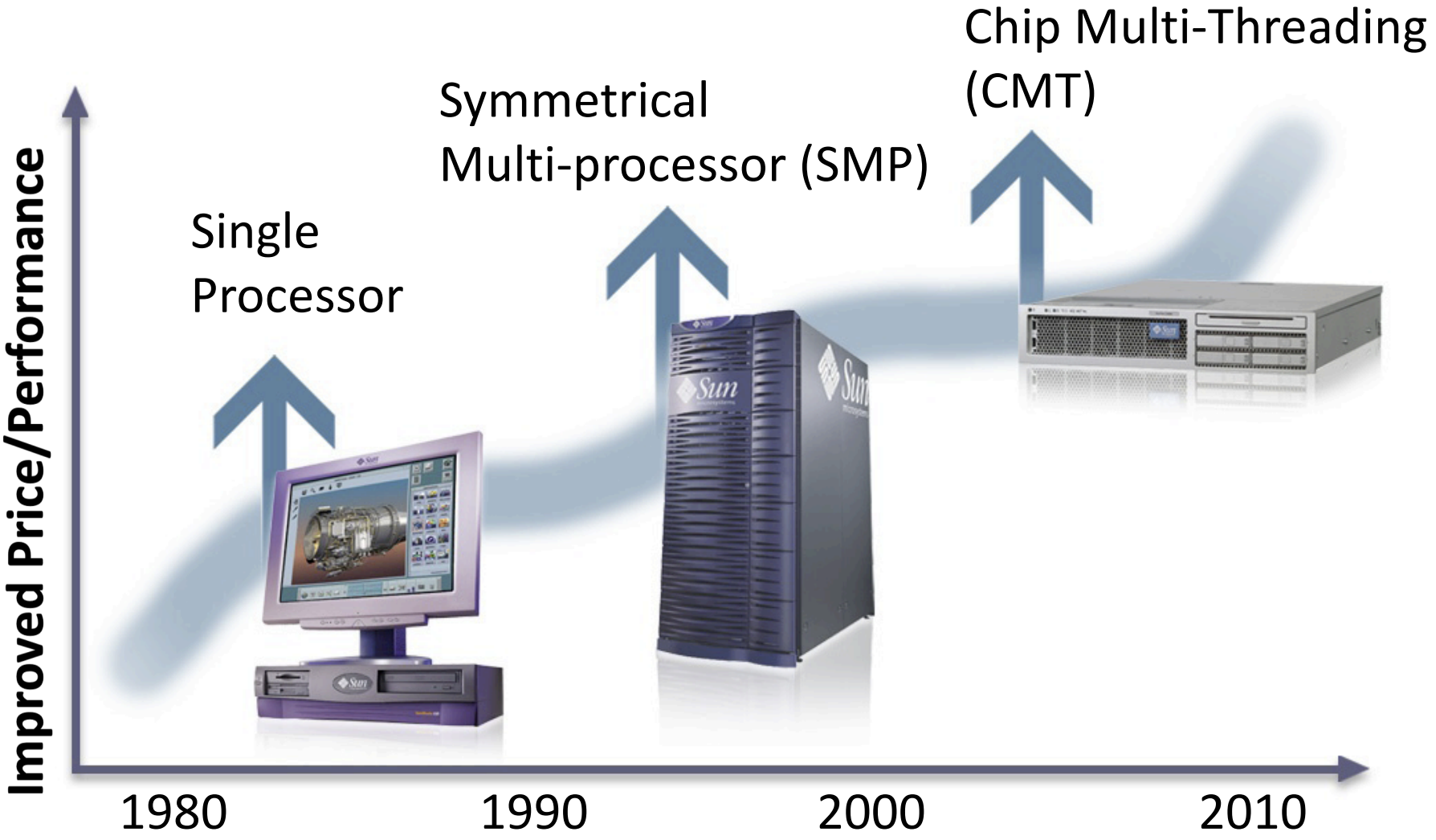
Multi-core Computing

Multi-core age (1)



Multi-core age (2)

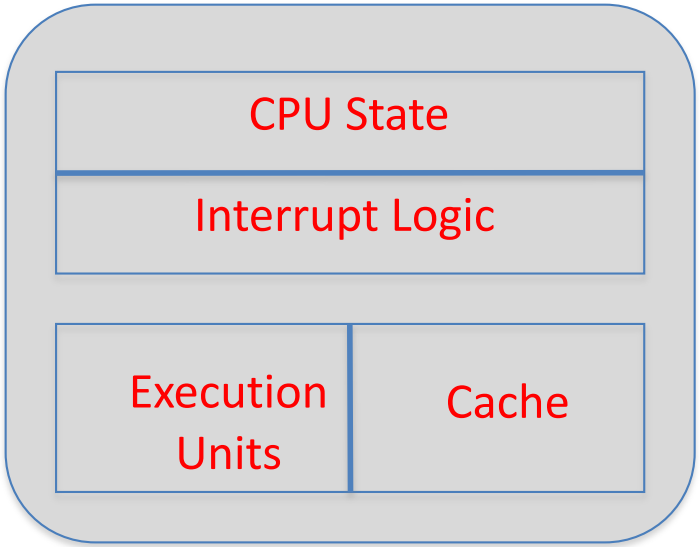
- Physical limits for chip manufacturing
  - Light speed, heat dissipation, power consumption
- Moore's law is still true!
  - Moore's Law: The number of transistors that can be placed in a chip doubles approximately every two years
  - A silicon die can accommodate more and more transistors
- New direction
  - To build multiple (hundreds or thousands) less powerful cores (CPUs) into a single chip

Memory wall

- Memory wall is the growing disparity of speed between CPU and RAM
  - From 1986 to 2000, CPU speed improved at 55% annually while memory speed only improved at 10%.
  - Now there are only a few cycles for the execution of one instruction, but hundreds of cycles for memory access outside chip.
  - 90% CPU time is waiting for data from RAM

