Solutions to the multiprocessor exercises

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1 Exam exercises

Exercise 1 *Exam, June 2015*

Given a computer with two processors, each with its private cache that use write-back policy to main memory. Initially all entries of the caches are invalid. The memory position corresponding to variable **A**, initially contains the value **255**.

The following sequence of operations is executed on this computer:

Time	Processor 1	Processor 2
1	**lw $t0, A**	
2		**lw $t1, A**
3	**sw $zero, A**	
4		**lw $t2, A**

Show for each of these operations the status and value of the memory address **A** in both caches and main memory.

Time	Cache state 1	Cache value 1	Cache state 2	Cache value 2	Memory value
0	I	–	I	–	255
1					
2					
3					
4					

Solution 1

Table 1 shows the states and values:

Exercise 2 *January 2015 exam*

Given the following code fragments that execute 3 threads using the MSI consistency protocol.

```
// Thread 0 code
for(i=0;i<16;i++){
  a[i]= 16;
}
```

Time	Cache state 1	Cache value 1	Cache state 2	Cache value 2	Memory value
0	I	–	I	–	255
1	Shared	255	Invalid	–	255
2	Shared	255	Shared	255	255
3	Modified	0	Invalid	255	255
4	Shared	0	Shared	0	0

Table 1: Solution to exercise 1.

```
// Thread 1 code
tmp=0;
for(i=0;i<16;i++){
  a[i]=tmp;
  tmp+=a[i];
}

// Thread 2 code
cnt=0;
for(i=0;i<4;i++){
  cnt+=b[i];
}
```

The computer has a CC-NUMA architecture with:

- 2 32-bit processors with only cache memory level It is private to each processor and has a direct-mapped correspondence. The cache line size is 16 bytes.
- The cache memories are initially empty.
- Threads 0 and 2 are executed in the processor 0 and thread 1 runs on processor 1.
- The variables **i**, **tmp** and **cnt** are stored in registers (they are not stored in memory).
- The block containing **a[0]** is associated with the same line cache the block that contains **b[0]**.

Complete the following tables and justify the answer for the following sections:

1. Starting from the initial situation, indicate in the following table the state transitions of the block that stores **a[0]** when the thread 0 is executed before the thread 1. Include the bus traffic associated with each thread.
   - **Note**: if there are several transitions associated with a thread, indicate all of them.

Code	Transition P0	Transition P1	Bus signals
Thread 0			
Thread 1			

2. Starting from the initial situation, indicate in the following table the state transitions of the block that stores **a[0]** when the thread 1 is executed before the thread 0. Include the bus traffic associated with each thread.
   - **Note**: if there are several transitions associated with a thread, indicate all of them.

Code	Transition P0	Transition P1	Bus signals
Thread 0			
Thread 1			

3. Starting from the initial situation, indicate in the following table the state transitions of the block that stores **a[0]** when thread 0 is executed, then thread 2 and finally thread 1. Include the bus traffic associated with each thread.

   - **Note**: if there are several transitions associated with a thread, indicate all of them.

Code	Transition P0	Transition P1	Bus signals
Thread 0			
Thread 1			
Thread 2			

4. For each of the above scenarios, indicate the number of existing cache misses for each process.

Scenario	P0 cache misses	P1 cache misses
Thread 0 → Thread 1		
Thread 1 → Thread 0		
Thread 0 → Thread 2 → Thread 1		

Solution 2

Section 1 Table 2 shows the transitions.

Code	Transition P0	Transition P1	Bus signals
Thread 0	I → E	I	Write miss
Thread 1	E → I	I → E , E → E	Write miss; Write-back block

Table 2: Transtions for section 1, exercise 2

Section 2 Table 3 shows the transitions.

Code	Transition P0	Transition P1	Bus signals
Thread 1	I	I → E ; E → E	Write miss
Thread 0	I → E	E → I	Write miss; Write-back block

Table 3: Transtions for section 2, exercise 2

Section 3 Table 4 shows the transitions.

Each block has 4 words. So the Threads 0 and 1 access to 4 blocks and while Thread 2 is accessing to the block in Thread 1 the first line causes a cache miss while the second, a cache hit.

Section 4 Table 5 shows the transitions.

