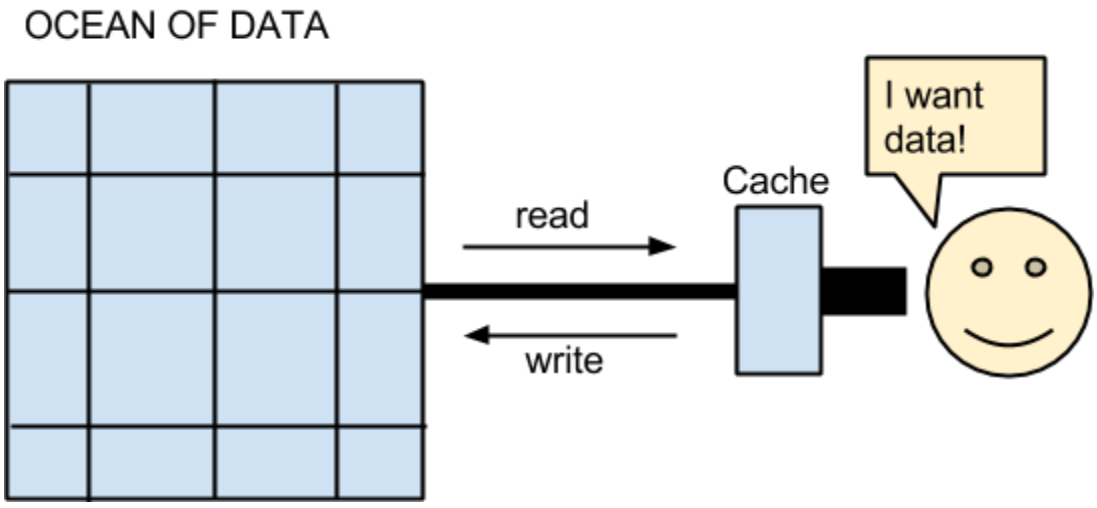
CS61 - Lecture 9 - September 30, 2014

Administrivia: Pset 1 grades will be released today!

Subjects:

- Models of Caches
- Processor Caches

Scenario: “ocean of data” that is far from the application that wants the data
In between the data and the application, there exists a thin pipe
To make moving data faster, we put a cache of data closer to the application



Data moves between the ocean of data and the cache in units called blocks
Block: a cacheable unit of data (lives in primary storage, some live in cache)

- The cache isn’t big enough to hold everything
- Data moves in “blocks”

Slot: a location in the cache that can hold one block

B: Block size (=64 for now)

```
void read_block (char *buf, off_t addr)
  // storage to memory: GIVEN
  // Fills [buf[0], buf[B]) with data from
  // storage starting at addr
  // This function is given to us
void read_byte (char *buf, off_t pos) {
  // Read storage @ pos into * buf
  // We must implement this function
```

```
  char cbuf[B];
  read_block(&cbuf[0], pos);
  memcpy(buf, cbuf, 1);
}
```

Note: off_t is a type for file positions (kind of like size_t)

Disadvantage: we are reading a whole block every time we try to read one byte

Random access pattern is hard to optimize using a cache, especially if size of primary storage is large.

Sequential access is easy to optimize using a cache. An example of a simple cache:

```
struct cache {
  char cbuf[B];
  off_t tag;
  int valid; // binary value that indicates whether or not the tag is meaningful
}
```

Our cache needs
-actual space for holding data (cbuf)
-a version of the position the data came from (tag)
-an indicator if there is stuff in the cache or not (valid)

How do we optimize read_byte using the above cache?
Assume that we have a global cache:

```
GLOBAL struct cache c;
```

This is a BAD way to try to use the cache:

```
void read_byte (char *buf, off_t pos) {
  read_block(&c.cbuf[0], pos);
  memcpy(buf, cbuf, 1);
}
```

What about this? (this is a blank version)

```
int cache_contains(off_t pos){

}
void read_byte (char *buf, off_t pos) {
  if (cache.contains(pos))
    goto found;
  read_block(&c.cbuf[0], _______);
  c.tag = ________;
```

```
  c.valid = 1;

  found: memcpy(buf, _______, 1);
}
```

