Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 18: Caching and ordering updates

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Project 3.
2. File systems recap.
3. Mapped files.
4. Ordering updates to a filesystem.

Agenda

1. Project 3.
2. File systems recap.
3. Mapped files.
4. Ordering updates to a filesystem.

Project 3: Due July 29

- Multi-process test cases
  - Needed even to test swap-backed no fork to test that you clean up when a process exits.
  - Call fork() before any calls to vm_map so you can test.
- To test fork(), write a test for every state a page can be in at the time of fork().
  - Swap-backed vs file-backed.
  - Resident vs non-resident.
  - Shared vs unshared.
  - …

Agenda

1. Project 3.
2. File systems recap.
3. Mapped files.
4. Ordering updates to a filesystem.

Recap: File system structure

Abstraction:

- Every file is an array of bytes.
- Can store large number of files.

File header contains

- Metadata, e.g., owner, permissions, size.
- Root of index structure to locate file’s contents.

Directory: (name, header disk block#) entries

- Stored in a file.
- Limited interface for users/apps to update.

Directories

Directory: mapping information for a set of files

- Name of file → file header's disk block # for that file.
- Once, array of (name, file header's disk block #) entries.
- Modern file systems: hash table or B-tree.

Directories and files are largely equivalent.

- Same storage structure.
- Directory entry points to inode for file or directory.

Directory Example

/ directory

Name	Block #
“bin”	100
“users”	35
“tmp”	43
“foo.txt”	254

/users directory

Name	Block #
“harshavm”	23
“pmchen”	99
“nham”	72
	0

/users/nham directory

Name	Block #
“482.txt”	44
	0
“src”	55
“foo.txt”	33

Any differences in allowing application to update file versus directory?

Users can put arbitrary data in a file. But a user can’t be allowed to corrupt the file system by writing junk to a directory, solved with limited set of system calls for updating directories.

Example: `/users/nham/482/notes`

1. Read the file header for `/` (root directory), which contains pointers to data blocks of the / directory.
2. Read data blocks of `/`, contains list of the files and directories in `/`. Each entry contains amapping from name → header's disk block #. One of those entries is "users".
3. Read file header for `/users`.
4. Read data blocks for `/users`.
5. Read file header for `/users/nham`.
6. Read data blocks for `/users/nham`.
7. Read file header for `/users/nham/482`.
8. Read data blocks for `/users/nham/482`.
9. Read file header for `/users/nham/482/notes`.
10. Read first data block for `/users/nham/482/notes`.

May be helped by caching the file header for the current working directory.

Unified view of multiple storage devices

- Combine multiple storage devices into a file system
  - Each device contains own file system (starting with its root)
  - A filesystem on a different device can be *mounted* over a directory, called a *mount point*, using the mount command.
- Example:
  - / (root)
    - bin (same device as /)
    - etc (same device as /)
    - tmp (separate storage device)
    - afs (network storage “device”)
- Directory entry: 1) file, 2) directory, or 3) device

Data types for disk blocks

File systems store lots of data structures on disk.

- Data blocks.
- Directories.
- File headers (inodes), indirect blocks.
- Free lists (bitmaps of used, unused blocks).

How can you tell what type a block is?

- Each is just a fixed number of bytes.
- By what points to it (just like data structure in memory) and the reason you got there.

File Cache

- Caches file system blocks in physical memory.
  - Each block indexed by (device, logical block number).
- Should cache be in physical or virtual memory?
  - If you need to write-back a dirty block, you should probably write it to the actual device. The file cache is usually kept in physical memory.
- Should cache be write-through or write-back?
  - Write-through: poor performance.
  - Write-back: loses data on OS crash, power failure.
- Current file systems:
  - Write-back but limit the time a dirty block can stay in the cache before being written out to the device.
  - Background daemon writes dirty pages.

File cache vs. Virtual memory

Both use physical memory as a cache for disk.

- Virtual memory: Use disk for increased capacity.
- File systems: Use memory for faster performance.

Both compete for physical memory.

- Another instance of local vs. global replacement.
- Common to use global replacement.

Both are about managing memory. The big difference is that the filesystem must be persistent. But they can overlap.

Agenda

1. Project 3.
2. File systems recap.
3. Mapped files.
4. Ordering updates to a filesystem.

Memory-mapped files

Use the paging system to cache both virtual address space *and* file system data.

- Map file into a virtual address space.
- Point the backing store for that part of the address space at the file's data blocks.
- Writes will only happen as dirty blocks are evicted, could be lost if the system crashes.