Exercise 3 *January 2014 exam*

Code	Transition P0	Transition P1	Bus signals
Thread 0	I → E (stores **a[0]**)	I	Write miss
Thread 2	E → S (stores **b[0]**)	I	Read miss; write back block
Thread 1	S → S (**b[0]**)	I → E ; E → E	Write miss

Table 4: Transtions for section 3, exercise 2

Escenario	Cache misses P0	Cache misses P1
Thread 0 → Thread 1	4	4
Thread 1 → Thread 0	4	4
Thread 0 → Thread 2 -> Thread 1	4 + 1	4

Table 5: Transtions for section 4, exercise 2

Given a multiprocessor with a symmetric shared memory architecture based on bus with protocol of snooping. Each processor has a cache whose consistency is maintained using the MSI protocol. Each cache uses direct-mapped correspondence and has four blocks each with two words. This cache uses the entire memory address as the label field.

The following tables show the status of each cache memory, with the word less significant on the left.

Processor P0			
Block	**State**	**State**	**Data**
B0	I	0x00100700	0x00000000 0x7FAABB11
B1	S	0x00100708	0x00000000 0x00001234
B2	M	0x00100710	0x00000000 0x0077AABB
B3	I	0x00100718	0x00000000 0x7FAABB11

Processor P1			
Block	**State**	**State**	**Data**
B0	I	0x00100700	0x00000000 0x7FAABB11
B1	M	0x00100728	0x00000000 0xFF000000
B2	I	0x00100710	0x00000000 0xEEEE7777
B3	S	0x00100718	0x00000000 0x7FAABB11

Processor P2			
Block	**State**	**State**	**Data**
B0	S	0x00100720	0x00000000 0x1111AAAA
B1	S	0x00100708	0x00000000 0X00001234
B2	I	0x00100710	0x00000000 0x7FAABB11
B3	I	0x00100718	0x00001234 0x1111AABB

For each of the cases that are presented below, start with the initial situation of the problem, without taking into account changes in previous cases. Indicate the changes that occur in the caches. In the case of readings, also indicate what is the value actually read.

1. P2: write 0x00100708, 0xFFFFFFFF
2. P2: read 0x00100708
3. P2: read 0x00100718

In each case you must fill in a table with the following format, justifying the answer:

Processor	Block	State previous	State Current	State	Data	

Solution 3

Section 1 A write occurs in block B1 of the P2 cache that is in state S. This produces the following changes in P2:

- Transition to modified state (M).
- A invalidation is placed on the bus

The invalidation is ignored by the processor P1, but is treated by the P0 processor.
In P0 the following changes occur:

- Transition to invalid state (I).

In this case there are no changes in the main memory.
The states in the different processors are shown in the Table 6.

Procesador P0				
Block	State	Label	Data	
B0	I	0x00100700	0x00000000	0x7FAABB11
B1	I	0x00100708	0x00000000	0x00001234
B2	M	0x00100710	0x00000000	0x0077AABB
B3	I	0x00100718	0x00000000	0x7FAABB11
Procesador P1				
Block	State	Label	Data	
B0	I	0x00100700	0x00000000	0x7FAABB11
B1	M	0x00100728	0x00000000	0xFF000000
B2	I	0x00100710	0x00000000	0xEEEE7777
B3	S	0x00100718	0x00000000	0x7FAABB11
Procesador P2				
Block	State	Label	Data	
B0	S	0x00100720	0x00000000	0x1111AAAA
B1	M	0x00100708	0xFFFFFFFF	0X00001234
B2	I	0x00100710	0x00000000	0x7FAABB11
B3	I	0x00100718	0x00001234	0x1111AABB

Table 6: Cache contents for Section 1, exercise 3.

The states in the different processors are shown in the Table 7.

Section 2 A read is perfomed in P2's cache block B1 that is in shared state. It is a cache hit the returns the value 0x00000000

Procesador	Block	State Previous state	New state	Label	Data	
P2	B1	I	M	0X00100708	0XFFFFFFFF	0X00001234
P0	B1	S	I	0X00100708	0X00000000	0X00001234

Table 7: State transitions for section 1, exercise 3.

Section 3 A read miss is produced in P2's cache for block B3 that is invalid (I). The read miss is placed on the bus, the block is adquired and the state changes to shared (S).

