Simulating MP Programs with Unreliable Communication on Shared Memory

CS403/534 - Distributed Systems
Programming Assignment #1, Fall 2022

Deadline: November 7, 2022 (23:55 UTC+03:00)

In the lectures, we have seen that message passing (MP) distributed programs can be simulated by shared-memory multi-threaded programs. Each thread might simulate the computation of a process and the shared memory can be used for simulating the communication mechanism. In this programming assignment (PA), you are supposed to implement a multi-threaded program that mimicks an asynchronous system with persistent communication channels.

Such a system can be abstractly represented by attaching a mailbox to every process. When a process A sends a message m to process B, it puts m into the mailbox of B. Then, B can asynchronously read messages intended for itself anytime by fetching messages from its mailbox. Depending on the underlying communication medium, the shape of mailbox changes. If processes are exchanging messages via TCP which preserves the order of messages by the same process, then we can think of queue mailboxes. Otherwise, if we assume UDP channels which can lose, duplicate or reorder messages, then a mailbox can be considered as a multi-set. It is an unordered collection in which the same element might appear multiple times. For this PA, we are assuming that processes are communicating through a stronger version of UDP such that messages by the same process might be reordered but no message can be lost or duplicated. Hence, mailboxes of processes can be represented by multi-sets.

For the first part of this PA, you will implement a multi-set for simulating the mailboxes of processes. Note that add and remove methods of the multi-set might be executed concurrently. Multiple processes can send a message to the same process at the same time. Similarly, a process might try to read a message while another process is trying to send a message to it. Hence, you have to consider concurrency problems like deadlock while developing your multi-set implementation.

In the second part, you will develop a multi-threaded PYTHON program that simulates a distributed leader election algorithm which uses your concurrent multi-set implementation for simulating the mailboxes of processes. Details are provided in the following sections of this document.

1 A Linearizable and Concurrent Multi-set Implementation

Your first task is to implement a class named *ConSet* in PYTHON programming language. This class will implement a linearizable concurrent (multi-threaded) multi-set. In lectures, we saw a sequence of concurrent sorted linked-list implementations from the Chapter 9 of the book *The Art of Multiprocessor Programming* by Herlihy and Shavit[1]. As the book claims (and provides informal intuition on its correctness), these concurrent linked-list implementations are linearizable with respect to a set specification. You can modify any one of these concurrent linked-list algorithms for your *ConSet* implementation or develop one multi-set implementation yourself.

In order to adapt the set for our purposes, we have to change the API provided by the book. The original API consists of `add`, `remove` and `contains` methods. For representing mailboxes, we do not need `contains` method and we remove it from the API. In addition, we modify the `remove` method's signature. In the original API, `remove` has an input parameter and a boolean return value that is true if and only if the input value was in the set before the operation. We change it such that `remove` does not take any input parameters and returns an element from the generic type `Element`. Depending on your choice, you can remove the first, the last or another random element in your linked list implementation. We want `remove` method to be blocking. If the multi-set is empty, then the `remove` must wait until another entity inserts an element to the multi-set. The signature of the `add` changes such that it takes an integer parameter as input and the new return type is `None`. The semantics of the add method must be slightly modified as well. Since we are implementing a multi-set, multiple nodes with the same element can occur inside the list. Hence, `add` method must be able to add an existing element to the list successfully. Order of the nodes with same element does not matter. When an existing element is to be re-added to the list, new node corresponding to this element could be inserted before, after or between existing nodes with the same element as long as the ascending order is preserved. With these changes, our new API looks like this:

```
interface Set {
   None add(Element x);
   Element remove();
}
```

There are two main routes for implementing the new API using the off-the-shelf implementations from the book. The first way is to take `add`, `remove` and `contains` method implementations as they are (but probably rename them) and implement the new API using them. For instance, `remove` in the new API can be implemented using `contains` and `remove` in the old API. The second way is to inspire from `add` and `remove` implementations from the book, but modifying and optimizing them according to the changes described above.

In both cases, ensuring linearizability with respect to the multi-set specification is your duty. We are expecting you to include a report in your sub-

[1]You might find this chapter on SUCourse under Architectural Styles and Concurrency header with title *Concurrent Linked Lists*.

mission bundle that explains which algorithm you picked from the book, how you modified it for the new API and your high level arguments on why your implementation is still linearizable. If you implement your own algorithm, you must describe it in the report as a replacement for the modifications in the book algorithm. For both cases, we expect a discussion on the linearizability of your implementation with respect to the multi-set specification described above.

