Advanced Operating Systems (CS 202)

Memory Consistency and RCU intro

Lets start with an example

Code:
Initially A = Flag = 0

P1
```
A = 23;
Flag = 1;
```

P2
```
while (Flag != 1) {;}
B = A;
```

Idea:
- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.
- What possible values can B have?

More realistic architecture

- Two key assumptions so far:
  1. processors do not cache global data
     - improving execution efficiency:
       - allow caching
         - » leads to cache coherence solved as we discussed
  2. Instructions are executed in order
     - improving execution efficiency:
       - allow processors to execute instructions out of order subject to data/control dependences
         - » this can change the semantics of the program!
         - » Reordering happens for other reasons too
         - » preventing this requires attention to memory consistency model of processor

Recall: uniprocessor execution

- Processors reorder or change operations to improve performance
  - Registers may eliminate some loads and stores
  - Load/store buffers may delay/reorder memory accesses
  - Lockup free caches; split transactions buses; …
- Constraint on reordering: must respect dependences
  - But only sequential ones
- Reorderings can be performed either by compiler or processor

Permitted memory-op reorderings

- Stores to different memory locations can be performed out of program order

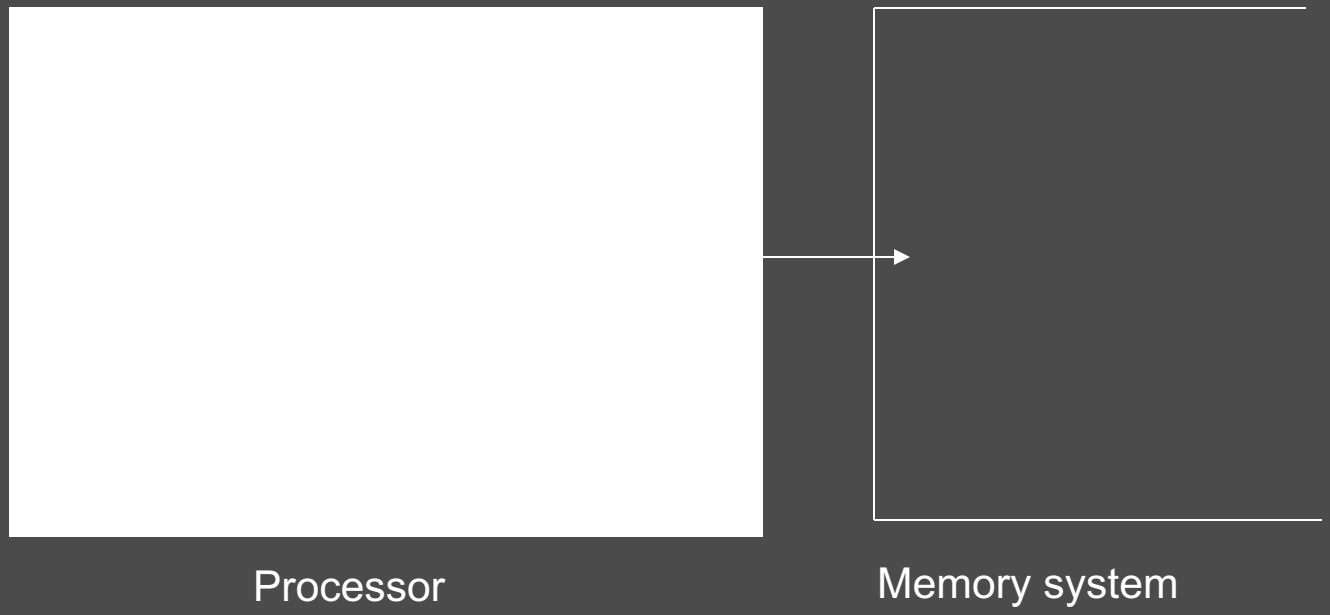
```
store v1, data                    store b1, flag
store b1, flag        ←→          store v1, data
```

- Loads from different memory locations can be performed out of program order

```
load flag, r1                     load data,r2
load data, r2         ←→          load flag, r1
```

- Load and store to different memory locations can be performed out of program order

Example cause of hardware reordering



- Store buffer holds store operations that need to be sent to memory
- Loads are higher priority operations than stores since their results are needed to keep processor busy, so they bypass the store buffer
- Load address is checked against addresses in store buffer, so store buffer satisfies load if there is an address match
- Result: load can bypass stores to other addresses

Problem in multiprocessor context

- Canonical model
  - operations from given processor are executed in program order
  - memory operations from different processors appear to be interleaved in some order at the memory
- Question:
  - If a processor is allowed to reorder independent operations in its own instruction stream, will the execution always produce the same results as the canonical model?
  - Answer: no. Let us look at some examples.