The processor P0, has this block in invalid State and ignores the read miss.

The processor P1, has this Block in shared State (S) and it continues in this State.

The value read is 0x00000000

The States in the different processors are shown in the Table 8.

Procesador P0				
Block	State	Label	Data	
B0	I	0x00100700	0x00000000	0x7FAABB11
B1	S	0x00100708	0x00000000	0x00001234
B2	M	0x00100710	0x00000000	0x0077AABB
B3	I	0x00100718	0x00000000	0x7FAABB11
Procesador P1				
Block	State	Label	Data	
B0	I	0x00100700	0x00000000	0x7FAABB11
B1	M	0x00100728	0x00000000	0xFF000000
B2	I	0x00100710	0x00000000	0xEEEE7777
B3	S	0x00100718	0x00000000	0x7FAABB11
Procesador P2				
Block	State	Label	Data	
B0	S	0x00100720	0x00000000	0x1111AAAA
B1	S	0x00100708	0x00000000	0X00001234
B2	I	0x00100710	0x00000000	0x7FAABB11
B3	S	0x00100718	0x00000000	0x7FAABB11

Table 8: States for Section 3, exercise 3.

The state transitions are shown in Table 9.

Procesador	Block	State previous state	New state	Label	Data	
P2	B3	I	S	0X00100718	0X00000000	0X7FAABB11

Table 9: State transitions for section 3, exercise 3.

Exercise 4 *June 2014 exam*

Given a processor with architecture **Intel P6 or later**, in which there are two variables (**z** and **t**) that initially have the value **42**. Two threads run concurrently.

Thread 1 executes:

```
mov [_z], 1
mov r1, [_z]
mov r2, [_t]
```

Thread 2 executes:

```
mov [_t], 1
mov r3, [_t]
mov r4, [_z]
```

Is it possible that at the end of execution of thread 2 the records **r2** and **r4** have both the value of **42**? Justify your answer.

Solution 4

It is possible because writes **can be perceived in a different order for each processor**. In this way, in thread 1, you can have **r1=1** and **r2=42**, while in thread 2, you can have **r3=1** and **r4=42**.

Exercise 5 *June 2014 exam*

Given a multiprocessor with a symmetric shared memory architecture based on the bus with protocol of snooping. Each processor has a cache whose consistency is maintained using the MSI protocol. Each cache block has a single word.

The following table shows the initial state of four different variables in each of the caches.

	Initial states of the variables			
Processor	**A**	**B**	**C**	**D**
P0	Shared	Exclusive	Shared	Shared
P1	Invalid	Invalid	Invalid	Shared
P2	Invalid	Invalid	Shared	Shared

The following table shows the final state of these variables after doing a series of accesses to memory.

	Final states of the variables			
Processor	**A**	**B**	**C**	**D**
P0	Invalid	Invalid	Invalid	Shared
P1	Invalid	Invalid	Shared	Exclusive
P2	Exclusive	Exclusive	Invalid	Shared

Complete the following:

- For each variable (A, B, C and D) describe the accesses made and the processes involved that have allowed to reach the final state.
  - **Note 1**: to reach the final state may be enough a single access or a sequence of accesses.
  - **Note 2**: there may be a final state that is unreachable (that is, that has no solution).
- For each previous case describe the generated bus traffic.

Solution 5

- Variable **A**:

  - **P2** writes **A** (*exclusive*) invalidating **P0**.
  - **P2** produces a write miss that is captured by **P0** and produces no traffic.

- Variable **B**:
  - **P2** writes **B** (*exclusive*) invalidating the copy of **P0**
  - **P2** produces a write miss that is captured by **P0** that performs a write-block to the cache of **P2**

- Variable **C**:
  - **P1** writes **B** (*exclusive*) invalidating the other two copies
  - **P1** produces a write miss that is captured by **P0** and **P1**. No bus traffic is generated.
  - **P1** reads of writes in **D** another variable with a related block allocated in the same line as **C** producing a conflict miss. This makes block **C** to be replaced and given it was modified it is written into memory. Block **C** is not present in cache (equivalent to *invalid* state).
  - This action of **P1** produces a *write-back* of the block in memory.
  - **P1** reads **C**. Produces a read miss and goes to shared state.

