PROGRAMING ASSIGNMENT 3

SHARED MEMORY

2013

Mahboobeh Abdal Mahmood Abadi & Mehran Nasseri

kth

2/20/2013

Contents

[1. Exercise 1 3](#_Toc349182789)

[2. Exercise 2 4](#_Toc349182790)

[3. Exercise 3 5](#_Toc349182791)

[4. Exercise 4 11](#_Toc349182792)

## Exercise 1

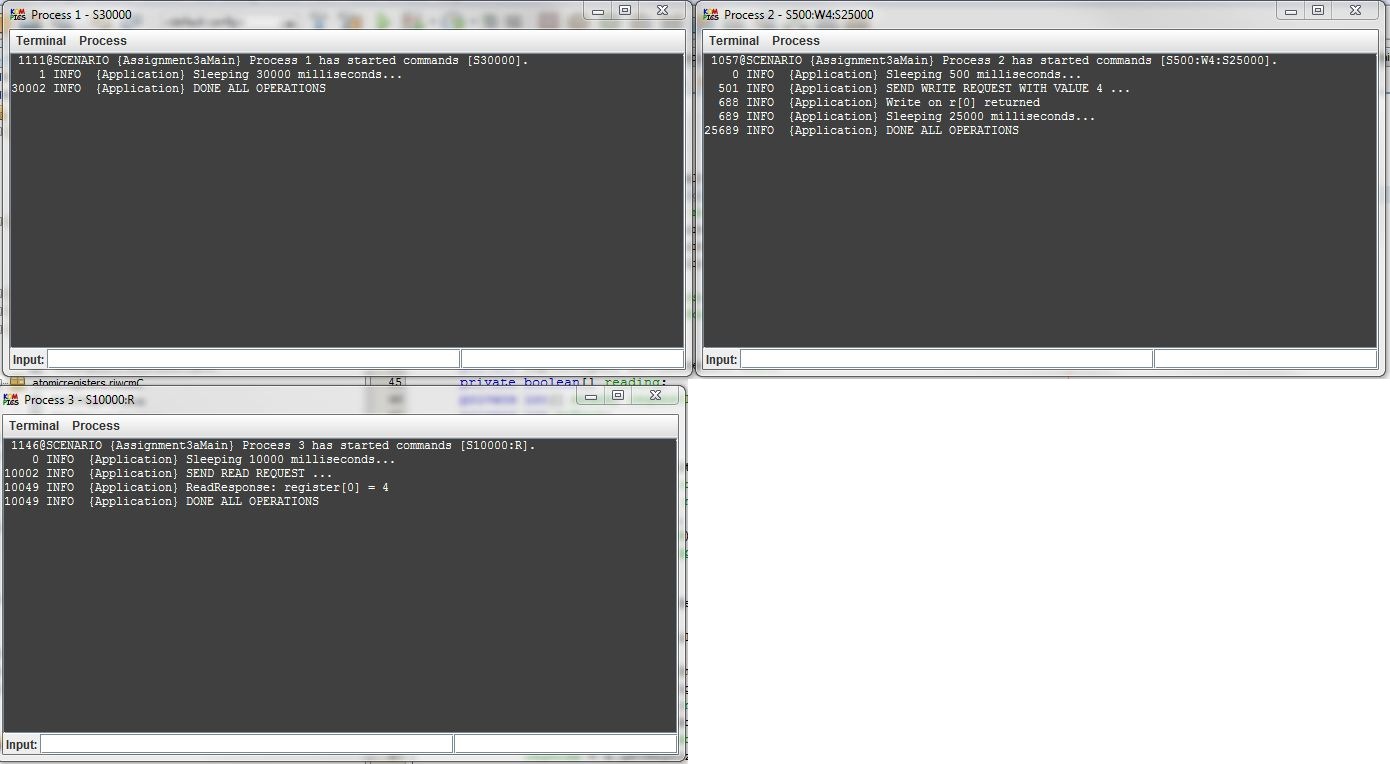


Figure 1: result of experiment using RIWC

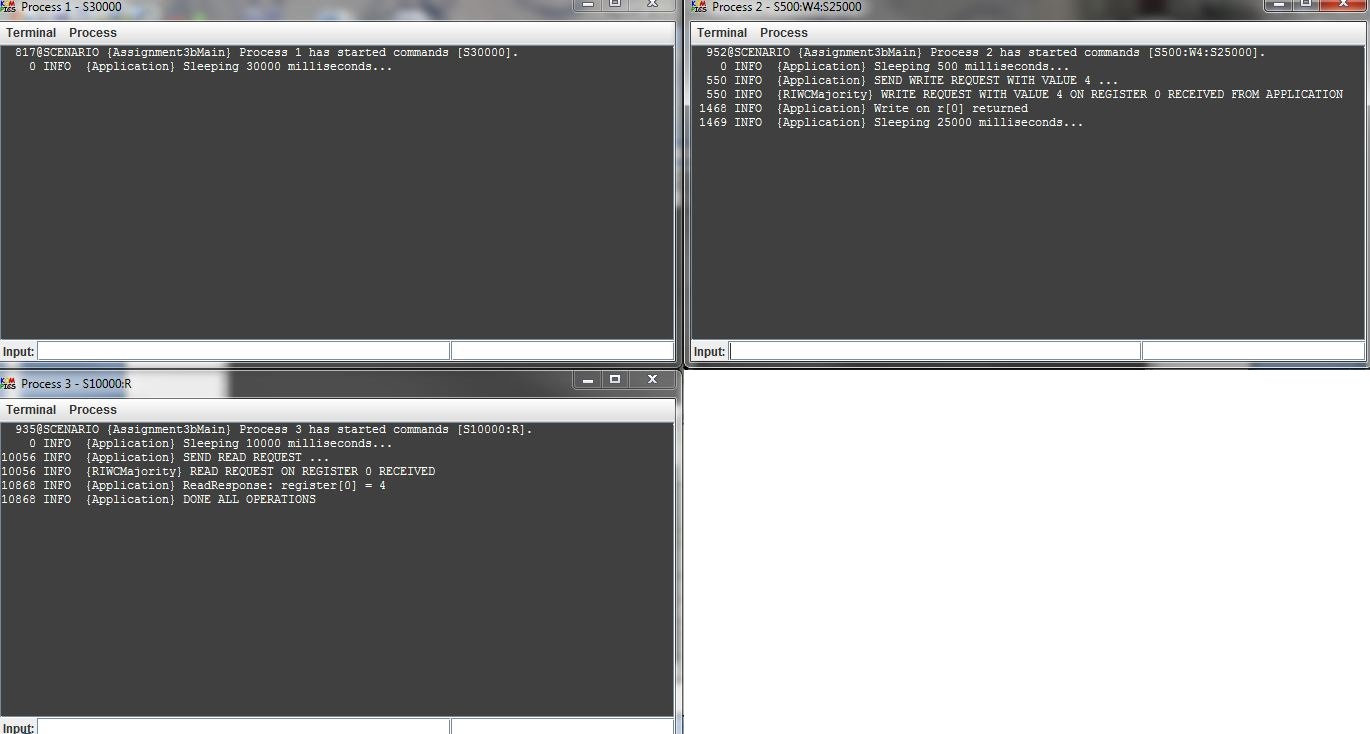


Figure 2: result of experiment using RIWCM

As can be seen in figure 1 and 2 the process 3 could successfully read the last value.

## Exercise 2

* 1. Value read by process 3

As the result of experiment shows in figure 3, the initially dead process could read the last value, which is 6. This is obviously because of the algorithm used to read the value. The algorithm does not read the value locally; instead, it sends the read request to other nodes, and then as soon as receiving values from the majority of nodes it selects the one with highest timestamp and rank as most recent value. Here it must be notice that even though, process 1 and 2 wrote two different values concurrently to the register with the same timestamp but the process 3 returned the value written by process 2, since it has higher rank.