Example (I)

Code:
Initially A = Flag = 0

P1
```
A = 23;
Flag = 1;
```

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

Idea:
- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

Execution Sequence for (I)

Code:

Initially A = Flag = 0

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

Possible execution sequence on each processor:

P1	P2	
Write A 23	Read Flag	//get 0
Write Flag 1	……	
	Read Flag	//get 1
	Read A	//what do you get?

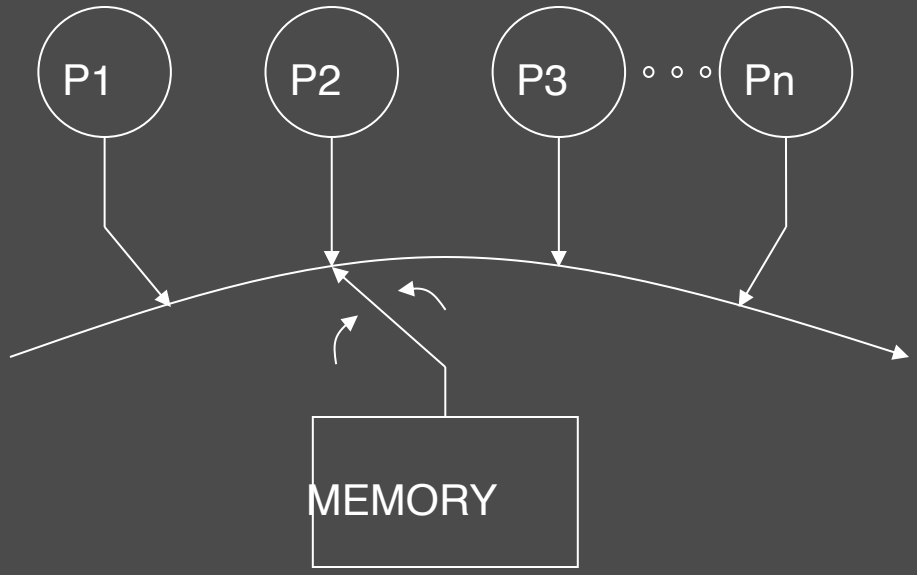
Problem: If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A.
Can happen on most modern processors.

Lessons

- Uniprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory context
  - simple parallel programs may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that
  - shared-memory programming is intuitive
  - but we do not lose uniprocessor performance?
- Many answers to this question
  - answer is called memory consistency model supported by the processor

Simplest Memory Consistency Model

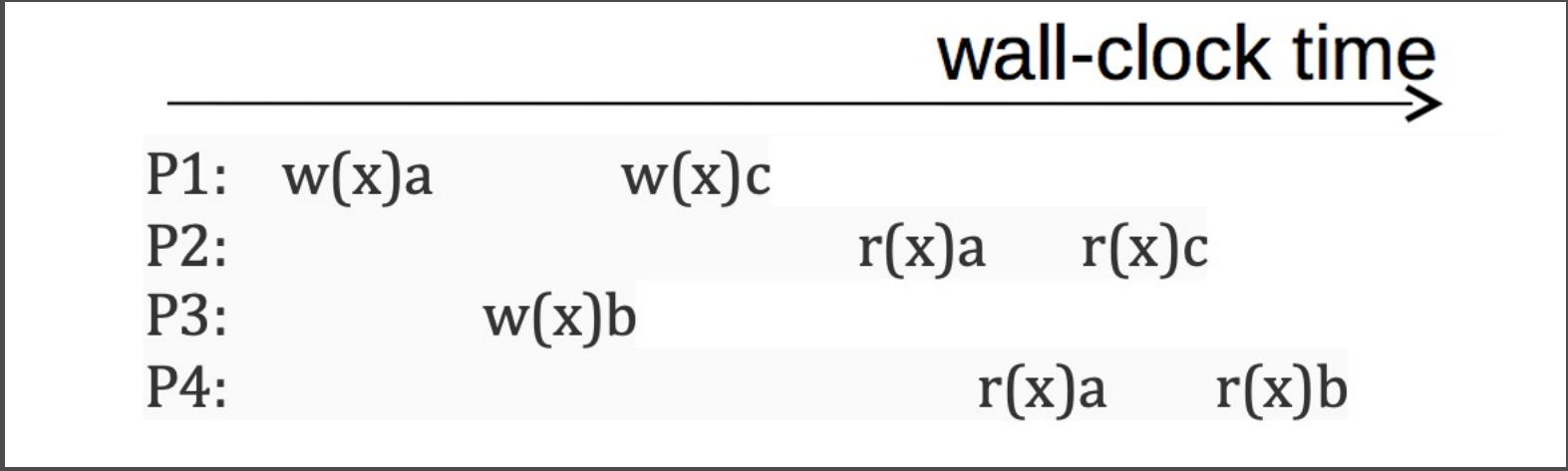
- Sequential consistency (SC) [Lamport]
  - our canonical model: processor is not allowed to reorder reads and writes to global memory