- Variable **D**:
  - The final state can not be reached because a block can not be exclusive and shared at the same time.

Exercise 6 *January 2013 exam.*

A computer has a 32-bit Intel Core Duo processor with two cores (cores) with the configuration shown in the figure. The computer uses the MSI cache coherence protocol, which is applied in all cache levels. The block size is 16 bytes and each word is 4 bytes, which applies to all the cache levels.

The cache memory has the following characteristics:

- **L1I**: 16 KB size, 1ns access time and write-through policy.
- **L1D**: 32 KB size, 1ns access time and write-through policy.
- **L2**: Size of 2 MB, access time of 5ns and write-back policy.
- **Main memory**: Size of 4 GB, access time of 100 ns.

The code executed by each core is shown below. Both cores start to execute their code at the same time but due to the sleeps They will execute each loop at different times. The labels T0, T1, T2, T3, T4 and T5 refer to different execution points in each program. Note that T0 <T1 <T2 <T3 <T4 <T5.

The indexes of the **i** and **j** loops and the variables **tmp1** and **tmp2** are stored in registers. The vector **a** can be cached but **is initially located only in main memory** (it is not in the cache). Each element of a occupies a word. Assume that the execution time of each loop is so small that it is negligible.

```
// Code executed in core 1
sleep(seconds(2));
// T1
for (i=0;i<40;++i) {
  tmp1 = tmp1 + a[i];
}
sleep(seconds(3));
// T2
for (j=0;j<40;++j) {
  a[j] = j;
}
// T3


// Code executed in core 2
// T0
for (i=0;i<40;++i) {
  tmp2 = tmp2 + a[i];
}
sleep(seconds(10));
// T4
for (j=0;j<40;++j) {
  tmp2 = tmp2 + a[j];
}
// T5
```

Se pide:

1. Describe in what states of the protocol **MSI** is the block that store **a[0]** for the L1D cache of the **core 1** in the times T0, T1, T2, T3, T4 and T5. Justify the answer.

	T0	T1	T2	T3	T4	T5
State of **a[0]** in L1D of core 1						

2. Describe in what states of the protocol **MSI** is the block that store **a[0]** for the L1D cache of the **core 2** in the times T0, T1, T2, T3, T4 and T5. Justify the answer.

	T0	T1	T2	T3	T4	T5
State of **a[0]** in L1D of core 2						

3. Calculate the execution time of the code executed by **core 2** considering only the access times to the data (assuming that the access time to the instructions and that of execution of the loops is negligible).

Solution 6

Section 1 The **a[0]** block is initially in memory. During the first iteration of the first loop, **a[0]** is accessed and a miss occurs for cache in L1D of core 1 that propagates to L2, producing a hit. He block is transferred from the L2 cache to the L1D of the core 2. The state of the block passes from *invalid* to *shared* (I → S).

Subsequently, after 3 seconds, core 1 writes in **a[0]** from shared to modified (S → M) and invalidating the copy of the core 2. At t = 5 seconds, core 2 reads **a[0]** passing state modified to shared (M → S).

	T0	T1	T2	T3	T4	T5
State of**a[0]** in L1D of core 1	I	S	S	I	I	S

Section 2 The **a[0]** block is initially in memory. During the first iteration of the first loop, the access to **a[0]** produces a cache miss in L1D of core 2 which propagates to L2 producing another miss. The block is transferred from memory to the L2 cache and from there to the L1D of the core 1. The state of the block goes from *invalid* to *shared* (I → S).

Subsequently, at =5 seconds, core 1 writes to **a[0]** invalidating the copy of core 2, going from *shared* to *invalid* (S → I). At t=10 seconds, the core 2 reads **a[0]** going from modified to shared state (M → S).