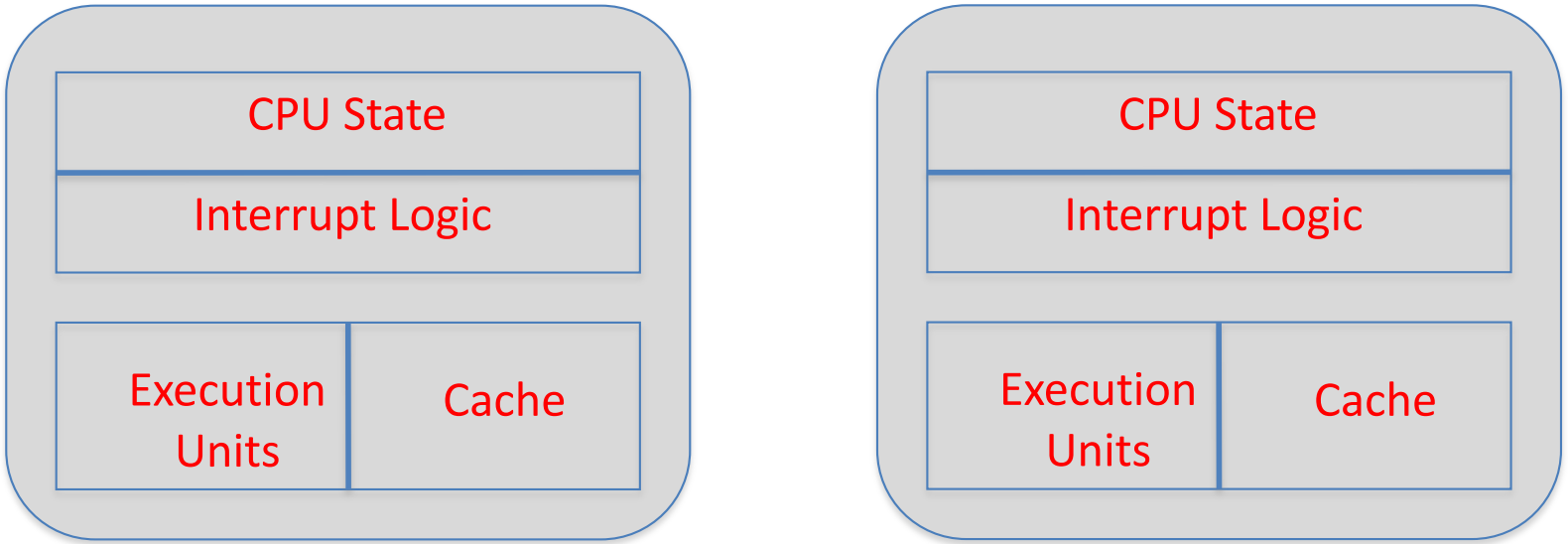
Multi-core architectures (1)

- Single core



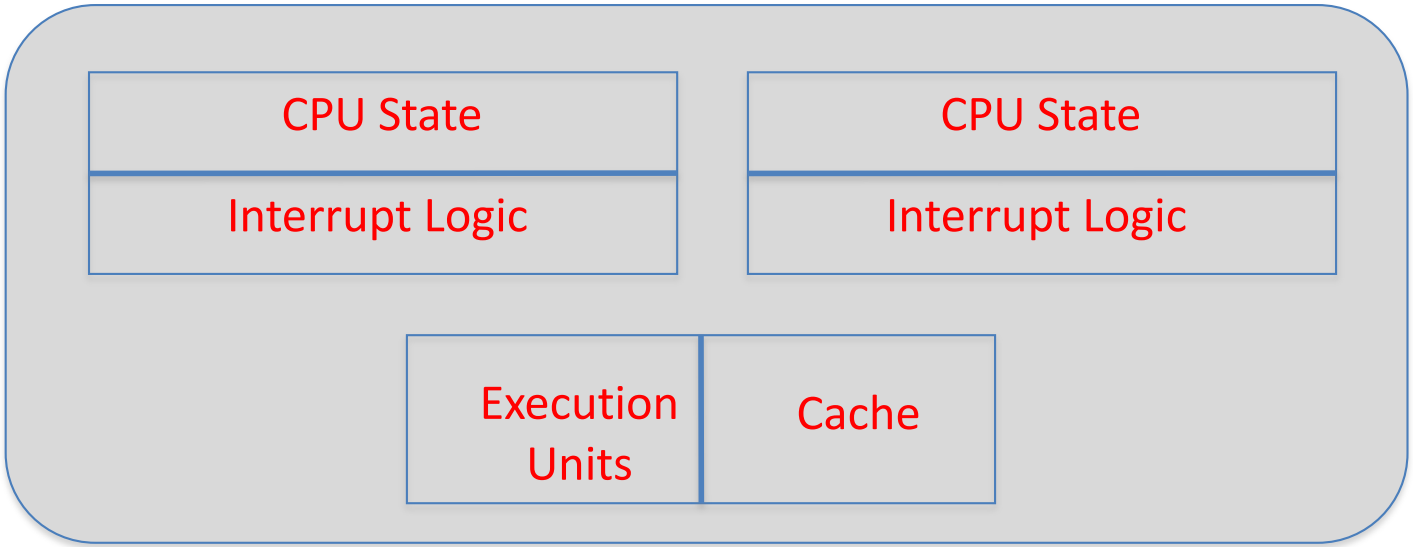
Multi-core architectures (2)

- Multiprocessor (SMP)



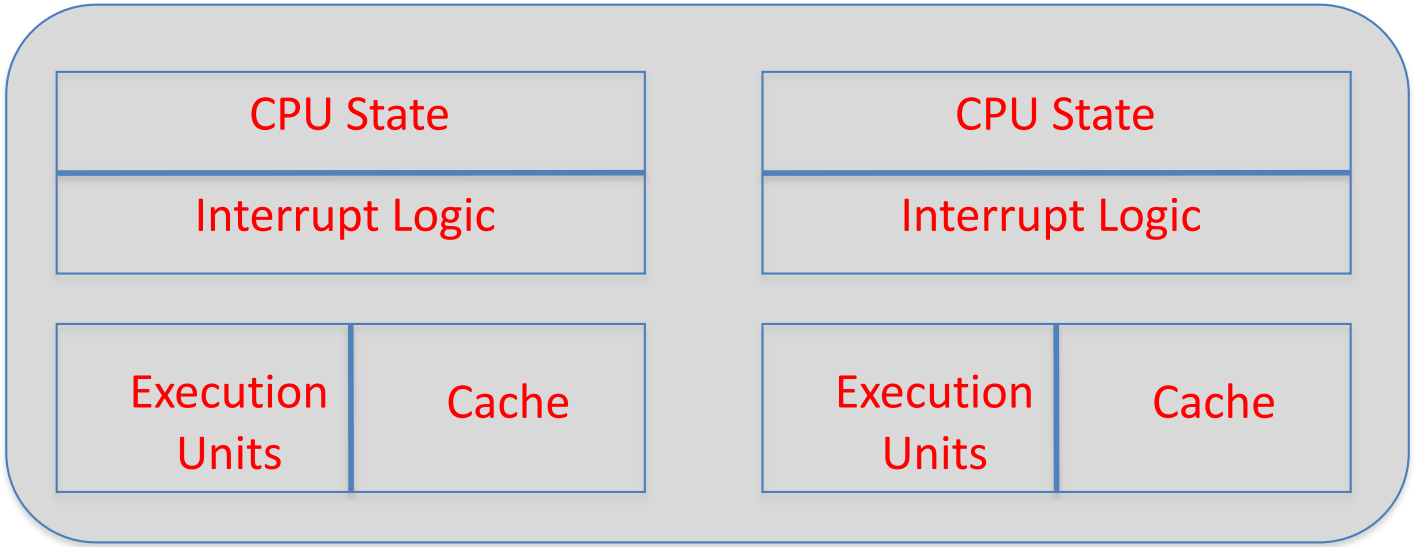
Multi-core architectures (3)