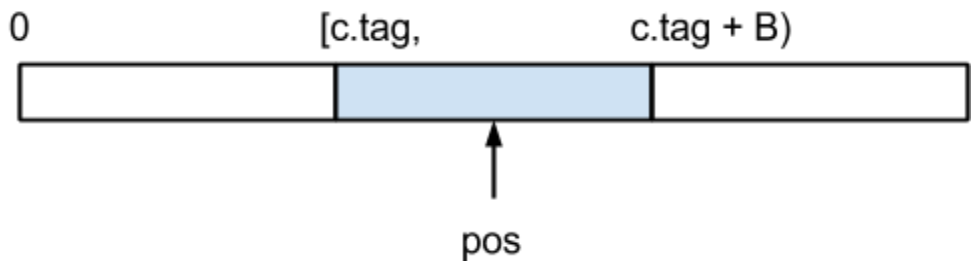
Here is the complete version:

```
int cache_contains(off_t pos){
  return pos >= c.tag && pos < c.tag + B && c.valid;
}
void read_byte (char *buf, off_t pos) {
  if (cache.contains(pos))
    goto found;
  read_block(&c.cbuf[0], pos);
  c.tag = pos;
  c.valid = 1;

  found: memcpy(buf, c.cbuf + pos - c.tag, 1); //assert cache_contains(pos)
}
```

If valid is 0, the cache does not contain meaningful data. If valid is 1, then we have B bytes of data starting at c.tag.

The tag tells us what range of data in storage is currently in the cache.

If c.valid is 0, then the cache is meaningless. If c.valid is 1, then the cache stores the data in the half-open interval [c.tag,c.tag+B). . Thus, cbuf[0] contains the data at c.tag, cbuf[1] contains the data at c.tag+1, and so on. Hence, to get data at pos, we calculate the offset pos-c.tag and go that many bytes into the cache.

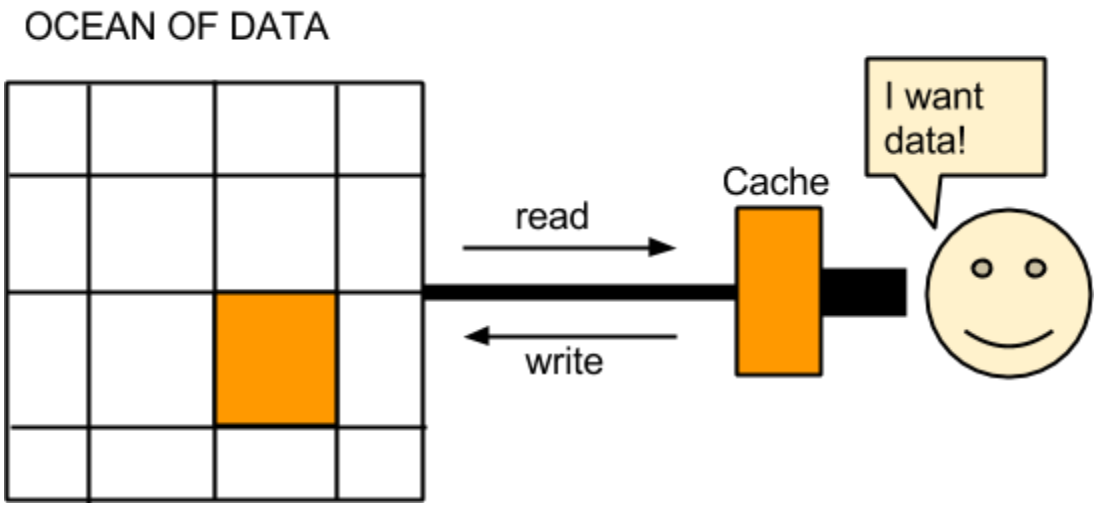


We need to make the following assertion at found (see the comment at the line in the code):

```
assert(cache_contains(pos));
```

This assert is redundant but you put it in anyway because it's an assumption, not because it may be false.

This is what the cache should look like when the cache is storing some data:



This cache does not work for reverse sequential accesses:

```
Value requested -> data stored by cache
19 -> [19, 83)
18 -> [18, 82)
17 -> [17, 81)
//this is very inefficient
.
.
```

How do we make the cache work for reverse sequential access? The key has to do with alignment. Load blocks into the cache aligned based on B.

So new read_byte:

```
void read_byte (char *buf, off_t pos) {
  if (cache.contains(pos))
    goto found;
  read_block(&c.cbuf[0], pos - pos % 64);
  c.tag = pos - pos % 64;
  c.valid = 1;

  found: memcpy(buf, c.cbuf + pos - c.tag, 1); //assert cache_contains(pos)
}
```

There is no universal answer to what's the best cache: there are only good answers for certain access patterns.

