15-213 Recitation 7
Caches and Blocking

8 October 2018

Agenda

- Reminders
- Revisiting Cache Lab
- Caching Review
- Blocking to reduce cache misses
- Cache alignment

Reminders

- Due Dates
  - Drop Date (Today 10/8)
  - Cache Lab (Thursday 10/11)
  - Midterm Exam (Monday 10/15 – Friday 10/19)
- Practice Problems
  - Exam Server
  - Website (32-bit, but still useful)
- Midterm Review Session
  - Sunday 10/14

Reminders: Cache Lab

- Part 1: Write a cache **simulator**
  - Substantial amount of C code!
- Part 2: Optimize some code to minimize cache misses
  - Substantial amount of thinking!
- Part 3: Style Grades
  - Worth about a letter grade on this assignment
  - Few examples in appendix
  - Full guide on course website
  - Git matters!

Cache Lab: Parsing Input with fscanf

- fscanf(FILE *stream, const char *format, …)
  - “scanf” but for files

- Arguments
  1. A stream pointer, e.g. from fopen()
  2. Format string for parsing, e.g “%c %d,%d”
  3+. **Pointers** to variables for parsed data
     - Can be pointers to stack variables

- Return Value
  - Success: # of parsed vars
  - Failure: EOF
- man fscanf

fscanf() Example

```
FILE *pFile;
pFile = fopen(“trace.txt”, "r"); // Open file for reading

// TODO: Error check sys call

char access_type;
unsigned long address;
int size;

// Line format is " S 2f,1" or " L 7d0,3"
//      - 1 character, 1 hex value, 1 decimal value
while (fscanf(pFile, " %c %lx, %d", &access_type, &address, &size) > 0)
{
    // TODO: Do stuff
}

fclose(pFile); // Clean up Resources
```

Cache Lab: Cache Simulator Hints

- Goal:
  - Count hits, misses, evictions and # of dirty bytes
- Procedure
  - Least Recently Used (LRU) replacement policy
  - Structs are great ways to bundle various parts of cache line (valid bit, tag, LRU counter, etc.)
  - A cache is like a 2D array of cache lines

```
struct cache_line cache[S][E];
```

- Your simulator needs to handle different values of S, E, and b (block size) given at run time
  - Dynamically allocate memory!

Class Question / Discussions

- We’ll work through a series of questions
- Write down your answer for each question
- You can discuss with your classmates

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void who(int *arr, int size) {
  for (int i = 0; i < size-1; ++i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void who(int *arr, int size) {
  for (int i = 0; i < size-1; ++i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void coo(int *arr, int size) {
  for (int i = size-2; i >= 0; --i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

What Type of Locality?