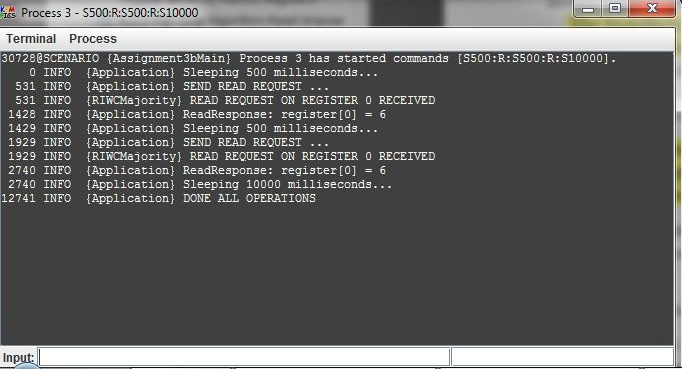


Figure 3

* 1. Extending RIWCM to be used in fail recovery model

` RIWCM cannot be used in fail recovery model by. We should consider the different meaning of correct process in fail recovery model. In fail recovery model, a correct process may crash and recover finite number of times. Consider an execution in which process q invokes either write or read operation and never crash. To terminate the operation the process need to receive ack from the majority. Assume that majority of processes crash before receiving the write/ read message, and recover later, so they will never be aware of such message they will never send any ack message. Consequently, process q can never terminate the operation and this violates the termination property. (Termination: If a process invokes an operation and ***never crashes***, then the operation eventually completes.)

Therefore, we need to apply following changes. In this way, the algorithm can be transformed to deal with processes crash and recovery.

* **Persistence storage:** The simplest registers must have access to stable storage and require that a majority of the processes is correct.
* **Store operation:** The algorithm stores some variables in persistent storage in particular, the current timestamp and value stored in the register, the request identifier, and the write timestamp.
* **Retrieve operation:** the process accesses stored variables again upon recovery with a retrieve operation.
* **Communication**: stubborn point-to-point links abstraction and a stubborn best-effort broadcast abstraction. This ensures that the process, even if it crashes and recovers a finite number of times, will eventually process every message sent to it.
* **Boolean writing:** Logging the writing variable is needed for restarting a write operation that may have been left in a partially completed state after a crash.

## Exercise 3

For all cases, the topology given in the assignment is used.

* + 1. Case 1

command(1, "S500:W1:R:S500:R:S8000");

command(2, "S500:W2:R:S500:R:S8000",1000);

command(3, "S500:W3:R:S500:R:S8000",2000);

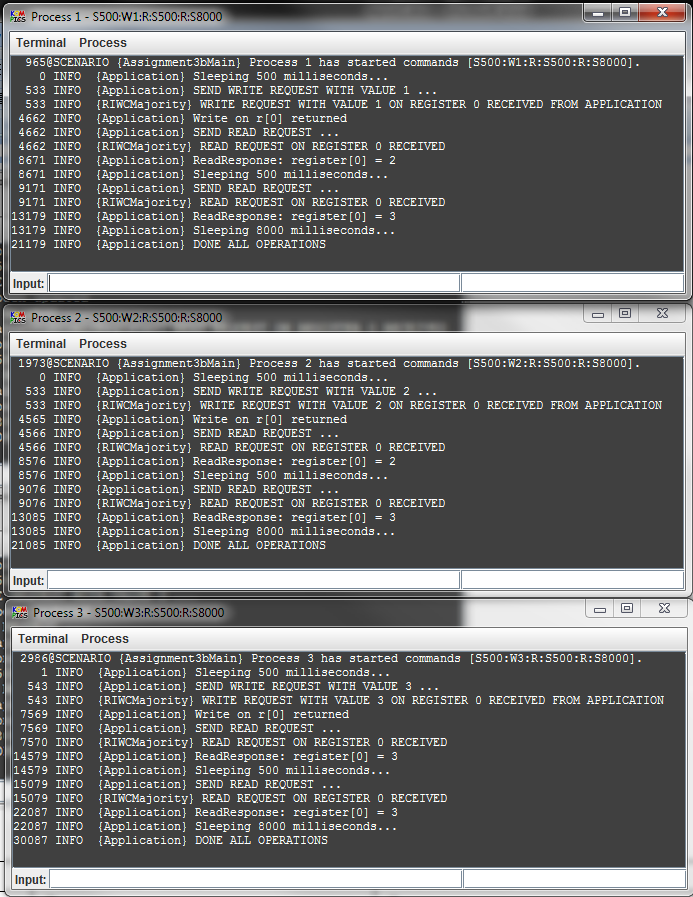


Figure 4

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 2 | 3 |
| 2 | 2 | 3 |
| 3 | 3 | 3 |