- Simultaneous Multithreading (SMT)
  - Also called Hyper-Threading Technology (HTT) as Intel's trademarked name
  - SMT uses multiple simultaneous hardware threads (aka strands) to utilize the stalling time such as cache miss and branch misprediction in a processor.
  - Can help hide memory wall.
  - E.g. Intel Xeon, Pentium 4



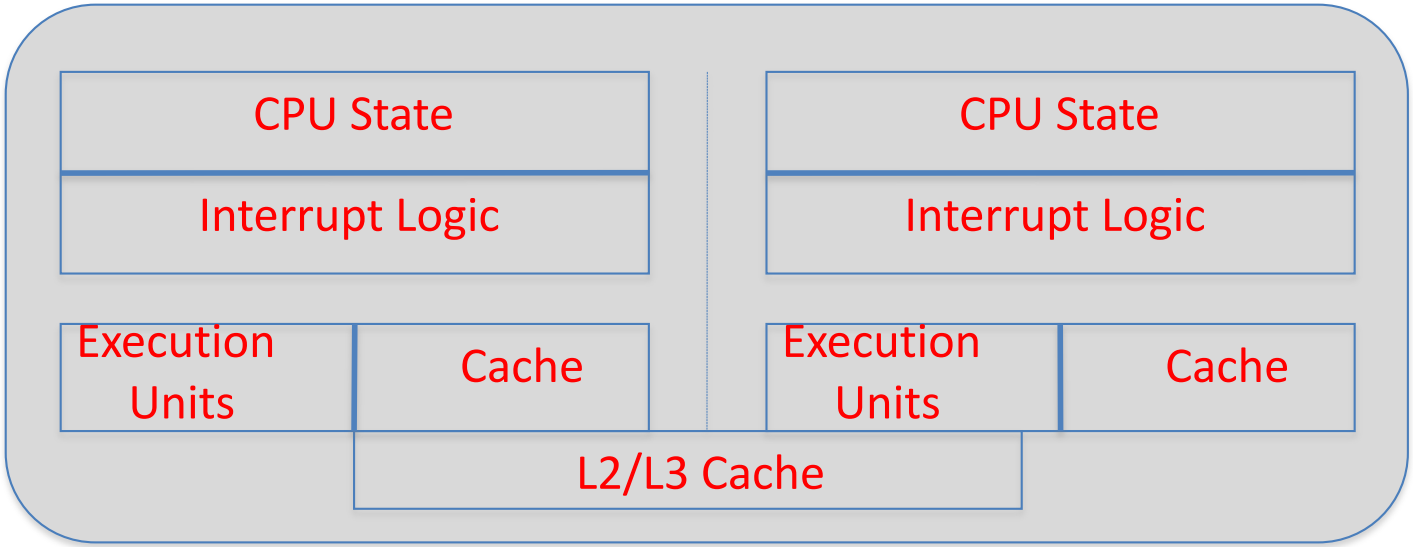
Multi-core architectures (4)

- Multi-core (CMP, Chip Multi-Processing)



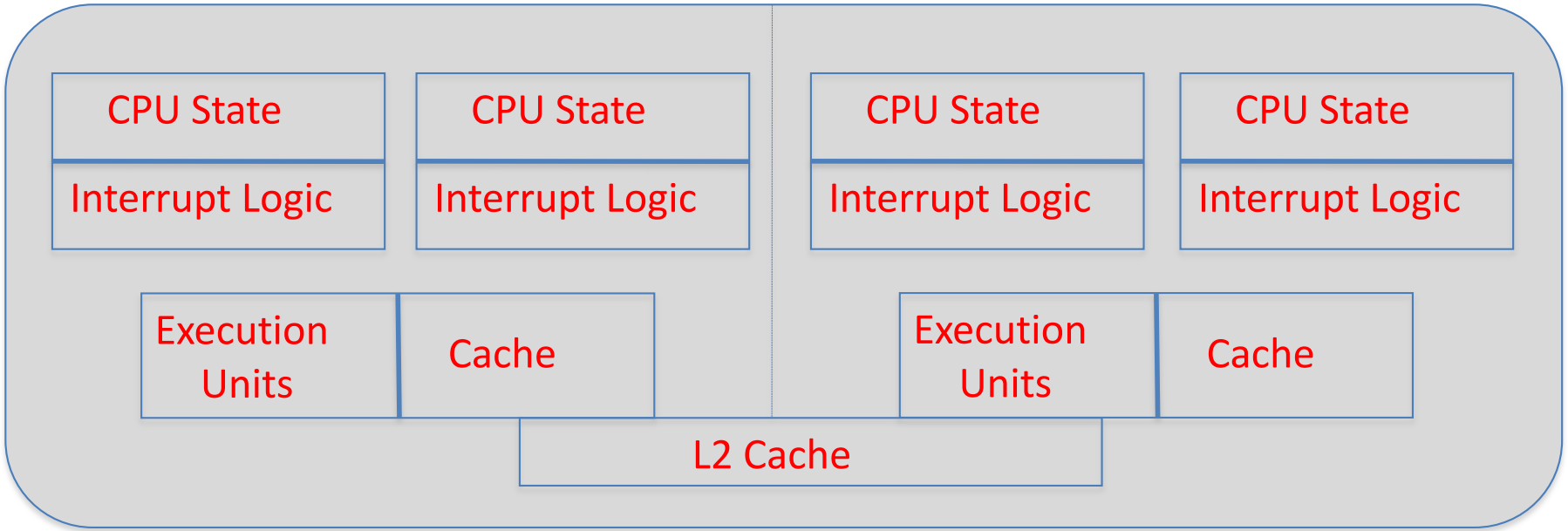
Multi-core architectures (5)

- Multi-core with shared cache (CMP)
- E.g. Intel Core Duo, AMD Barcelona (Opteron 8300)



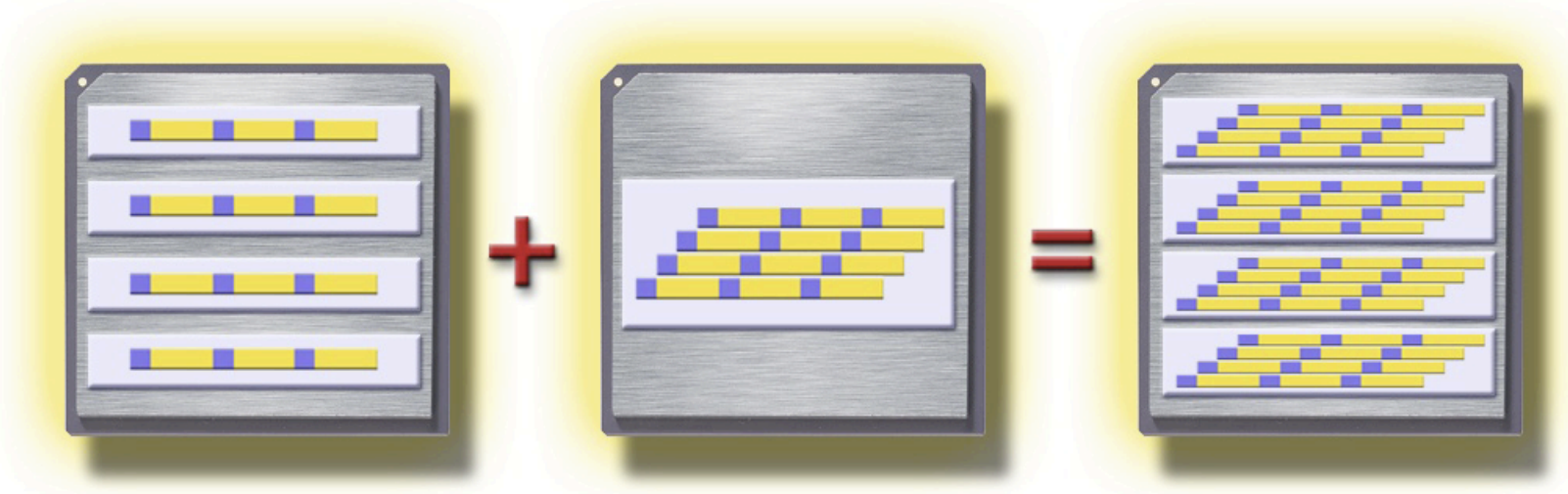
Multi-core architectures (6)

