COS3043
System Fundamentals

Lecture 4

List of Discussion

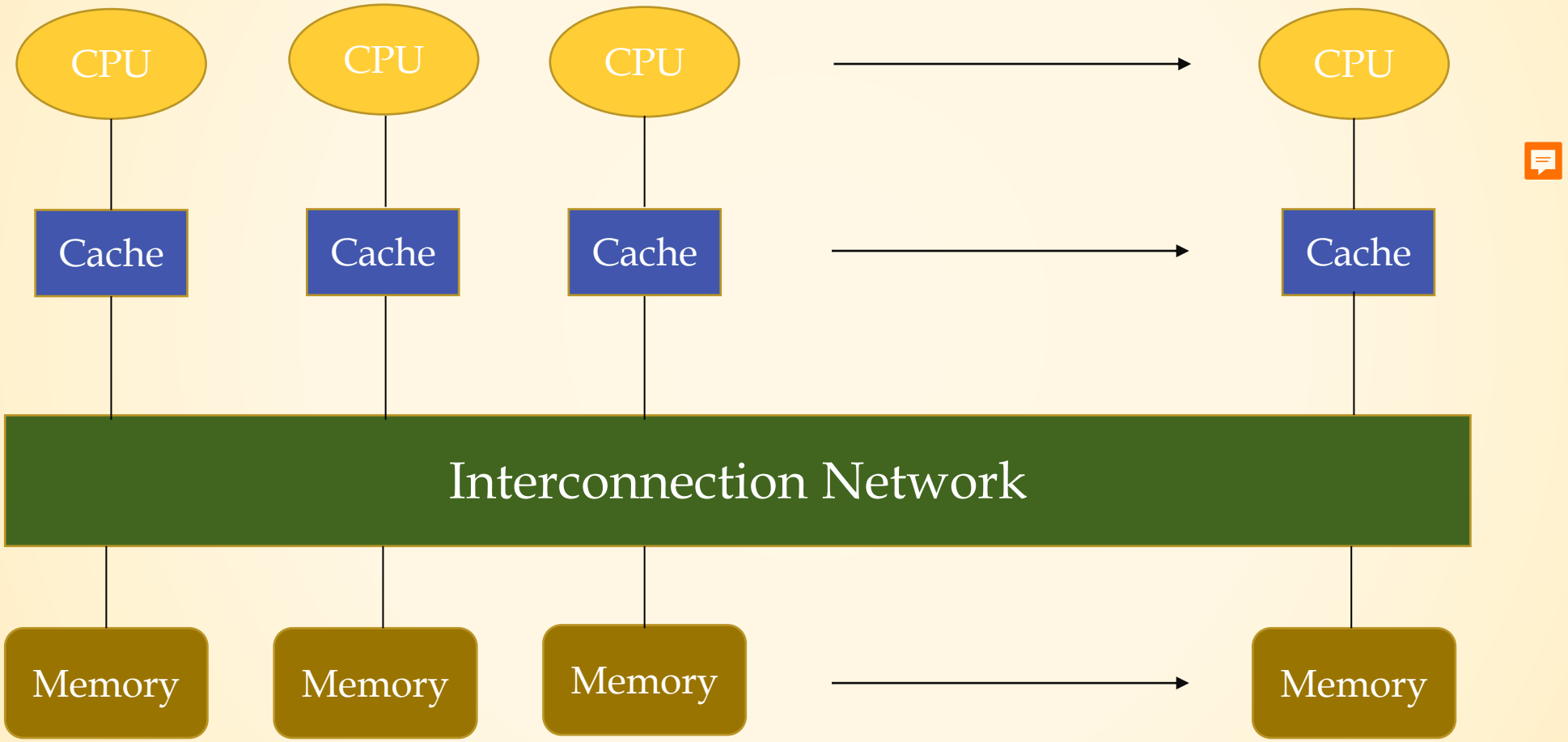
- Lecture 4:
  - Shared Memory Machine
  - Synchronization in Parallel System
- Lecture 5 (will be covered next lecture):
  - Communication in Parallel System
  - Scheduling in Parallel System

