So, lets try our hand at some synchronization

What is synchronization?

- Making sure that concurrent activities don’t access shared data in inconsistent ways

- int i = 0; // shared

```
 Thread A                  Thread B
  i=i+1;                    i=i-1;
```

What is in i?

What are the sources of concurrency?

- Multiple user-space processes
  - On multiple CPUs
- Device interrupts
- Workqueues
- Tasklets
- Timers

Pitfalls in `scull`

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Scull is the Simple Character Utility for Locality Loading (an example device driver from the Linux Device Driver book)

Pitfalls in `scull`

- *Race condition*: result of uncontrolled access to shared data

```
⇉if (!dptr->data[s_pos]) {
   dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
   if (!dptr->data[s_pos]) {
     goto out;
   }
 }
```

Pitfalls in `scull`

- *Race condition*: result of uncontrolled access to shared data

```
→if (!dptr->data[s_pos]) {
→   dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
       goto out;
    }
 }
```

Pitfalls in `scull`

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Memory leak

Synchronization primitives

- Lock/Mutex
  - To protect a shared variable, surround it with a lock (critical region)
  - Only one thread can get the lock at a time
  - Provides mutual exclusion
- Shared locks
  - More than one thread allowed (hmm…)
- Others? Yes, including Barriers (discussed in the paper)

Synchronization primitives (cont’d)

- Lock based
  - Blocking (e.g., semaphores, futexes, completions)
  - Non-blocking (e.g., spin-lock, …)
    - Sometimes we have to use spinlocks
- Lock free (or partially lock free ☺)
  - Atomic instructions
  - seqLocks
  - RCU
  - Transactions

How about locks?

- Lock(L): Unlock(L):

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

```
L=0;
```

Check and lock are not atomic!

Can we do this just with atomic reads and writes?

Yes but not easy—Decker's algorithm
Easier to use read-modify-update atomic instructions

Naïve implementation of spinlock

- Lock(L):

```
While(test_and_set(L));
//we have the lock!
//eat, dance and be merry
```

- Unlock(L)

```
L=0;
```

Why naïve?

- Works? Yes, but not used in practice
- Contention
  - Think about the cache coherence protocol
  - Set in test and set is a write operation
    - Has to go to memory
    - A lot of cache coherence traffic
    - Unnecessary unless the lock has been released
    - Imagine if many threads are waiting to get the lock
- Fairness/starvation

Better implementation: Spin on read

- Assumption: We have cache coherence
  - Not all are: e.g., Intel SCC
- Lock(L):

```
while(L==locked); //wait
 if(test_and_set(L)==locked) go back;
```

- Still a lot of chattering when there is an unlock
  - Spin lock with backoff

Bakery Algorithm

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

- Acquire_lock:

```
int my_ticket = fetch_and_inc(L->next_ticket);
while(L->new_serving!=my_ticket); //wait
//Eat, Dance and me merry!
```

Still too much chatter

- Release_lock:

```
L->now_serving++;
```

Comments? Fairness? Efficiency/cache coherence?

Anderson Lock (Array lock)

- Problem with bakery algorithm:
  - All threads listening to next_serving
    - A lot of cache coherence chatter
  - But only one will actually acquire the lock
  - Can we have each thread wait on a different variable to reduce chatter?

Anderson's Lock

- We have an array (actually circular queue) of variables
  - Each variable can indicate either lock available or waiting for lock
    - Only one location has lock available

```
Lock(L):
    my_place = fetch_and_inc (queuelast);
    while (flags[myplace mod N] == must_wait);
Unlock(L)
    flags[myplace mod N] = must_wait;
    flags[mypalce+1 mod N] = available;
```

Fair and not noisy – compare to spin-on-read and bakery algorithm
Any negative side effects?