- Multi-core with SMT and shared L2 cache (Sun Niagara CMT, Chip Multi-Threading)



CMP, SMT and CMT

- CMP + SMT = CMT



Parallel computing

- Parallel computing, computing with more than one processor, has been around for more than 50 years now.

Gill writes in 1958:

"... There is therefore nothing new in the idea of parallel programming, but its application to computers. The author cannot believe that there will be any insuperable difficulty in extending it to computers. It is not to be expected that the necessary programming techniques will be worked out overnight. Much experimenting remains to be done. After all, the techniques that are commonly used in programming today were only won at the cost of considerable toil several years ago. In fact the advent of parallel programming may do something to revive the pioneering spirit in programming which seems at the present to be degenerating into a rather dull and routine occupation ..."

Gill, S. (1958), "Parallel Programming," The Computer Journal, vol. 1, April, pp. 2-10.

What is different now?

- Parallel computers are easily accessible
  - Cluster computers (Beowulf cluster)
  - Network of Workstations (NOW)
  - Grid computing resources
  - Multi-core
- No free ride on CPU speed. CPU speed is limited by
  - Heat dissipation problem, and
  - Eventually light speed.

Speedup

- The performance of a parallel program is mainly measured with speedup
  - **Speedup = Time of sequential program / Time of parallel program**

Amdahl's Law

- Proposed by Gene Amdahl in 1967
- Speedup = n / (1 + (n – 1)f),
  - where **f** is the fraction of the computation that cannot be parallelized in a parallel algorithm, and **n** is the number of processors working on the concurrent parts of the algorithm.
- What does Amdahl's Law tell us?
  - Even with an infinite number of processors, the maximum speedup is bounded by **1/f**

Gustafson's Law

- Amdahl's Law does not take into account problem size
- John L. Gustafson proposed the following formula in 1988
  - Speedup = n – α (n-1)
  - Where **n** is the number of processors, and **α** is the time for the non-parallelizable part of the parallel computation and **β** is the time for the parallel part (α+β=1 is the parallel execution time)
  - When **α** diminishes with problem size, the speedup approaches **n**

A metaphor

- Suppose a person is traveling between two cities 10 km apart, and has already spent one hour walking half the distance at 5 km/h.
- Amdahl's Law suggests:
  - “No matter how fast the person runs the last half, it is impossible to achieve 20 km/h average before reaching the second city. Since it has already taken 1 hour and there is only a distance of 10 km total; running infinitely fast would only achieve 10 km/h.”
- Gustafson's Law argues:
  - “Given enough time and distance to walk, the person's average speed can always eventually reach 20 km/h, no matter how long the person has already walked. In the two-cities case this could be achieved by running at 25 km/h for additional three hours.”

Counter-Amdahl's Law

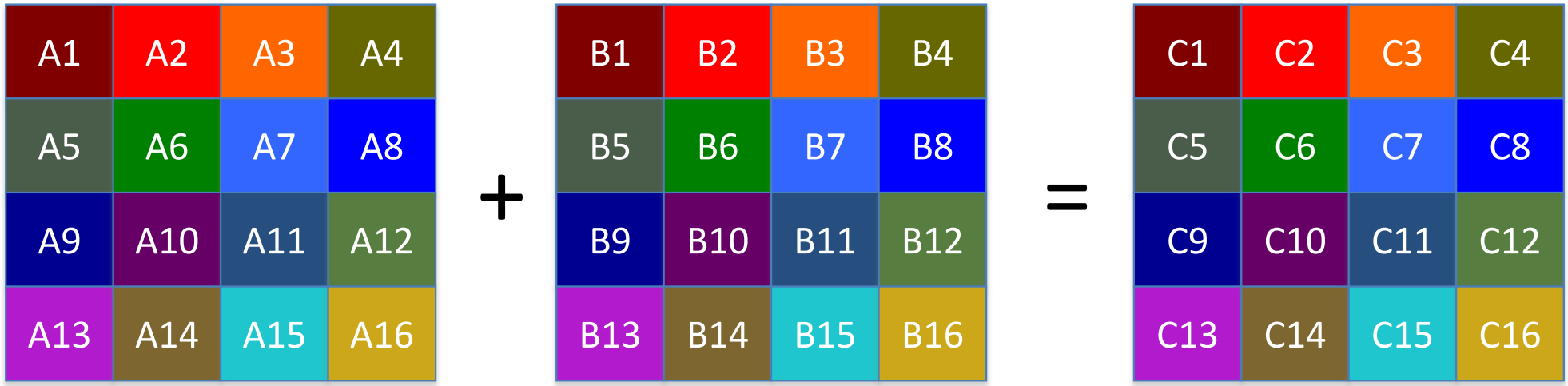
- Suppose **f** is the fraction of the computation that is serial in a parallel algorithm, and **p** is the speedup for the parallel fraction of the algorithm. Then, if **p > (1 – f)/f**, which means **p** is greater than the ratio between the parallel fraction and the serial fraction, it is more efficient to improve the serial fraction rather than the parallel fraction in order to increase the overall speedup of the algorithm.
- Detailed proof can refer to
  - Z. Huang, et al, Virtual Aggregated Processor in Multi-core Computers, NZHPC Workshop, In Proceedings of the International Conference on Parallel Distributed Computing, Applications and Technologies 2008 (PDCAT08), IEEE Computer Society, Dunedin, 2008.