Linearization:

* + 1. Case 2

command(1, "S500:W1:R:S500:R:S8000");

command(2, "S500:W2:R:S500:R:S8000",2000);

command(3, "S500:W3:R:S500:R:S8000",1000);

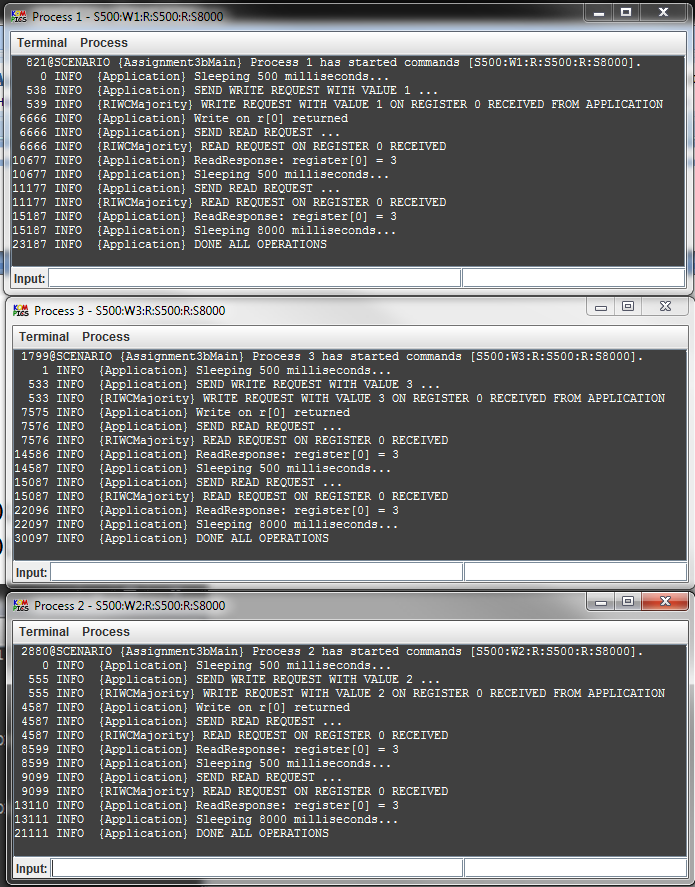


Figure 5

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 3 | 3 |
| 2 | 3 | 3 |
| 3 | 3 | 3 |

Linearization:

* + 1. Case 3

command(1, "S500:W1:R:S500:R:S8000", 1000);

command(2, "S500:W2:R:S500:R:S8000");

command(3, "S500:W3:R:S500:R:S8000",2000);

