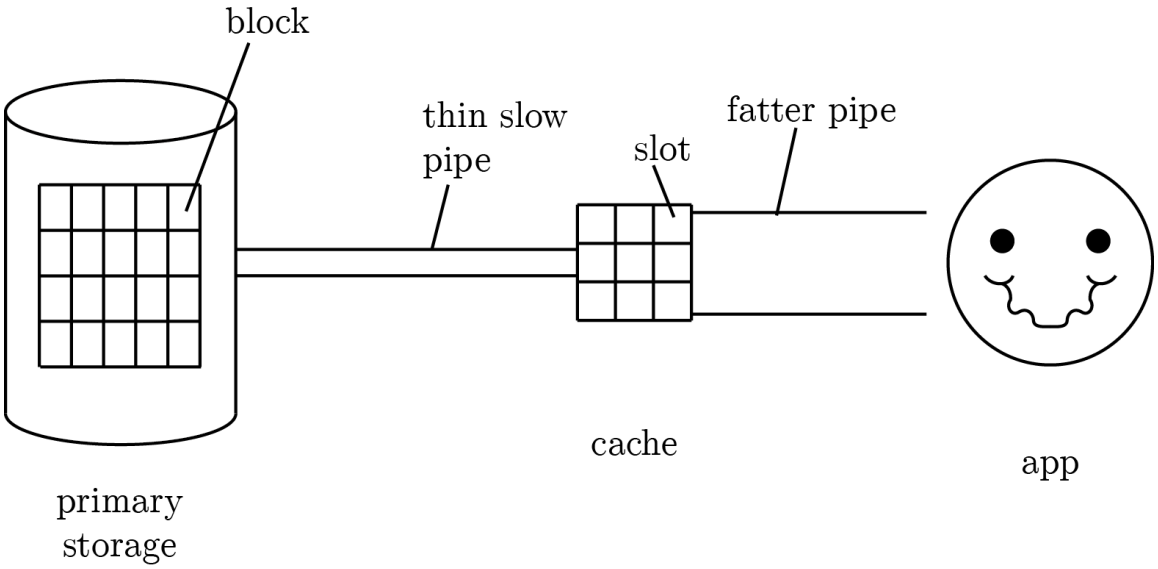
Today, we are going to develop pseudo code for how cache might work.

In the generic picture of a cache, we have:



Caching is a very general idea. In the caches we are going to focus on in class, the blocks have fixed size B.

Cache is fast local storage that is used to speed up access to primary storage. Cache is divided into slots. Slots have indices while storage has addresses. This is a hierarchy, so there are caches all the way up and all the way down.

What's a buffer cache?
It's cache that the operating system uses to store files. The OS uses most of memory as a buffer cache.

Analogies:
Your refrigerator is a cache for the grocery store. Your refrigerator is speeding up your access to food. Your digestive system/fat is used as cache for energy. What happens if you add a toilet in there??

Let's stay at the generic level. Let's say our cache has S slots total. In C-like pseudo code:

```
struct cache_slot {
        address addr;
        char* data; (or  char data[B])
        (additional information for eviction policy)
        time access_time;
        doubly_linked_list_nodes;
```

```
};

cache_slot cache[s];
```

What's is the algorithm we use to access caches?

First, how do we read data from the cache?

```
char* read (address a) {
        foreach slot s∈ [0, s):
        if (cache[s].addr == a)
                return cache[s].data
        s = eviction_policy();); // eviction policy: free up some slots
        dirty write back

        read_block(a, cache[s].data;
        cache[s].dirty = 0;

        return cache[s].data;
        cache[s].addr = a;
}
```

High level overview:
Look for the a in the cache.

Now let's talk about eviction policy:
Maximize the amount of hits in the cache. (as few misses as possible)

Work with reference strings (list of addresses): 012301401234

I want to build a cache that speeds up this reference string. Suppose we have s=3 slots.

A cold miss means that the data that misses was never in the cache.

The simplest policy is to use a queue. Evict the first thing that you load. FIFO (first in first out). Evict the slot loaded furthest in the past.

What is the hit rate for this reference string?