Shared Memory Machine

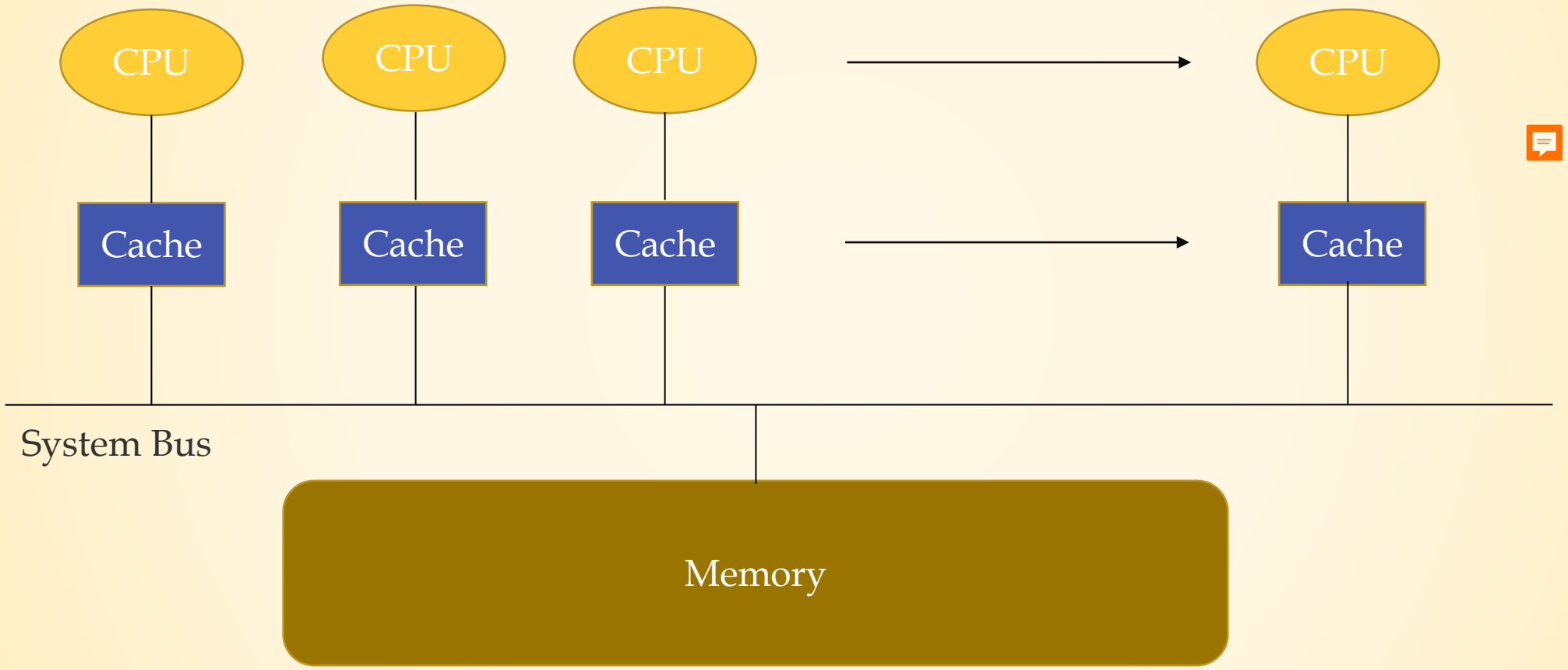
Shared Memory Multiprocessors

- There are three different structures of shared memory machines:
  - Dance Hall Architecture
  - Symmetric Multiprocessors (SMP)
  - Distributed Shared Memory (DSM)
- Common things in different structures:
  - There will be CPU, cache (associated with the CPU) , memory and interconnection network.
  - Entire address space that is defined by the memory, it can be accessed from any of the CPUs.