Sequential Consistency

- SC constrains all memory operations:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- Simple model for reasoning about parallel programs
  - You can verify that the examples considered earlier work correctly under sequential consistency.
- However, this simplicity comes at the cost of performance.
- Question: how do we reconcile sequential consistency model with the demands of performance?

Is this sequentially consistent?



- P1 to P4 are different programs running on different cores
- r(x)a means we read memory location x and found the value a
- w(x)a means we wrote a to memory location x

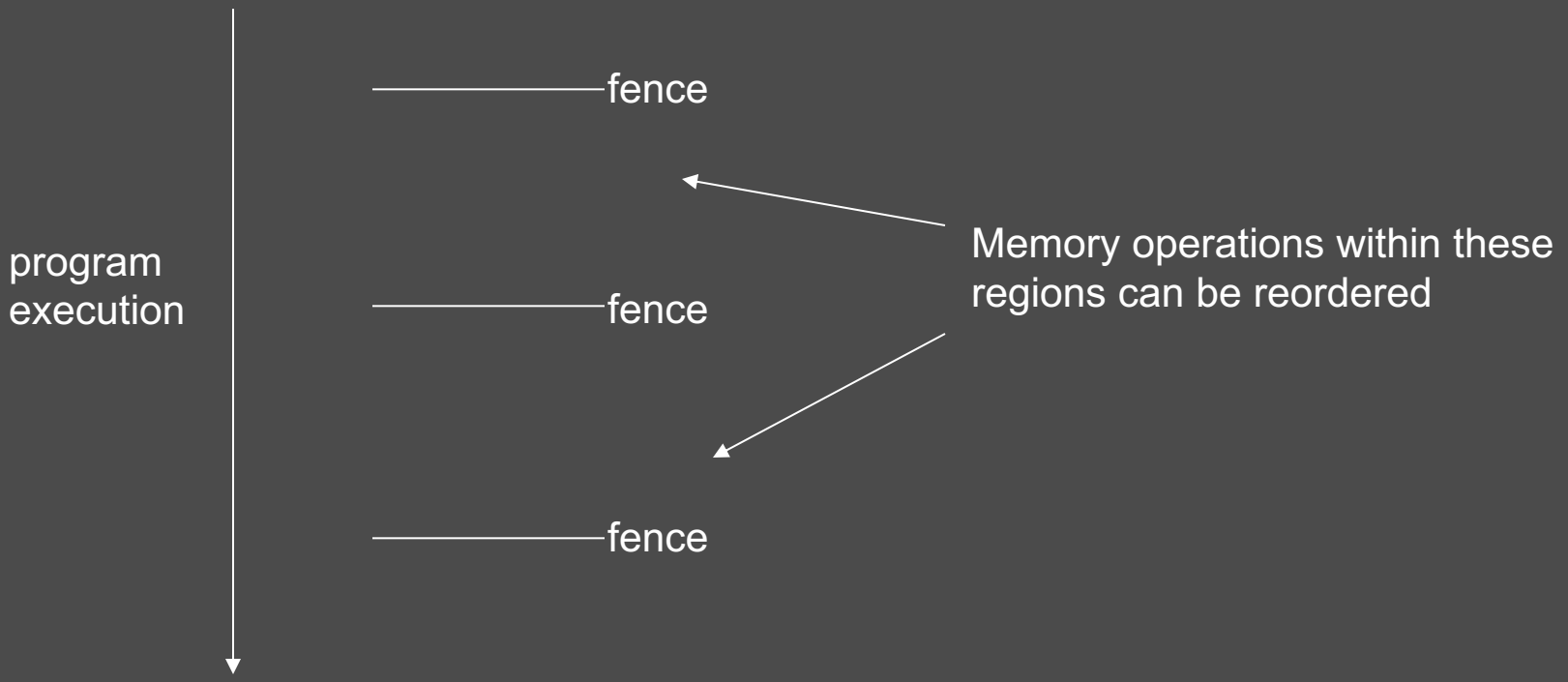
Consistency models

- Consistency models are not about memory operations from different processors.
- Consistency models are not about dependent memory operations in a single processor's instruction stream (these are respected even by processors that reorder instructions).
- Consistency models are all about ordering constraints on independent memory operations in a single processor's instruction stream that have some high-level dependence (such as flags guarding data) that should be respected to obtain intuitively reasonable results.

