Department of Computer Science, Institute of System Architecture, Operating Systems Group

Distributed Operating Systems

Exercise 5: MEMORY CONSISTENCY, CACHE COHERENCE AND LOCKS

In the tutorial, all solutions will be presented by students. Please be prepared for all questions as the exercise will focus on discussion, not on understanding the question and gathering the knowledge.

Memory Consistency

1) **Sequential Consistency**
   In a system with sequential consistency each processor always executes memory operations in the order specified by its program (program order). The order in which the individual memory operations of each processor become visible to the other processors on the shared interconnect (e.g., the bus) is called visibility order.
   Three processors (*P1*, *P2* and *P3*) in a shared-memory system execute the following code (initially *A* = *B* = *0*).

P1	**P2**	**P3**
a1: A := 1	a2: u := A	a3: v := B
	b2: B := 1	b3: w := A

Here a1 denotes the first operation of processor *P1*, *a2* denotes the first operation of *P2* and *b2* denotes the second operation of *P2*, etc.

The outcome of the execution, denoted by the tuple *(u,v,w)*, may vary depending on the order in which the individual operations of each processor become globally visible. Some outcomes may not be possible on a sequentially consistent system. Complete the following table with the possible results for *(u,v,w)*. For each row describe if the result is sequentially consistent and if so, specify a visibility order that produces the result. An example is given in the second row.

u	v	w	sequential consistent	visibility order
0	0	0		
0	0	1	yes	a2, a3, a1, b3, b2
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

2) **Relaxed Consistency: Peterson-Algorithm**
A well-known algorithm for mutual exclusion is Peterson's algorithm (shown in pseudo-code below). Explain why Peterson's algorithm does not break on machines with a store buffer where reads are not permitted to bypass writes to the same memory location and why it does break if reads are permitted to bypass writes to the same memory location on systems with store forwarding (e.g., SPARC TSO).

```
// global variables and initial values
bool flag0 = false, flag1 = false;
int turn = 0;

// Process on CPU0
flag0 = true;
turn = 1;
while (turn == 1 && flag1);
// critical section
flag0 = false;

// Process on CPU1
flag1 = true;
turn = 0;
while (turn == 0 && flag0);
// critical section
flag1 = false;
```

3) **Fence Instructions**
Machines with relaxed memory consistency typically provide programmers with fence instructions to tighten the ordering of memory instructions. Insert *MFENCE* (memory fence) instructions in Dekker's and Peterson's algorithms to ensure their correct behavior on a multi-processor system that implements a store buffer with store forwarding. Use as few fence instructions as necessary.

Cache Coherency

Multiprocessor systems with caches use a coherency protocol, which ensures that writes by one processor eventually become visible to all other processors and that no two processors write to the same memory location simultaneously. One invalidation-based protocol discussed in the lecture is the MESI protocol.

Two processors (*P1* and *P2*) and uniform memory are connected to a shared bus, which implements the MESI cache coherency protocol. In memory there exists a data structure with the following layout: *A* (8 bytes), *B* (24 bytes), *C* (8 bytes). The cacheline size is 32 bytes, so that *A* and *B* reside in cacheline *X*, whereas *C* resides in cacheline *Y*.

Processor *P1 and P2* execute the following code:

```
// Processor P1
read(A);
read(B);
// do something with A
read(B);
write(A);

// Processor P2
read(B);
read(C);
// do something with C
read(B);
write(C);
```

4) **Cache Coherency in Action**
The following table lists the memory operations of the individual processors as they appear on the shared bus. The first column lists the processor and the second column specifies the memory operation being carried out. The next two columns list the MESI state of the cachelines *X* and *Y* in each of the processors. The last two columns describe if the operation caused a cacheline transfer to/from memory or to/from another cache. Initially all cachelines are invalid *(I)*.

CPU	Operation	P1	P2	Memory Transfer	Cache Transfer
		I I	I I		
1	read(A)				
1	read(B)				
2	read(B)				
2	read(C)				
1	read(B)				
1	write(A)				
2	read(B)				
2	write(C)				

5) **False Sharing**
Because *B* is contained in cacheline *X*, false sharing occurs with *A*. Discuss how to remedy this problem and explain the impact on memory and cache transfers.