For reverse sequential access, this new cache makes one call to read_block per every 64 calls to read_byte.

HIT RATE
Hit: An access when the cache is useful
Miss: An access when the cache is not useful
Hit rate: #hits / #accesses
Better hit rate means better cache

Apparently everybody's name begins with A and not B.

Analogy: a paper encyclopedia is a collection of books, each of which has a letter on its spine
The book that has "A" on the spine knows everything about things that start with "A,"
but you can only hold one book at once, and the books are far away at the library.
When you ask about things sequentially, you don't have to go to the library very often.
When you ask about things in reverse sequential order, you also don't have to go the library very often.

Encyclopedias A and J

Sequential Access:
-Apple?
-A'a?
-Alimentary Canal?
-Jeans?
-Jackalopes?
5 accesses, 3 hits

Reverse Sequential Access
-Jerks?
-Ju?
-Ack?
3 accesses, 1 hit

The tag is the first letter, and the block size is the size of the encyclopedia. The alignment in this case is that everything has a first letter.

The way we were doing it before was analogous to ripping pages out of the encyclopedias, potentially out of multiple encyclopedias, and taking those home instead of taking the entire encyclopedia home.

What's an access pattern that doesn't work so well for our cache now?

```
0, 64, 128, 192, 256,
1, 65, 129, 193, 257,
2, 66, 130, . . .
This is called a strided access pattern, and it is important for big data because of
```

```
matrices
```

How do we make our cache work with this strided access pattern? Prof. Kohler claims we can give it a 63/64 hit rate.
We can store more than one slot in the cache!

New cache:

```
struct cache {
  struct slot {
    char cbuf[B];
    off_t tag;
    int valid; // binary value that indicates whether or not the tag is meaningful
  } s[5];
}
```

Changes to read_byte and cache_contains:

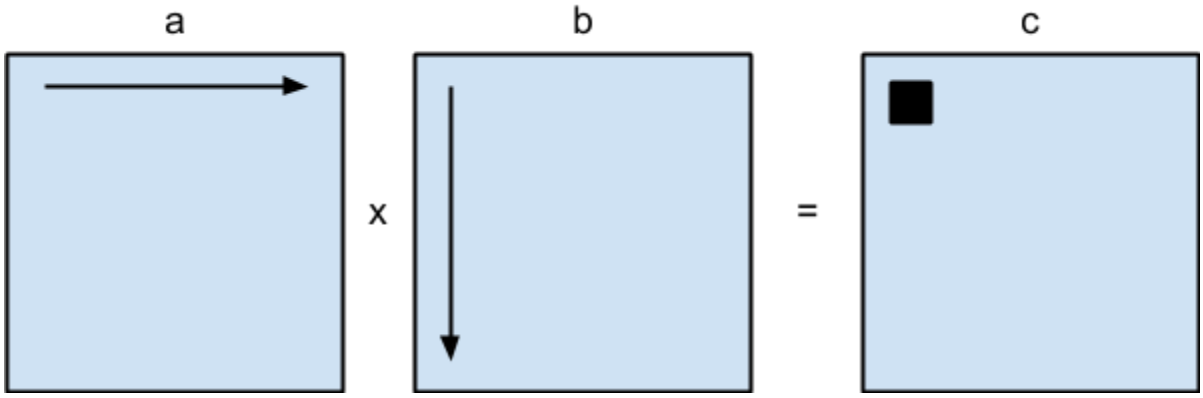
```
int cache_contains(struct slot *s, off_t pos){
  return pos >= s->tag && pos < s->tag + B && s->valid;
}
void read_byte (char *buf, off_t pos) {
  // Read storage @ pos into *buf;
  struct slot *s;
  for (int i = 0; i < 5; i++) {
    if (cache.contains(&c.s[i], pos)) {
      s = &c.s[i];
      goto found;
    }
  }
  // In this new cache, we need to choose which slot to put the data in
  s = eviction_policy(); // what type of eviction policy? see notes below
  read_block(&s->cbuf[0], pos - pos % 64);
  s->tag = pos - pos % 64;
  s->valid = 1;

  found: memcpy(buf, s->cbuf + pos - s->tag, 1); //assert cache_contains(s, pos)
}
```

How do we decide where to put data that hasn't been cached (how do we define our eviction policy?)