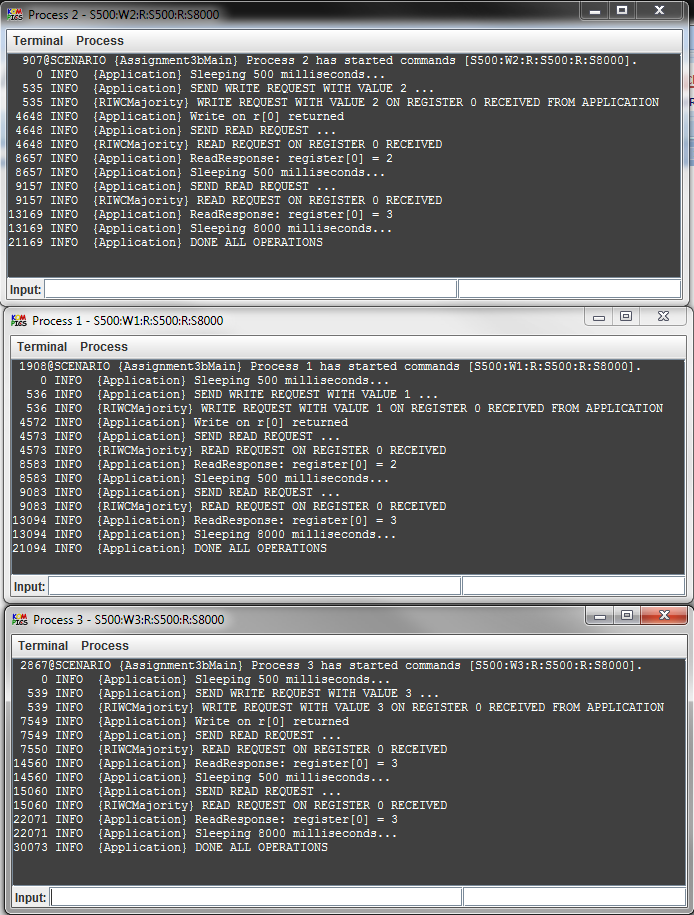


Figure 6

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 2 | 3 |
| 2 | 2 | 3 |
| 3 | 3 | 3 |

Linearization:

* + 1. Case 4

command(1, "S500:W1:R:S500:R:S8000", 2000);

command(2, "S500:W2:R:S500:R:S8000");

command(3, "S500:W3:R:S500:R:S8000",1000);

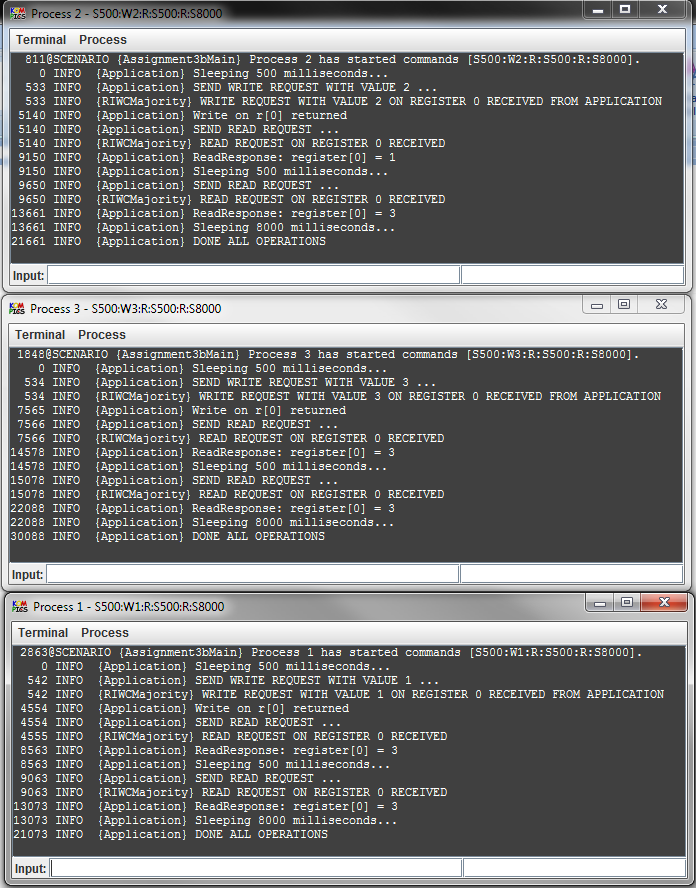


Figure 7

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 3 | 3 |
| 2 | 1 | 3 |
| 3 | 3 | 3 |

Linearization:

* + 1. Case 5

command(1, "S500:W1:R:S500:R:S8000", 1000);

command(2, "S500:W2:R:S500:R:S8000", 2000);

command(3, "S500:W3:R:S500:R:S8000");