Example: How to load a program executable from disk to memory?

```
tcsh-1% cat Sleep.cpp
#include <iostream>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int main( )
   {
   cout << "pid = " << getpid( ) << endl;
   sleep( 1000 );
   }
tcsh-2% g++ Sleep.cpp -o Sleep
tcsh-3% ./Sleep &
[1] 89
tcsh-4% pid = 89
```

```
tcsh-5% cat /proc/89/maps
7f4a188b0000-7f4a188c7000 r-xp 00000000 00:00 164943                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7f4a188c7000-7f4a188c8000 ---p 00017000 00:00 164943                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7f4a188c8000-7f4a18ac6000 ---p 00000018 00:00 164943                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7f4a18ac6000-7f4a18ac7000 r--p 00016000 00:00 164943                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7f4a18ac7000-7f4a18ac8000 rw-p 00017000 00:00 164943                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7f4a18ad0000-7f4a18c6d000 r-xp 00000000 00:00 757460                     /lib/x86_64-linux-gnu/libm-2.27.so
7f4a18c6d000-7f4a18c70000 ---p 0019d000 00:00 757460                     /lib/x86_64-linux-gnu/libm-2.27.so
7f4a18c70000-7f4a18e6c000 ---p 000001a0 00:00 757460                     /lib/x86_64-linux-gnu/libm-2.27.so
7f4a18e6c000-7f4a18e6d000 r--p 0019c000 00:00 757460                     /lib/x86_64-linux-gnu/libm-2.27.so
7f4a18e6d000-7f4a18e6e000 rw-p 0019d000 00:00 757460                     /lib/x86_64-linux-gnu/libm-2.27.so
7f4a18e70000-7f4a19057000 r-xp 00000000 00:00 757397                     /lib/x86_64-linux-gnu/libc-2.27.so
7f4a19057000-7f4a19060000 ---p 001e7000 00:00 757397                     /lib/x86_64-linux-gnu/libc-2.27.so
7f4a19060000-7f4a19257000 ---p 000001f0 00:00 757397                     /lib/x86_64-linux-gnu/libc-2.27.so
7f4a19257000-7f4a1925b000 r--p 001e7000 00:00 757397                     /lib/x86_64-linux-gnu/libc-2.27.so
7f4a1925b000-7f4a1925d000 rw-p 001eb000 00:00 757397                     /lib/x86_64-linux-gnu/libc-2.27.so
7f4a1925d000-7f4a19261000 rw-p 00000000 00:00 0
7f4a19270000-7f4a193e9000 r-xp 00000000 00:00 165051                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7f4a193e9000-7f4a193f6000 ---p 00179000 00:00 165051                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7f4a193f6000-7f4a195e9000 ---p 00000186 00:00 165051                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7f4a195e9000-7f4a195f3000 r--p 00179000 00:00 165051                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7f4a195f3000-7f4a195f5000 rw-p 00183000 00:00 165051                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7f4a195f5000-7f4a195f9000 rw-p 00000000 00:00 0
7f4a19600000-7f4a19626000 r-xp 00000000 00:00 757373                     /lib/x86_64-linux-gnu/ld-2.27.so
7f4a19626000-7f4a19627000 r-xp 00026000 00:00 757373                     /lib/x86_64-linux-gnu/ld-2.27.so
7f4a19827000-7f4a19828000 r--p 00027000 00:00 757373                     /lib/x86_64-linux-gnu/ld-2.27.so
7f4a19828000-7f4a19829000 rw-p 00028000 00:00 757373                     /lib/x86_64-linux-gnu/ld-2.27.so
7f4a19829000-7f4a1982a000 rw-p 00000000 00:00 0
7f4a19940000-7f4a19942000 rw-p 00000000 00:00 0
7f4a19950000-7f4a19952000 rw-p 00000000 00:00 0
7f4a19960000-7f4a19962000 rw-p 00000000 00:00 0
7f4a19a00000-7f4a19a01000 r-xp 00000000 00:00 1203170                    /mnt/c/Users/hamil/Google Drive/eecs482/W20Lectures/Sleep
7f4a19c00000-7f4a19c01000 r--p 00000000 00:00 1203170                    /mnt/c/Users/hamil/Google Drive/eecs482/W20Lectures/Sleep
7f4a19c01000-7f4a19c02000 rw-p 00001000 00:00 1203170                    /mnt/c/Users/hamil/Google Drive/eecs482/W20Lectures/Sleep
7fffe43bf000-7fffe43e0000 rw-p 00000000 00:00 0                          [heap]
7fffead68000-7fffeb568000 rw-p 00000000 00:00 0                          [stack]
7fffeb582000-7fffeb583000 r-xp 00000000 00:00 0                          [vdso]
tcsh-6%
```

Agenda

1. Project 3.
2. File systems recap.
3. Mapped files.
4. Ordering updates to a filesystem.

Multiple updates and reliability

File system must ensure reliability/durability.