Different options for eviction policies:

- Remove the slot that has been accessed the least
- Remove the one that is furthest on the disc
- First in first out (remove slot that was loaded longest ago)
- Randomly choose one

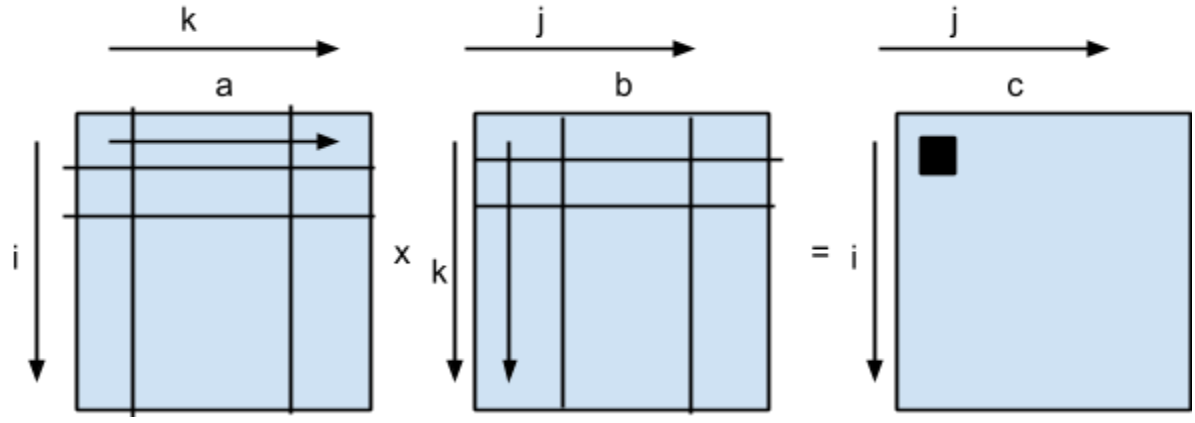
Matrix Multiplication:



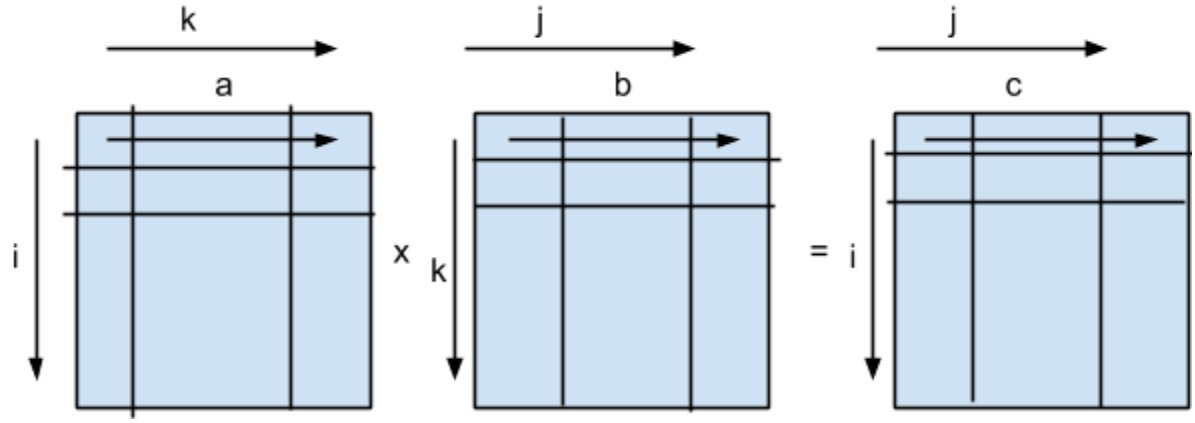
In the lecture code, the matrices are stored one-dimensionally. The way in which the data is stored here is called row-major order.
To access, m[i, j], need to access m[i*sz + j]
At this point, Prof. Kohler draws horizontal zigzags across matrix a to demonstrate row-major order - we are not going to do that (he erases the zigzags a few minutes later anyways).

matrixmultiply-ijk and matrixmultiply-ikj are virtually the same codewise, but ikj is much faster. Why? The k's and the j's are switched in the loops, with a speed difference of a factor of 1000.

ijk:



ikj:



The processor has its own cache that is organized in terms of aligned addresses. The ijk multiplication creates a strident access pattern by traversing the column of the matrix, which requires many different blocks of memory to be loaded into the processor's cache. On the other hand, the ikj multiplication does not require strided access.

Notes compiled by Dan Fu, Jimmy Jiang, Yuan Jiang, and Andrew Mauboussin