Cache Coherency and Locks

The cache-coherency protocol is sometimes crucial for the scaling of a particular lock implementation. The reason is, that some lock implementations produce too many bus messages and thereby slow down the execution of the processor.

6) **Test-and-Set Locks**
The Test-and-Set locks presented within the lecture do not scale well since they perform exclusive write operations to the lock variable even if the lock is currently taken by a different thread.

A possible implementation of a simple Test-and-Set locks can be done as follows:

```
class TSLock {
    private:
     unsigned int _lock

    public:
     TSLock() : _lock(0) { }

     void lock() {
         while (test_and_set(_lock) == 1) { }
     }

     void unlock() {
         _lock = 0;
     }
 };
```

Check the scalability problem of the Test-and-Set locks by completing the below-mentioned table.

The first column of the table states which CPU is executing an instruction at the moment. The second column contains the executed operation. The state of the lock variable should be written in column three. The remaining four columns are supposed to contain the state of the cache line containing the lock variable on the corresponding core according to the MESI cache coherency protocol.

CPU	Operation	Lock	P1	P2	P3	P4
		0	I	I	I	I
1	lock (T&S)	1	M	I	I	I
2	wait (T&S)	1	I	M	I	I
4	wait (T&S)	1	I	I	I	M
2	wait (T&S)					
3	wait (T&S)					
4	wait (T&S)					
3	wait (T&S)					
1	unlock					
3	lock (T&S)					
2	wait (T&S)					
4	wait (T&S)					
3	unlock					
2	lock (T&S)					
4	wait (T&S)					
4	wait (T&S)					
2	unlock					

7) **Test-Test-and-Set Locks**
Within the lecture, an extension of the simple Test-and-Set lock was presented. This new lock implementation contains an additional Test-operation before the actual taking of the lock is performed. A possible implementation of a Test-Test-and-Set lock could look as follows:

```
class TTSLock {
  private:
    unsigned int _lock;

  public:
    TTSLock() : _lock(0) { }

    void lock() {
        while (true) {
            while (_lock == 1) { }
            if (test_and_set(_lock) == 0)
                break;
         }
    }

    void unlock() { _lock = 0; }
};
```

Repeat the previous exercise and use a Test-Test-and-Set lock instead of only a Test-and-Set Lock. Keep in mind that the Test-operation of the lock is only a read access and not a write access.

CPU	**Operation**	**Lock**	**P1**	**P2**	**P3**	**P4**
		0	I	I	I	I
1	test	0	E	I	I	I
2	test	0	S	S	I	I
1	lock (T&S)	1	M	I	I	I
2	lock (T&S)					
3	wait					
4	wait					
2	wait					
3	wait					
4	wait					
1	unlock					
3	test					
3	lock (T&S)					
2	wait					
4	wait					
3	unlock					
2	test					
4	test					
2	lock (T&S)					
4	lock (T&S)					
4	wait					
4	wait					
2	unlock					

8) **False Sharing and Locks**
Earlier within this exercise session there was one exercise about false sharing. Discuss the consequences if two independent lock variables *l1* and *l2* share the same cache line.

Extra Tasks

The following tasks will not be discussed during the lecture but are there for more advanced and interested students to broaden their knowledge.

9) **Lock Free**
   Implement double-address compare-and-swap with the help of compare-and-swap.

10) **Reader-Writer-Lock**
   Implement a reader writer lock using the atomic read-modify-write instructions presented in the lecture. Make sure the lock is fair, that is, both readers and writers get the lock after a bounded time.

11) **Fair Fast Scalable Reader-Writer**
   Lock In their paper "A Fair Fast Scalable Reader-Writer Lock", Krieger et al. present a scalable reader-writer lock implementation based on the MCS lock discussed in the lecture.

   a) Search for typos in the readerUnlock function and correct them. Justify why there is a bug. If you do not spot any errors, justify why the function is correct.

   b) Describe how the readerUnlock function works.

   c) The reader-writer lock implementation is fair, though inherently unfair spin-lock implementations are used in the readerUnlock function. Why?

Material

- Orran Krieger, Michael Stumm, Ron Unrau, Jonathan Hanna: "A Fair Fast Scalable Reader-Writer Lock", International Conference on Parallel Processing (ICPP), 1993
- Mellor-Crummey Scott: "Algorithms for scalable synchronization on shared memory multiprocessors", ACM Transactions on Computer Systems, Volume 9, 1991