- Okay to lose data in address space.
- Data must survive crashes and power outages.
- Assume: Only the update of single block is atomic and durable.
- Challenge: Crashes in midst of multi-step updates.

Example: Transfer $100 between accounts.

1. Deduct $100 from savings.
2. Add $100 to checking.

Crash between steps 1 and 2 = lose $100.

Other Examples

Move file from directory a to directory b.

1. Delete file from dir a.
2. Add file to dir b.

Create new (empty) file

1. Update directory to point to new file header.
2. Write new file header to disk.

How to fix these problems?

Ordering of updates

Careful ordering can fix some problems.

Example: Create file `482.txt` in directory `nham`

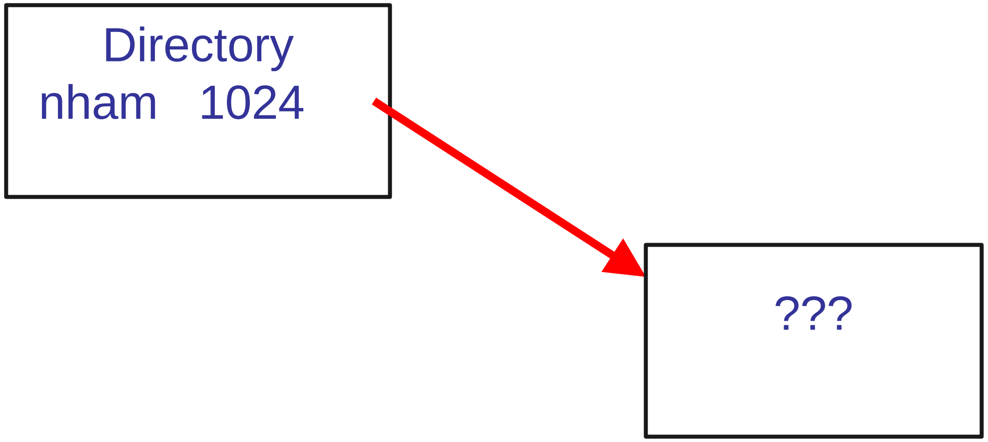
- Update directory first?
- Create inode for new file first?

Ordering of updates

Careful ordering can fix some problems:

For example, creating file `482.txt` in directory `nham`

Update directory first?

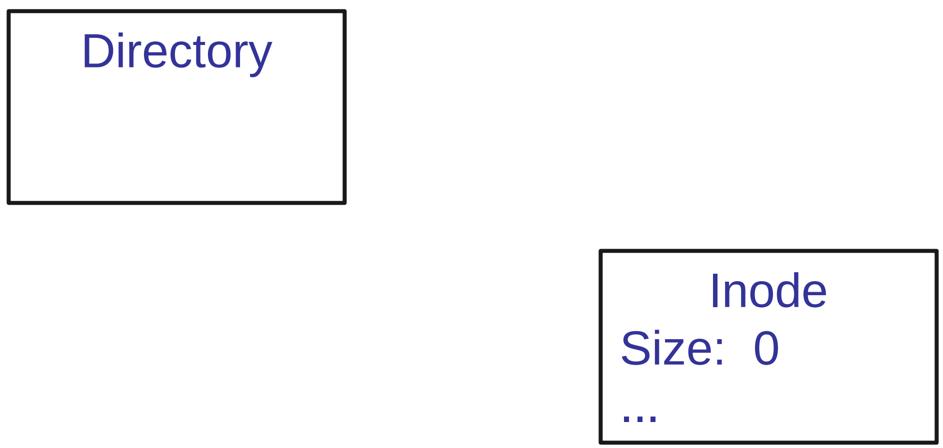


Never have a pointer from valid block to invalid one!

Ordering of updates

Careful ordering can fix some problems:

- For example, creating file `482.txt` in directory `nham`
- Create inode first?