	T0	T1	T2	T3	T4	T5
State of**a[0]** of L1D of core 1	I	S	S	I	I	S

Section 3

- **First loop** (**i** index):
  - When the first loop of the core 2 is executed, the a entries are in the main memory. The number of entries per block is $\frac{16}{4} = 4$ and the number of blocks accessed is $\frac{40}{4} = 10$.
  - The access time for the first entry of each block is the cost of accessing the memory: $t = 1 + 5 + 100$ ns For the rest of the entries in the block, since they are already in L1, the time will be the L1 access time: $t = 1$ ns.
  - The time for each block will be: $T_{block} = 106 + 3 \cdot 1$ ns.
  - The time of the first loop will be: $T_{loop1} = 10 \cdot T_{block} = 1090$ ns.
- **Second loop** (index **i**):
  - When the second loop of the core 2 is executed, the inputs of they are already in the L2 cache. The number of entries per block is $\frac{16}{4} = 4$ and the number of blocks accessed is $\frac{40}{4} = 10$.
  - The access time for the first entry of each block is the cost of accessing L2 cache: $t = 1 + 5$ ns.
  - For the rest of the block entries, since they are in L1, the time will be the access to the L1: $t = 1$ ns.
  - The time for each block will be: $T_{block} = 6 + 3 \cdot 1$ ns
  - The time of the second loop will be: $T_{loop2} = 10 \cdot T_{block} = 90$ ns.
- **Total time**:
  - The total time is: $T = T_{loop1} + T_{loop2} + sleep = 1180ns. + 10 * 10^9$ ns

Exercise 7 *January 2013 exam*

Given the following implementation of *lock / unlock*::

```
lock(location) {
  acquired = 0;
  while (! acquired) { // Waits if lock has been adquired
    acquired = test_and_set(location);
      if (!acquired) // if test and set fails, waits 2 ms
        sleep (2ms);
  }
}
```

```
unlock(location) {
  location = 0;
}
```

Four processors (P0, P1, P2 and P3) try to acquire the lock by simultaneously executing the following code:

```
lock(location);
// Compute time with a duration of 3 ms
unlock(location);
```

Complete the following:

1. Calculate how much time (in ms) it takes to acquire the lock by each of the four processors assuming they acquire it in an orderly manner (first P0, then P1, then P2 and finally P3). Assume that the execution time of each sentence and that the overload of the coherence protocol are negligible.
2. Is the implementation of the lock efficient? Justify your answer.
3. Indicate a possible improvement for this implementation. Justify your answer.

Solution 7

Section 1 The execution times that take to acquire are:

- P1: 0 ms
- P2: 4ms
- P3: 8ms
- P4: 12 ms

Section 2 It is not efficient because the waiting process waits longer than necessary and wastes time (1ms) in this wait.

Section 3 More efficient alternatives:

1. **test-and-set** with exponential backoff.
2. reduce the waiting time in the current version (at the expense of increasing the traffic of net),
3. Use of *load-linked/store-conditional.*
4. Use of a *mutex.*

Exercise 8 *June 2013 exam*

Consider 2 processes running on both processors P1 and P2 in a multiprocessor shared memory uysing a 100 Mbps bus. Processors P1 and P2 have an initially empty shared cache and the Consistency is achieved using the MSI protocol. In the same moment of time the processes execute the following code, where the variable l stores the memory address of a lock.

```
lock (l)
//cpu time for t=100 ms.
unlock(l)
```

Complete the following:

1. Write the implementation for synchronization primitives **lock(l)** and **unlock(l)** based on the instruction **test_and_set** whose syntax is:

   ```
   test_and_set register address
   ```

2. Suppose that a cache miss implies an 8 byte transfer for notify the miss and an invalidation implies a transfer of 16 bytes. What would be the total execution time of this program? Consider only the times of miss, invalidation and computation. Assume that the caches are initially empty.

3. What difference in performance exists between the primitives of synchronization **test_and_set** and **LL/SC** (*Load Linked* and *Store Conditional*)? Justify your answer.

Solution 8

Section 1 The implementation using the atomic primitive *test-and-set* would be:

```
lock:   TST R1 address_l
        bnz R1 lock
        ret
unlock: sw 0 address_l
```

where, the implementation of **TST** is equivalent to performing the following in an atomic fashion:

```
TST:    lw R1 address_l #R1 is returned
        sw 1 address_l
        ret
```

Section 2 The execution time of the program is requested, for which we initially calculate the cost in time of a miss and an invalidation. A cache miss transfers 8B over a 100Mbps bus, so the transfer time would be:

$$t_{transf} = \frac{8 \cdot 8}{100Mbps} = \frac{64}{100} \cdot 10^{-6} = 0.64 \cdot 10^{-6} s$$

An invalidation involves transferring 16B over a 100Mbps bus:

$$t_{transf} = \frac{16 \cdot 8}{100Mbps} = \frac{128}{100} \cdot 10^{-6} = 1.28 \cdot 10^{-6} s$$

Suppose that P1 process acquires the lock:

- **lock**: *test-and-set* R1 address_l $\rightarrow$ miss + invalidation $(0.64 + 1.28) \cdot 10^{-6}$ s
- **cpu**: 0.1s
- **unlock**: miss + invalidation $(0.64 + 1.28) * 10^{-6}$ s.