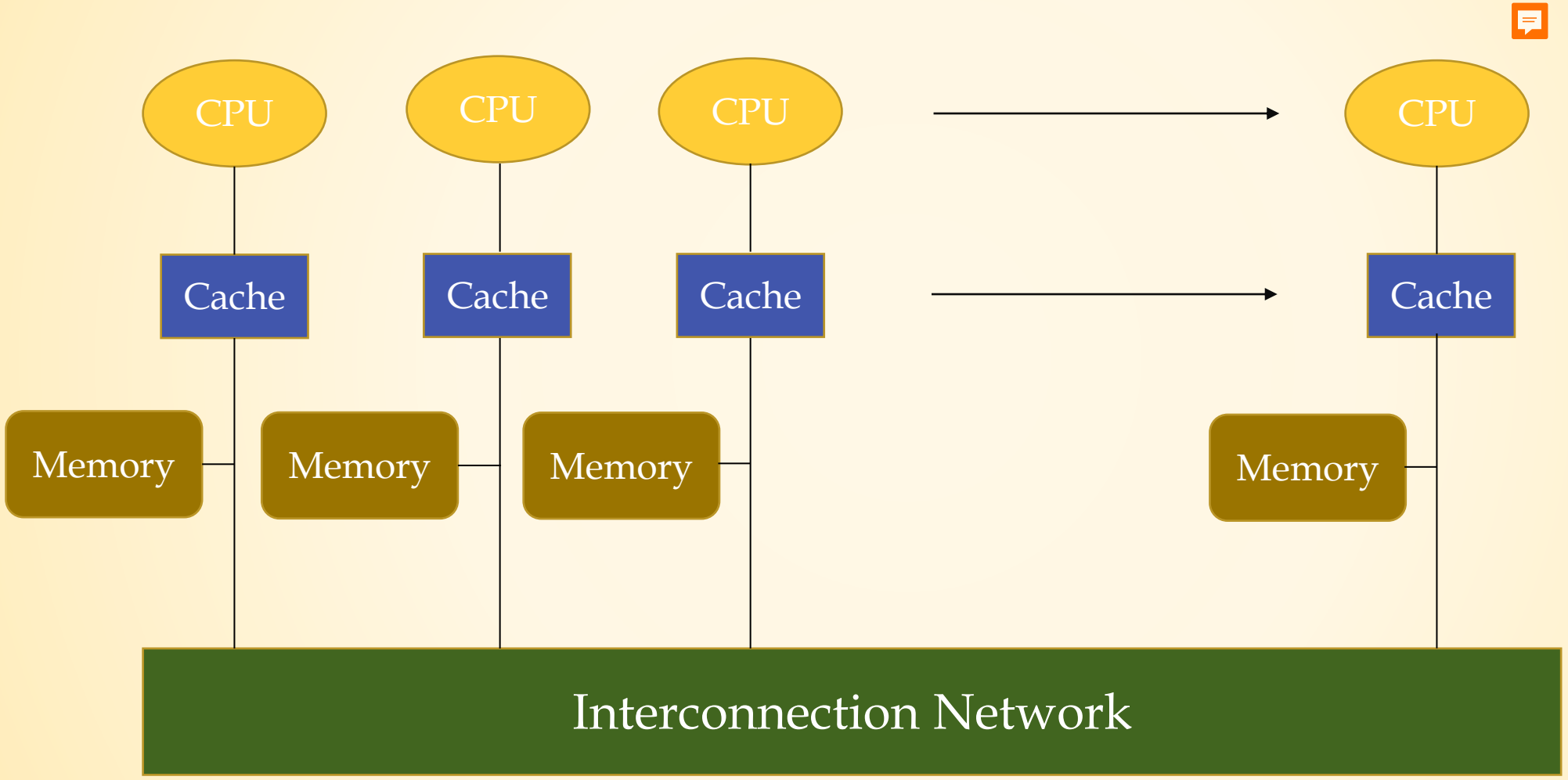
Dance Hall Architecture



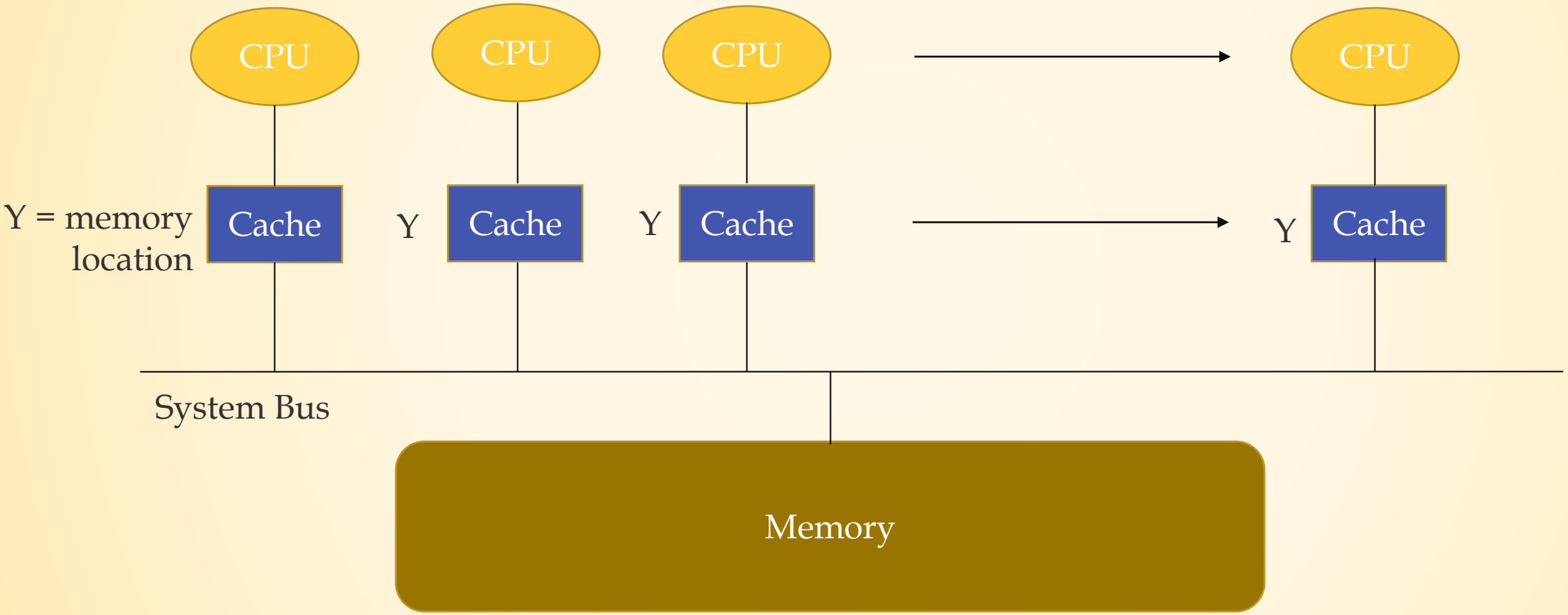
Symmetric Multiprocessors (SMP)



