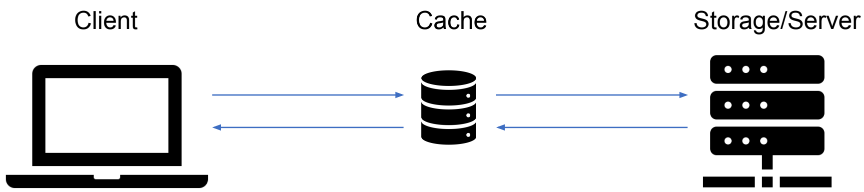
COS 316 Precept #6:
Caching
+
Eviction (Replacement)

Please visit ed discussion to download the precept 6 exercise code (here)

What is caching?

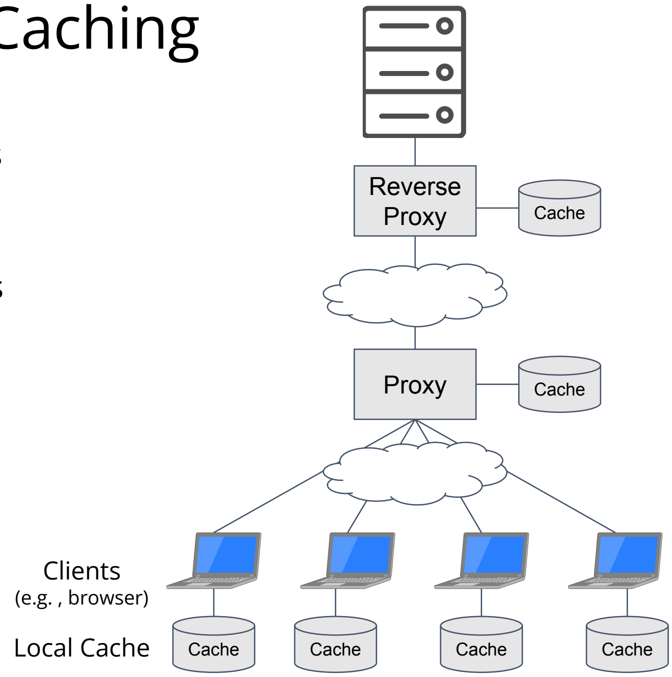


- This week, you’ve learned a bit about web caching in lecture
- Let’s talk about what a cache is and why’d you use one
- As a general computing term, a cache is a place where some objects from a larger dataset are stored for more convenient access
- You may have encountered a few caches in your computer science education
- There are caches in the CPU memory hierarchy that store a subset of main memory’s contents
- The TLB, which you learned about earlier, is a cache that stores virtual to physical address translations
- And the list goes on
- So why would we want to store objects in two places?
- Really there are two reasons:
    - Accessing a cache is often faster than accessing the main datastore, because caches tend to be closer to the entity that’s requesting the object
    - They also result in less traffic to the storage device or server that holds the entire dataset, which lowers the bandwidth and computing requirements of the storage server
- Caches generally take advantage of something called *locality*
- In this context, it refers to the tendency of data accesses to be correlated in some way
- Temporal locality is a type of locality that describes the tendency of things to be reused
- For example, if I access a particular object, I’m likely to use it again in the near

- future
- I’ve described this in a very vague and general sense so far. Let’s dive a bit deeper with an example from lecture

Overview of Web Caching

- Basic idea:
  - Bring objects “closer” to clients
- Three primary features:
  - Reduce network bandwidth
  - Reduce client-perceived delays
  - Reduce load on server
- Cache Replacement Strategy
  - When a cache becomes full, which object should be **evicted/replaced**?