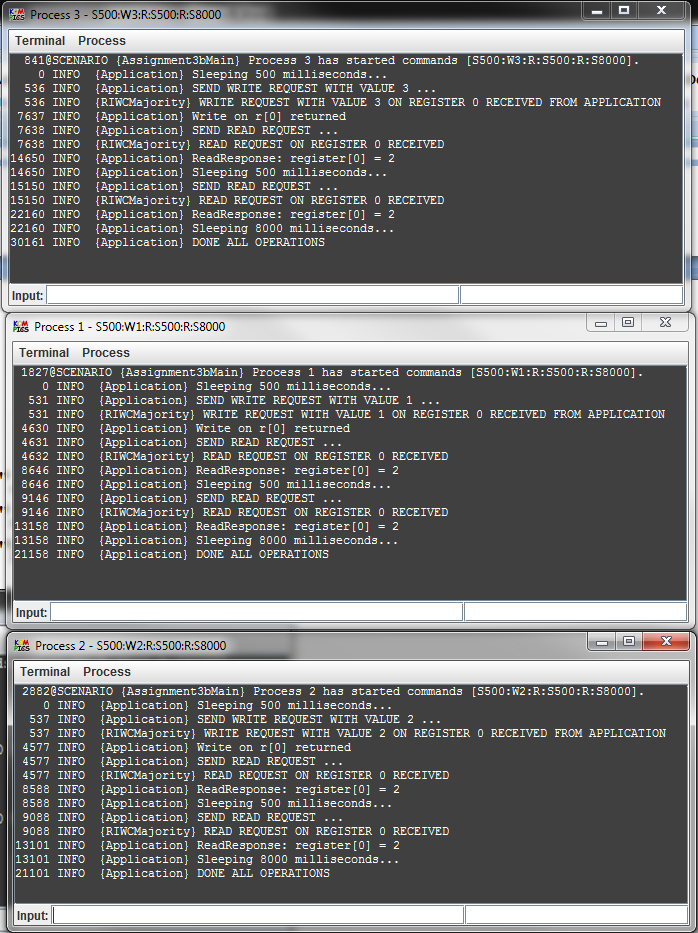


Figure 8

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 2 | 2 |
| 2 | 2 | 2 |
| 3 | 2 | 2 |

Linearization:

* + 1. Case 6

command(1, "S500:W1:R:S500:R:S8000", 2000);

command(2, "S500:W2:R:S500:R:S8000", 1000);

command(3, "S500:W3:R:S500:R:S8000");

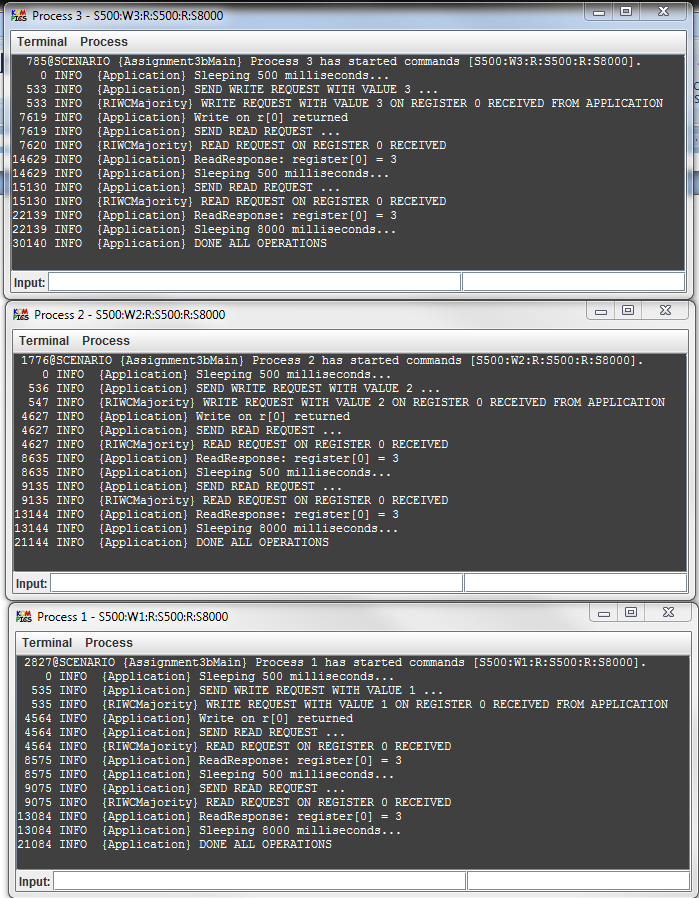


Figure 9

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 3 | 3 |
| 2 | 3 | 3 |
| 3 | 3 | 3 |

Linearization:

In conclusion, we can say that executions in all cases are linearizable since they satisfied the linearization conditions. I.e. every read returned the last value written and if an operation *x* precedes another operation *y* in actual execution, shown in figures 6-9, then *x* also appears before *y* in the linearization.