The execution time of process 1 implies a cache miss time at read the direction of the lock since initially the cache is empty. It also implies an invalidation to P2 since the *test-and-set* writes to the memory address. Subsequently, the computation with a duration of 0.1 s and finally the instruction **unlock ()** which implies a new miss (since invalidates the cache by the P2 that is constantly performing *test-and-set*) and an invalidation, since it is also modified in the other processor.

$$Total_{P1} = 1.92 \cdot 10^{-6} + 0.1$$

Process 2 accesses too the lock once it has been blocked by P1. This means that will perform active wait by execution *test-and-set* until the lock is released. The first time the process invokes lock is a miss and an invalidation.

- **First lock**: *TST* R1 address_l → miss + Invalidation $\frac{0.64+1.28}{\cdot}10^{-6}$ s

  item The subsequent times that process 2 invokes the *lock*, there are no misses or invalidations since it is already in cache in a modified state. Once the process 1 executes the *unlock*, process 2 performs again *test-and-set* (now succesfully), and involves 1 miss and one invalidation.

- **last lock**: *TST* R1 address_l → miss + invalidation $\frac{0.64+1.28}{\cdot}10^{-6}$ s

- **cpu**: = 0.1s

- **Unlock**: no hay ni fallo ni invalidación

  $Total_{P2} = 1.92 \cdot 10^{-6} + 0.1$

  $Total = 1.92 \cdot 10^{-6} + 0.1 + 1.92 \cdot 10^{-6} + 0.1 = 3.84 \cdot 10^{-6} + 0.2$

Section 3 To look for differences in performance, let's look at the implementation of *ll-sc*:

```
lock:   ll R1 address_l
        bnez R1 lock
        sc 1 address_l error
        bnez error lock
        ret
```

The special instruction **sc** (*store conditional*) only writes a 1 in the lock address if the value previously read by **ll** in the register **R1** (*load linked*) has not been modified by any other process. Therefore, when the *lock* is executed using the implementation *ll-sc*, it is always read from the variable of the lock but it is not always written, causing a lower number of accesses to memory and invalidations. For *test-and-set* a read and a write is performed in the direction of the lock each time that is executed.

2 Complementary consistency exercises

Exercise 9

The following code is executed on a machine with 3 processors. The variables that used are located in shared memory and initialized to **0**.

Processor 1 executes:

```
X=1;
```

Processor 2 executes:

```
Print(X);
Print(Y);
```

Processor 3 executes:

```
X=1;
Print(X);
```

Assume that all the variables are atomics. Answer the following questions:

- Analyze the following outputs
  - Is it possible to produce this output?
  - If so, provide an execution order that produces this output.

Print(X) [P2]	Print(Y)	Print(X) [P3]
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

- Show a global execution order that is not possible under sequential consistency, but that is possible under a relaxed consistency model.

Solution 9

The solution to this exercise consists of drawing a dependence graph with the program order and for each case analyse the order relationships.

Case 0,0,0:

Print X(2)
Print Y
Y=1
Print X(3)
X=1

Case 0,0,1

Print X(2)
Print Y
Y=1
X=1
Print X(3)

Case 0,1,0

Print X(2)
Y=1
Print Y
Print X(3)
X=1

Case 0,1,1

Print X(2)
Y=1
Print Y
X=1
Print X(3)

Case 1,0,0: not possible

Case 1,0,1:

X=1
Print X(2)
Print Y
Y=1
Print X(3)