Distributed Shared Memory (DSM)



Shared Memory and Caches



Question?

Assume a = b = 0 initially

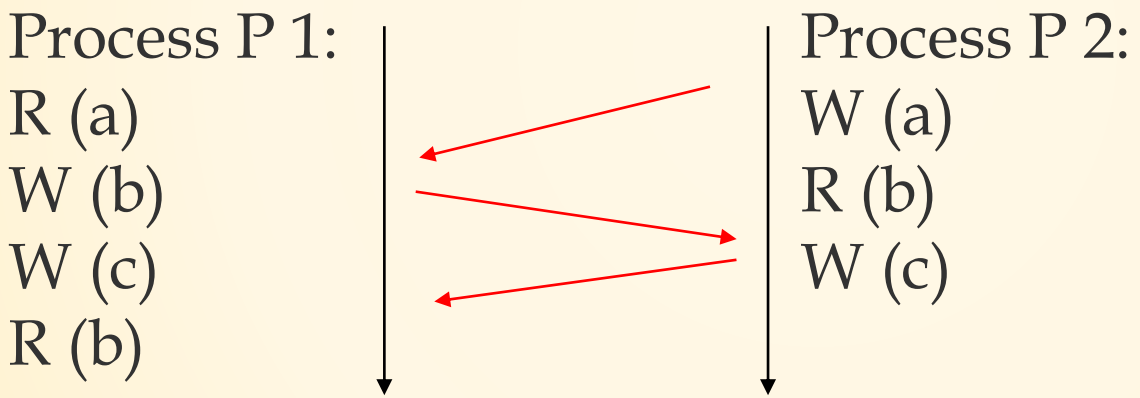
Process P 1:	Process P 2:
a = a + 1	d = b
b = b + 1	c = a

What is the possible value for c & d:

- c = d = 0
- c = d = 1
- c = 1; d = 0
- c = 0; d = 1

Memory Consistency Model

- There are several consistency models available.
- But based on the example previously, we will discuss Sequential Consistency (SC) model