## Exercise 4

1. **Case 1**

link(1, 2, 1000, 0).bidirectional();

link(1, 3, 200, 0).bidirectional();

link(2, 3, 1750, 0).bidirectional();

command(1, "S500:W1:R:S500:R:S8000", 200);

command(2, "S500:W2:R:S500:R:S8000", 100);

command(3, "S500:W3:R:S500:R:S8000");

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 3 | 3 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |

Linearization:

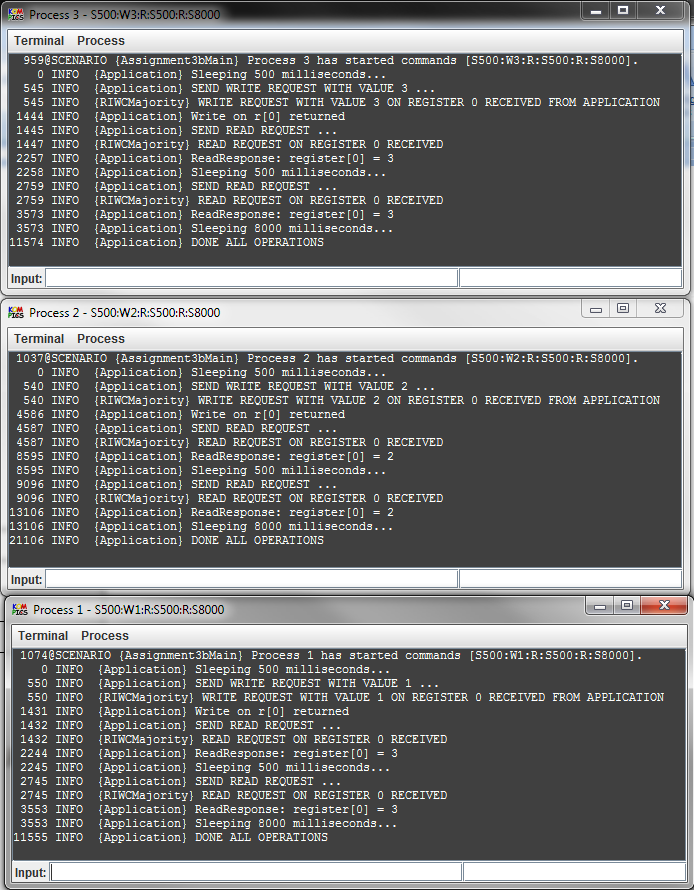


Figure 10: actual execution of the case 1

1. **Case 2**

link(1, 2, 1000, 0).bidirectional();

link(1, 3, 1000, 0).bidirectional();

link(2, 3, 0, 0).bidirectional();

command(1, "S500:W1:R:S500:R:S8000");

command(2, "S500:W2:R:S500:R:S8000", 100);

command(3, "S500:W3:R:S500:R:S8000", 200);

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 1 | 1 |
| 2 | 3 | 3 |
| 3 | 3 | 3 |

Linearization:

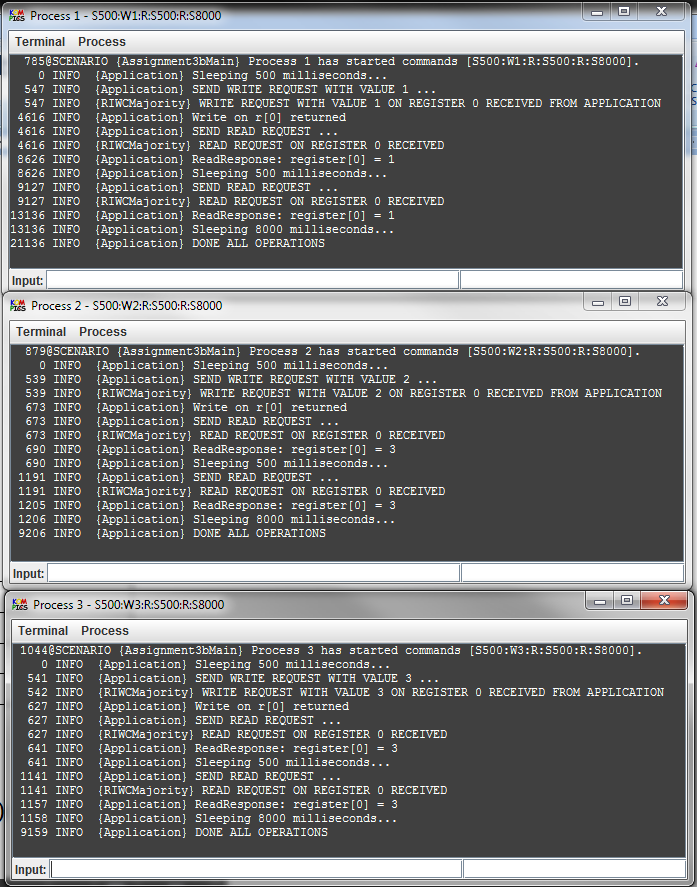


Figure 11: actual execution of the case 2

1. **Case 3**

link(1, 2, 0, 0).bidirectional();

link(1, 3, 0, 0).bidirectional();

link(2, 3, 500, 0).bidirectional();

command(1, "S500:W1:R:S500:R:S8000");

command(2, "S500:W2:R:S500:R:S8000", 1000);

command(3, "S500:W3:R:S500:R:S8000", 2000);

|  |  |  |
| --- | --- | --- |
| Process id | First read value | Second read value |
| 1 | 1 | 2 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |

Linearization:

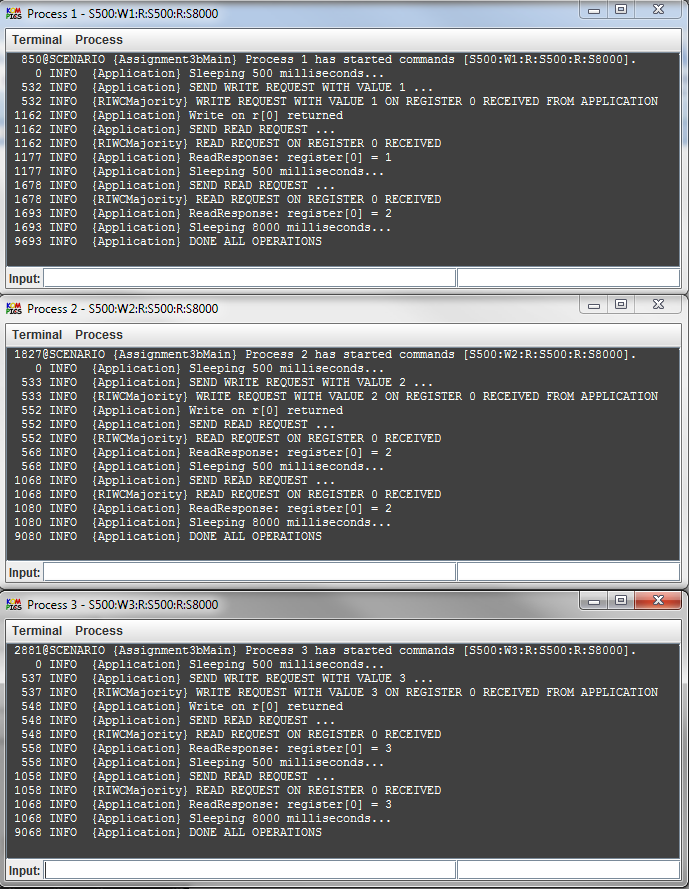


Figure 12: actual execution of the case 3