Memory Consistency

Chris Rossbach

Outline for Today

- Questions?
- Administrivia
  - Lab 3 looms large: Go go go!
- Agenda
  - Memory Consistency
  - Message Passing background
  - Concurrency in Go
  - Thoughts and guidance on Lab 3

- Acknowledgements: Rob Pike's 2012 Go presentation is excellent, and I borrowed from it: https://talks.golang.org/2012/concurrency.slide

Memory Consistency

Memory Consistency

- Formal specification of memory semantics
  - Statement of how shared memory will behave with multiple CPUs
  - Ordering of reads and writes

Memory Consistency

- Formal specification of memory semantics
  - Statement of how shared memory will behave with multiple CPUs
  - Ordering of reads and writes

- Memory Consistency != Cache Coherence
  - Coherence: propagate updates to cached copies
    - Invalidate vs. Update
  - Coherence vs. Consistency?
    - **Coherence:** ordering of ops. at a single location
    - **Consistency:** ordering of ops. at multiple locations

Consistency: Canonical Challenge

```
Initially, Flag1 = Flag2 = 0
```

P1

```
Flag1 = 1
if (Flag2 == 0)
  enter CS
```

P2

```
Flag2 = 1
if (Flag1 == 0)
  enter CS
```

Consistency: Canonical Challenge

```
Initially, Flag1 = Flag2 = 0
```

P1

```
Flag1 = 1
if (Flag2 == 0)
  enter CS
```

P2

```
Flag2 = 1
if (Flag1 == 0)
  enter CS
```

Can both P1 and P2 wind up in the critical section at the same time?

Consistency: Canonical Challenge

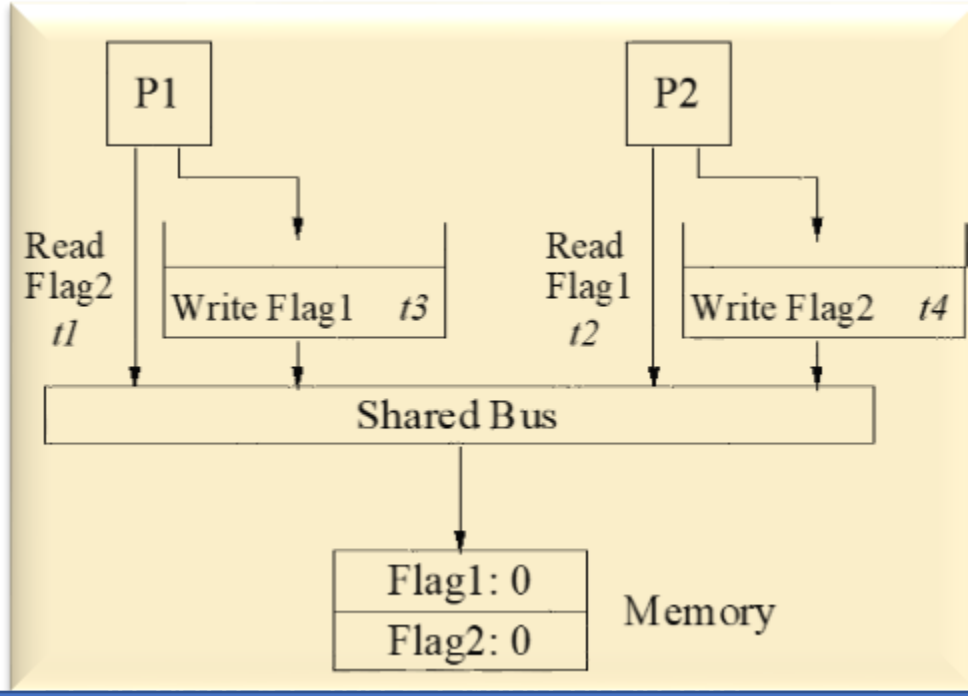
```
Initially, Flag1 = Flag2 = 0
```

P1

```
Flag1 = 1
if (Flag2 == 0)
  enter CS
```

P2

```
Flag2 = 1
if (Flag1 == 0)
  enter CS
```

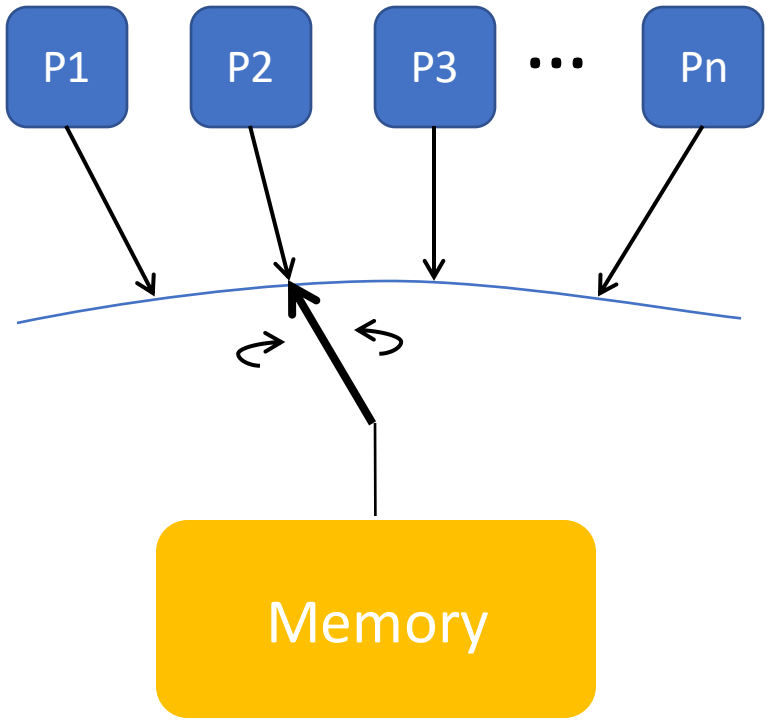


Write Buffers

- P_0 write → queue op in write buffer, proceed
- P_0 read → look in write buffer,
- P_(x != 0) read → old value: write buffer hasn't drained

