Chapter 5: Multiprocessors (Thread-Level Parallelism)– Part 2

- Introduction
  - What is a parallel or multiprocessor system?
  - Why parallel architecture?
  - Performance potential
  - Flynn classification
- Communication models
- Architectures
- Centralized sharedmemory
- Distributed sharedmemory
- Parallel programming
- Synchronization
- **Memory consistency models**

Memory Consistency Model - Motivation

Example shared-memory program

Initially all locations = 0

Processor 1	*Processor 2*
Data = 23	while (Flag != 1) {;}
Flag = 1	… = Data

Execution (only shared-memory operations)

Processor 1	*Processor 2*
Write, Data, 23	
Write, Flag, 1	
	Read, Flag, 1
	Read, Data, ___

Memory Consistency Model: Definition

Memory consistency model

- Order in which memory operations will appear to execute
  - ⇒ What value can a read return?

Affects ease-of-programming and performance

The Uniprocessor Model

Program text defines total order = *program order*

Uniprocessor model

- Memory operations appear to execute one-at-a-time in program order
- ⇒ Read returns value of last write

BUT uniprocessor hardware

- Overlap, reorder operations

Model maintained as long as

- maintain control and data dependences

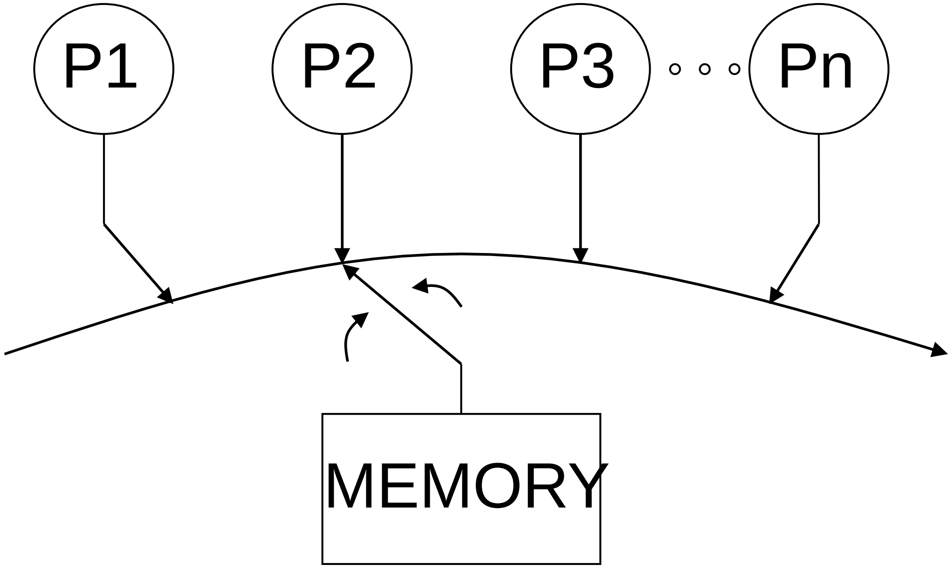
⇒ Easy to use + high performance

Implicit Memory Model

Sequential consistency (SC) [Lamport]

Result of an execution appears as if

- All operations executed in some sequential order (i.e., atomically)
- Memory operations of each process in program order



Understanding Program Order – Example 1

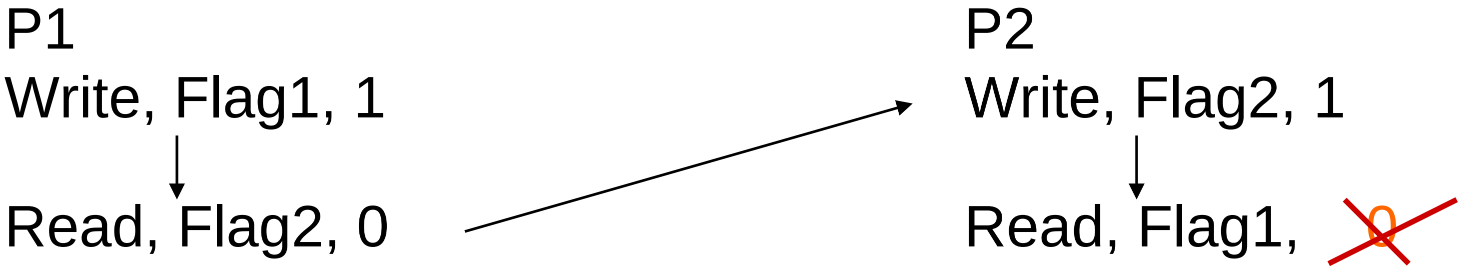
Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Execution:

P1	P2
(Operation, Location, Value)	*(Operation, Location, Value)*
Write, Flag1, 1	Write, Flag2, 1
Read, Flag2, 0	Read, Flag1, ___

Understanding Program Order – Example 1



P1
Write, Flag1, 1
Read, Flag2, 0

P2
Write, Flag2, 1
Read, Flag1, ~~0~~

Can happen if

- Write buffers with read bypassing
- Overlap, reorder write followed by read in h/w or compiler
- Allocate Flag1 or Flag2 in registers

Understanding Program Order - Example 2

Initially A = Flag = 0

P1	P2
`A = 23;`	`while (Flag != 1) {;}`
`Flag = 1;`	`... = A;`

P1	P2
Write, A, 23	Read, Flag, 0
Write, Flag, 1	
	Read, Flag, 1
	Read, A, _____

Understanding Program Order - Example 2

Initially A = Flag = 0

P1	P2
A = 23;	while (Flag != 1) {;}
Flag = 1;	... = A;

P1	P2
Write, A, 23	Read, Flag, 0
Write, Flag, 1	
	Read, Flag, 1
	Read, A, ~~0~~

Can happen if

Overlap or reorder writes or reads in hardware or compiler

Understanding Program Order: Summary

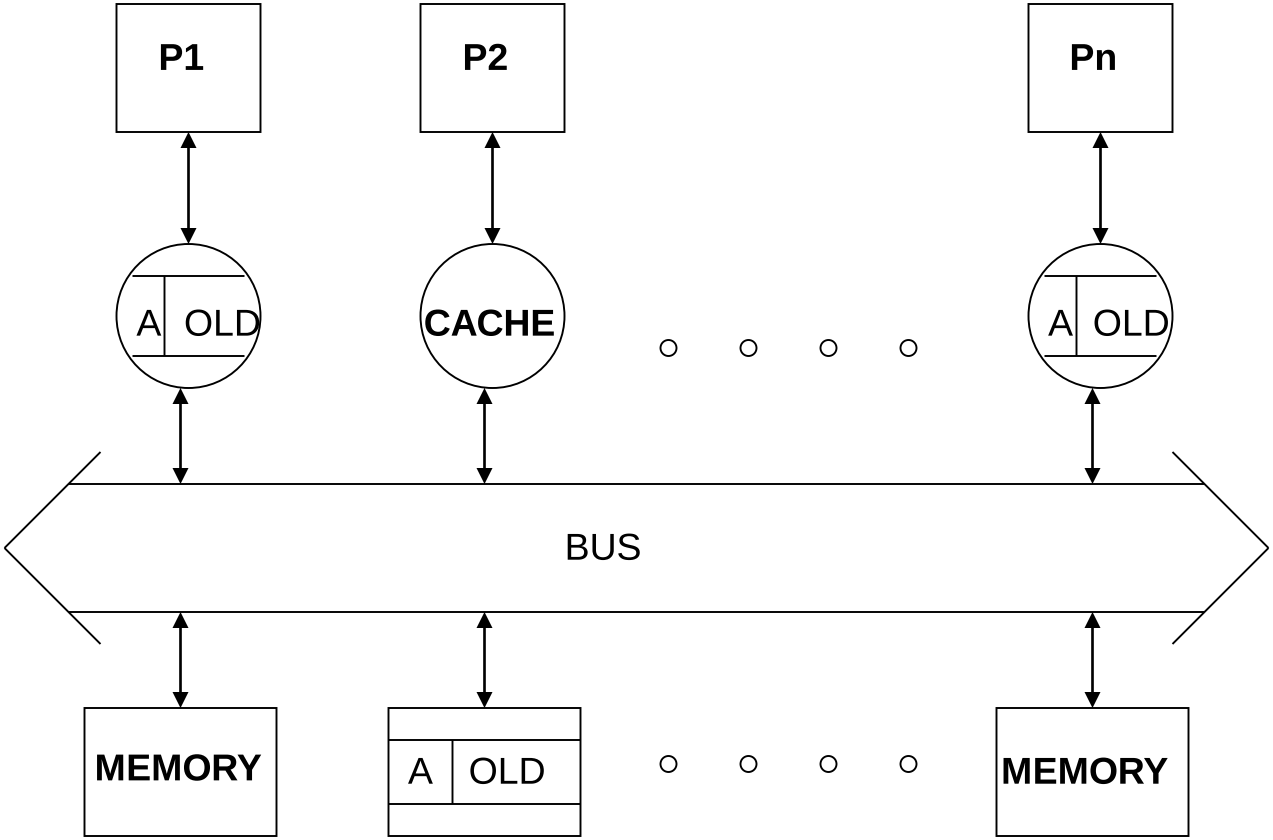
SC limits program order relaxation:

Write → Read

Write → Write

Read → Read, Write

Understanding Atomicity



A mechanism needed to propagate a write to other copies

⇒ Cache coherence protocol

Cache Coherence Protocols

How to propagate write?