Case 1,1,0

Y=1
Print X(3)
X=1
Print X(2)
Print Y

Case 1,1,1

```
X=1
Y=1
Print X(2)
Print Y
Print X(3)
```

Therefore, the only option that does not occur with sequential consistency is the 1.0.0 combination. However, with a relaxed consistency by eliminating the program order requirement, we could have:

```
Print Y
Y=1
Print X(2)
Print X(3)
X=1
```

Exercise 10

Repeat the exercise 9 for the following program, with all variables initialized to 0. Processor 1:

```
A=1;
Print(flag);
```

Processor 2:

```
while (flag==0) ;
Print(A);
```

Processor 3:

```
flag=1;
Print(A);
```

Solution 10

The solution to this exercise consists of drawing a dependency graph of the program order and for each case analyse the additional order relationships.

The possible events are:

- Print flag
- Print A(2)
- Print A(3)

Using a graph we observe that if processor 1 prints the flag value as 0:

- It has to happen before flag = 1 (Processor 3). This implieas that the processor 3 will print a value of 1.
- Once the while loop has been completed, it is also completed the assignment 1 for the flag, so the processor 2, also has to print 1

Caso 1,0,0

```
flag=1
Print A(3)
while (flag==0)
Print A(2)
A=1
Print flag
```

Caso 1,0,1

```
flag=1
while (flag==0)
Print A(2)
A=1
Print A(3)
Print flag
```

Caso 1,1,0

```
flag=1
while (flag==0)
Print A(3)
A=1
Print A(2)
Print flag
```

Caso 1,1,1

```
flag=1
A=1
while (flag==0)
Print A(3)
Print A(2)
Print flag
```

If the constraints of relaxed consistency are removed, it would be possible to use the following order:

```
Print A(2)
A=1
Print flag
while (flag==0)
flag=1
Print A(3)
```

That corresponds to the combination A(2) $\rightarrow$ 0, flag $\rightarrow$ 0, A (3) $\rightarrow$ 1.

Exercise 11

Repeat the exercise 9 for the following program, with all variables initialized to 0.
Processor 1:

```
A=1;
```

Processor 2:

```
U=A;
B=1;
```

Processor 3:

```
V=B;
W=A;
```

Solution 11

An analysis must be done for the possible values of **U**, **V** and **W**:
0,0,0

```
U=A
V=B
W=A
A=1
B=1
```

0,0,1

U=A
V=B
A=1
W=A
B=1

0,1,0

U=A
B=1
U=B
W=A
A=1

0,1,1

U=A
A=1
B=1
V=B
W=A

1,0,0

V=B
W=A
A=1
U=A
B=1

1,0,1

A=1
U=A
V=B
W=A
B=1

If both **U** and **V** are 1, this means that necessarily the last instruction to execute must be **W** = **A**, and for both **A** must also take the value 1.

Therefore, it is impossible that the case produces the output 1,1,0

For such a case to occur, the execution order of the program has to be the following sequence:

A=1
U=A
W=A
B=1
V=B

Exercise 12

The following program is executed in a computer with two processors. The variables are stored in shared memory and initialized to 0.

Processor 1:

X=1;
Y=X;

Processor 2:

Y=2;
X=3;

- Which values of X and Y are possible under sequential consistency?
- Find an output that is possible under relaxed consistency and not under sequential consistency.

- Is the output (3,0) possible?

Solution 12

On the one hand, the final value for **X**, can only be either 1 or 3.

If the final value of **X** is 1, the mapping **X** = 3, must precede **X**=1, and therefore **Y** must also take the value 1.

If the final value of **X** is 3, the following commands can be given:

Case 1 (X=3, Y=1)

Y=2
X=1
Y=X
X=3

Case 2 (X=3, Y=3)

Y=2
X=1
X=3
Y=X

Case 3 (X=3, Y=2)

X=1
Y=2
Y=X
X=3

Case 4 (X=3, Y=2)

X=1
Y=X
Y=2
X=3

Case 5 (X=3, Y=2)

X=1
Y=2
X=3
Y=X

The following case is possible under relaxed consistency:

Y=2
X=3
Y=X
X=1

The result (**X = 1**, **Y = 3**), is not possible under sequential consistency.

Finally, the case (3.0) is not possible even on a relaxed consistency model because **X = 1** must always be executed before **Y = X**. Therefore, it is impossible for **Y** to take the value 0.