5. **(12 pts)** The three functions below perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Also, write an explanation for your ranking.

```
#define n 100
typedef struct{
    int x[3];
    int y[3];
}pt;

pt p[n];
```

```
void f1(pt *p, int n){
    int i, j;
    for (j=0; j<3; j++){
        for (i=0; i<n; i++)
            p[i].x[j] = 0;
        for (i=0; i<n; i++)
            p[i].y[j] = 0;
    }
}
```

```
void f2(pt *p, int n){
    int i, j;
    for (i=0; i<n; i++){
        for (j=0; j<3; j++)
            p[i].x[j] = 0;
        for (j=0; j<3; j++)
            p[i].y[j] = 0;
    }
}
```

```
void f3(pt *p, int n){
    int i, j;
    for (i=0; i<n; i++)
        for (j=0; j<3; j++){
            p[i].x[j] = 0;
            p[i].y[j] = 0;
        }
}
```

a) best spatial locality f2

b) mid spatial locality f3

c) worst spatial locality f1

pt+0 x[3] pt+12 y[3] pt+24

mult. of 4

f2 (B) f3 (B) f1

f2
p[0].x[0]
p[0].x[1]
p[0].x[2]
p[0].y[0]
p[0].y[1]
p[0].y[2]
p[1].x[0]
p[1].x[1]
p[1].x[2]

f3
p[0].x[0]
p[0].y[0]
p[0].x[1]
p[0].y[1]
p[0].x[2]

f1
p[0].x[0]
p[1].x[0]
p[2].x[0]
p[99].x[0]
p[0].y[0]
p[1].y[0]
p[2].y[0]
p[99].y[0]

p[0]
p[1]

- The block size is 4 bytes (B = 4)
- The cache has 4 sets (S = 4)
- The cache is direct mapped (E = 1)

a) What is the total capacity of the cache? (in number of data bytes) 16

b) How long is a tag? (in number of bits) 4 bits

c) Assuming that the cache starts clean (all lines invalid), please fill in the following tables, describing what happens with each operation. Addresses are given in both hex and binary for your convenience.

Operation	Set index?	Hit or Miss?	Eviction?
load 0x00 $(0000\ 0000)_2$	0	Miss	no
load 0x04 $(0000\ 0100)_2$	1	Miss	no
load 0x08 $(0000\ 1000)_2$	2	Miss	no
store 0x12 $(0001\ 0010)_2$	0	Miss	yes
load 0x16 $(0001\ 0110)_2$	1	Miss	yes
store 0x06 $(0000\ 0110)_2$	1	Miss	yes
load 0x18 $(0001\ 1000)_2$	2	Miss	yes
load 0x20 $(0010\ 0000)_2$	0	Miss	yes
store 0x1A $(0001\ 1010)_2$	2	Hit	no

SI	V	tag	Blocks
00	1	0010	20, 21, 22, 23
01	1	0000	04, 05, 06, 07
10	1	0001	18, 19, 1A, 1B
11	0		

v. Give the contents of header and footer for each block (in binary). Don't forget to indicate the change in header and footer when there is a need.

a) p1 = malloc(3)
b) p2 = malloc(5)
c) free(p1)
d) p3 = malloc(9)
e) free(p2)
f) p4 = malloc(4)

Show all your work and briefly explain your answer!

3 bytes + H + F = 3 + 2 + 2 = 7 ← 4-byte alignment

Payload + H + F = 8 bytes

5 + 2 + 2 = 9 → 12 bytes

9 + 2 + 2 = 1

0000000000001001 = 8

Your heap will be like the following, and you will draw the heap 5 times (i.e., for each request). You can insert more rows as you need.

Address	Content	Header/Footer?	is pointed ?
0xb000	0000000000001000	Header	
0xb002	Free		
0xb004	Free		
0xb006	0000000000001000	Footer	
0xb008	0000000000001101	Header	
0xb00a	Allocated		p2 points at this m.l.
0xb00c	Allocated		
0xb00e	Allocated		

Here is an example that shows what you should include in your answer. Note that this example is for a totally different allocation.