Program order + arbitrary interleaving

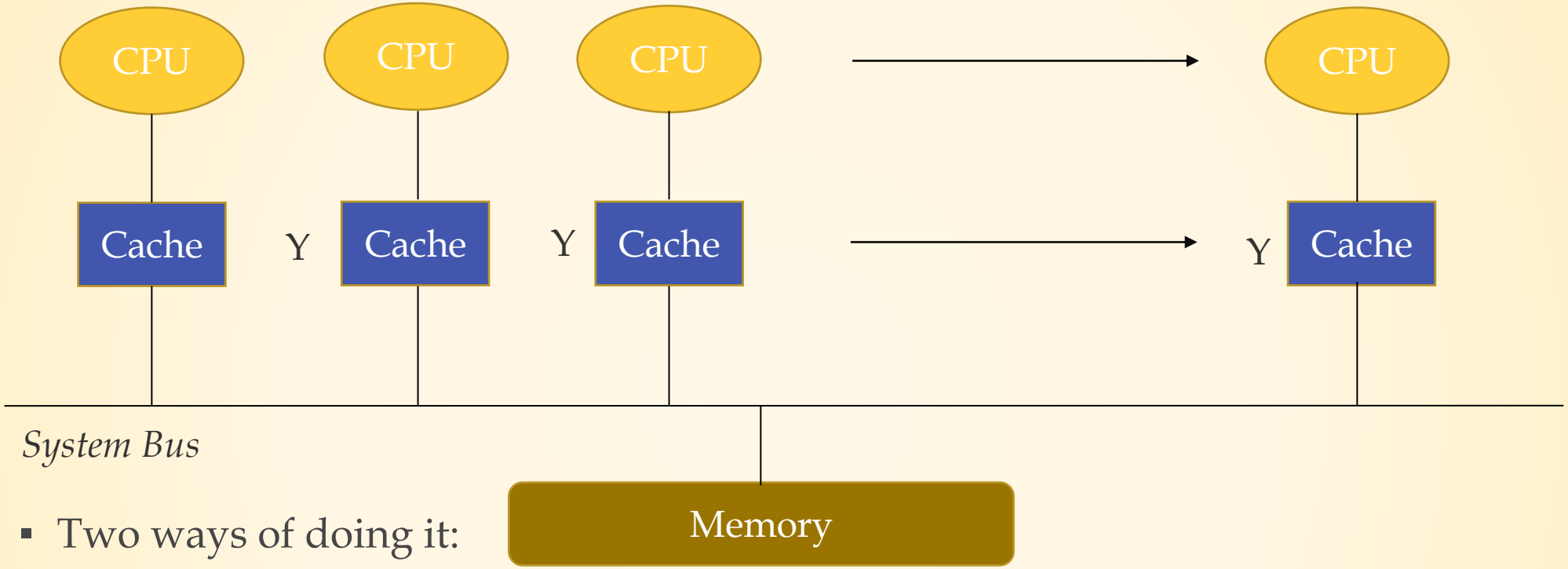
Memory Consistency and Cache Coherence

- There must be strong relationship/partnership between the memory consistency (software) and cache coherence (hardware).

- Memory consistency: what's the model presented to the programmer?
  - Is a contract between the app programmer and the system
  - Answering the **"when"** question.
  - How soon the change going to be made visible to other processes that have the same memory location in their respective cache.

- Cache coherence: how is the system implementing the model in the presence of private cache?
  - Answering the **"how"** question
  - How the system (software + hardware working together) implementing the contract of the memory consistency model.
    - CC: hardware will make sure the cache is coherent => Hardware Cache Coherence.
    - NCC: No hardware cache coherent, let system software to ensure cache is coherent.

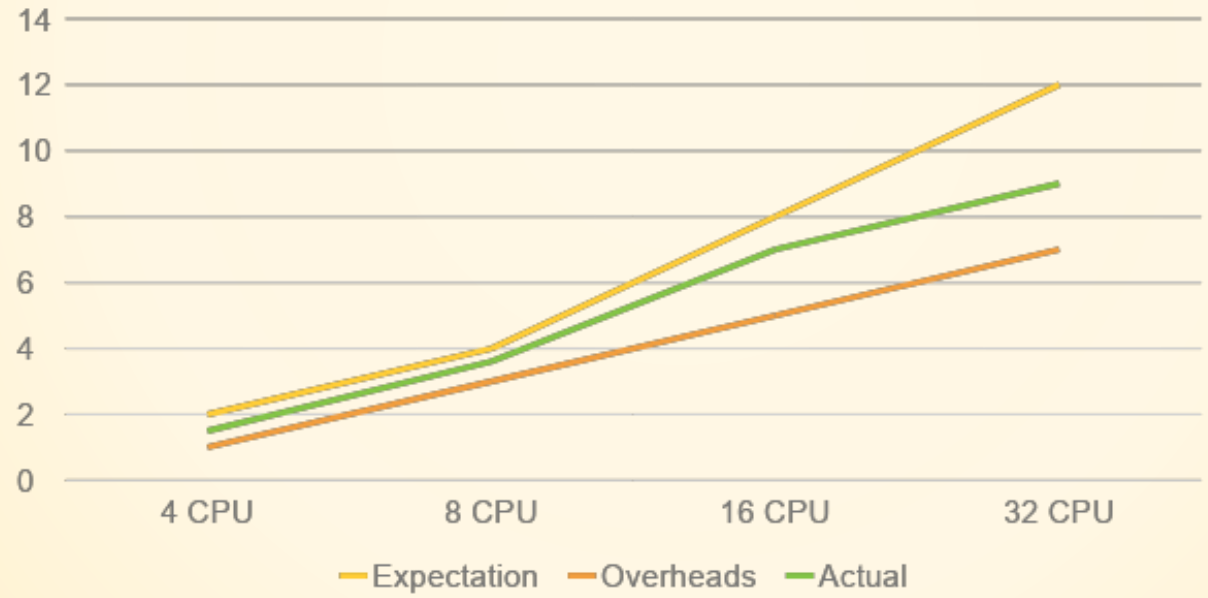
Hardware Cache Coherence



- Two ways of doing it:
  - Write invalidate
  - Write update
- However, such operation will increase overhead if there are more CPU and cache in the system

Scalability

- Remember scalability => when resources ↑ , the performance ↑
- However in the case of shared memory, when there are more CPU, it will be more overheads because of the cache coherence.