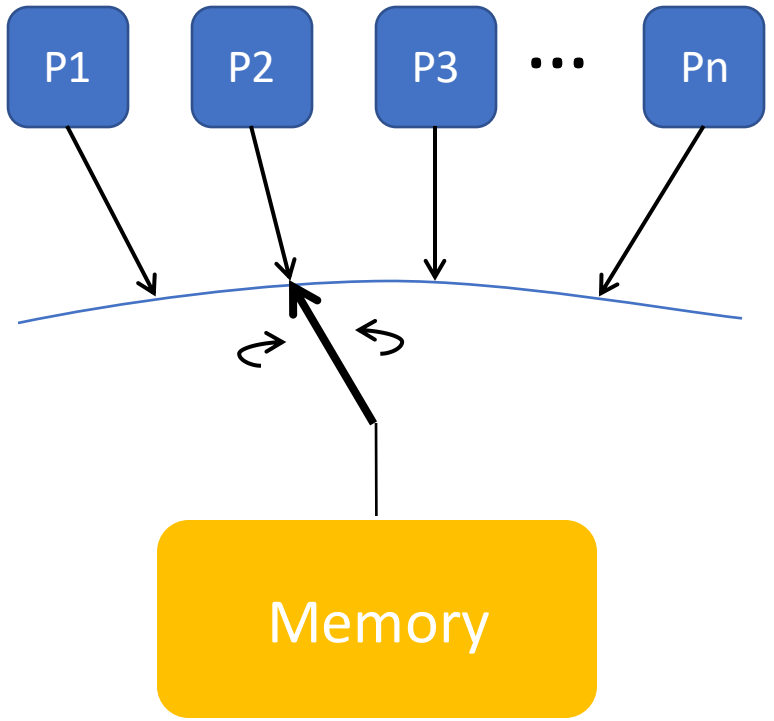
Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor



Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor



Trying to mimic Uniprocessor semantics:

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write

Requirements for Sequential Consistency

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order
- *Write Atomicity*
  - Writes to the same location seen by all other CPUs
  - Subsequent reads must not return value of a write until propagated to all
  - Note: write atomicity → property of *schedule*: writes *appear* atomic

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order
- *Write Atomicity*
  - Writes to the same location seen by all other CPUs
  - Subsequent reads must not return value of a write until propagated to all
  - Note: write atomicity → property of *schedule*: writes *appear* atomic
- Write acknowledgements are necessary
  - Cache coherence provides these properties for a *cache-only* system

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order
- *Write Atomicity*
  - Writes to the same location seen by all other CPUs
  - Subsequent reads must not return value of a write until propagated to all
  - Note: write atomicity → property of *schedule*: writes *appear* atomic
- Write acknowledgements are necessary
  - Cache coherence provides these properties for a *cache-only* system

Disadvantages:

- Difficult to implement!
  - Coherence to (e.g.) write buffers is hard
- Sacrifices many potential optimizations
  - Hardware (cache) and software (compiler)
  - Major performance hit

Sequential Consistency: Canonical Example

```
Initially, Flag1 = Flag2 = 0
```

P1	**P2**
`Flag1 = 1`	`Flag2 = 1`
`if (Flag2 == 0)`	`if (Flag1 == 0)`
`  enter CS`	`  enter CS`

Sequential Consistency: Canonical Example

```
Initially, Flag1 = Flag2 = 0
```

P1

```
Flag1 = 1
if (Flag2 == 0)
  enter CS
```

P2

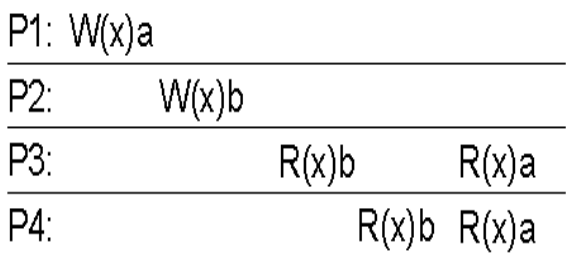
```
Flag2 = 1
if (Flag1 == 0)
  enter CS
```

Can both P1 and P2 wind up in the critical section at the same time?

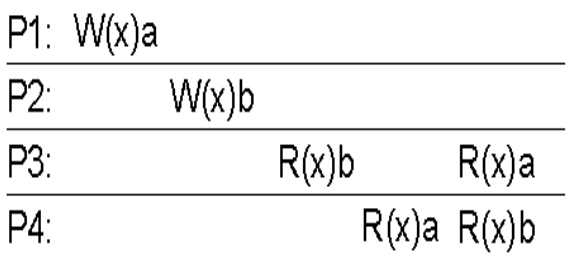
In an SC system NO

Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



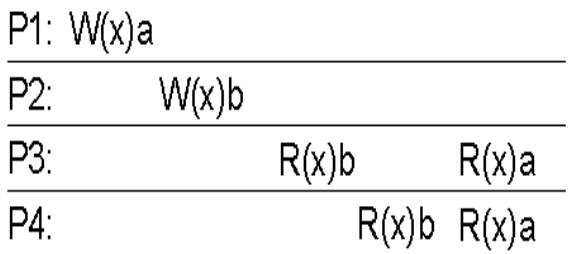
(a)

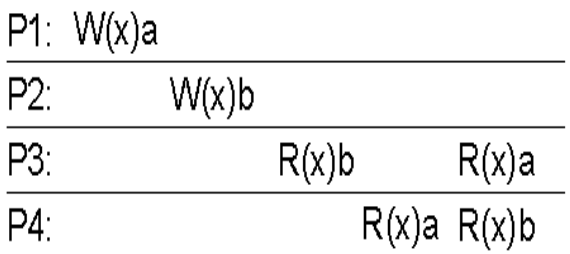


(b)

Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



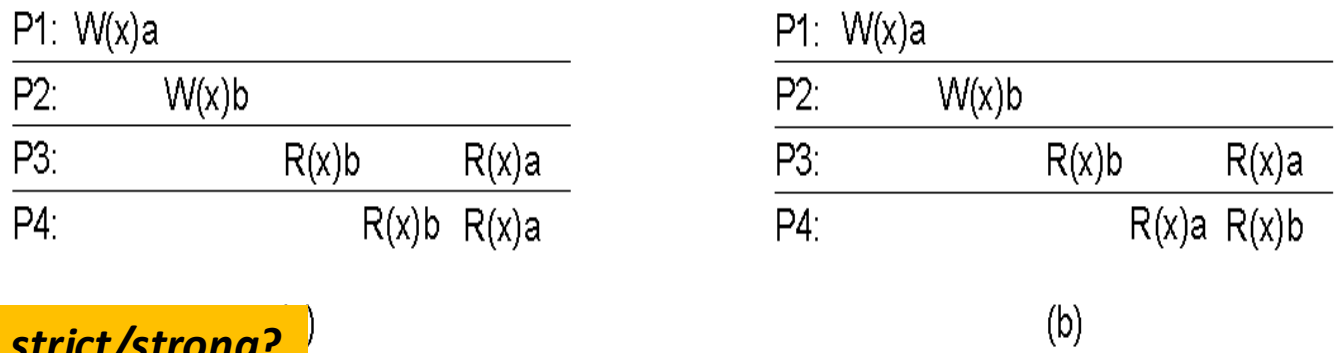


(b)

- ***Why is this weaker than strict/strong?***

Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



- ***Why is this weaker than strict/strong?***
- ***Nothing is said about "most recent write"***

More Consistency Motivation

Initially, A = B = 0

How many possible final values of register1?

P1	**P2**	**P3**
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register1 = A

More Consistency Motivation

Initially, A = B = 0

How many possible final values of register1?

P1	**P2**	**P3**
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register1 = A

Key issue:

- P2 and P3 may not see writes to A, B in the same order
- Implication: P3 can see B == 1, but A == 0 which is incorrect
- Wait! Why would this happen?

More Consistency Motivation

Initially, A = B = 0

How many possible final values of register1?

P1	**P2**	**P3**
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register1 = A

Key issue:

- P2 and P3 may not see writes to A, B in the same order
- Implication: P3 can see B == 1, but A == 0 which is incorrect
- Wait! Why would this happen?

Sources of re-ordering:

- Post-retirement store queues
- Load queues
- O-o-O instruction processing
- Non-Uniform topologies
- Compiler optimizations

Consistency:
Each “flavor” is some combination of allowed/supported optimizations

Why Relax Consistency?

- Motivation, originally
  - Allow in-order processors to overlap store latency with other work
  - “Other work” depends on loads, so loads bypass stores using a *store queue*
- PC (processor consistency), SPARC TSO, IBM/370
  - Just relax read-to-write program order requirement
- Subsequently
  - Hide latency of one store with latency of other stores
  - Stores to be performed OOO with respect to each other
  - Breaks SC even further
- This led to definition of SPARC PSO/RMO, WO, PowerPC WC, Itanium
- What’s the problem with relaxed consistency?
  - Shared memory programs can break if not written for specific cons. model

Relaxed Consistency Models

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's Write early

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's Write early
- *Requirement:* synchronization primitives for safety
  - Fence, barrier instructions etc

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's W
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consis

```
static inline void arch_write_lock(arch_rwlock_t *rw) {
   asm volatile(LOCK_PREFIX WRITE_LOCK_SUB(%1) "(%0)\n\t"
        "jz 1f\n"
         "call __write_lock_failed\n\t"
        "1:\n"
        ::LOCK_PTR_REG (&rw->write), "i" (RW_LOCK_BIAS) : "memory"); }
```

x86

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's V
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's W
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W

ons

cessor's V

tion pri

etc

```
static inline unsigned long
__arch_spin_trylock(arch_spinlock_t *lock)
{
  unsigned long tmp, token;
  token = LOCK_TOKEN;
  __asm__ __volatile__(
    "1: "  PPC_LWARX(%0,0,%2,1) "\n\
           cmpwi 0,%0,0\n\
           bne- 2f\n\
           stwcx. %1,0,%2\n\
           bne- 1b\n"
           PPC_ACQUIRE_BARRIER
    "2:“ : "=&r" (tmp)
         : "r" (token), "r" (&lock->slock)
         : "cr0", "memory");
    return tmp;
}
```

PowerPC

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's W
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Some Key Consistency Models

TSO: Total Store Order

- Stores are totally ordered, reads not
- Differs from PC by allowing early reads of processor's own writes

PC: Processor consistency

- Writes from processor always respect program order
- Different processors may see different interleavings from different processors

RC: Release Consistency

- Key insight: only synchronization references need to be ordered
- Hence, relax memory for all other references
  - Enable high-performance OOO implementation
- Programmer **labels** synchronization references
  - Hardware must carefully order these labeled references
- Labeling schemes:
  - Explicit synchronization ops (acquire/release)
  - Memory fence or memory barrier ops:
    - All preceding ops must finish before following ones begin

Another Good SC Exercise

Initially, x = 0, y = 0

P0:		P1:	
1.	x = 1;	1.	a = y;
2.	y = 1;	2.	b = x;

What final values of (a, b) are possible under SC?

Another Good SC Exercise

Initially, x = 0, y = 0

P0:		P1:	
1.	x = 1;	1.	a = y;
2.	y = 1;	2.	b = x;

What final values of (a, b) are possible under SC?