Address	Content	Header/Footer ?	is pointed ?
0x400AFFC	0x00000013	Header	
0x400B000	Allocated		p1 points at this memory location
0x400b004	Allocated		
0x400b008	0x00000013	Footer	

Array Example

Practice Problem 3.38 (solution page 377)

Consider the following source code, where M and N are constants declared with `#define`:

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

```
long sum_element(long i, long j)
i in %rdi, j in %rsi
sum_element:
  leaq    0(,%rdi,8), %rdx
  subq    %rdi, %rdx
  addq    %rsi, %rdx
  leaq    (%rsi,%rsi,4), %rax
  addq    %rax, %rdi
  movq    Q(,%rdi,8), %rax
  addq    P(,%rdx,8), %rax
  ret
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

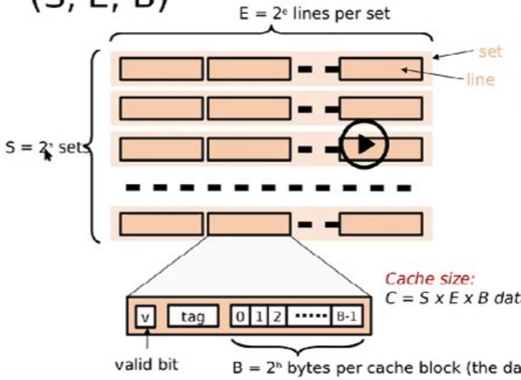
8 is scale value
Offset of matrix P 7i + j →
P has 7 columns

Offset of matrix Q 5j + i →
Q has 5 columns

So, M = 5 and N = 7

Compute 8i → rdx
Compute 7i → rdx
Compute 7i + j → rdx
Compute 5j → rax
Compute i + 5j → rdi
Retrieve $M[x_Q + 8*(5j + i)]$ → rax
Add $M[x_p + 8*(7i+j)]$ → rax

General Cache Organization (S, E, B)



- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at block offset

Address of word:

t bits	s bits	b bits
tag	set index	block offset

Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

What about writes?

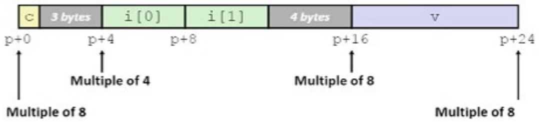
- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

2

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- **Example:**
  - K = 8, due to double element

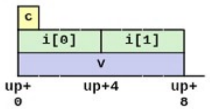
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Union Allocation

- **Allocate according to largest element**
- **Can only use one field at a time**

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

FP Basics

- **Arguments passed in %xmm0, %xmm1, ...**
- **Result returned in %xmm0**
- **All XMM registers caller-saved**

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
  addss    %xmm1, %xmm0
  ret
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

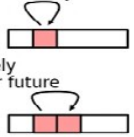
```
# x in %xmm0, y in %xmm1
  addsd    %xmm1, %xmm0
  ret
```

return	%rax	%eax	%al	%r8	%r8d	%r8b
	%rbx	%ebx	%bl	%r9	%r9d	%r9b
4	%rcx	%ecx	%cl	%r10	%r10d	%r10b
3	%rdx	%edx	%dl	%r11	%r11d	%r11b
2	%rsi	%esi	%sil	%r12	%r12d	%r12b
1	%rdi	%edi	%dil	%r13	%r13d	%r13b
	%rsp	%esp	%spl	%r14	%r14d	%r14b
	%rbp	%ebp	%bpl	%r15	%r15d	%r15b

- Can reference low-order byte

Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

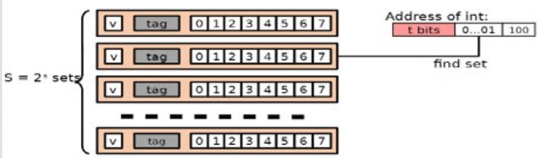


General Caching Concepts: Types of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Conflict miss
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Example: Direct Mapped Cache (E = 1)

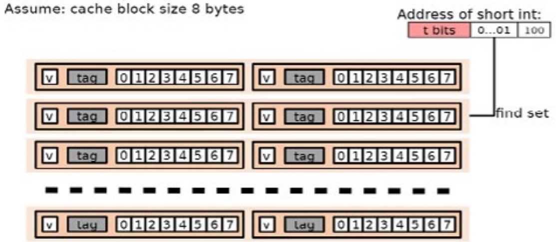
Direct mapped: One line per set
Assume: cache block size 8 bytes



If tag doesn't match: old line is evicted and replaced

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



Keeping Track of Free Blocks

- **Method 1: *Implicit list* using length—links all blocks**

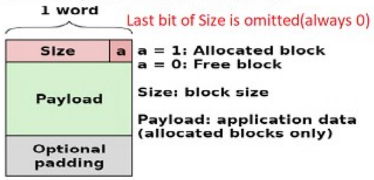
- **Method 2: *Explicit list* among the free blocks using pointers**

- **Method 3: *Segregated free list***
  - Different free lists for different size classes
- **Method 4: *Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Method 1: Implicit List

- **For each block we need both size and allocation status**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks



Implicit List: Finding a Free Block

- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits:
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- ***Next fit:***
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- ***Best fit:***
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit

Nested Array Element Access

- **Array Elements**
  - **A[i][j]** is element of type *T*, which requires *K* bytes
  - Address **A + i * (C * K) + j * K**

    = A + (i * C + j) * K

`int A[R][C];`