Relaxed consistency

- Allow reordering for performance, but provide a safety net
  - E.g., Processor has fence instruction:
  - Accesses before fence in program order must complete before fence
  - Accesses after fence in program order must wait for fence
  - Fences are performed in program order
- Weak consistency: programmer puts fences where reordering is not acceptable
- Implementation of fence:
  - processor has counter that is incremented when data op is issued, and decremented when data op is completed
    - Example: PowerPC has SYNC instruction
    - Language constructs:
      - OpenMP: flush
      - All synchronization operations like lock and unlock act like a fence

Weak ordering picture



Example revisited

Code:
Initially A = Flag = 0

P1		P2
A = 23;		
flush;	←memory fence	while (Flag != 1) {;}
Flag = 1;		B = A;

Execution:

- P1 writes data into A
- Flush waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.
- Question: does P2 need a flush between the two statements?

Another relaxed model: release consistency

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
- However,
  - acquire does not wait for accesses preceding it
  - accesses after release in program order do not have to wait for release
    - operations which follow release and which need to wait must be protected by an acquire

Implementations on Current Processors

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA–RISC)	Y	Y	Y	Y				
PA–RISC CPUs								
POWER	Y	Y	Y	Y	Y	Y		Y
SPARC RMO	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86	Y	Y		Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries				Y				Y

Comments

- In the literature, there are a large number of other consistency models
  - E.g., Eventual consistency
  - We will revisit some later…
- It is important to remember that these are concerned with reordering of independent memory operations within a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.
  - Therefore, we have to be careful with concurrency primitives!
  - How do we get them right?
  - How do we make them portable?

Advanced Operating Systems (CS 202)

Read Copy Update (RCU)

(some slides from Dan Porter)

Linux Synch. Primitives

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction	All
Memory barrier	Avoid instruction re-ordering	Local CPU
Spin lock	Lock with busy wait	All
Semaphore	Lock with blocking wait (sleep)	All
Seqlocks	Lock based on access counter	All
Local interrupt disabling	Forbid interrupt on a single CPU	Local
Local softirq disabling	Forbid deferrable function on a single CPU	Local
Read-copy-update (RCU)	Lock-free access to shared data through pointers	All

Also Read-write locks

Why are we reading this paper?

- Example of a synchronization primitive that is:
  - Lock free (mostly/for reads)
  - Tuned to a common access pattern
  - Making the common case fast
- What is this common pattern?
  - A lot of reads
  - Writes are rare
    - Prioritize writes
  - Stale copies are short lived – time heals all wounds
  - Ok to read a slightly stale copy
    - But that can be fixed too

Traditional OS locking designs

- poor concurrency
  - Especially if mostly reads
- Fail to take advantage of event-driven nature of operating systems
- Locks have acquire and release cost
  - Use atomic operations which are expensive
  - Can dominate cost for short critical regions
  - Locks become the bottleneck

Readers/Writers

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
```

```
reader {
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);      // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount
}
```

Naïve implementation – can be done using just atomic instructions

Lock free data structures

- Do not require locks
- Good if contention is rare
- But difficult to create and error prone
- RCU is a mixture
  - Concurrent changes to pointers a challenge for lock-free
  - RCU serializes writers using locks
  - Win if most of our accesses are reads

Example of lock free synchronization

```
int retry = 0;
while (true)
{
    // DO NOT worry about writer access for now - this is for demonstrating atomic operation only

    int prev_readers = _readers;            // current count
    int new_readers = prev_reader + 1;      // new count - note this is using the local value prev_readers
                                            // in case _readers has changed in between
    if (_readers.compare_exchange_weak(prev_readers, new_readers))
    {
        // we've won the race
        break;
    }

    // we've failed, retry
    retry++;
    if (retry > RETRY_COUNT)
    {
        retry = 0;
        std::this_thread::yield();
    }
}
```

Correction: second argument should be _readers

Credit: https://yizhang82.dev/lock-free-rw-lock