Modern programming models

- SIMD (Single Instruction Multiple Data)
  - The same instruction is executed at any time by many cores but on different data
  - E.g. GPU, GPGPU with languages like CUDA and OpenCL
- SPMD (Single Program, Multiple Data)
  - Multiple processors simultaneously execute the same program, but on different data sets.
  - Unlike SIMD, there is no lockstep at instruction level and can thus be implemented on general purposed processors
  - E.g. MPI 1.0 and VOPP
- MPMD (Multiple Program, Multiple Data)
  - Multiple processors simultaneously execute different programs on different data sets.
  - Can spawn new processes/threads during execution.
  - E.g. OpenMP, PVM, and MPI 2.0

Parallelization methods (1)

- Data decomposition
  - Breaks down a task by the data it works on
- Example
  - Matrix addition A+B=C

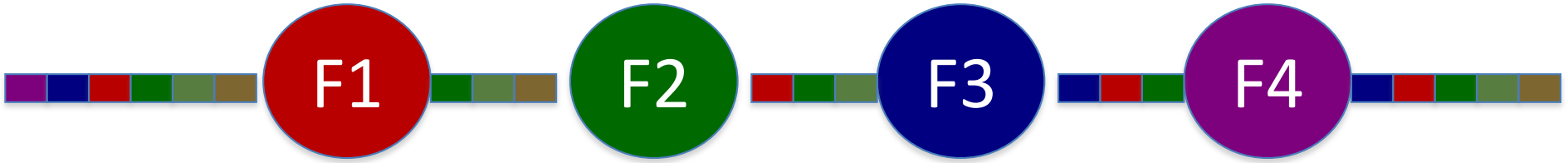


Parallelization methods (2)

- Task decomposition
  - Decompose a task by the functions it performs.
- Example
  - Gardening decomposed into mowing, weeding, and trimming
  - The task of a simulated soccer game can be divided into 22 sub-tasks, each of which simulate one player.

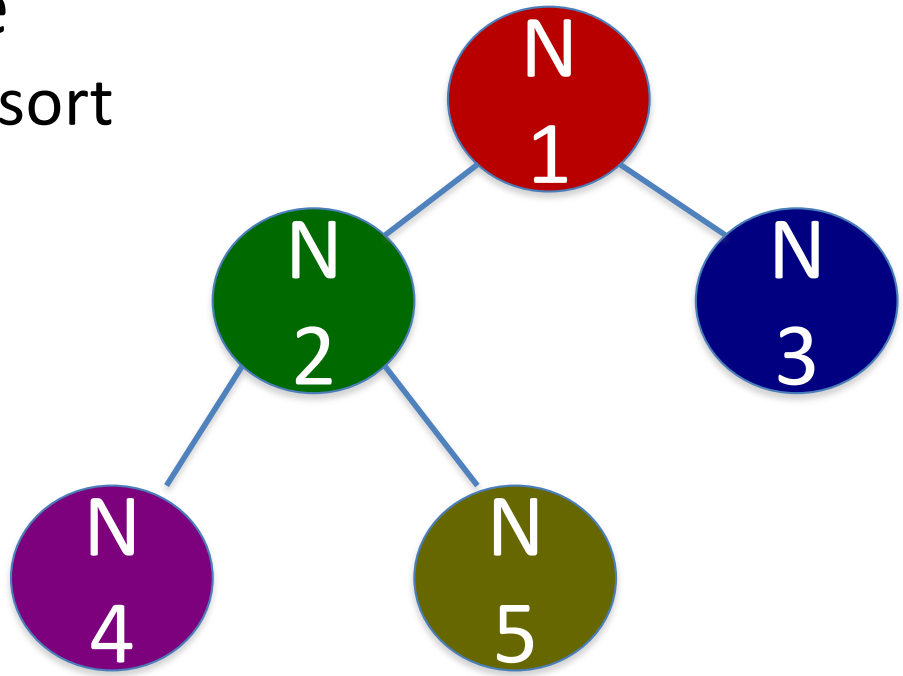
Parallelization methods (3)

- Data flow decomposition
  - Decompose a task by how data flows between different stages.
- Example
  - Producer/consumer problem, pipelining



Parallelization methods (4)

- Divide and Conquer
  - Decompose a task into sub-tasks dynamically during execution. Often task queues are used.
- Example
  - mergesort



Multithreading

- Threads are divided into
  - Heavy-weight, traditionally called processes
  - Light-weight, also called light-weight processes
- The light-weight threads
  - Unlike processes which have individual address spaces, they share the same address space and the following
    - process instructions, heap, open files (desrciptors), signal handlers, current working directory, user and group IDs
  - However, LWT has its own of the following
    - thread ID, set of registers (including program counter and stack pointer), stack (for local variables and return addresses), signal mask, priority

Shared memory

- No matter with LWT or HWT, if shared memory space is used, we need to deal with the data race issue
- Race conditions
  - Determinacy race: A race condition that occurs when two threads access the same memory location and at least one of the threads performs a write.
  - Data race: A determinacy race that occurs without mutual exclusive mechanisms protecting the involved memory location.

Data race (1)

- Data race causes bugs in programs.
- Example
  - Suppose two threads execute the same statement, and X is 0 initially

T1	T2
X=X+1	X=X+1

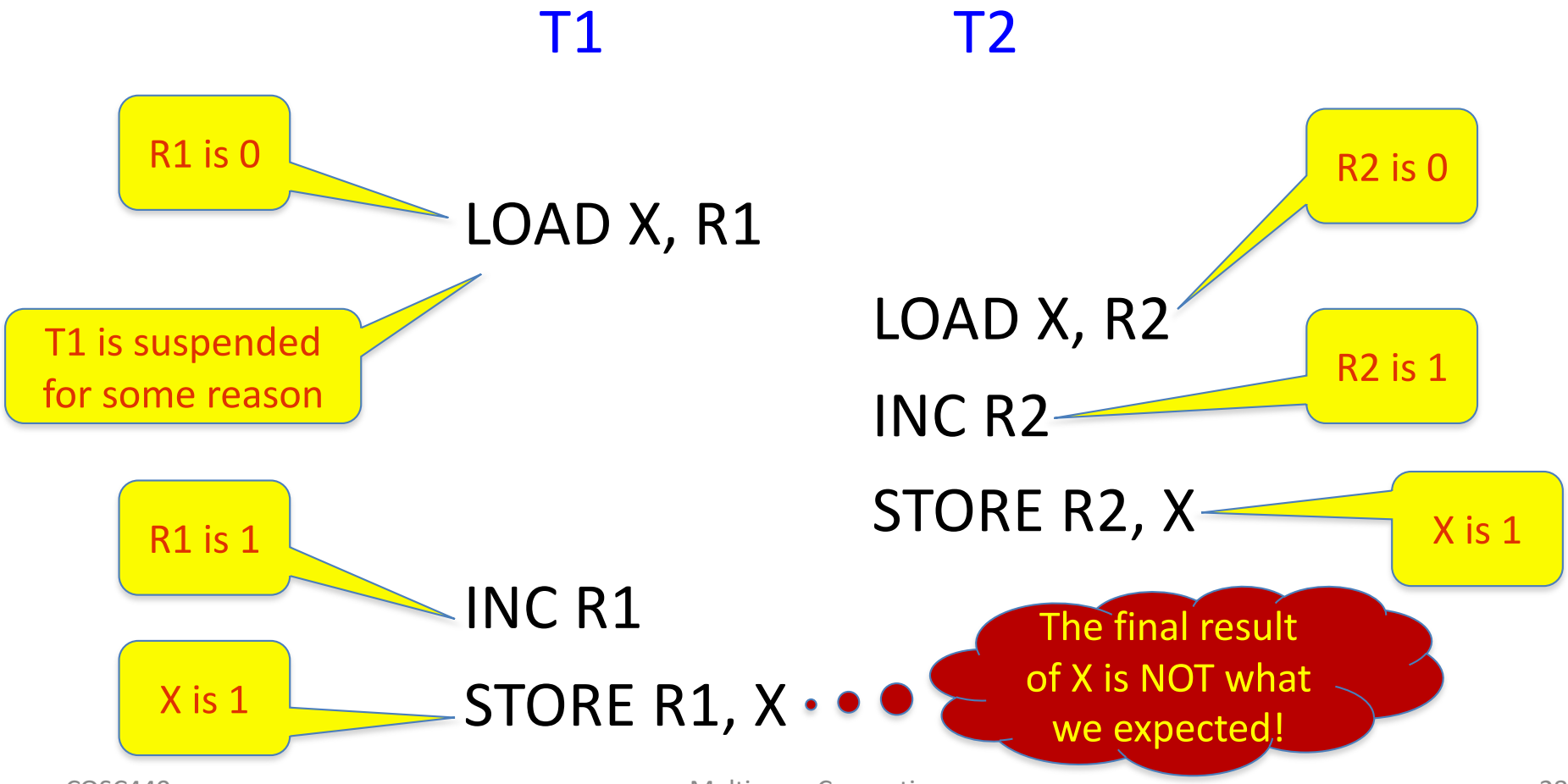
Data race (2)