(0, 0), (0, 1), (1, 1)

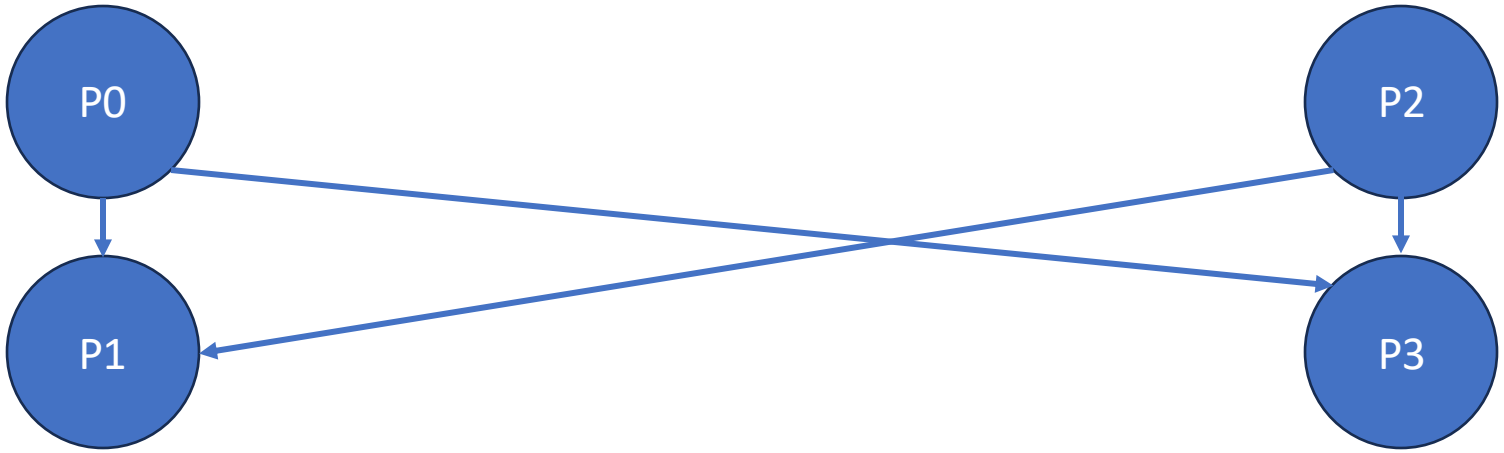
Not 1, 0

PC: Processor Consistency

- Writes from a single processor are received by all other processors in the order they were issued
- Writes from different processors may be seen in a different order by different processors
- Key idea:
  - reflect reality of networks
  - latency between nodes may be different

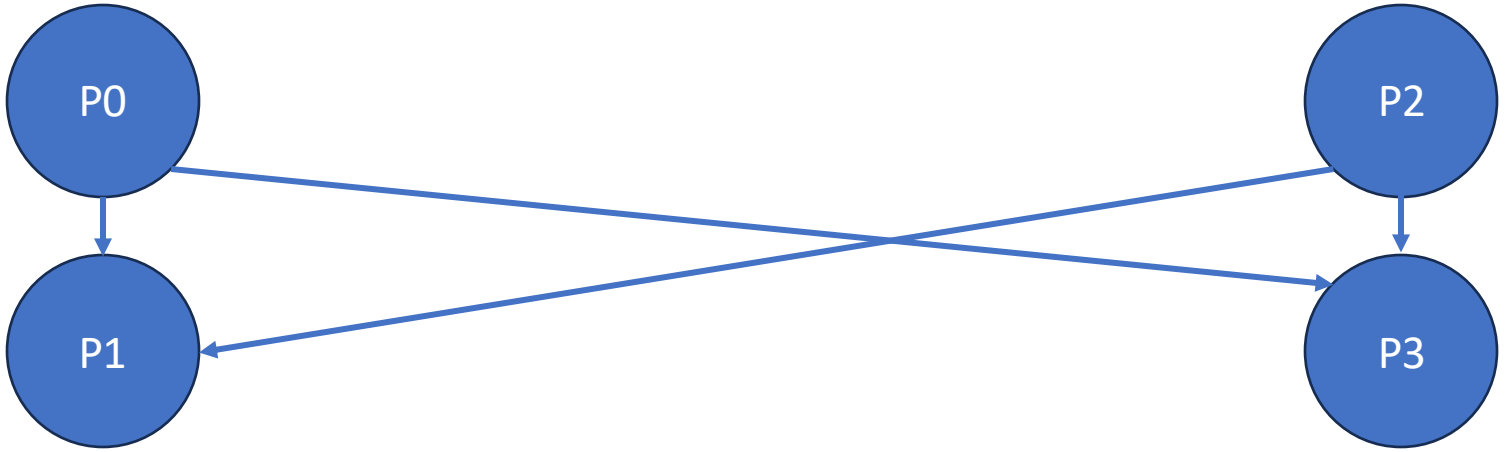
PC: Processor Consistency

- Writes from a single processor are received by all other processors in the order they were issued
- Writes from different processors may be seen in a different order by different processors
- Key idea:
  - reflect reality of networks
  - latency between nodes may be different



PC: Processor Consistency

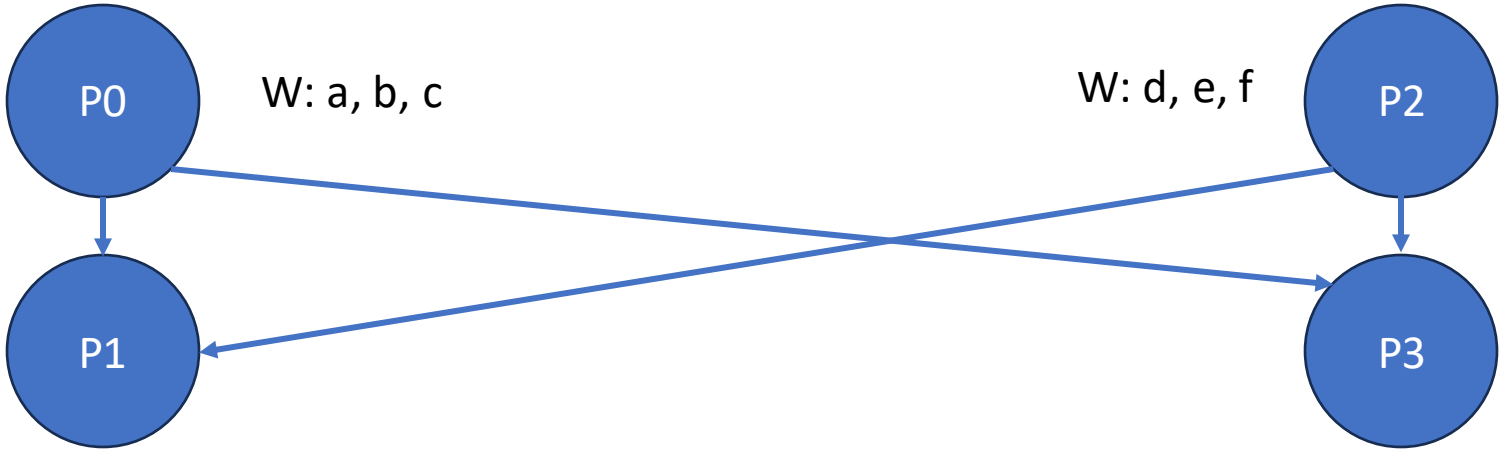
- Writes from a single processor are received by all other processors in the order they were issued
- Writes from different processors may be seen in a different order by different processors
- Key idea:
  - reflect reality of networks
  - latency between nodes may be different