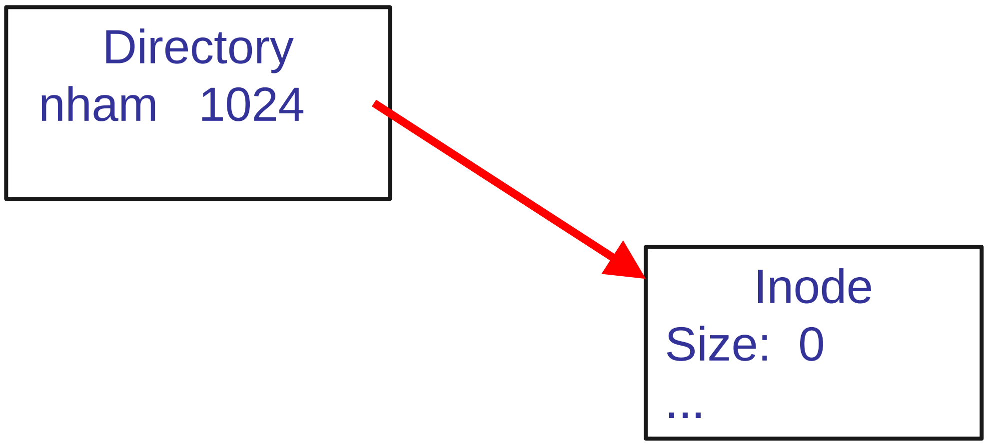
OK to modify unreachable blocks on disk.

Ordering of updates

Careful ordering can fix some problems:

For example, creating file `482.txt` in directory `nham`

Create inode first?



Careful ordering goes from one consistent state to another.

Ordering not always enough

Example: Create a file and update the free block list.

1. Write new file header to disk.
2. Update directory to point to new file header.
3. Write the new free map.

Is 1, 2, 3 correct?

What about 3, 1, 2?

What about the bank account example?

ACID terminology

Database systems are commonly describing as offering ACID properties. For a filesystem, we mostly care about atomicity and durability.

Atomicity	All or nothing. The operation either succeeds or does nothing.
Consistency	Representation invariants observed before and after an operation.
Isolation	Any intermediate states are invisible to other transactions which only see the state before or after.
Durability	Once an operation succeeds, the changes persist and will not be undone, even in the event of a system failure.

Transactions

Need a way to create transactions with atomicity and durability. But only writes to a single sector to a disk are atomic.

How to make a sequence of updates atomic?

Two main methods:

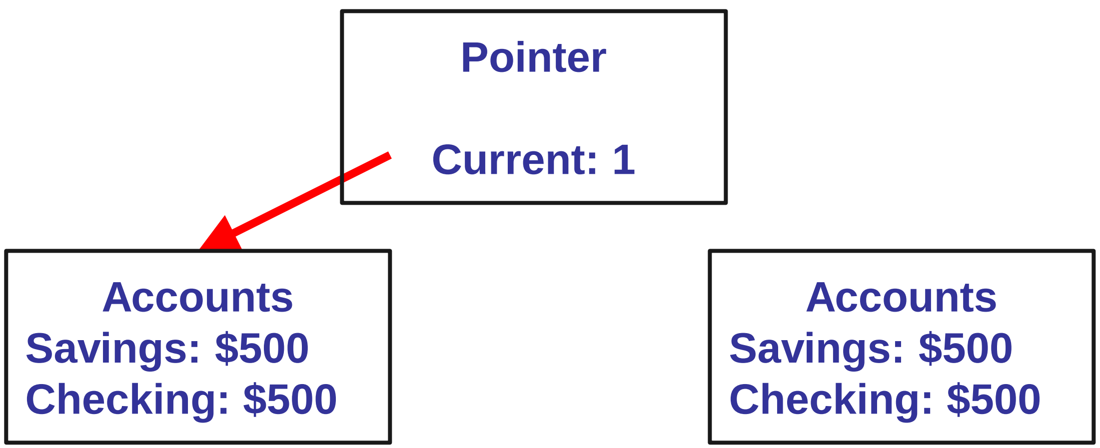
1. Shadowing.
2. Logging

```
begin
   write disk
   write disk
   write disk
end // commit the transaction)
```

Shadowing

Replicate the data across two stores:

- One is current version, other is backup
- Current pointer points to the current version