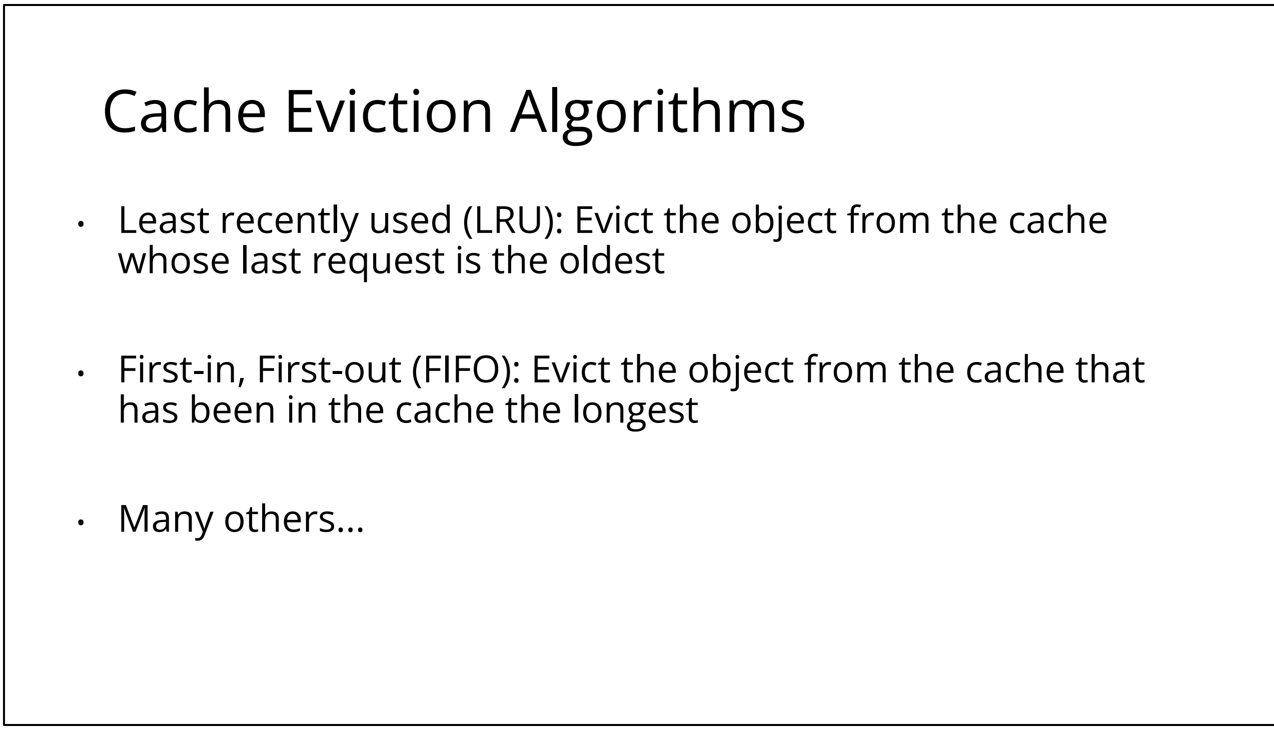


- Web caching is a specific instance of caching where objects are requested over the web and stored at one or more caches between the client and the server
- The cache may sit somewhere in the network between the client and the server
- Or the cache may be on the client itself, meaning the client has decided to store objects from previous requests
- This is done for several reasons:
  - First, it reduces network bandwidth usage
    - The network is a shared resource and if less data has to go over many of its connections, it can reduce congestion
  - Second, it reduces delays
    - Remember that caches are closer to clients than the servers are. This means that when a client needs to request an object, it will receive it faster from a cache
  - Third, it reduces load on the server
    - Simply put: If requests are handled by a cache, that means they don’t need to be handled by a server.
- But caches require management
- Caches are necessarily limited in size for various reasons, principal among them being that storage is expensive
- What happens when we want to insert an object into a cache that’s already full?
- We have to evict something!