1. P1 sees P0's writes in P0 order
2. P1 sees P2's writes in P2 order
3. Same for P3
4. P3 may see different interleavings of P0, P2 writes than P1 observes

PC: Processor Consistency

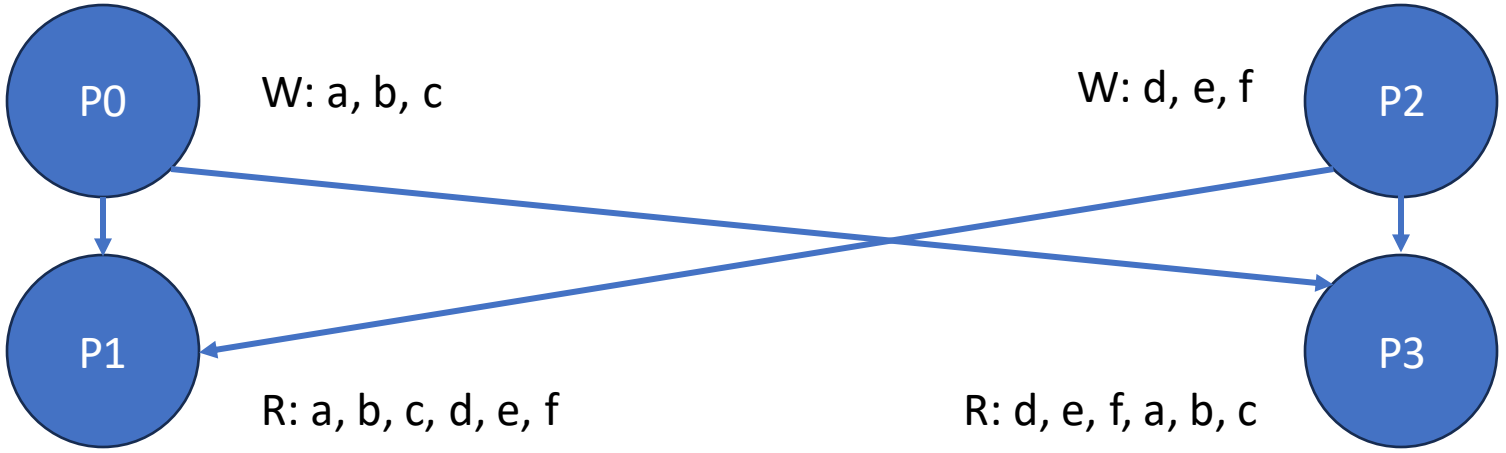
- Writes from a single processor are received by all other processors in the order they were issued
- Writes from different processors may be seen in a different order by different processors
- Key idea:
  - reflect reality of networks
  - latency between nodes may be different



1. P1 sees P0's writes in P0 order
2. P1 sees P2's writes in P2 order
3. Same for P3
4. P3 may see different interleavings of P0, P2 writes than P1 observes

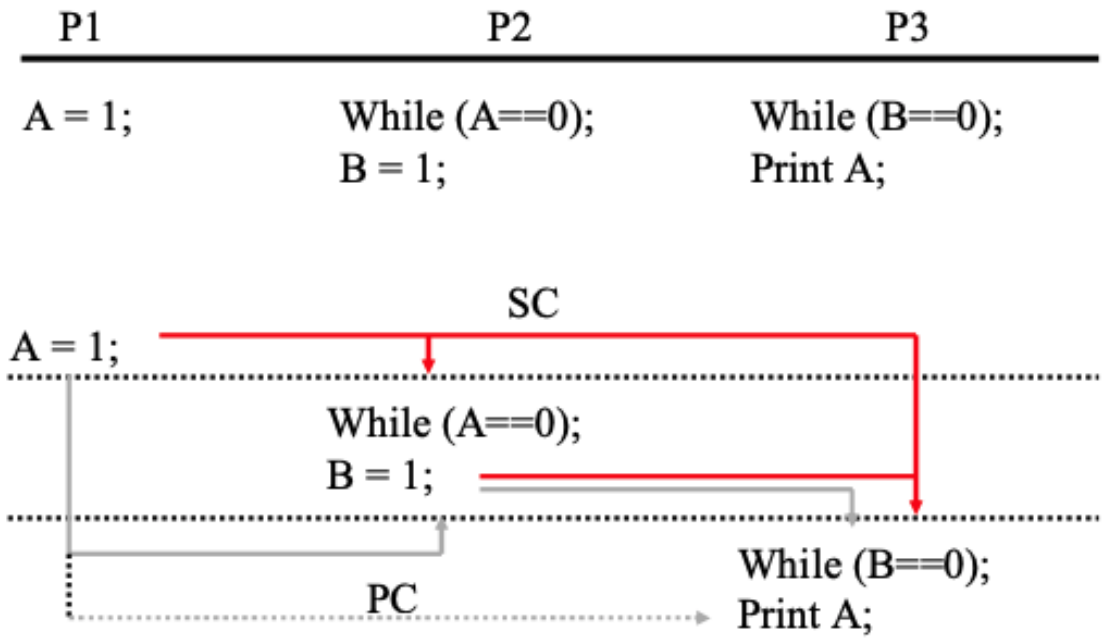
PC: Processor Consistency

- Writes from a single processor are received by all other processors in the order they were issued
- Writes from different processors may be seen in a different order by different processors
- Key idea:
  - reflect reality of networks
  - latency between nodes may be different