Synchronization

Overview - Synchronization

- Synchronization Primitives.
- Algorithms Related to Parallel OS Supporting Multithread Operations.
- Various Locks and Operations/Algorithms.

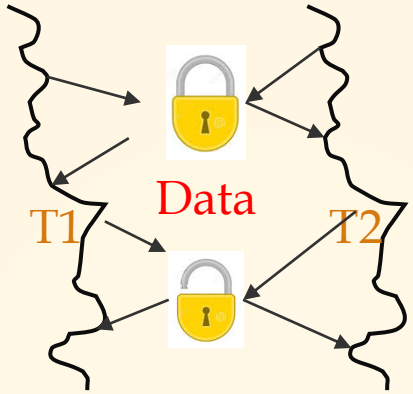
Synchronization Primitives

- It’s imperative for shared memory programming.
- Two different primitives: “Lock” & “Barrier”.

Lock

- What is a “Lock”? – It’s to safeguard something important or precious from being taken or used by strangers/outsiders.
- Likewise in parallel programming, many threads are sharing same data structure, it’s crucial to ensure each thread doesn’t mess up with other thread’s works.
- With a lock, a thread can be sure that when it’s using some piece of shared data, no other thread can come to interfere.

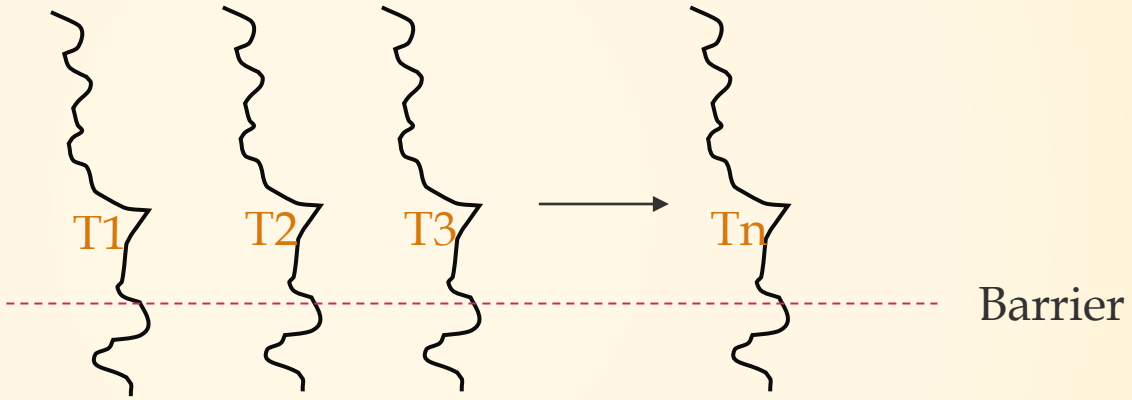
Lock Operation



- Once T1 has got the Data, it will lock the Data so that no other threads can access to the Data from there on.
- Once T1 is done with its intended operations, it will release the lock.
- 2 different types of lock:
  - Mutual Exclusive Lock – one at a time. Why?
  - Shared Lock – simultaneous used, shared by multithreads. How?

Barrier

- The other popular primitive in multithread programming is “barrier”.



- The main idea is that all threads must reach the barrier first before they proceed to next step although some of the threads may reach the barrier point before others.

Algorithm – Applying Lock for Mutual Exclusion

P1

- modify structure A

P2

- wait for modification to complete
- use structure A

Solution

flag = 0; //init a shared variable

P1

- modify structure A
- flag = 1; //signals P2

P2

- while (flag == 0) { wait };
- use structure A
- flag = 0; //re-init

Atomic Operations

Lock

```
if(L == 0)
    L = 1;
else
    while (L == 1)
        wait;
    go back
```

Group of 3 instructions:

Unlock

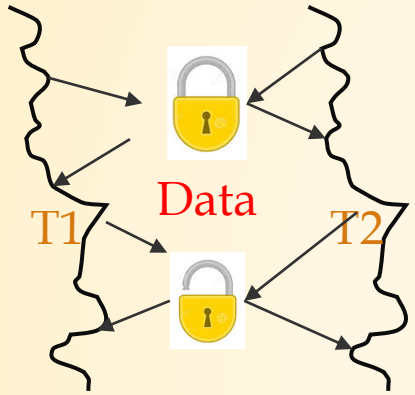
```
L = 0;
```

- *This set of instructions has to be executed 'atomically' so that no other comes to interfere.*
- *Read and write are atomic operation themselves, but when they are combined, they are not atomic.*
- *Thus new semantic atomic instruction need to be introduced:* ***RMW (read-modify-write) instruction***