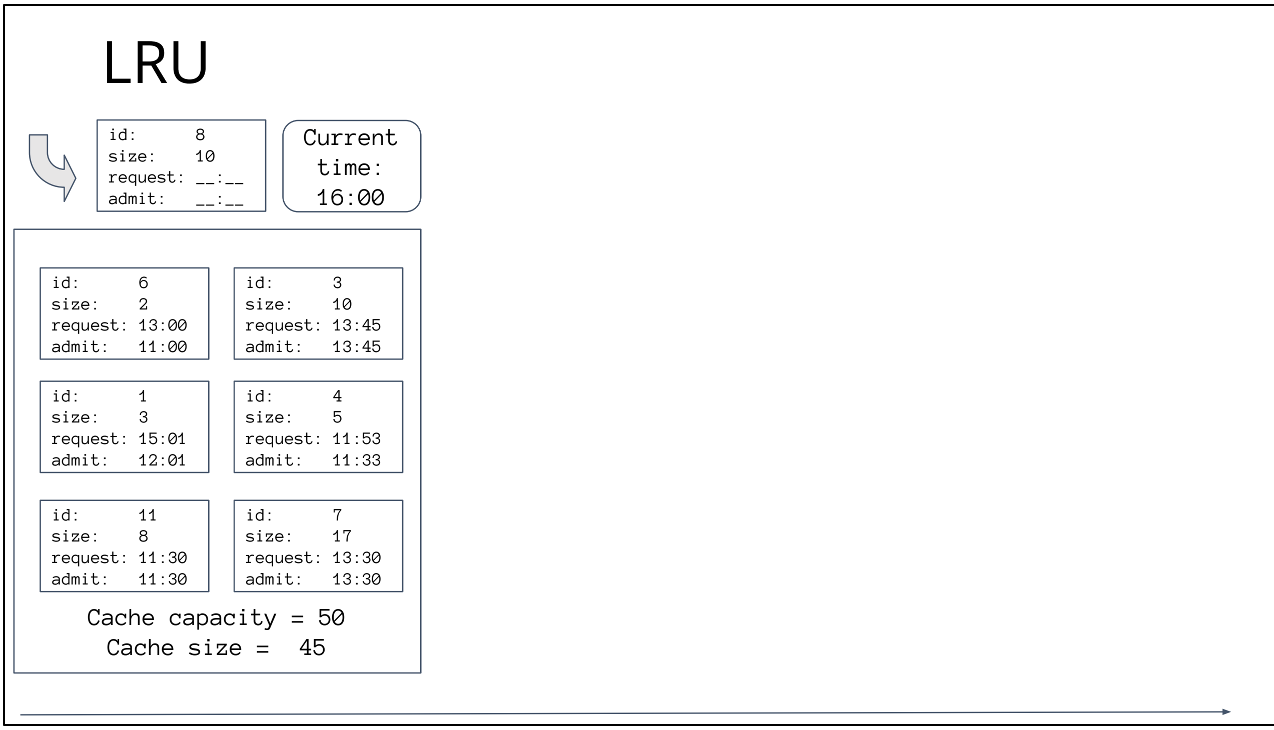
- It turns out that what we choose to evict from a cache is important and there’s an entire field of research based around this question

Cache Eviction Algorithms

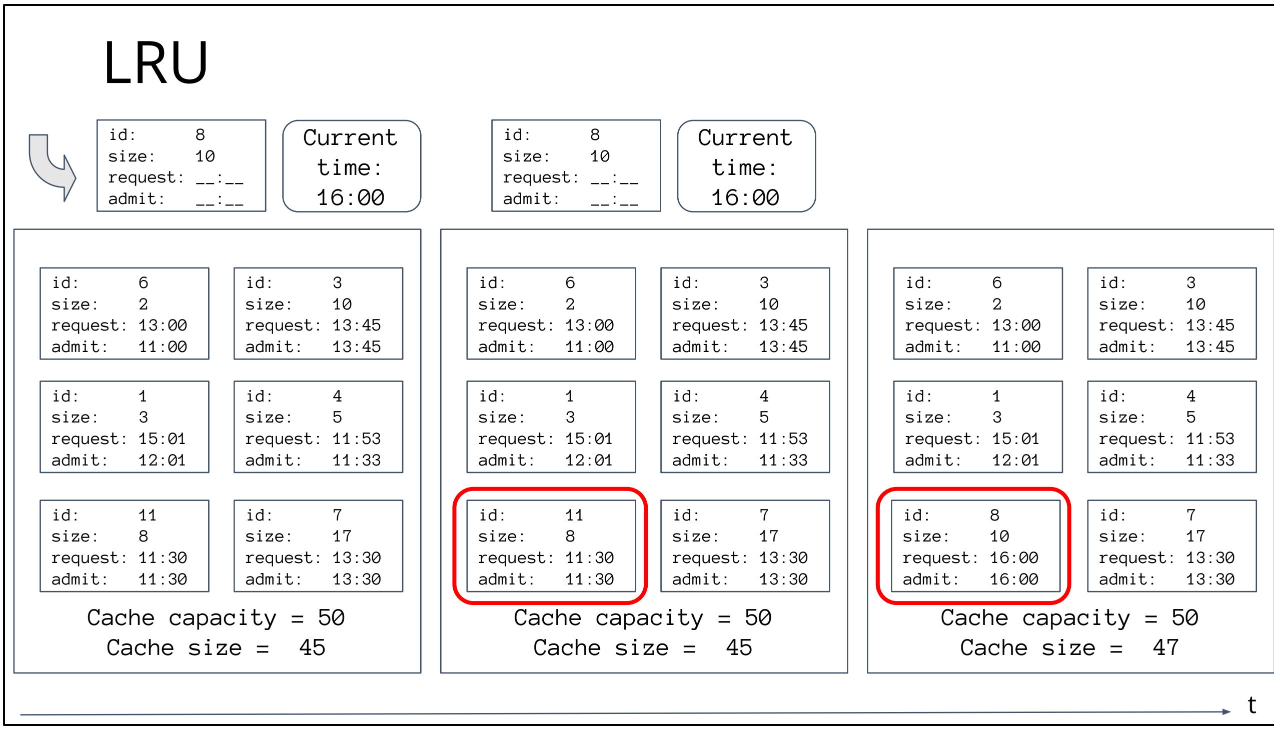
1. Client requests a new object
2. If object in cache
    1. return the object
3. Else:
    1. Get object from server/provider and return the object to client
    2. Attempt to insert the object into the cache:
        1. If Cache full:
            1. Identify an object in cache to evict
            2. Evict the object in the cache
            3. Replace with new object (insert new object)
    3. Admit the new object to the cache