Concurrency and Memory Consistency

References:

- **Shared Memory Consistency Models: A Tutorial**, Sarita V. Adve & Kourosh Gharachorloo, September 1995
- **A primer on memory consistency and cache coherence**, Sorin, Hill and wood, 2011 (chapters 3 and 4)
- **Memory Models: A Case for Rethinking Parallel Languages and Hardware**, Adve and Boehm, 2010

Memory Consistency

- Formal specification of memory semantics
- Guarantees as to how shared memory will behave on systems with multiple processors
- Ordering of reads and writes
- Essential for programmer (OS writer!) to understand

Why Bother?

- Memory consistency models affect *everything*
  - Programmability
  - Performance
  - Portability
- Model must be defined at all levels
- Programmers and system designers care

Uniprocessor Systems

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write
  - Only matters if location is the same or dependent
  - Many possible optimizations

- **Intuitive!**

How does a core reorder? (1)

- Store-store reordering:
  - Non-FIFO write buffer
- Load-load or load-store/store-load reordering:
  - Out of order execution

- Should the hardware prevent any of this behavior?

Multiprocessor: Example

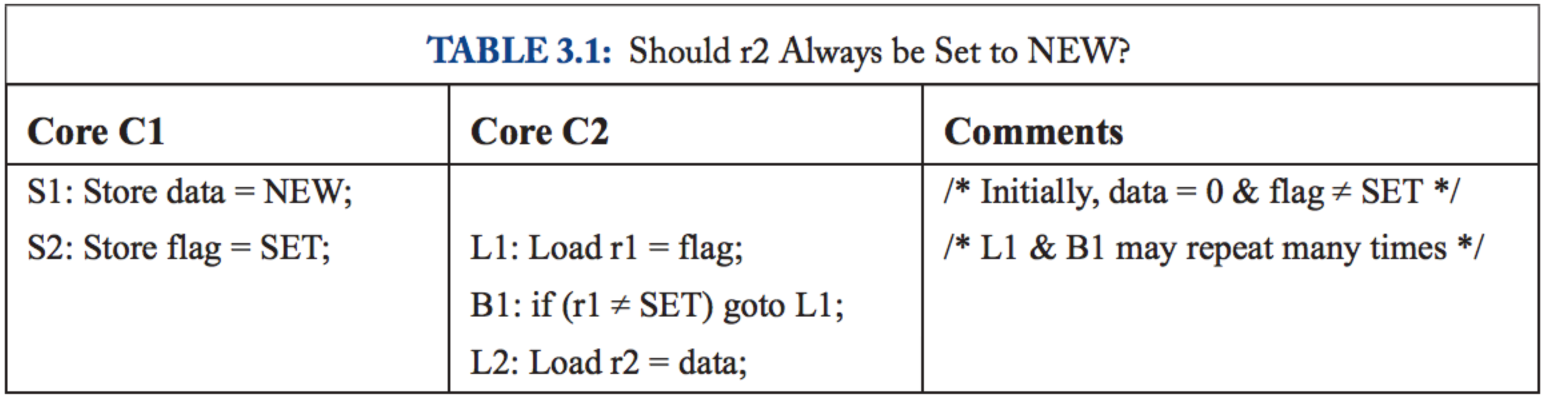


TABLE 3.1: Should r2 Always be Set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