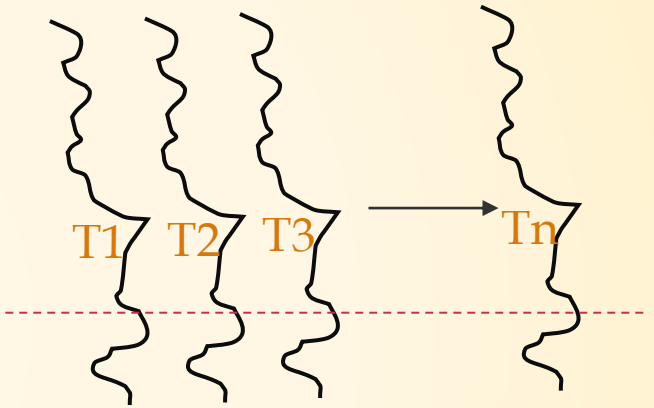
Variations of RMW Instruction

- Test-and-set (mem-loc)
  - Return current value in mem-loc; set mem-loc to 1
- Fetch–and-increment (mem-loc)
  - Return current value in mem-loc; increment of mem-loc

Scalability Issue with Synchronization



Lock Algorithm

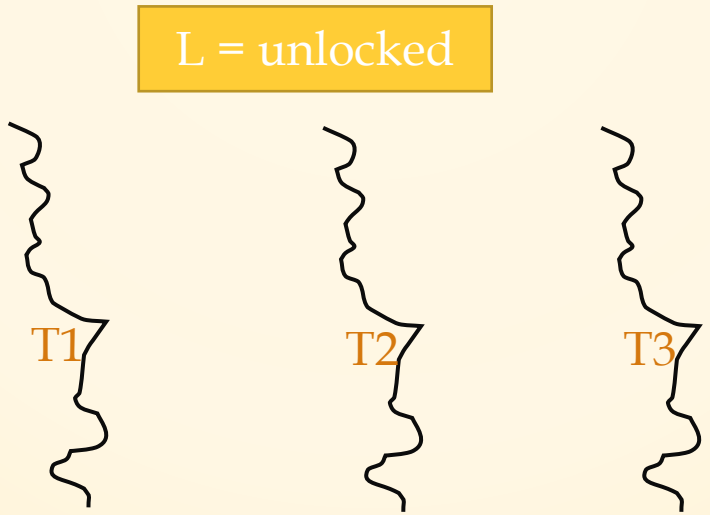


Barrier Algorithm

1. Latency – How long it takes for a thread to acquire the lock when the lock is free?
2. Waiting Time – How long it takes to wait for a lock?
3. Contention – Competition among other threads which are waiting.

Spinlock (Spin on T&S)

- Lock
  - while (T&S(L) == locked) { wait };
  - NOTE: wait = ***spin***ning waiting for ***lock***
- Unlock
  - L = unlocked;



Question?

What is/are the problem(s) could be faced by spinlock?

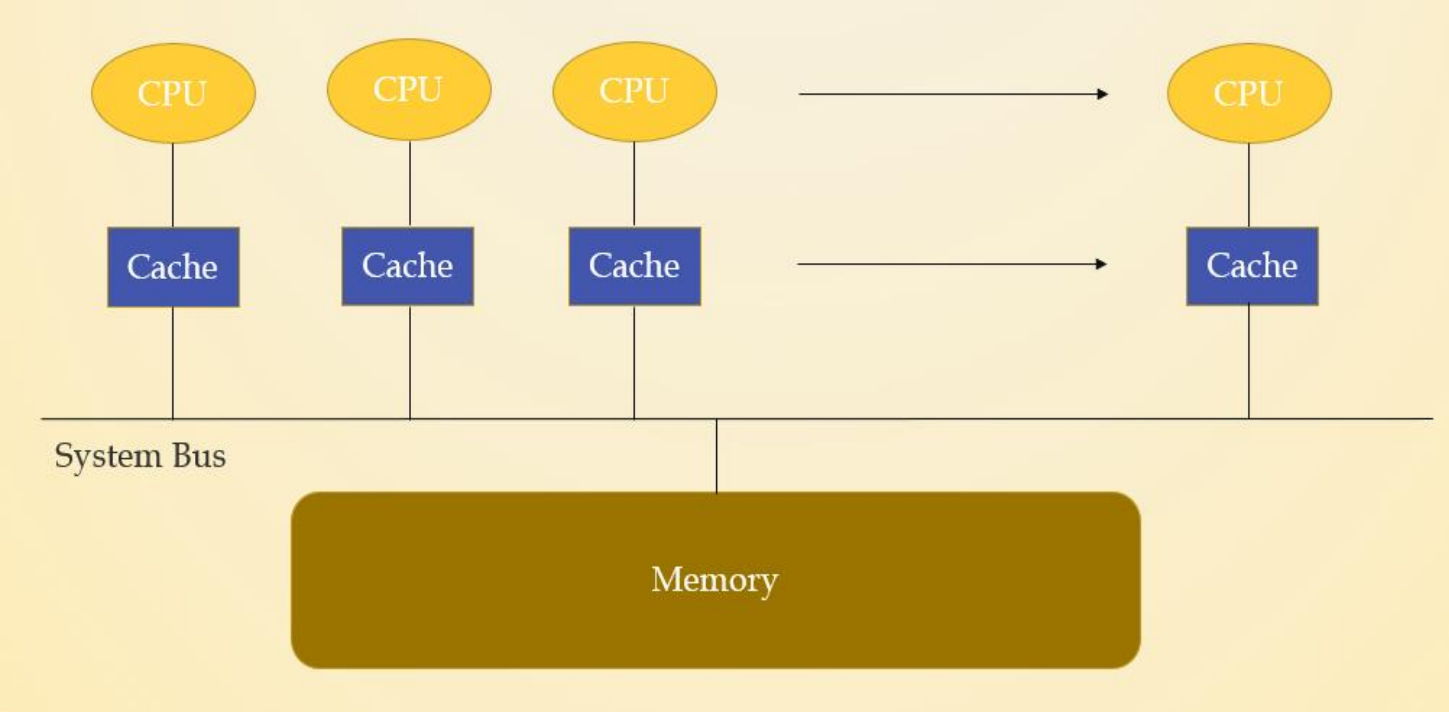
- Too much contentions?
- Does not exploit caches?
- Disrupts useful work?