- Here are a few important cache eviction algorithms that are commonly used and taught
- The first is least recently used, or LRU for short.
- LRU caches keep track of the last time an object was accessed.
- When an object needs to be evicted from the cache, it chooses the object whose most recent access is the furthest in the past
- LRU is a very intuitive algorithm; Just think about your closet.
- If you need to free up space in your closet, are you going to donate the shirt that you wore yesterday or the one that you can't even remember wearing?
- This kind of cache exploits temporal locality: Objects that were used recently are more likely to be used again
- Another algorithm that gets some use is FIFO. With this algorithm, you just evict the object that has been in the cache for the longest time
- There are many caching algorithms out there, each meant to maximize hit rate, which is the fraction of object requests that the cache can handle
- A high hit rate means that a cache is able to handle a large fraction of the requests
- A low hit rate means that most of the requests end up being handled by the server

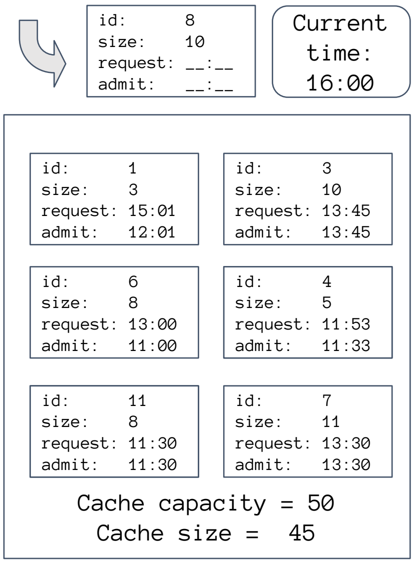


- Here we have an example of the LRU eviction algorithm in action
- This is a cache with capacity 50 units
- Each of the smaller rectangles is an object
- Notice the metadata associated with each object
- They each have an ID (just so we know what object we're referring to), size, most recent request time, and admission time
- In this example, the cache currently contains 45 units of objects when an object of size 10 comes in
- This would push the cache over its capacity, so something has to evicted
- Since this is an example of LRU, we have to look at the object that's currently in the cache with the oldest request time
- That is the object with id 11
- So we evict it!
- We then have enough space to place object 8