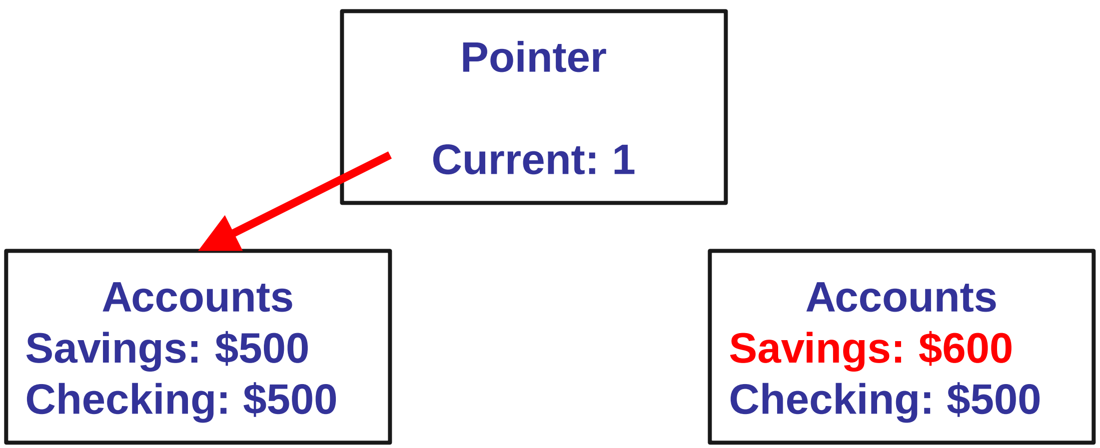


At beginning of transaction, both replicas are identical

Shadowing

Transaction updates the backup (shadow)

First add $100 to savings

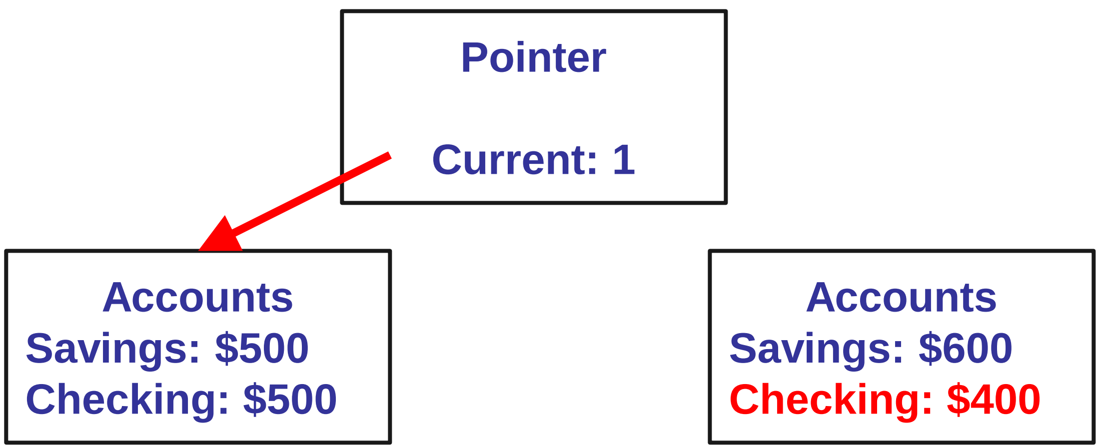


Note: modifying “unreachable” block

Shadowing

Transaction updates the backup (shadow)