Cont'd

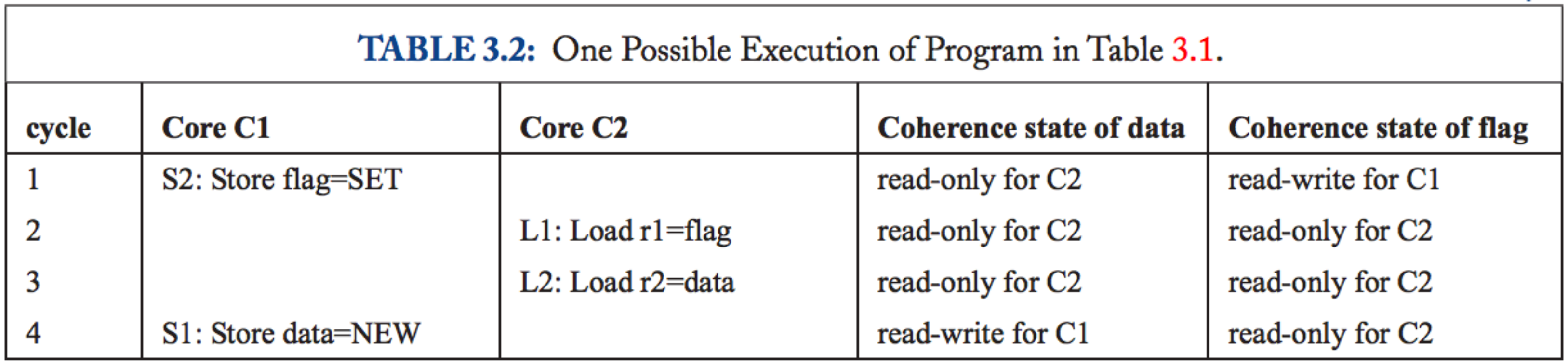


TABLE 3.2: One Possible Execution of Program in Table 3.1.

cycle	Core C1	Core C2	Coherence state of data	Coherence state of flag
1	S2: Store flag=SET		read-only for C2	read-write for C1
2		L1: Load r1=flag	read-only for C2	read-only for C2
3		L2: Load r2=data	read-only for C2	read-only for C2
4	S1: Store data=NEW		read-write for C1	read-only for C2

- S2 and S1 reordered
  - Why? How?

Example 2

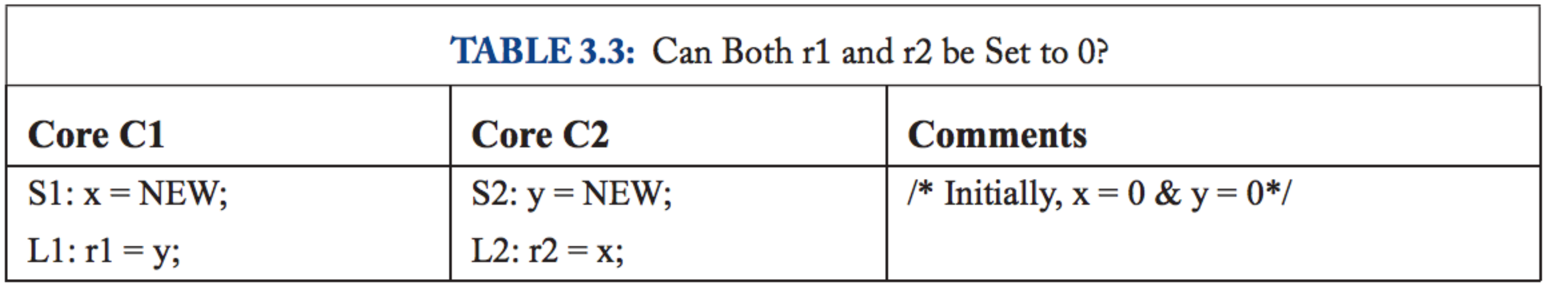
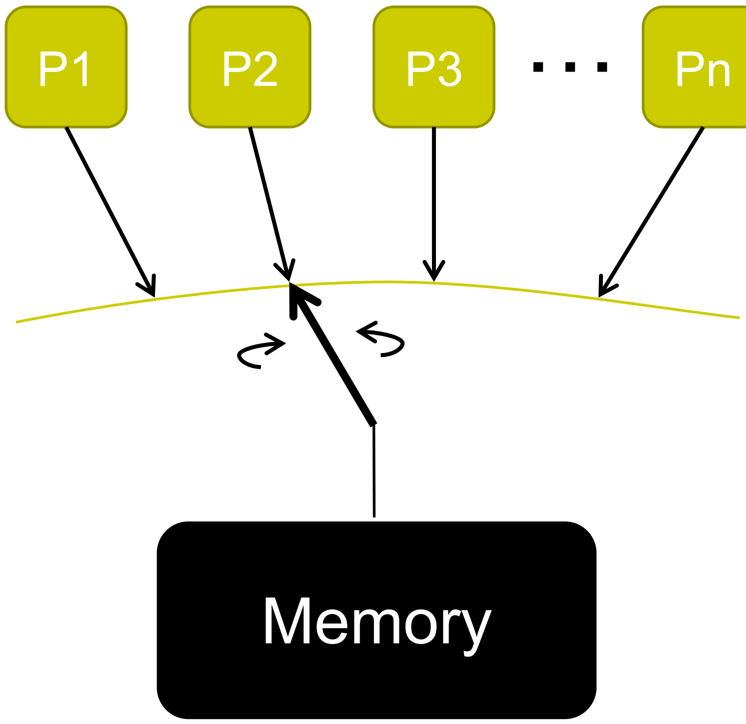