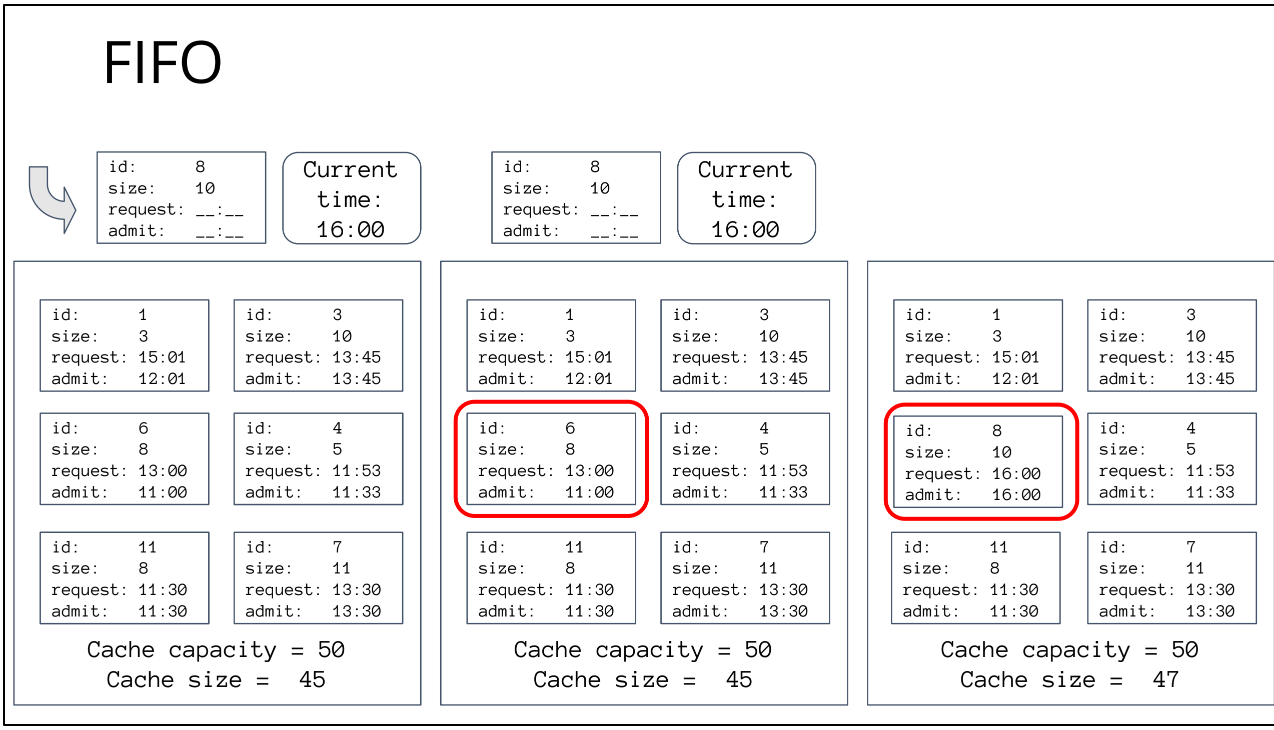


- Here we have an example of the LRU eviction algorithm in action
- This is a cache with capacity 50 units
- Each of the smaller rectangles is an object
- Notice the metadata associated with each object
- They each have an ID (just so we know what object we're referring to), size, most recent request time, and admission time
- In this example, the cache currently contains 45 units of objects when an object of size 10 comes in
- This would push the cache over its capacity, so something has to evicted
- Since this is an example of LRU, we have to look at the object that's currently in the cache with the oldest request time
- That is the object with id 11
- So we evict it!
- We then have enough space to place object 8

FIFO



- Next we have an example of the FIFO eviction algorithm
- Again, we have objects that keep track of all of the same metadata
- Again, an object comes in whose insertion would push the cache over its capacity
- We look for the object with the oldest admit time and evict it!



- Next we have an example of the FIFO eviction algorithm
- Again, we have objects that keep track of all of the same metadata
- Again, an object comes in whose insertion would push the cache over its capacity
- We look for the object with the oldest admit time and evict it!

Experiments

› Download exercise code from ed

› cd precept6/webcachesim-master

› make

- There's a web cache simulator on ed that we can use to experiment with different algorithms and cache sizes

Trace File Form

- Request traces must be given in a space-separated format with three columns
- time - long long int
- id - long long int, used to uniquely identify objects
- size should be a long long int, object's size in bytes

- Example

time	id	size
1	1	120
2	2	64
3	1	120
4	3	14
4	1	120

- See `test.tr`

Using the Simulator*

```
> ./webcachesim test.tr LRU 1000

LRU:1000 bytes, 10492 reqs, 8495 hits, 81 hits/reqs(%)

> ./webcachesim test.tr FIFO 1000

FIFO:1000 bytes, 10492 reqs, 8206 hits, 78 hits/reqs(%)
```

* Derived from https://github.com/dasebe/webcachesim

- Now we can see how these eviction algorithms perform by running the above commands

Experiments

- LRU and FIFO
- Vary cache sizes
  - 80
  - 160
  - 320
  - 640
  - 1280
  - 2560
  - 5120

- Create a Google Sheet
- Three columns
- SIZE LRU FIFO
- Copy results accordingly
- Select three columns to create line chart

Experiments