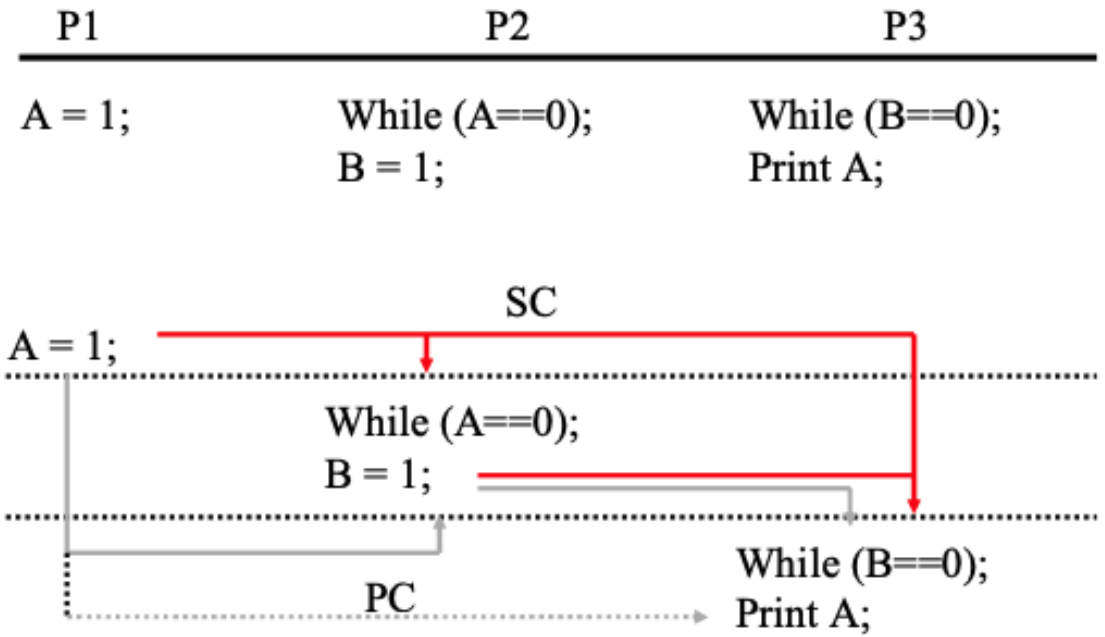


1. P1 sees P0's writes in P0 order
2. P1 sees P2's writes in P2 order
3. Same for P3
4. P3 may see different interleavings of P0, P2 writes than P1 observes

PC Example

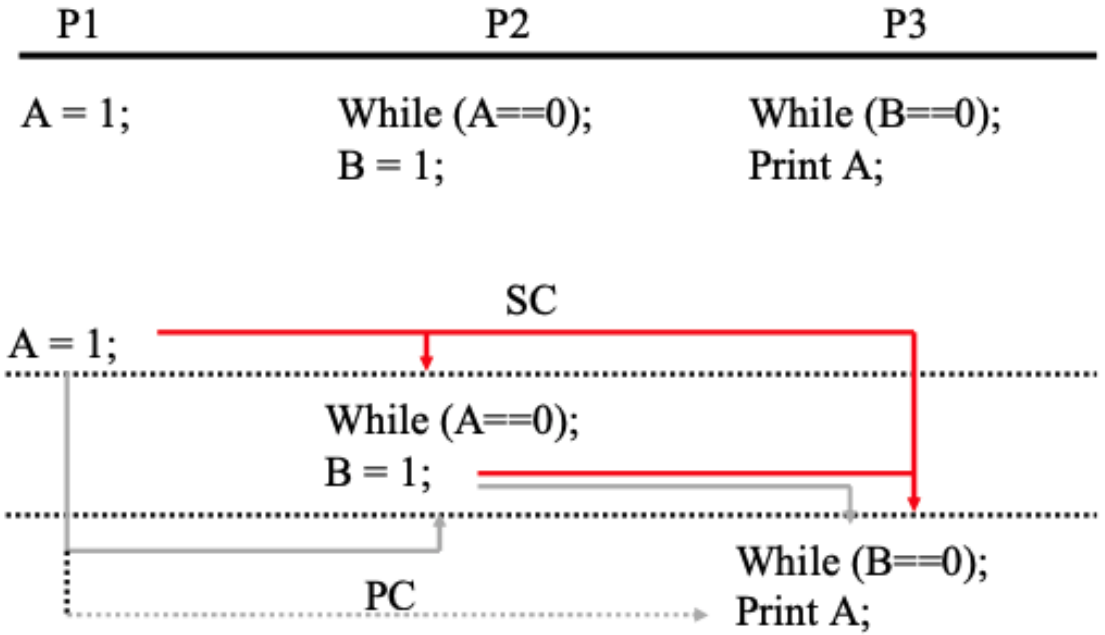

P1
P2
P3
A = 1;
While (A==0);
B = 1;
While (B==0);
Print A;
SC
A = 1;
While (A==0);
B = 1;
While (B==0);
Print A;
PC

PC Example



- How many different outputs from P3
  - For SC?
  - For PC?

PC Example



- How many different outputs from P3
  - For SC?
  - For PC?

PC Implementation:
- Store Queues Drain in Order
- Loads check Store Queue to "read own writes"

WO: Weak Ordering

WO: Weak Ordering

- Instructions are either “data” or “sync”

WO: Weak Ordering

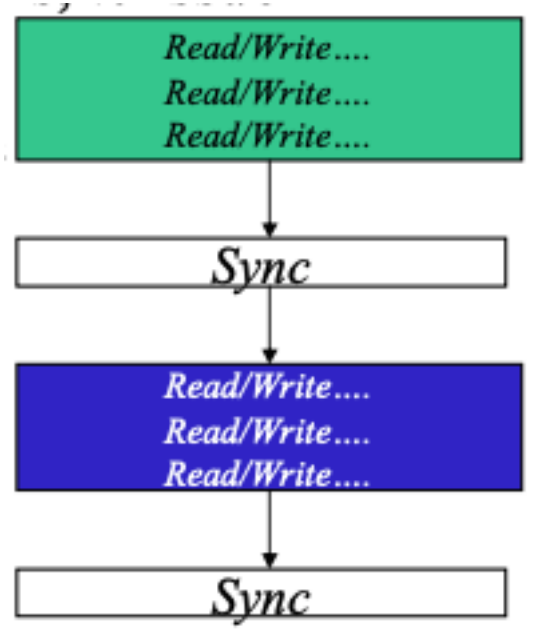
- Instructions are either “data” or “sync”
- reordering reads and writes between sync ops ok