- In machine language, the two threads will execute the following instructions:

T1	T2
LOAD X, R1	LOAD X, R2
INC R1	INC R2
STORE R1, X	STORE R2, X

Data race (3)

- In a concurrent environment, there is no guarantee regarding in which order the instructions of the two threads are executed
- Suppose the instructions are executed in the following order:



Solution for data race

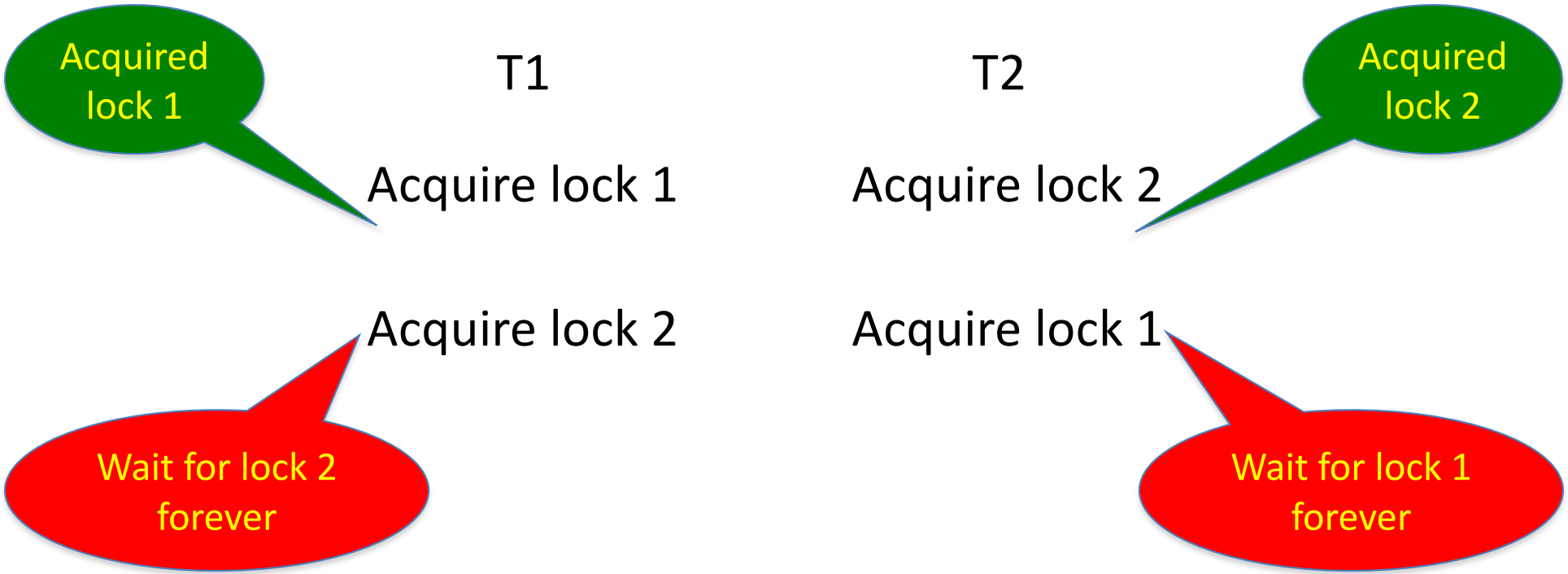
- The problem of the previous example is the violation of atomicity of **X=X+1**
  - The execution should be uninterruptible by competitors.
  - Atomicity means the set of operations should have all-or-nothing feature: they should all be completed or none of them is done before other competing operations start.
- There are two solutions to guarantee atomicity
  - Mutual exclusion (pessimistic approach)
  - Transactional memory (optimistic approach)

Mutual exclusion

- Atomic variables
  - Use hardware supported CAS (conpare-and-swap) instruction
- Semaphore
  - An integer variable S (≥0) which is only accessed by two atomic operations: **down** (P, test and decrease) and **up** (V, increase)
  - If S is 0, **down** has to wait.
- Mutex
  - When a semaphore is initialized to 1, it is called mutex, which allows only one thread to get in with **down** at any time
  - It is generally called lock, which is the only interesting application of a semaphore.
  - The code section protected by mutex through **down** and **up** is often called critical section

Deadlock

- When locks are nested, deadlock can happen.
  - Deadlock is a situation where threads are waiting for locks that will never be released.



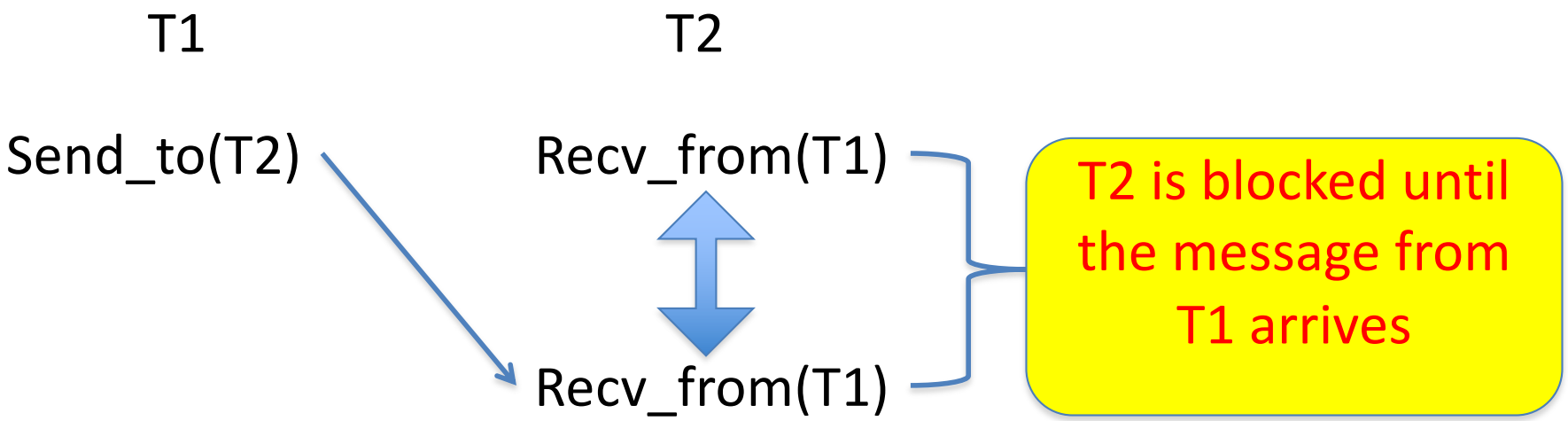
Synchronization (1)