TABLE 3.3: Can Both r1 and r2 be Set to 0?

Core C1	Core C2	Comments
S1: x = NEW;	S2: y = NEW;	/* Initially, x = 0 & y = 0*/
L1: r1 = y;	L2: r2 = x;	

Sequential Consistency

- The result of any execution is the same as if all operations were executed on a single processor
- Operations on each processor occur in the sequence specified by the executing program



One execution sequence

TABLE 3.1: Should r2 Always be Set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

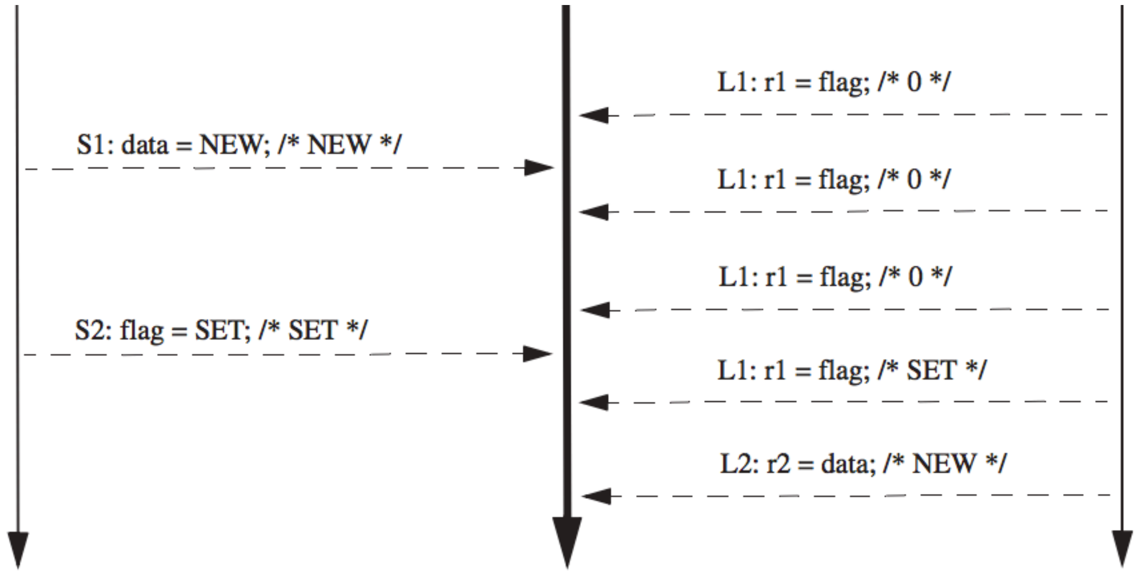
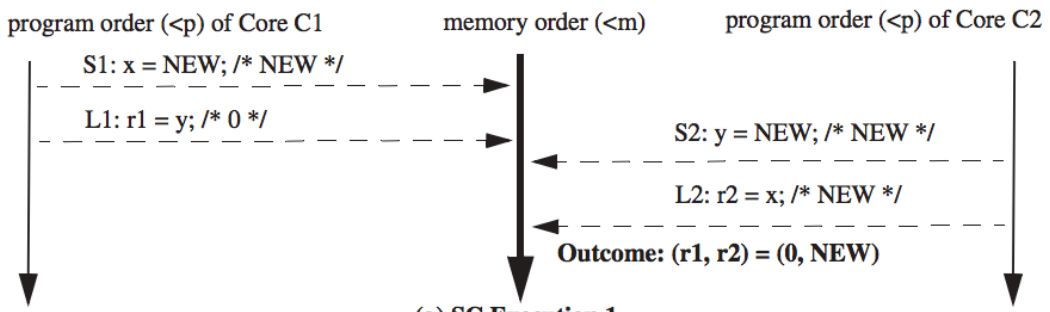
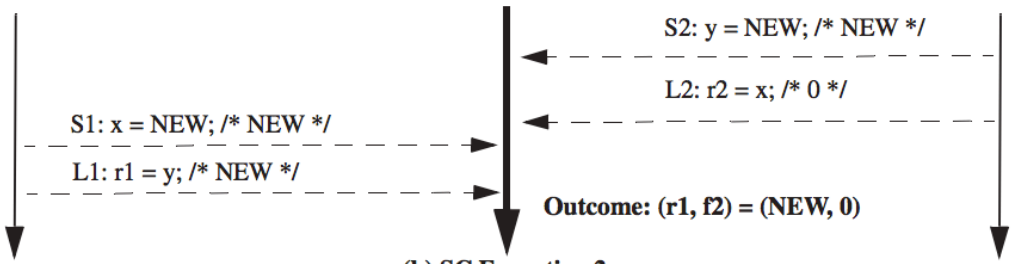