WO: Weak Ordering

- Instructions are either “data” or “sync”
- reordering reads and writes between sync ops ok
- Sync ops must be SC

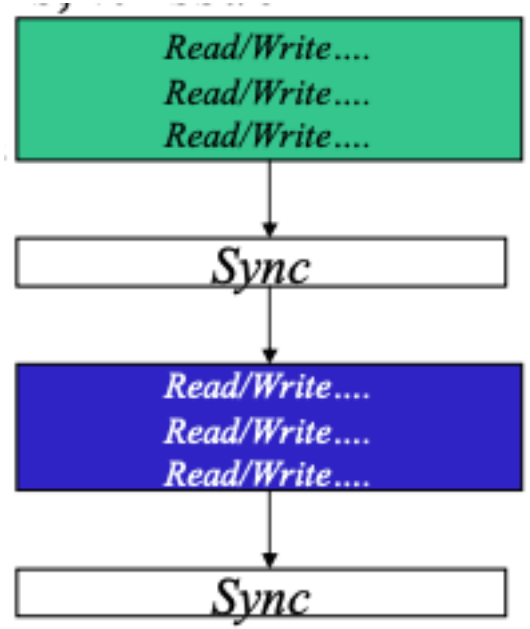
WO: Weak Ordering

- Instructions are either “data” or “sync”
- reordering reads and writes between sync ops ok
- Sync ops must be SC



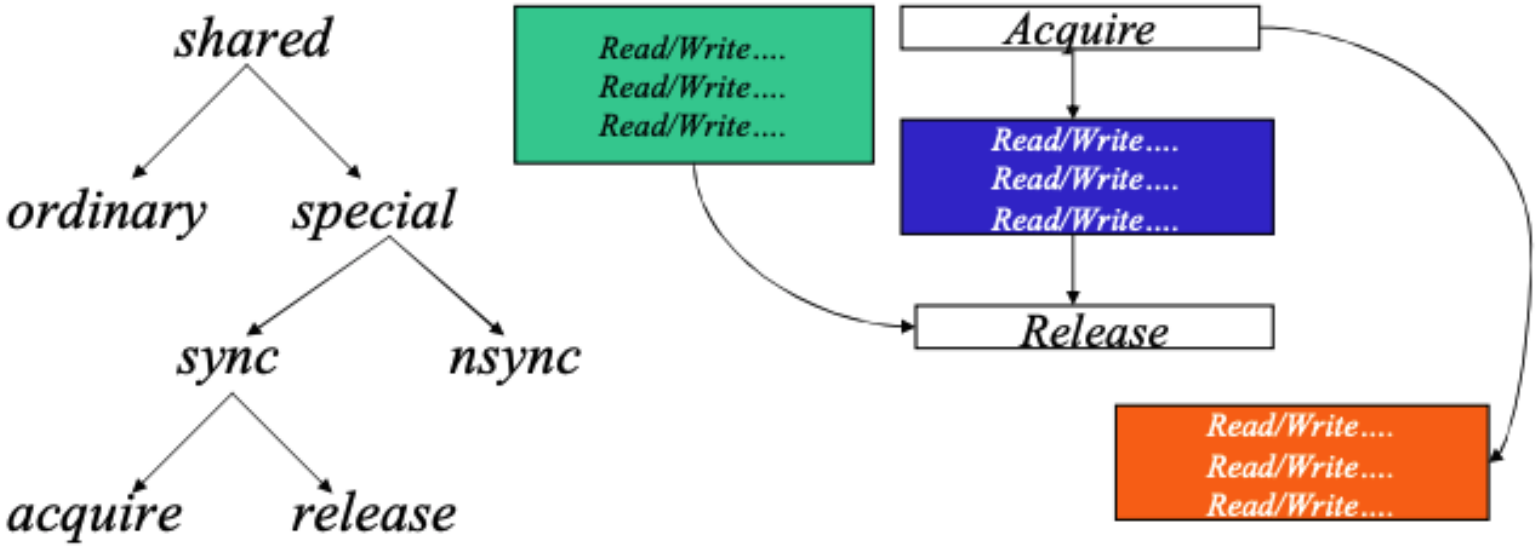
WO: Weak Ordering

- Instructions are either “data” or “sync”
- reordering reads and writes between sync ops ok
- Sync ops must be SC
- Implementation:
  - Use counters for outstanding ops
  - Counter must be zero for sync to issue
  - No ops can issue until previous sync retires



RC: Release Consistency

- Extends WO to richer taxonomy of sync and non-sync ops
- Two flavors:
  - RCsc → special operations must be SC
  - RCpc → special operations must be PC



Understanding How “Safety Nets” Work

- Post—wait synchronization

Initially, x = 0, y = 0, dataReady = 0

P0:		P1:	
1.	x = 5;	1.	while(!dataReady);
2.	dataReady = 1;	2.	y = x;

Understanding How “Safety Nets” Work

- Post—wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:                          P1:
1.    x = 5;                 1.    while(!dataReady);
2.    dataReady = 1;         2.    y = x;
```

- In SC, this “just works”
- In PC, this works for 2 processors
- In WO, RC, this requires fences

WO: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

P0:
1. x = 5;
2. dataReady = 1;

P1:
1. while(!dataReady);
2. y = x;

P0:
1. st &x, #5
2. st.**SYNC** &dataReady, 1

P1:
1. L: ld.**SYNC** R1, &dataReady
2. sub R1, #1
3. bnz R1, L
4. ld R2, &x

WO: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:                          P1:
1.    x = 5;                 1.    while(!dataReady);
2.    dataReady = 1;         2.    y = x;
```

```
P0:                          P1:
1.    st &x, #5              1.  L: ld.SYNC R1, &dataReady
2.    st.SYNC &dataReady, 1  2.     sub R1, #1
                             3.     bnz R1, L
                             4.    ld R2, &x
```