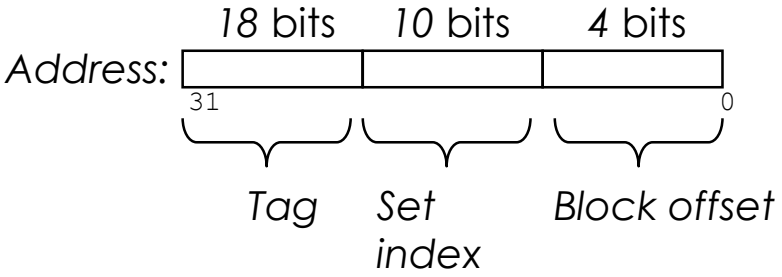
- The following function exhibits which type of locality? Consider *only* array accesses.

```
void coo(int *arr, int size) {
  for (int i = size-2; i >= 0; --i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

Calculating Cache Parameters

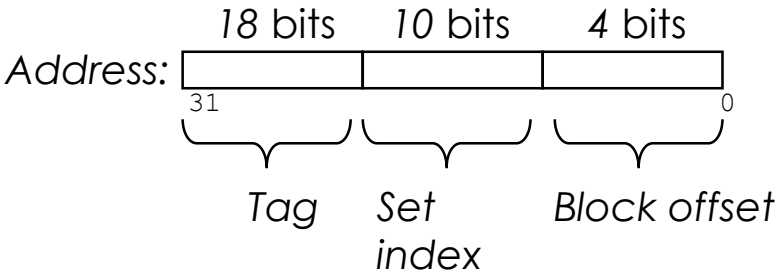
- Given the following address partition, how many `int` values will fit in a single data block?



	# of int in block
A.	0
B.	1
C.	2
D.	4
E.	Unknown: We need more info

Calculating Cache Parameters

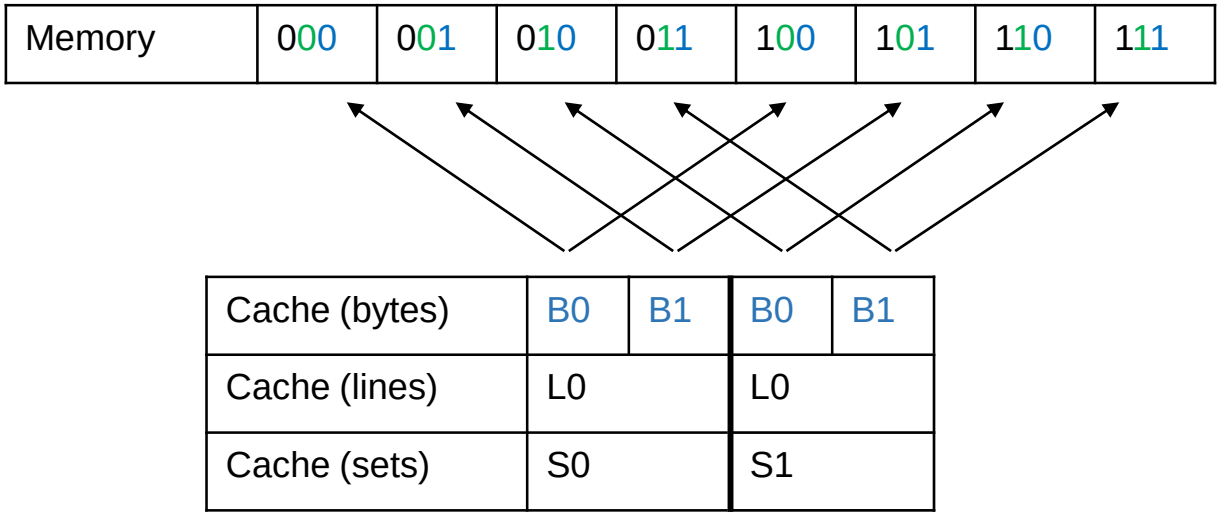
- Given the following address partition, how many `int` values will fit in a single data block?



	# of int in block
A.	0
B.	1
C.	2
D.	4
E.	Unknown: We need more info

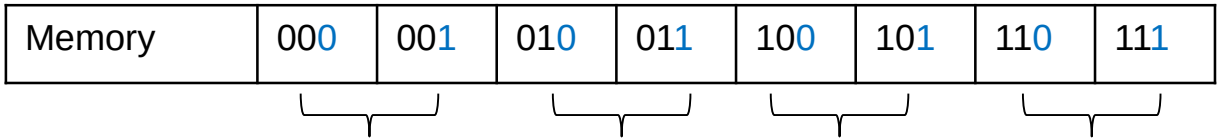
Interlude: terminology

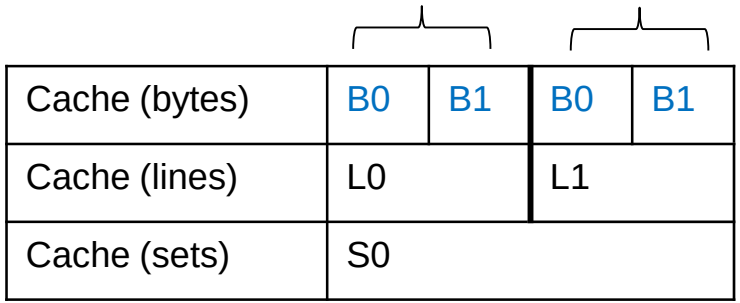
- A **direct-mapped** cache only contains one line per set. This means E = 2^e = 1.



Interlude: terminology

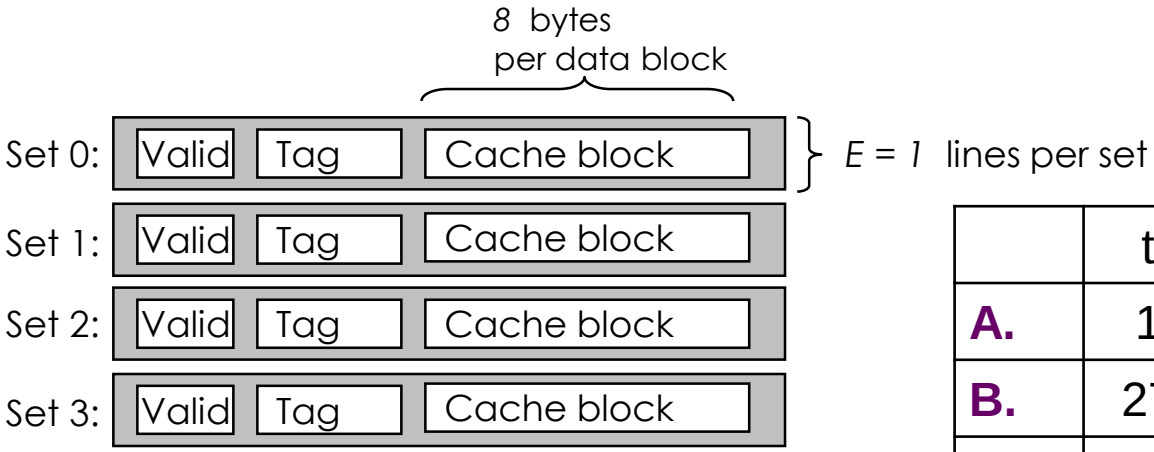
- A **fully associative** cache has 1 set, and many lines for that one set. This means S = 2^s = 1.





Direct-Mapped Cache Example

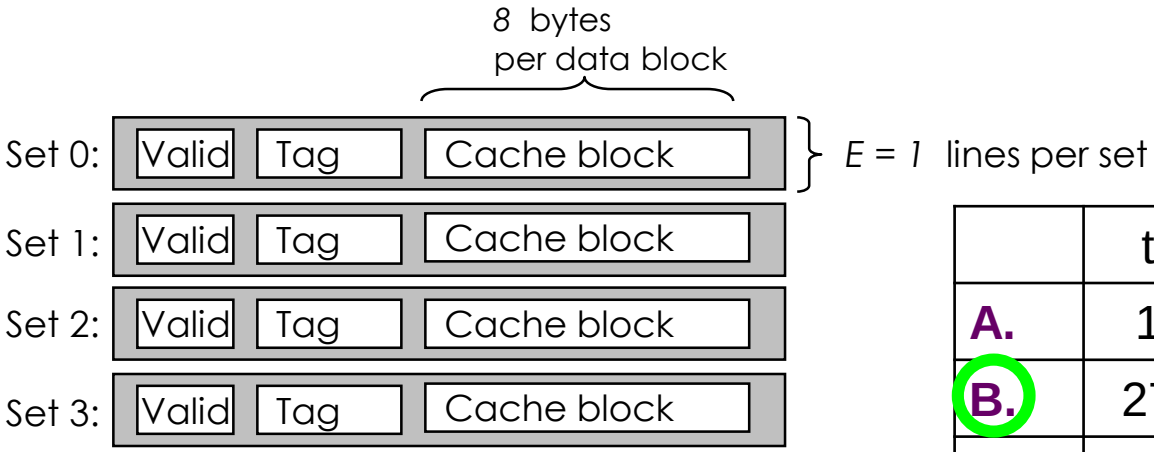
- Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).



	t	s	b
A.	1	2	3
B.	27	2	3
C.	25	4	3
D.	1	4	8
E.	20	4	8

Direct-Mapped Cache Example

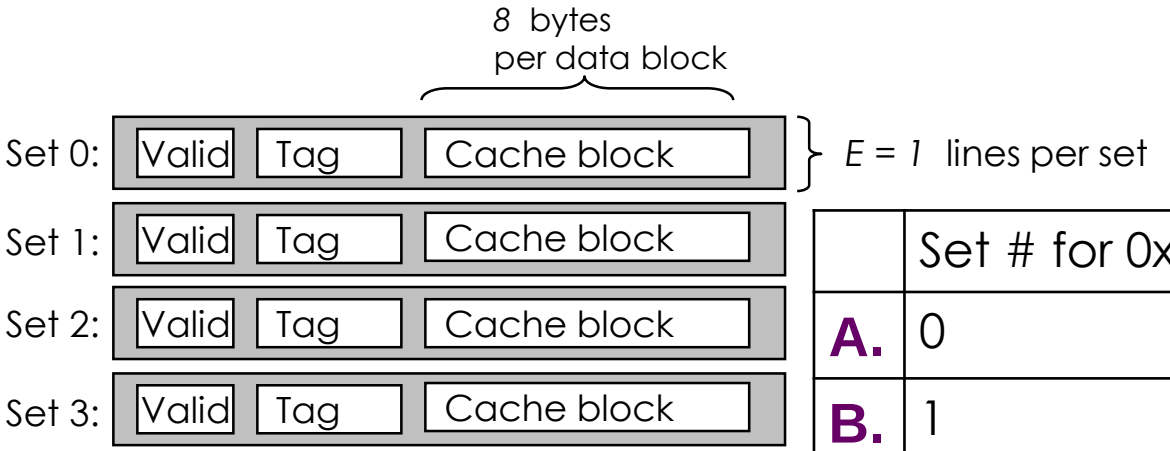
- Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).



	t	s	b
A.	1	2	3
B.	27	2	3
C.	25	4	3
D.	1	4	8
E.	20	4	8

Which Set Is it?

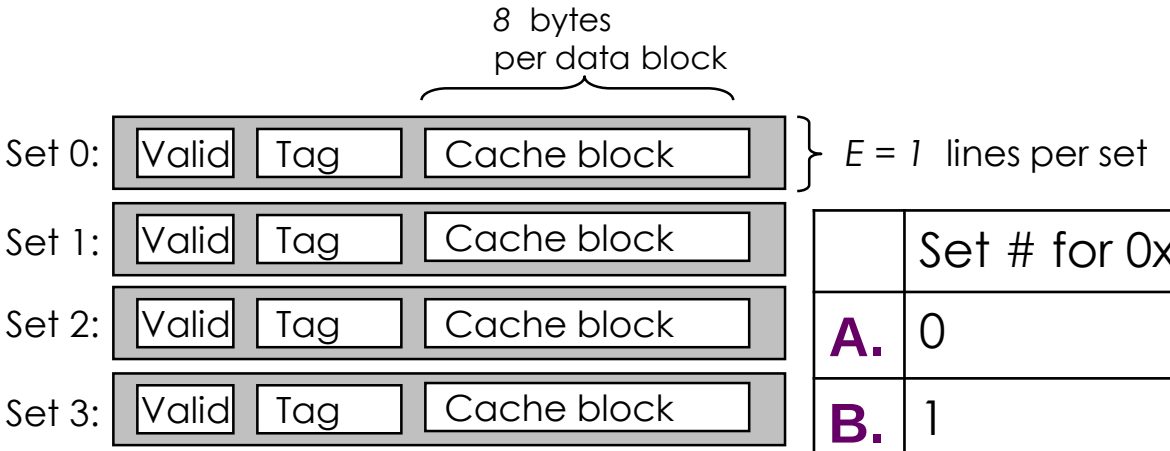
- Which set is the address **0xFA1C** located in?

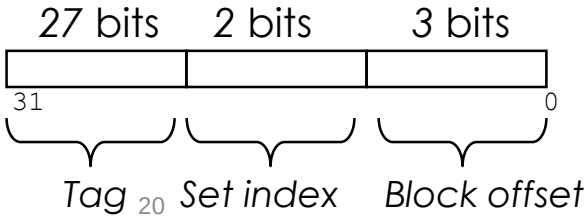


	Set # for 0xFA1C
A.	0
B.	1
C.	2
D.	3
E.	More than one of the above

Which Set Is it?

- Which set is the address **0xFA1C** located in?

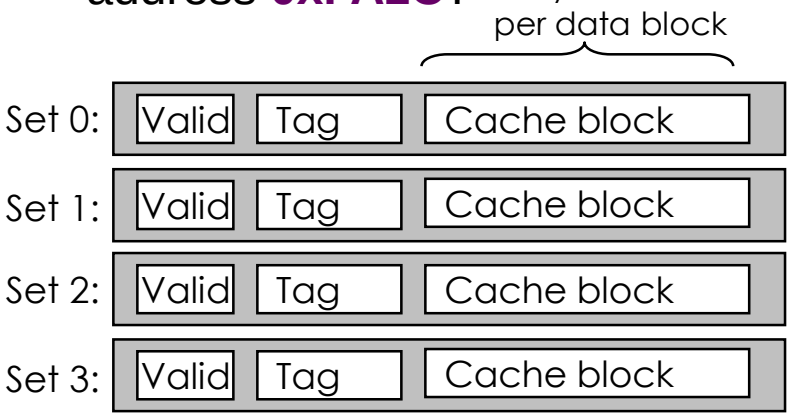


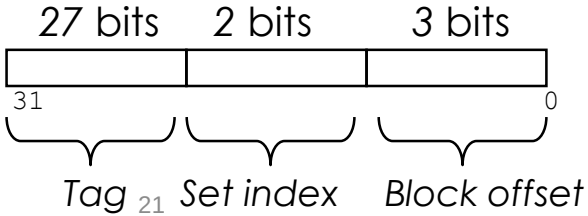


	Set # for 0xFA1C
A.	0
B.	1
C.	2
D.	3
E.	More than one of the above

Cache Block Range

- What range of addresses will be in the same block as address **0xFA1C**?

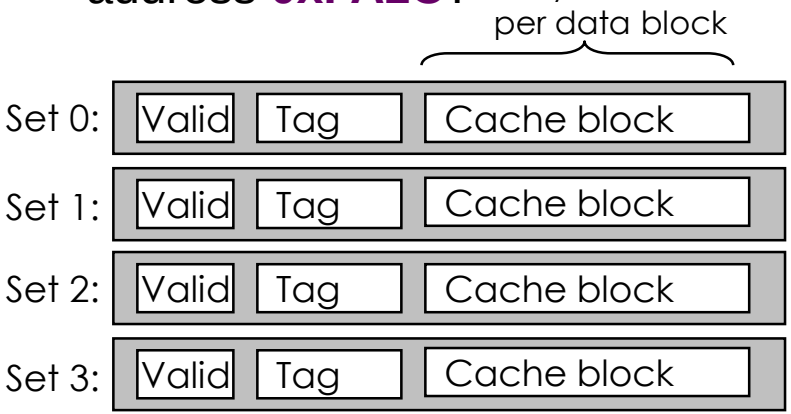


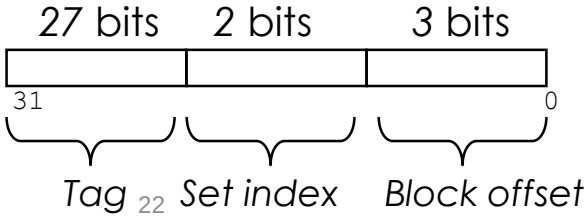


	Addr. Range
A.	0xFA1C
B.	0xFA1C – 0xFA23
C.	0xFA1C – 0xFA1F
D.	0xFA18 – 0xFA1F
E.	It depends on the access size (byte, word, etc)

Cache Block Range

- What range of addresses will be in the same block as address **0xFA1C**?





	Addr. Range
A.	0xFA1C
B.	0xFA1C – 0xFA23
C.	0xFA1C – 0xFA1F
