CSE 351 Section 6 – Structs and Caches

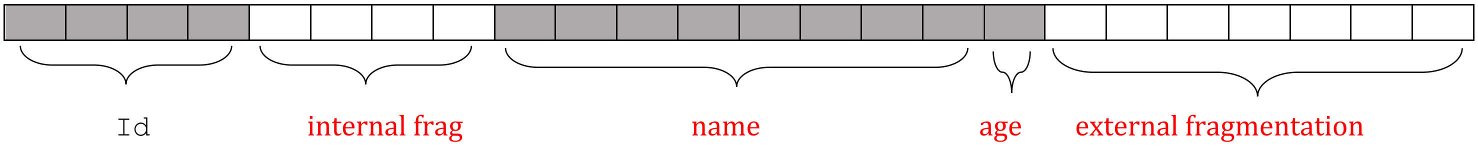
Hi there! Welcome back to section, we're happy that you're here ☺

Structs

- Structs are contiguously allocated chunks of memory that hold a programmer-defined collection of potentially disparate variables.
- Individual fields appear in the struct in the order that they are declared
- Each field follows its variable alignment requirement, with internal fragmentation added between fields as necessary.
- The overall struct is aligned according to the largest field alignment requirement, with external fragmentation added at the end as necessary.

```
struct Student {
  int id;
  char* name;
  char age;
};
```

a) Fill in which bytes are used by which variables and label the rest as internal or external fragmentation. The first variable "id" is given.



b) What is the size of `struct Student`? **24 bytes**

c) Give a reordering of the fields in `struct Student` such that there is no internal fragmentation

```
struct Student {

      char* name;
      int id;
      char age;

};
```

d) How much external fragmentation does this new `struct Student` have? **3 bytes**

e) What is the size of this new `struct Student`? **16 bytes** (smaller than before)

Caches: Locality!

Recall that we have two types of locality that we can have in code:

Temporal locality: when recently referenced items are likely to be referenced again in the near future.
Spatial locality: when nearby addresses tend to be referenced close together in time.

For each type of locality, can you give an example of when we might see it in code?

Temporal Locality:

Spatial Locality:

Accessing a sum counter over and over; reading and writing to the same variable; etc.

Accessing a[0] in an array, then a[1], then a[2] in order; accessing the first field in a struct, then the second, then the third; etc.

Accessing a Cache (Hit or Miss?)

Assume the following caches all have block size $K = 4$ and are in the current state shown (you can ignore "—"). All values are shown in hex. Tag fields are padded, while bytes of the cache blocks are shown in full. The word size for the machine with these caches is 12 bits (i.e. addresses are 12 bits long)

Direct-Mapped:

Set	Valid	Tag (8 bits)	B0	B1	B2	B3
0	1	15	63	B4	C1	A4
1	0	—	—	—	—	—
2	0	—	—	—	—	—
3	1	0D	DE	AF	BA	DE
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	13	31	14	15	93
7	0	—	—	—	—	—
8	0	—	—	—	—	—
9	1	00	01	12	23	34
A	1	01	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	—	—	—	—	—
D	1	11	C0	04	39	AA
E	0	—	—	—	—	—
F	1	0F	FF	6F	30	0

Offset bits: 2

Index bits: 4

Tag bits: 6

	Hit or Miss?	Data returned
a) Read 1 byte at `0x7AC`	Miss	—
b) Read 1 byte at `0x024`	Hit	0x01
c) Read 1 byte at `0x99F`	Miss	—

2-way Set Associative:

Set	Valid	Tag (8 bits)	B0	B1	B2	B3	Set	Valid	Tag (8 bits)	B0	B1	B2	B3
0	0	—	—	—	—	—	0	0	—	—	—	—	—
1	0	—	—	—	—	—	1	1	2F	01	20	40	03
2	1	03	4F	D4	A1	3B	2	1	0E	99	09	87	56
3	0	—	—	—	—	—	3	0	—	—	—	—	—
4	0	06	CA	FE	F0	0D	4	0	—	—	—	—	—
5	1	21	DE	AD	BE	EF	5	0	—	—	—	—	—
6	0	—	—	—	—	—	6	1	37	22	B6	DB	AA
7	1	11	00	12	51	55	7	0	—	—	—	—	—

Offset bits: 2

Index bits: 3

Tag bits: 7

	Hit or Miss?	Data returned
a) Read 1 byte at `0x435`	Hit	0xAD
b) Read 1 byte at `0x388`	Miss	—
c) Read 1 byte at `0x0D3`	Miss	—

Fully Associative:

Set	Valid	Tag (12 bits)	B0	B1	B2	B3
0	1	1F4	00	01	02	03
0	0	—	—	—	—	—
0	1	100	F4	4D	EE	11
0	1	077	12	23	34	45
0	0	—	—	—	—	—
0	1	101	DA	14	EE	22
0	0	—	—	—	—	—
0	1	016	90	32	AC	24

Set	Valid	Tag (12 bits)	B0	B1	B2	B3
0	0	—	—	—	—	—
0	1	0AB	02	30	44	67
0	1	034	FD	EC	BA	23
0	0	—	—	—	—	—
0	1	1C6	00	11	22	33
0	1	045	67	78	89	9A
0	1	001	70	00	44	A6
0	0	—	—	—	—	—

Offset bits: 2

Index bits: 0

Tag bits: 10

	Hit or Miss?	Data returned
a) Read 1 byte at `0x1DD`	Hit	0x23
b) Read 1 byte at `0x719`	Hit	0x11
c) Read 1 byte at `0x2AA`	Miss	—

Cache Sim

If you need help on using the cache sim, take a look at additional supplemental material that will guide you through using the cache sim (posted with today's section handouts)! The cache sim is very useful for lab 4 and corresponding homework assignments.