- SYNC is a fence:
  - all previous memory ops complete before SYNC
  - No subsequent memory ops issue until after SYNC

WO: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:                          P1:
1.    x = 5;                 1.    while(!dataReady);
2.    dataReady = 1;         2.    y = x;
```

```
P0:                          P1:
1.    st &x, #5              1.  L: ld.SYNC R1, &dataReady
2.    st.SYNC &dataReady, 1  2.     sub R1, #1
                             3.     bnz R1, L
                             4.    ld R2, &x
```

- SYNC is a fence:
  - all previous memory ops complete before SYNC
  - No subsequent memory ops issue until after SYNC

Does SYNC require communication with other processors?

WO: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:
1.      x = 5;
2.      dataReady = 1;
```

```
P1:
1.      while(!dataReady);
2.      y = x;
```

```
P0:
1.      st &x, #5
2.      st.SYNC &dataReady, 1
```

```
P1:
1.  L: ld.SYNC R1, &dataReady
2.     sub R1, #1
3.     bnz R1, L
4.   ld R2, &x
```

- SYNC is a fence:
  - all previous memory ops complete before SYNC
  - No subsequent memory ops issue until after SYNC

Does SYNC require communication with other processors?

No. SYNC ensures no one can see W(dataReady) -> W(x) by forcing st &x to complete before st &dataReady issues

RC: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:
1.      x = 5;
2.      dataReady = 1;
```

```
P1:
1.      while(!dataReady);
2.      y = x;
```

```
P0:
1.      st &x, #5
2.      st.rel &dataReady, 1
```

```
P1:
1.   L: ld.acq R1, &dataReady
2.      sub R1, #1
3.      bnz R1, L
4.    ld R2, &x
```

RC: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:
1.      x = 5;
2.      dataReady = 1;
```

```
P1:
1.      while(!dataReady);
2.      y = x;
```

```
P0:
1.      st &x, #5
2.      st.rel &dataReady, 1
```

```
P1:
1.  L: ld.acq R1, &dataReady
2.     sub R1, #1
3.     bnz R1, L
4.   ld R2, &x
```

- rel → all previous memory ops must complete before
- acq → no subsequent memory can ops issue until after

RC: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:                          P1:
1.    x = 5;                 1.    while(!dataReady);
2.    dataReady = 1;         2.    y = x;
```

```
P0:                          P1:
1.    st &x, #5              1.  L: ld.acq R1, &dataReady
2.    st.rel &dataReady, 1   2.     sub R1, #1
                             3.     bnz R1, L
                             4.    ld R2, &x
```

- rel → all previous memory ops must complete before
- acq → no subsequent memory can ops issue until after

Does acq/rel require communication with other processors?

RC: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:                          P1:
1.    x = 5;                 1.    while(!dataReady);
2.    dataReady = 1;         2.    y = x;
```

```
P0:                          P1:
1.    st &x, #5              1.  L: ld.acq R1, &dataReady
2.    st.rel &dataReady, 1   2.     sub R1, #1
                             3.     bnz R1, L
                             4.    ld R2, &x
```

- rel → all previous memory ops must complete before
- acq → no subsequent memory can ops issue until after

Does acq/rel require communication with other processors?

No. rel ensures no one can see W(dataReady) -> W(x)

RC: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:
1.      x = 5;
2.      dataReady = 1;
```

```
P1:
1.      while(!dataReady);
2.      y = x;
```

```
P0:
1.      st &x, #5
2.      st.rel &dataReady, 1
```

```
P1:
1.  L: ld.acq R1, &dataReady
2.     sub R1, #1
3.     bnz R1, L
4.   ld R2, &x
```

- rel → all previous memory ops must complete before
- acq → no subsequent memory can ops issue until after

Does acq/rel require communication with other processors?

No. rel ensures no one can see W(dataReady) -> W(x)

Why do we need ld.acq on P1.1?

RC: Post-wait synchronization

Initially, x = 0, y = 0, dataReady = 0

```
P0:                          P1:
1.    x = 5;                 1.    while(!dataReady);
2.    dataReady = 1;         2.    y = x;
```

```
P0:                          P1:
1.    st &x, #5              1.  L: ld.acq R1, &dataReady
2.    st.rel &dataReady, 1   2.     sub R1, #1
                             3.     bnz R1, L
                             4.    ld R2, &x
```

- rel → all previous memory ops must complete before
- acq → no subsequent memory can ops issue until after

Does acq/rel require communication with other processors?

No. rel ensures no one can see W(dataReady) -> W(x)

Why do we need ld.acq on P1.1?

So that P1.4 can't execute before P1.1 completes

Comparing Safety Net Usage

IBM PowerPC style

Post synchronization code

```
datum = 5;
lwsync
datumIsReady = 1;
```

Wait synchronization code

```
while (!datumIsReady) {};
isync
... = datum;
```

Lock release code

```
#end of critical section
lwsync      #full fence
stw r4,r3   #write 0 in r4 to
            #lock address in r3
```

LL/SC lock acquisition code

```
loop: lwarx  r6,0,r3 #load linked
      cmpw   r4,r6   #is lock free?
      bne-   wait    #go to wait if not free
      stwcx. loop    #store conditional
      bne-   loop    #if SC fails, repeat
      isync          #acquire fence
      # begin critical section

wait: ...            # wait until lock is free
```

IA-64 style

Post synchronization code

```
// suppose R1=5, R2=1

st &datum, R1
st.rel &datumIsready,R2
```

Wait synchronization code

```
wait: ld.acq R1, &datumIsReady
      sub R2, R1, #1
      biz R2, wait
      ld R3, &datum
```

Exercise: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);
```

```
Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

Exercise: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);
```

```
Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

1. Q_head is last write in Q_put, so Q_get never gets “ahead”.
2. *single* p,c only (as advertised)
3. Requires ??? before setting Q head
4. Devil in the details of “wait”
5. No lock → “optimistic”

Questions?