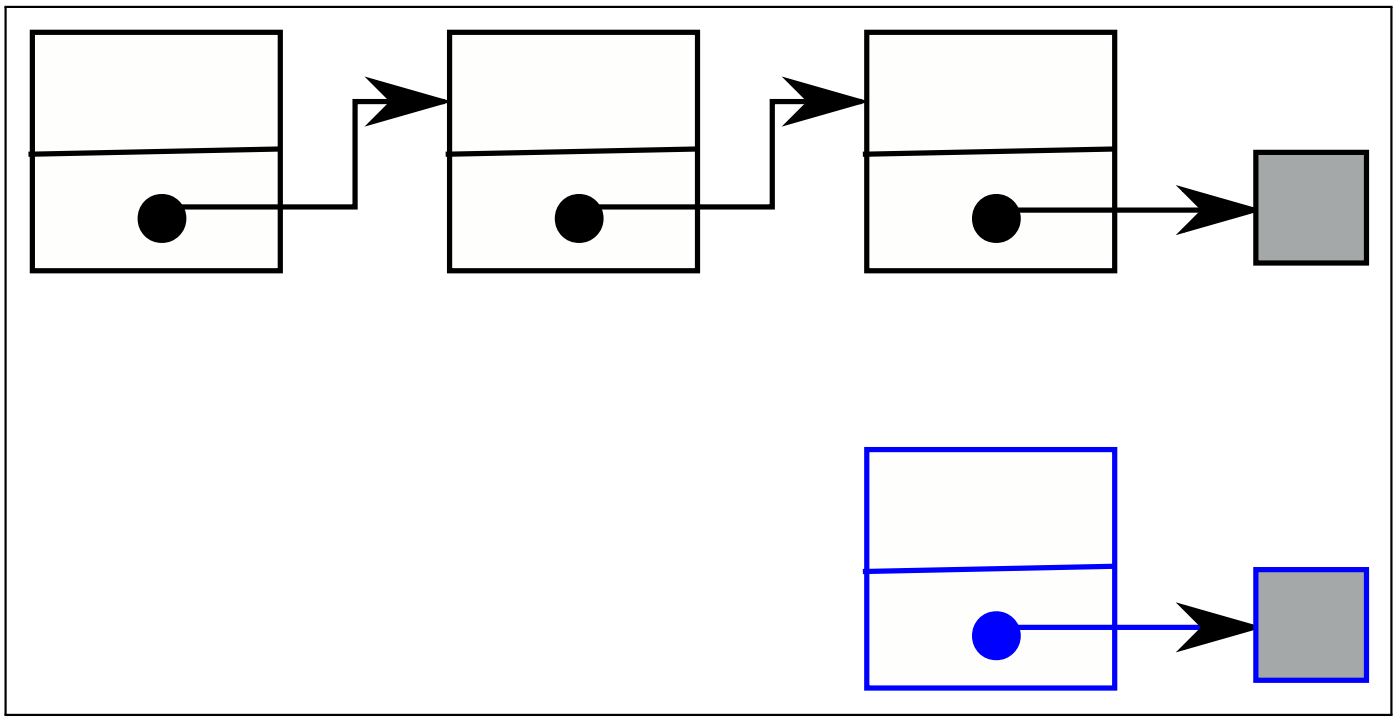
Resiliency

Corrupting a data structure



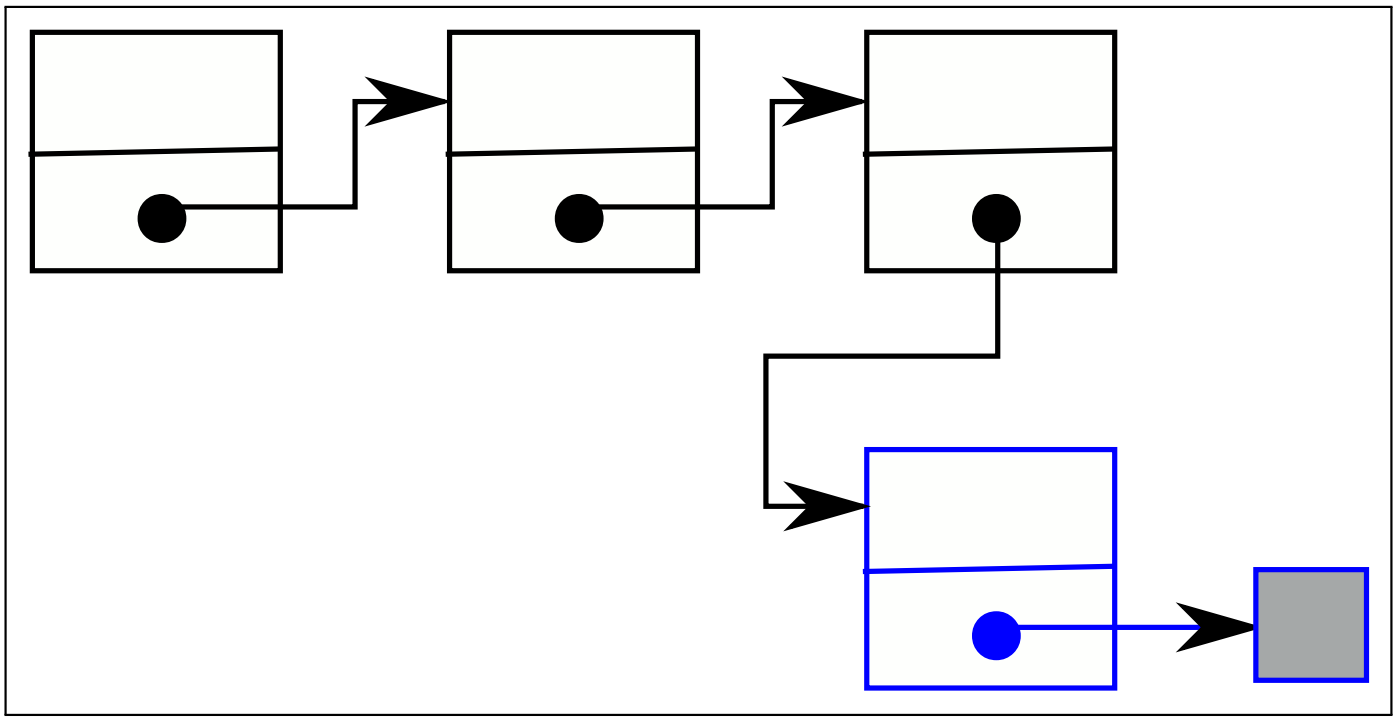
Resiliency

Corrupting a data structure



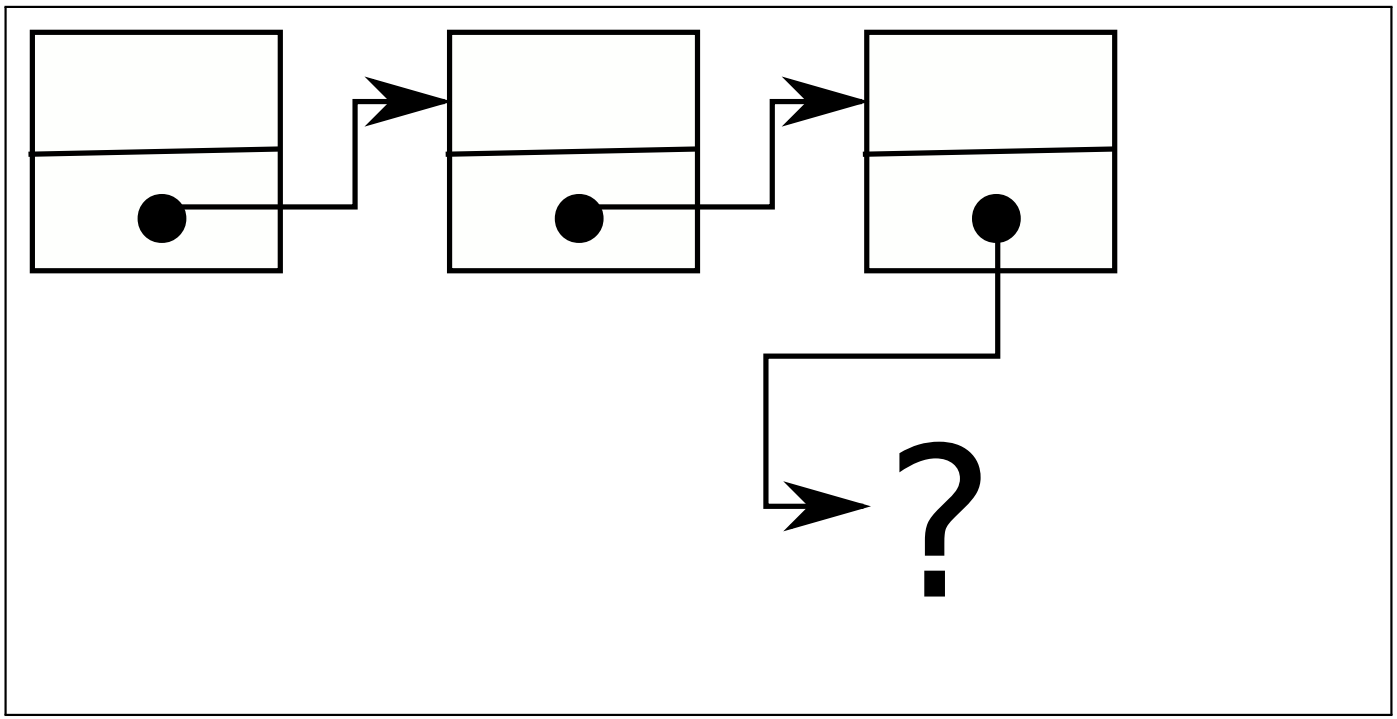
Resiliency

Corrupting a data structure



Resiliency

Corrupting a data structure



Resiliency

When a cache is being used, the user may actually have instructed the OS to save their work and been given feedback that the task is done, whilst in fact the changes are still in the cache and will not survive a crash.

We require the metadata structures on disk (list of free space, inodes, diskmaps within inodes, directories, etc) to be well formed at all times even if, at a push, they might not contain the latest updates.

If this is the case, then the data is consistent.

Resiliency

The measures taken by file systems to ensure consistency have normally focused on metadata consistency, rather than the contents of user files, although this is obviously important to the user.

One approach is to ensure that every change to the system (`write`, `creat`, `rename`, `unlink`) takes the system from one consistent state to another.

This is called a consistency-preserving approach, as used by FFS (80s UNIX file system, one of the successors to S5FS) for example. This is difficult in practice as every system change involves many individual changes.

Resiliency

A second approach is called the transactional approach.

Using this approach, we collect updates into groups called transactions, inspired by the database world.

This group of updates can be treated as a single action with the ACID properties.

Resiliency

- Atomic: each transaction is *all-or-nothing* – either all of it takes place or none of it does.
- Consistent: the system is always consistent. None of the inconsistent states that might exist while a transaction is taking place are visible outside of the transaction.
- Isolated: a transaction has no effect until it is committed to disk, and is not affected by any other ongoing (uncommitted) transactions.
- Durable: once a transaction is committed, its effects persist.

The two main approaches to providing these properties are journalling and shadow-paging, which we will look at in turn.