D.	0xFA18 – 0xFA1F
E.	It depends on the access size (byte, word, etc)

Cache Misses

If N = 16, how many bytes does the loop access of a?

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Accessed Bytes
A	4
B	16
C	64
D	256

Cache Misses

If N = 16, how many bytes does the loop access of a?

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Accessed Bytes
A	4
B	16
C	64
D	256

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 1’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
D	50 %
E	66 %

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 1’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
D	50 %
E	66 %

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 2’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
D	50 %
E	66 %

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 2’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
D	50 %
E	66 %

Detailed explanation in Appendix!

Cache-Friendly Code

- Keep memory accesses bunched together
  - In both time and space (address)
- The working set at any time should be smaller than the cache
- Avoid access patterns that cause conflict misses
- Align accesses to use fewer cache sets (often means dividing data structures into pieces whose sizes are powers of 2)

Cache Alignment

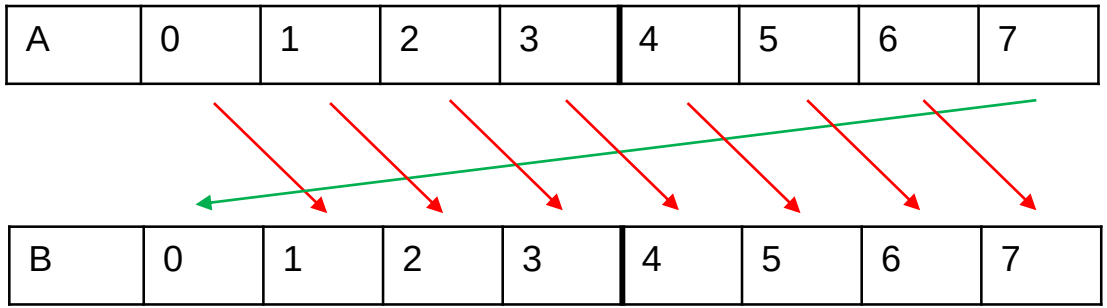
Suppose you have arrays:

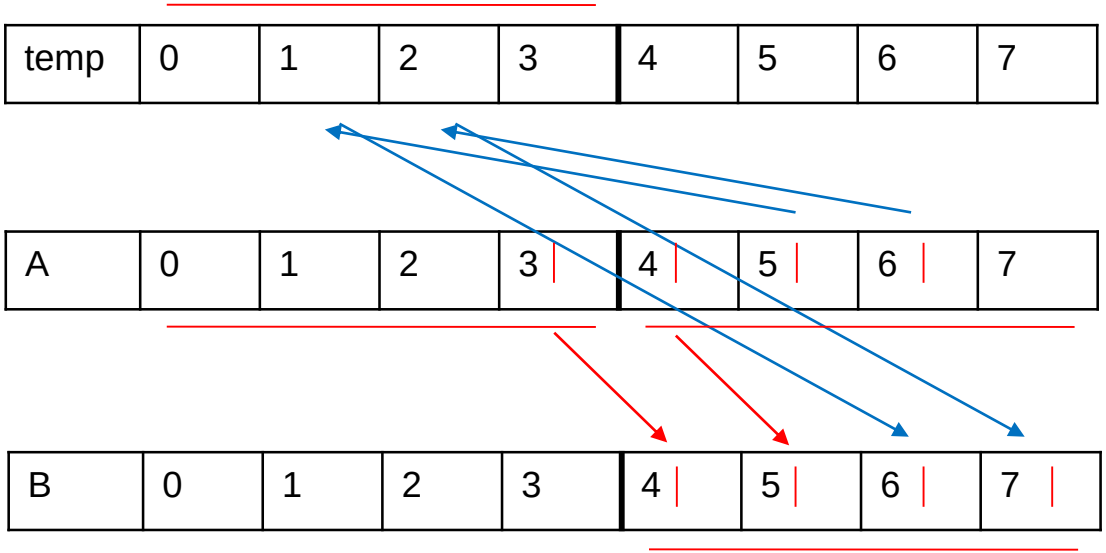
```
int[8] A, B, temp;
```

- `A[0]`, `B[0]` and `temp[0]` all correspond to byte 0 of set 0 on the cache. We say that all three arrays are cache-aligned.
  - For example, suppose we use a direct-mapped cache. If we request first `A[0]` then `B[0]`, the cache will evict the line containing `A[0]`.

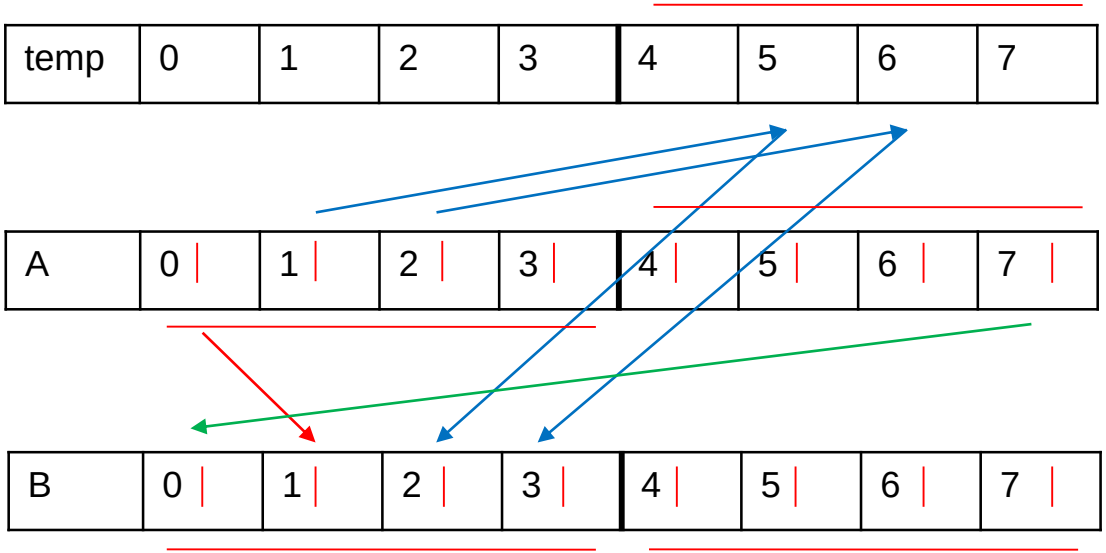
Very Hard Cache Problem

- We will use a direct-mapped cache with 2 sets, which each can hold up to 4 `int`'s.
- How can we copy A into B, shifted over by 1 position?
  - The most efficient way? (Use `temp`!)





Number of misses:



Number of misses:

← Could've been 16 misses otherwise!
We would save even more if the block size were larger, or if `temp` were already cached

If You Get Stuck

Please read the writeup

Read it again after doing ~25% of the lab

- CS:APP Chapter 6
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- `man malloc`, `man gdb`, gdb's `help` command
- http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf

Appendix: C Programming Style