- *Invalidate* -- Remove old copies from other caches
- *Update* -- Update old copies in other caches to new values

Understanding Atomicity - Example 1

Initially A = B = C = 0

```
P1         P2         P3                      P4
A = 1;     A = 2;     while (B != 1) {;}      while (B != 1) {;}
B = 1;     C = 1;     while (C != 1) {;}      while (C != 1) {;}
                      tmp1 = A;               tmp2 = A;
```

Understanding Atomicity - Example 1

Initially A = B = C = 0

P1	P2	P3	P4
A = 1;	A = 2;	while (B != 1) {;}	while (B != 1) {;}
B = 1;	C = 1;	while (C != 1) {;}	while (C != 1) {;}
		tmp1 = A; ~~1~~	tmp2 = A; ~~2~~

Can happen if updates of A reach P3 and P4 in different order

Coherence protocol must serialize writes to same location

(Writes to same location should be seen in same order by all)

Understanding Atomicity - Example 2

Initially A = B = 0

P1	P2	P3
A = 1	while (A != 1) ;	while (B != 1) ;
	B = 1;	tmp = A

P1	P2	P3
Write, A, 1		
	Read, A, 1	
	Write, B, 1	
		Read, B, 1
		Read, A, ~~0~~

Can happen if read returns new value before all copies see it

SC Summary

SC limits

- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- When a processor can read the value of a write
- Unserialized writes to the same location

Alternative

- (1) Aggressive hardware techniques proposed to get SC w/o penalty using speculation and prefetching
  - But compilers still limited by SC
- (2) Give up sequential consistency
  - Use relaxed models

Classification for Relaxed Models

Typically described as system optimizations - system-centric

Optimizations

- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- Read others’ write early
- Read own write early

All models provide safety net

All models maintain uniprocessor data and control dependences, write serialization

Some System-Centric Models

Relaxation:	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

System-Centric Models: Assessment

System-centric models provide higher performance than SC

BUT 3P criteria

- Programmability?
  - Lost intuitive interface of SC
- Portability?
  - Many different models
- Performance?
  - Can we do better?
- Need a higher level of abstraction

An Alternate Programmer-Centric View

One source of consensus

- Programmers need SC to reason about programs

But SC not practical today

- How about the next best thing…

A Programmer-Centric View

Specify memory model as a contract

System gives sequential consistency

IF programmer obeys certain rules

+ Programmability

+ Performance

+ Portability

The Data-Race-Free-0 Model: Motivation

Different operations have different semantics

```
P1                              P2
A =  23;                        while (Flag != 1)  {;}
B =  37;                       … = B;
Flag = 1;                      … = A;
```

Flag = Synchronization; A, B = Data

Can reorder data operations

Distinguish data and synchronization

Need to

- Characterize data / synchronization
- Prove characterization allows optimizations w/o violating SC

Data-Race-Free-0: Some Definitions

Two operations conflict if

- Access same location
- At least one is a write

Data-Race-Free-0: Some Definitions (Cont.)

(Consider SC executions ⇒ global total order)

Two conflicting operations race if

- From different processors
- Execute one after another (consecutively)

P1	P2
Write, A, 23	
Write, B, 37	
	Read, Flag, 0
Write, Flag, 1	
	Read, Flag, 1
	Read, B, ___
	Read, A, ___

Races usually “synchronization,” others “data”

Can optimize operations that *never race*

Data-Race-Free-0 (DRF0) Definition

Data-Race-Free-0 Program

All accesses distinguished as either synchronization or data

All races distinguished as synchronization

(in any SC execution)

Data-Race-Free-0 Model

Guarantees SC to data-race-free-0 programs

It is widely accepted that data races make programs hard to debug independent of memory model (even with SC)

Distinguishing/Labeling Memory Operations

Need to distinguish/label operations at all levels

- High-level language
- Hardware

Compiler must translate language label to hardware label

Java: volatiles, synchronized

C++: atomics

Hardware: fences inserted before/after synchronization

Data-Race-Free Summary

The idea

- Programmer writes data-race-free programs
- System gives SC

For programmer

- Reason with SC
- Enhanced portability

For hardware and compiler

- More flexibility

Finally, convergence on hardware and software sides

(BUT still many problems…)