- Barrier is often used to synchronize threads.
  - When a barrier is called in a thread, it is blocked until all other threads arrive at the barrier.
  - Similar mechanism like **join** is used in Pthreads
- Barrier can be used to avoid data race as well.

T1	T2
Work on **red** part of A	Work on **blue** part of A
barrier	barrier
Work on **blue** part of A	Work on **red** part of A

Synchronization (2)

- In message passing based programming, messages are used for synchronization.



Determinacy race

- Even with locks, there exists a kind of non-determinacy in parallel programs called determinacy race, which will cause the non-determinate results of parallel programs

Suppose X=0 initially

T1	T2
Acquire lock 1	Acquire lock 1
X=X+100	printf("The value of X is %d\n", X
Release lock 1	Release lock 1

The printed result of X is not determined, depending on which thread acquires the lock first!

Processor optimizations

- Memory ordering is often optimized for performance reason
  - For example, for two writes W1 and W2 from P1 can be seen by P1 as W1->W2, but by P2 as an order W2->W1 (which is allowed by Partial Store Order, PSO in SPARC)
  - In Intel processors, if P1 writes W1 and P2 writes W2, it is possible for P1 to observe W1->W2 while P2 observes W2->W1
- Read operations can overtake previous write operations waiting to be completed in the write buffer.
- These optimizations can cause problems for programmers assuming Sequential Consistency.

Sequential Consistency

- SC defined by Lamport in 1979
  - The result of any execution is the same as if the operations of all processes were executed in some global sequential order, and the operations of each individual process appear in this sequence in the order specified by its own program
  - Example

What are the possible Values of T1 and T2?

Initially X=Y=0;

P1	P2
X = 1;	Y = 2;
T1 = Y;	T2 = X;

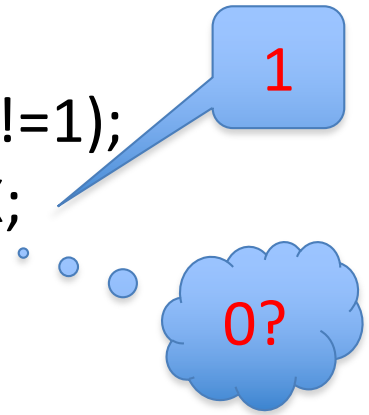
T1	T2	
2	1	✔
2	0	✔
0	1	✔
0	0	✖

Example 1

What value can T3 get from X?

Initially X=Y=0;

T1	T2	T3
X = 1;	while(X!=1) ; Y = 1;	while(Y!=1); tmp = X;



Suppose T1, T2, and T3 run on C1, C2, and C3 respectively.
Since the updates on X and Y can be seen by C3 as Y->X, it is possible for T3 to see Y:1 and execute tmp=X before X:1 arrives.

Example 2

Initially flag1 and flag2 are set 0

T1	A possible order:	T2
	T2 read flag1(0);	
flag1 = 1;	T1 write flag1(1);	flag2 = 1;
If(flag2 == 0)	T1 read flag2(0)	If(flag1 == 0)
critical section;	T2 write flag2 (1)	critical section;

The critical section should be executed by at most one thread at any time.

However, if a read operation can overtake write operations, the critical section may be executed by both threads at the same time.