0	1	2	3	0	1	4	0	1	2	3	4
0m	0	0	3m	3	3	4m	4	4	4	4	4h
	1m	1	1	0m	0	0	0h	0	2m	2	2
		2m	2	2	1m	1	1	1h	1	3m	3

The hit rate for us is: 3/12=25%.
How many of these misses are cold misses? 5 (we're not considering pre-fetching)
The others are capacity misses. They are called such because these misses go away if the capacity increases.

Belady's anomaly: if we use a FIFO eviction policy, the hit rate can actually decrease if we increase capacity. Let's increase the capacity to 4, the same input string with FIFO eviction policy now gives:

0	1	2	3	0	1	4	0	1	2	3	4
0m	0	0	0	0h	0	4m	4	4	4	3 m	3
	1m	1	1	1	1h	1	0m	0	0	0	4m
		2m	2	2	2	2	2	1m	1	1	1
			3m	3	3	3	3	3	2m	2	2

The hit rate has dropped to 2/12=17%!

Optimal policy:
Look ahead and evict the slot that is furthest in the future. Let's look at the hit rate for that:

0	1	2	3	0	1	4	0	1	2	3	4
0m	0	0	0	0h	0	0	0h	0	0	3m	3
	1m	1	1	1	1h	1	1	1h	1	1	1
		2m	2	2	2	2	2	2	2h	2	2
			3m	3	3	4m	4	4	4	4	4h

The hit rate is now 6/12=50%. That is the best we could possibly do with this cache.

What is the smartest reasonable policy that doesn't require reading the future?

Aside
There are some operating systems that allow apps to provide information about "the future".

For example, Windows "super fetch": track past accesses based on past accesses.

The buffer cache is shared among different applications. The processor cache is normally partitioned among algorithms.

It is common for eviction policy to take in hints.

Caches benefit when past accesses are a good predictor of future accesses.

In reality, without hints, the best we can reasonably do it the Least recently used (LRU) policy: we evicts slots used least recently in the past. It's not optimal but better than FIFO. Let's check its hit rate:

0	1	2	3	0	1	4	0	1	2	3	4
0m	0	0	0	0h	0	0	0h	0	0	0	4m
	1m	1	1	1	1h	1	1	1h	1	1	1
		2m	2	2	2	4m	4	4	4	3m	3
			3m	3	3	3	3	3	2m	2	2

It's 4/12=33%.

Writes

When we write we need a dirty flag, which is set to be 1 if data is not yet written to disk, in which case that data needs to be flushed to disk before it can be evicted.

```
void write(address a, modification mod) {
        foreach slot s∈ [0, s):
        if (cache[s].addr == a)
                goto done
        s = eviction_policy(); // eviction policy: free up some slots
        *

        [read_block(a, cache[s].data];
        cache[s].addr = a;
        done: modify cache[s].data;
        cache[s].dirty = 1;
}
```

Many files are read many more time than they are written

```
struct cache_slot {
        address addr;
        char* data; (or  char data[B])
        (additional information for eviction policy)
        time access_time;
        bool dirty;
};

*if (cache[s].dirty)
        write.block(cache[s].addr, cache[s].data);
```

Write back cache is a cache that delays writing. Write through cache writes it to primary storage straight away (and wouldn't need a dirty flag). But if you made several modifications to the same address you are not benefiting from batching.

Let's try this sequence:
0r 1r 2w 3r 4r

0r	1r	2w	3r	4r
0m	0	2m *dirty*	2 *dirty*	4m *2 written to disk*
	1m	1	3m	3

It's faster to evict clean slots rather than dirty slots, so our eviction policy should take that into account.

Cache correctness
Gives you some version of the data on primary storage and eventually write data back to primary storage.

Coherent cache and incoherent cache (both are correct)

Buffer cache is coherent. The OS makes sure that it is the only one that can access memory. It is doing work in maintaining coherence.
stdio cache is not coherent. It's just easier to maintain incoherent caches.

If there are two programs that are writing to files, one of them will clobber the other.

Something awesome you can do in problem set 2:

Preview: OS uses primary memory for application memory too.

File I/O at the speed of memory can be achieve with memory map: mapping files to memory and modifying it there. But the only files that can be memory mapped are files in the buffer cache. You can't use it for pipes or network connections.