A CPU cache is another storage media which is much faster than RAM. All memory accesses first look in the CPU cache for the data. If the data is not present in the cache, the data is first brought in the cache before the read/write.

Instead of bringing 1,2,4,8 bytes of data (that a typical x86 instruction will access), cache-lines (of size 64 bytes) that contain the data are brought into the cache.

Number of cache lines in a 4 KB page would be: 4096/64.

A write-back cache does not synchronize the data with memory immediately. It does so when a cache-line is replaced (due to the limited size of cache). The primary difference between write-back buffer cache (in the filesystem) and a write-back CPU cache is: a buffer cache synchronizes the data with the disk on commit. A commit does not necessarily mean replacement. Further, a commit can be delayed until the application prints something on the console or send a network packet outside the system (to preserve the consistency wrt user view). A file system must commit all the pending changes before the data can be externalized. In case of memory cache, because the data is anyway going to be written on a volatile media, we can delay the synchronization until eviction (replacement) of a cache-line.

A write-through cache, on the other hand, writes to the main memory on every store. In a write-through cache, the stores execute at the speed of RAM which is not good for performance. One advantage of write-through cache is that it simplifies cache coherence (discussed next) logic. Another case where we need write-through cache is a physical address corresponding to MMIO devices because there is no cache between the device and the physical memory.

Because, the multiple CPUs have their private caches, a cache coherence protocol is needed to ensure that all the CPUs see a consistent view of memory. x86 hardware implements the MESI protocol for cache coherence. MESI protocol states that a cache line can be in one of the following four states:

M: modified: only present in the current cache; not matches main memory

E: exclusive: only present in the current cache; matches main memory

S: shared: can be present in multiple caches; matches main memory

I: Invalid: the cache line is invalid

A cache line can be present in multiple caches only in shared mode. Before writing a cache line the CPU sends a request to other CPUs to invalidate the respective cache line.

Because, updating a cache line may require invalidation of the cache line on other CPUs, x86 has a store buffer that allows the processor to store the updates and continue execution without actually updating the cache line. Due to store buffer, the x86 processor doesn't have to stall on every update, that makes it very efficient. However, a drawback of store buffer is that the processor cannot see the updates made by other CPUs immediately. If a processor tries to read

a cache-line that is present in its store buffer, then the store buffer value is forwarded to the processor.

On x86, loads and stores are atomic.

P1: x = 1

P2: x = 2

assert (x == 1 || x == 2);

For example, if two processors try to update the same memory location, the final value would be the last store request received by the system memory.

Due to the store buffer, a single CPU can see the loads and stores in a different order. Let us see some of the examples:

1. Loads and stores are not reordered with like operations

P1	P2
x = 1	r1 = y
y = 1	r2 = x

Initially, x and y are zero.

r1 == 1 and r2 == 0 is not possible on x86.

2. Stores are not reordered with earlier store

P1	P2
r1 = x	r2 = y
y = 1	x = 1

Initially, x and y are zero.

r1 == 1 and r2 == 1 is not possible on x86.

3. Loads may be reordered with earlier stores to different memory location

P1	P2
x = 1	y = 1
r1 = y	r2 = x

Initially, x and y are zero.

r1 == 0 and r2 == 0 is possible.

Because the loads and store on a single CPU may get reordered this memory model is a weak memory model. On an architecture, with a weak memory model, it is the responsibility of the developers to consider reordering in their design. x86 also provides “mfence” instruction, that pause the execution until all the loads and stores prior to “mfence” are completed (prevents reordering).

Read Chapter – 8 from Intel software developers manual vol-3.

Read Peterson’s solution from chapter 6 of Silberschatz and Galvin.

Let us look at the Peterson’s solution for the implementing acquire and release for two threads. Here, i is the current thread and j is the remote thread.

```
volatile int turn;
volatile boolean flag[2];
acquire () {
   flag[i] = true;
   turn  = j;
   while (flag[j] && turn == j);
}
release () {
   flag[i] = FALSE;
}
```

This algorithm works on the CPUs that don’t do reordering. However, this won’t work on hardware that implements reordering (e.g., x86).

Load and stores in acquire:
store flag[i]
load j
store turn
load flag[j]
…

On x86, a load may be reordered with a prior store on a different memory location. This brings us to the following order:

load j
store flag[i]
store turn
load flag[j]
…

The flag[j] can be reordered with earlier stores to a different memory location. This brings us to:

load j
load flag[j]
store flag[i]
store turn
…

Notice, that due to reordering both the CPUs may see the value of flag[j] as FALSE and proceed after the while loop.

```
volatile int turn;
volatile boolean flag[2];
acquire ()
{
   flag[i] = true;
   turn  = j;
   asm volatile (“mfence”);
   while (flag[j] && turn == j);
}
release ()
{
   flag[i] = FALSE;
}
```

The above code will not reorder the loading of flag[j] and would work on x86.

x86 instructions are complex. A single instruction can do multiple reads and write. Let us see the following example:

add $1, 0x1000
This instruction adds one to memory location 0x1000. The hardware may break the instruction into the following microcode.

```
t1 = *0x1000
t1 += 1
*0x1000 = t1
P1:
for (i = 0; i < 1000; i ++)
{
  add $1, 0x1000;
}
P2:
for (i = 0; i < 1000; i ++)
{
  add $1, 0x1000;
}
```

Consider the above code concurrently executes “add $1, 0x1000” on processor 1 and 2. It is possible after the execution of this code the final value of 0x1000 is not 2000, because at some point both the CPUs read the same value before adding and updating 0x1000.

To make sure that this work correctly, we can rewire the code as:

```
P1:
for (i = 0; i < 1000; i ++)
{
   acquire ();
   add $1, 0x1000;
   release ();
}
P2:
for (i = 0; i < 1000; i ++)
{
   acquire ();
   add $1, 0x1000;
   release ();
}
```

x86 provides a lock prefix that can be used with an instruction to get the atomicity of the given instruction. The locked instruction drains the store buffer and locks the cache lines during the execution of the instruction.

```
P1:
for (i = 0; i < 1000; i ++)
{
   lock add $1, 0x1000;
}
P2:
for (i = 0; i < 1000; i ++)
{
   lock add $1, 0x1000;
}
```

The final value of 0x1000 will be 2000 if we use locked instructions.