Next remove $100 from checking

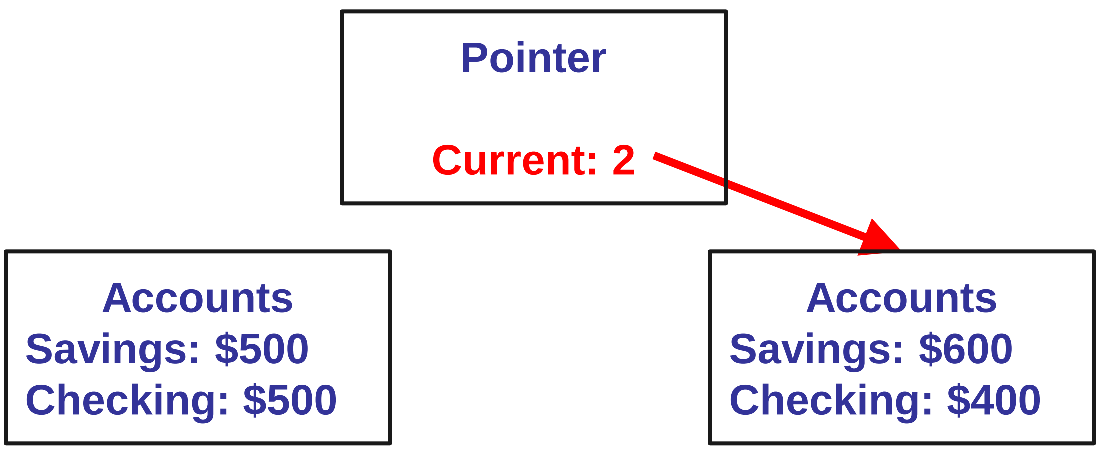


Note: modifying “unreachable” block

Shadowing

Transaction commit switches the pointer

This is point when updates become durable

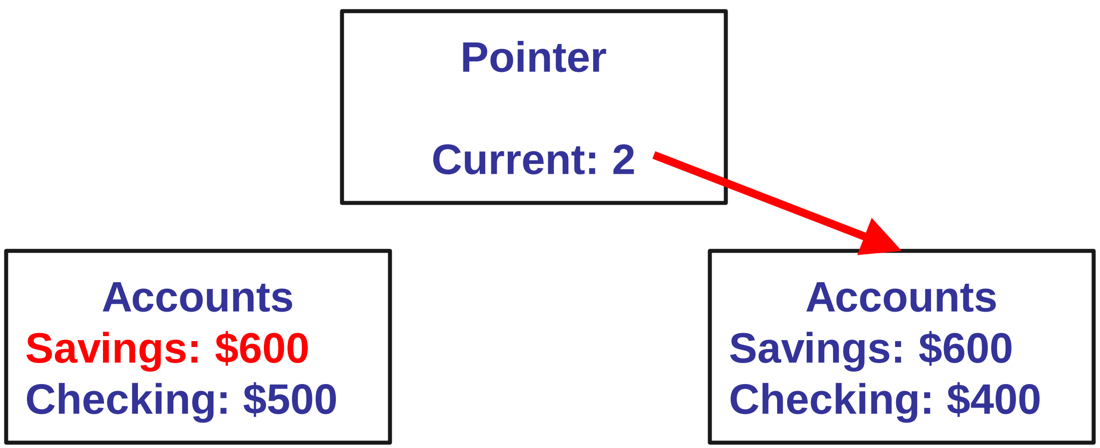


Note: updating single block = atomic update

Shadowing

Finally, must update new shadow

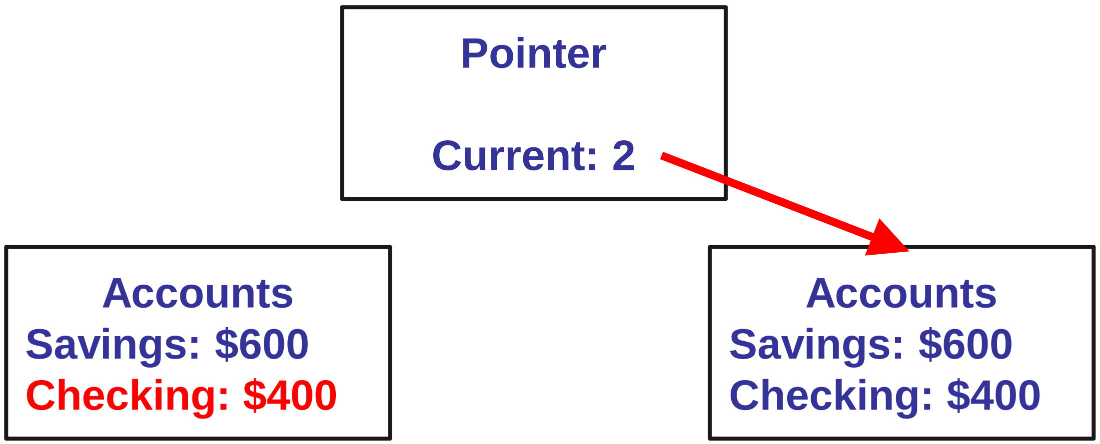
- First, update savings



Note: again, updating unreachable block

Shadowing

- Finally, must update new shadow
  - Next, update checking



Note: again, updating unreachable block

Shadowing summary

Can make arbitrary set of updates in transaction.

- Pointer switch is always an atomic commit.

Downside?

- Requires replicating data store.

Can reduce cost by shadowing on demand.

- Sometimes called shadow paging.
- Used in modern file systems (WAFL, ZFS, ...).