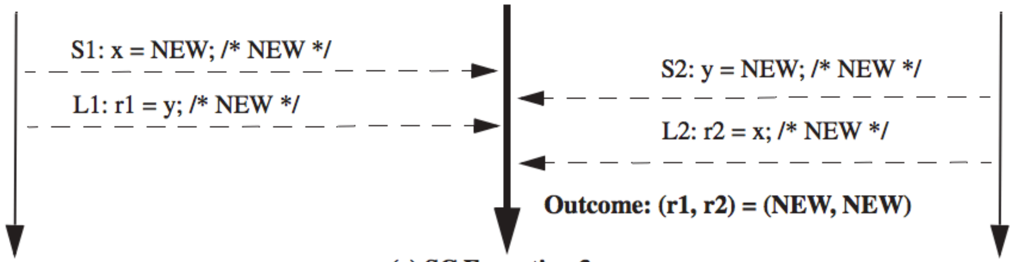
FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.



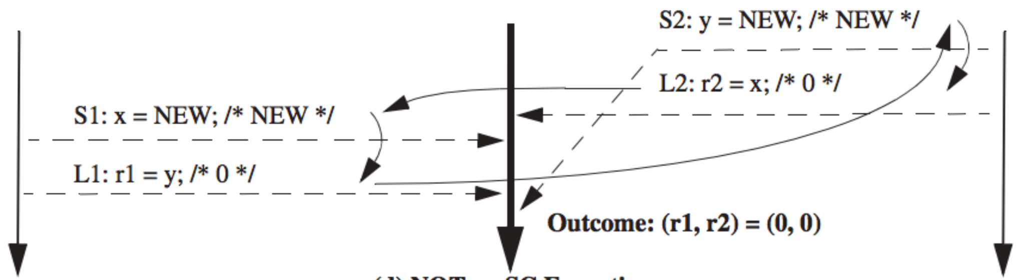
(a) SC Execution 1



(b) SC Execution 2



(c) SC Execution 3



(d) NOT an SC Execution

S.C. Disadvantages

- Difficult to implement!
- Huge lost potential for optimizations
  - Hardware (cache) and software (compiler)
  - Be conservative: err on the safe side
  - Major performance hit

Relaxed Consistency

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's Write early
- Combined relaxations
  - Read your own Write *(okay for S.C.)*
- *Safety Net* – available synchronization operations
- *Note: assume one thread per core*

Synchronization is broken!

- How can we solve this problem?

- Answer: Memory Barrier/Fence
  - A special complier or CPU instruction that enforces an ordering constraint
  - Compiler: asm volatile ("" ::: "memory");
  - CPU: mfence/lfence