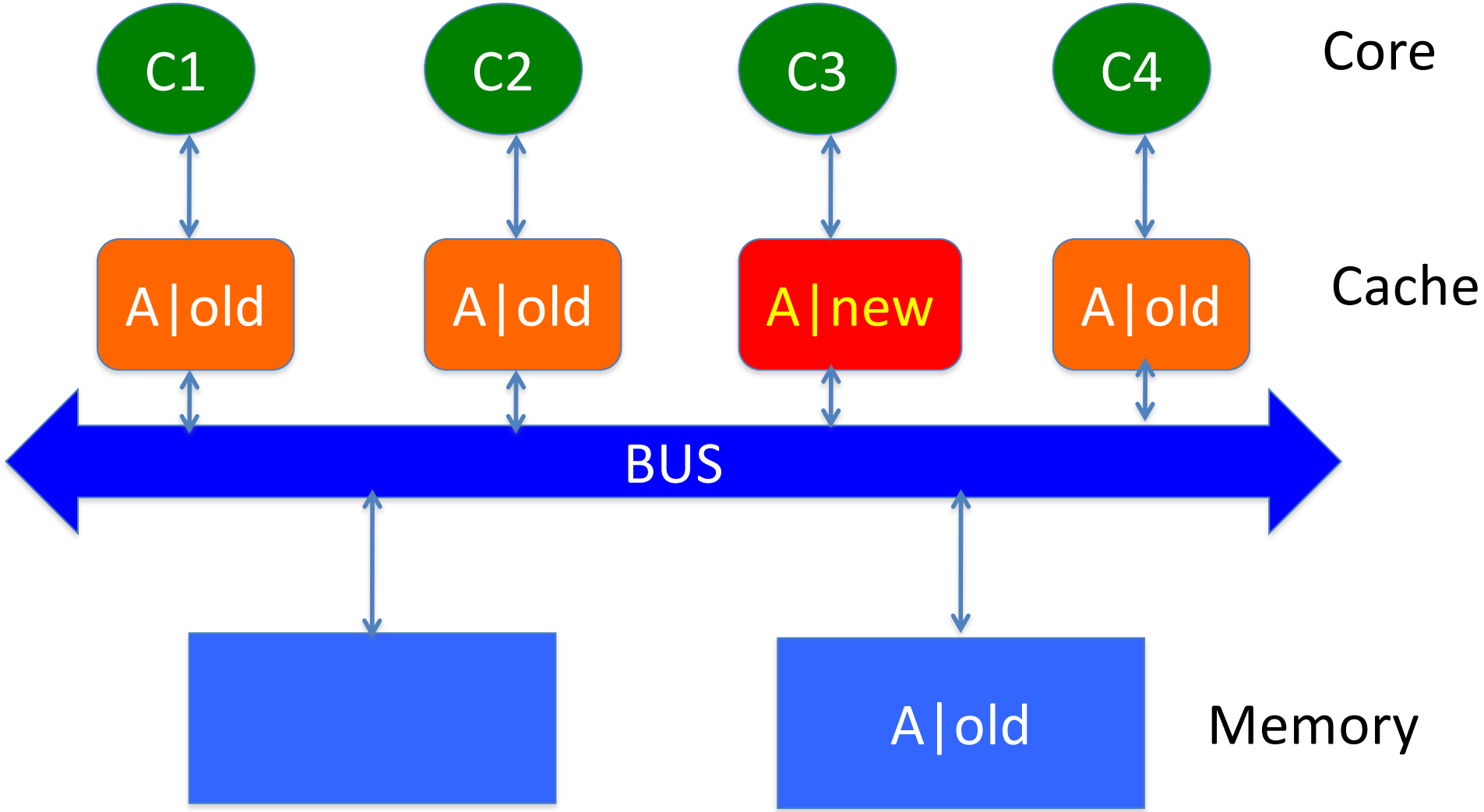
Solution--fence

- A *fence* instruction (called barrier in Linux) is often used to avoid previous problems.
- At execution time, a fence instruction guarantees completeness of all pre-fence memory operations and delays all post-fence memory operations until the completion of fence instruction cycles
  - Fences have performance problems. Intel has multiple kinds of fences to relieve the problems.
- The following gives an example of fence:

T1	T2

flag1 = 1;
fence;
If(flag2 == 0)
critical section;
	flag2 = 1;
	fence;
	If(flag1 == 0)
	critical section;

Cache coherence (1)



Cache coherence (2)

- Cache coherence protocol is used to propagate a write to other copies
- There are two basic protocols
  - Invalidate: remove old copies from other caches
  - Update: update old copies in other caches to the new value
- A cache line is used as the basic unit to invalidate or update caches.
- Normally an invalidate protocol is used in processors

False sharing

- A false sharing occurs when two independent variables are located in the same cache line
  - A cache line in modern processors can be up to 64 bytes or more
- False sharing can cause unnecessary performance loss
- Suppose eight threads work on an array A[8] in parallel, which happen to be in the same cache line.

T1	T2	...	T8
A[0] = 0;	A[1] = 1;		A[7] = 7;

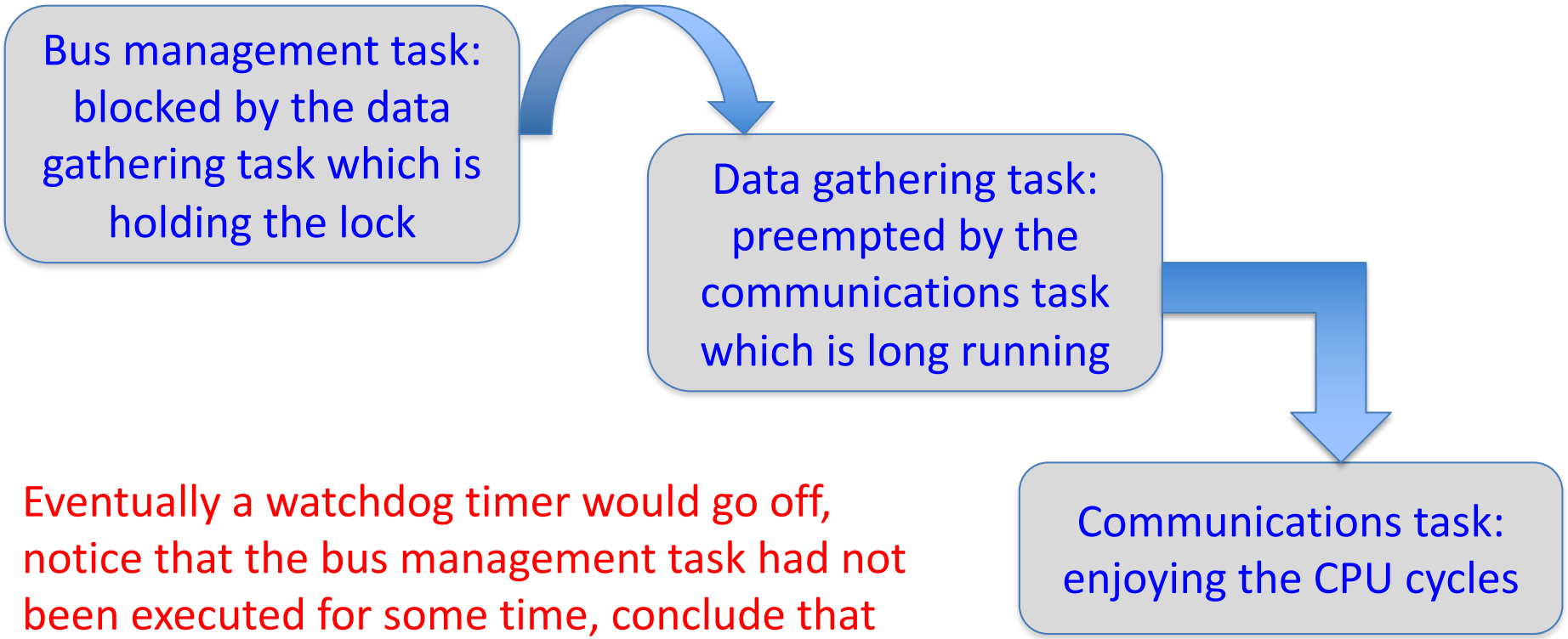
The performance will be the same as if these statements are executed serially, since each time a thread has to own the cache line before the write access. Even worse, each time the copies of the cache line in other caches are invalidated. Solution: padding!

Priority inversion (1)

- Locking can cause priority inversion in real time systems
  - priority inversion is a situation where a low priority task holds a lock that is required by a high priority task, which causes the high priority task to be blocked until the low priority task has released the lock and thus invert the priorities of the two tasks.
- A typical priority inversion was experienced by the Mars lander “Mars Pathfinder”.
  - There three tasks:
    - Data gathering task (DGT): low priority, infrequent, short running;
    - Bus management task (BMT): high priority, frequent;
    - Communications task (CT): medium priority, infrequent, long running;
  - GDT and BMT share an information bus and synchronized with a *mutex*.

Priority inversion (2)

- The following scenario causes frequent reset of the pathfinder:



Eventually a watchdog timer would go off, notice that the bus management task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

Transactional memory

- TM is a good solution to deadlock and priority inversion, though it still has data race issue and livelock problem.
- The idea is to treat shared memory operations as transactions, which can be committed or aborted but rolled back for execution again.
  - Operation conflicts such as W/W and R/W need to be detected by the conflict manager in order to commit or abort a transaction.

T1	T2
Begin_transaction; X = 1 ; if (Y == 0) {}; End_transaction;	Begin_transaction; Y = 1 ; if (X == 0) {}; End_transaction;

Livelock

- If a memory location is heavily contended, it is possible for transactions to abort each other, which causes livelock
- A livelock is a situation where transactions abort each other, resulting in no useful progress at all for any transaction.