- Properly document your code
  - Function + File header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - Use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - No magic numbers – use `#define` or `static const`
- Formatting
  - 80 characters per line (use Autolab's highlight feature to double-check)
  - Consistent braces and whitespace
- No memory or file descriptor leaks
- Git commit history

Appendix: Git Usage

- Commit early and often!
  - At minimum at every major milestone
  - Commits don’t cost anything!
- Popular stylistic conventions
  - Branches: short, descriptive names
  - Commits: A single, logical change. Split large changes into multiple commits.
  - Messages:
    - Summary: Descriptive, yet succinct
    - Body: More detailed description on **what** you changed, **why** you changed it, and what **side effects** it may have

Appendix: Blocking Example

- We have a 2D array `int[4][4] A;`
- Cache is fully associative and can hold two lines
- Each line can hold two `int` values

- Discuss the following questions with your neighbor:

- What is the best miss rate for traversing `A` once?
- What order does of traversal did you use?

- What other traversal orders can achieve this miss rate?

Appendix: Discussion Questions

- What did the optimal transversal orders have in common?
- How does the pattern generalize to `int[8][8] A` and a cache that holds 4 lines each of 4 `int`'s?

Appendix: Cache Misses

If there is a 48B cache with 8 bytes per block and 3 cache lines per set, how many misses if foo is called twice? N still equals 16.

NOTE: This is a contrived example since the number of cache lines must be a power of 2. However, it still demonstrates an important point.

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Misses
A	0
B	8
C	12
D	14
E	16

Appendix: Cache Misses

If there is a 48B cache with 8 bytes per block and 3 cache lines per set, how many misses if foo is called twice? N still equals 16.

NOTE: This is a contrived example since the number of cache lines must be a power of 2. However, it still demonstrates an important point.

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Misses
A	0
B	8
C	12
D	14
E	16

Appendix: 48KB Cache Explained (1)

We access the int array in strides of 8 (note the comment and the i += 8). Each block is 64 bytes, which is enough to hold 16 ints, so in each block:

```
| 8 ints = 32B  | 8 ints = 32B  |
+---------------+---------------+
|m| | | | | | |h| | | | | | | |
+---------------+---------------+
|       16 ints = 64B
```

The "m" denotes a miss, and the "h" denotes a hit. This pattern will repeat for the entirety of the array.

We can be sure that the second access is always a hit. This is because the first access will load the entire 64-byte block into the cache (since the entire block is always loaded if any of its elements are accessed).

So, the big question is why the first access is always a miss. To answer this, we must understand many things about the cache.

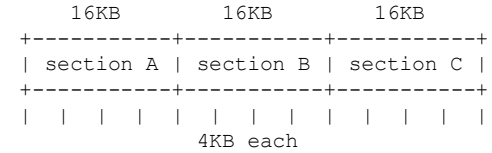
First of all, we know that s, the number of set bits, is 6, which means there are 64 sets. Since each set maps to 64 bytes (as there are b = 6 block bits), we know that every 64 * 64 bytes = 4 kilobytes we run out of sets:

```
   64B     64B             64B      64B
+-------+-------+--...--+--------+-------+--...
| set 0 | set 1 |       | set 63 | set 0 |
+-------+-------+--...--+--------+-------+--...
|        64 * 64B = 4KB          |
```

Clearly, this pattern will repeat for the entirety of the array.

Appendix: 48KB Cache Explained (2)

However, note that we have E = 8 lines per set. That means that even though the next 4KB map to the same sets (0-63) as the first 4KB, they will just be put in another line in the cache, until we run out of lines (i.e., after we've gone through 8 * 4KB = 32KB of memory). Splitting up the bigArr into 16KB chunks:

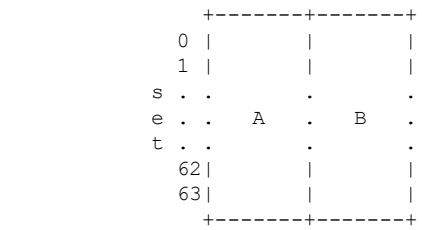


We see that section A will take up 16KB = 4 * 4KB; like we said, each of those 4KB chunks will take up 1 line each, so section A uses 4 lines per set (and uses all 64 sets).

Similarly, section B also takes up 16KB = 4 * 4KB; again, each of those 4KB chunks will take up 1 line each, so section B also uses 4 lines per set (and uses all 64 sets).

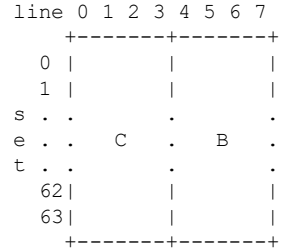
Note that as all of this data is being loaded in, our cache is still cold (does not contain any data from those sections), so the previous assumption about the first of every other access missing (the "m" above) is still true.

After we read in sections A and B, the cache looks like:



Appendix: 48KB Cache Explained (3)

However, once we reach section C, we've run out of lines! So what do we have to do? We have to start evicting lines. And of course, the least-recently used lines are the ones used to store the data from A (lines 0-3), since we just loaded in the stuff from B. So, first of all, these evictions are causing misses on the first of every other read, so that "m" assumption is still true. Second, after we read in the entirety of section C, the cache looks like:



Thus, we know now that the miss rate for the first pass is 50%.

Appendix: 48KB Cache Explained (4)

If we now consider the second pass, we're starting over at the beginning of bigArr (i.e., now we're reading section A). However, there's a problem - section A isn't in the cache anymore! So we get a bunch of evictions (the "m" assumption is still true, of course, since these evictions must also be misses). What are we evicting? The least-recently used lines, which are now lines 4-7 (holding data from B). Thus, the cache after reading section A looks like:

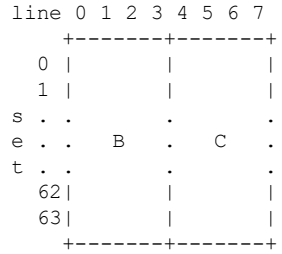
```
line 0 1 2 3 4 5 6 7
    +-------+-------+
  0 |       |       |
  1 |       |       |
s . .       .       .
e . .   C   .   A   .
t . .       .       .
  62|       |       |
  63|       |       |
    +-------+-------+
```

Then, we access B. But it isn't in the cache either! So we evict the least-recently-used lines (in this case, the lines that were holding section C, 0-3) (the "m" assumption still holds); afterwards, the cache looks like:

```
line 0 1 2 3 4 5 6 7
    +-------+-------+
  0 |       |       |
  1 |       |       |
s . .       .       .
e . .   B   .   A   .
t . .       .       .
  62|       |       |
  63|       |       |
    +-------+-------+
```

Appendix: 48KB Cache Explained (5)

And finally, we access section C. But of course, its data isn't in the cache at all, so we again evict the least-recently used lines (in this case, section A's lines, 4-7) (again, "m" assumption holds):



And so the miss rate is 50% for the second pass as well.

Thank you to Stan Zhang for coming up with such a detailed explanation!