Caching Spinlock (Spin on Read)

- Lock(L)

```
while (L == locked) { wait };
if (T&S(L) == locked) go back;
```

Waiters depend on cache coherent hardware, and will spin on cached copy of L



Spinlock with Delay

- Delay After Lock Release

```
while (T&S(L) == locked ) {
        delay (d[Pi]); //Pi=Processor ID
}
```

  - For Static delay, a CPU may still delayed although the lock is ready

- Delay With Exponential Backoff

```
while (T&S(L) == locked) {
        delay(d);
        d = d * 2;
}
```

  - With Dynamic delay, this method can still work in non cache coherent hardware.

Ticket Lock

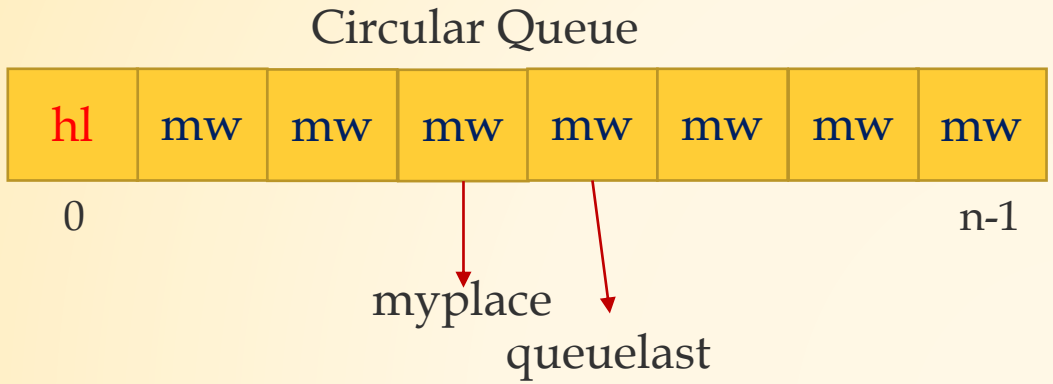
- So far, we have addressed issue of latency, waiting time and contention. But we have not talked about 'fairness'.
- Retail's ticketing system will be used to ensure 'fairness' – first come first served.

```
struct L {
        int next_ticket;
        int now_serving;
}
```

```
Acquire-Lock(L):
        int my_ticket = fetch-and-inc (L~>next_ticket);
        loop
                pause (my_ticket – L~>now_serving);   //pause to delay for waiting
                if(L~>now_serving == my_ticket ) return;   //got the lock now

Release-Lock(L):
        L~>now_serving++;
```

Array Based Queueing Lock



For each Lock L:

- *flags[n] //n=number of CPU*
  - *2 states:*
    - *has-lock (hl)*
    - *must-wait (mw)*
- *queuelast: 0 //init*

```
Lock(L):
        myplace = fetch-and-inc (queuelast);
        while (flags[myplace] == mw) { wait };    // wait until mw changes to hl

Unlock(L):
        flags[current] == mw;
        flags[current+1] == hl;
```