If you develop and implement your own linearizable concurrent multi-set `Element` type could be arbitrary. However, if you adopt one of the sorted linked-list implementations from the book, instances of the `Element` must be comparable.

Algorithms in the book are written in JAVA language. You have to translate them into PYTHON and use the synchronization primitives from THREADING library of PYTHON. This library contains mutex, condition variable and semaphore implementations. However, it does not provide complex atomic instructions like `compare-and-set`, `fetch-and-add` etc. If you are willing to implement one of the algorithms in the book that utilizes such statements, you have to establish their atomicity by yourselves (by most probably using existing synhcronization mechanisms like semaphores).

Algorithms towards the end of the chapter are more complicated to implement, yet they are faster and more efficient. We will provide bonus points to the fastest submissions. You can find the details of bonus points in the grading section. The burden of implementing more complex algorithms and solving implementation problems for `compare-and-set` kind of atomic statement will be rewarded by faster programs and bonus points.

2 A Distributed Leader Election Algorithm

In this part, you are asked to simulate a leader election protocol using your concurrent multi-set implementation. For this protocol, assume that there are n nodes. Each node has a mailbox that keeps its messages without respecting the order of their arrival but it does not allow message losses. So, you will use your concurrent multi-set to simulate mailboxes of nodes. Mailboxes will be represented by a global list that stores n *ConSet* instances.

To simulate nodes, you need to implement a $nodeWork$ function that will be run by different threads. Node identifier (a non-negative integer) and n will be arguments of this function. It will generate a random number between 0 and n^2, create a tuple with this number and its id and will put this tuple into the mailboxes of all nodes (including its own) using the `add` operation. Then, it will start receiving messages from its own mailbox using the `remove` operation. Since a node is aware of n it knows how many messages it is expecting. Among the received messages, its aim is to find the node with maximum generated value. If this value is unique, it prints this node's id and declares it as a leader. If there are more than one node with this maximal generated number, the whole process starts again by producing a new random number and putting it into mailboxes. Unfortunately, it is possible that nodes can create a livelock while choosing a leader meaning that they can keep generating the same maximal numbers and then restarting indefinitely. In the scope of PA, livelock is not a problem. We will ignore it hoping that eventually they will reach to a unique maximum.

Important Note: Since the maximal generated number might not be

unique, there can be multiple rounds of leader election. If the message contents are just $(number, id)$ pairs, a thread cannot differentiate messages coming from different rounds. This might cause incorrect computation. In order to prevent this problem, you might provide synchronization by using a barrier (you might implement one yourself or use an off-the-shelf implementation) at the end of a round or you might add new fields to the message content for differentiating messages from distinct rounds. In your report, you should have a section that explains how you solve this problem.

Important Note 2: Messages of the leader election algorithm are in the form of tuples (pairs in the most plain case. Yet, you can add more fields). If your *ConSet* class implements one of the sorted linked-lists from the book, you must provide a comparison method that implements $<$ operator so that *ConSet* can keep nodes ordered.

The *nodeWork* function must be implemented in a PYTHON file called `Leader.py` This file must have a global variable n which is initialized to some small positive integer like 3. Also, this file must initialize mailboxes and generate n threads by providing correct arguments. At the end, main thread must wait until all children threads terminate.

3 Sample Runs

In this section, we provide sample runs for the program implemented in `Leader.py`. Assume that $n = 4$ for all cases. In the first example, the leader is elected in the first round:

```
Node 0 proposes value 1 for round 1.
Node 1 proposes value 6 for round 1.
Node 2 proposes value 7 for round 1.
Node 3 proposes value 1 for round 1.
Node 1 decided 2 as the leader.
Node 2 decided 2 as the leader.
Node 0 decided 2 as the leader.
Node 3 decided 2 as the leader.
```

In the second run, decision could be made in the second round:

```
Node 0 proposes value 3 for round 1.
Node 1 proposes value 3 for round 1.
Node 2 proposes value 0 for round 1.
Node 3 proposes value 3 for round 1.
Node 0 could not decide on the leader and moves to the round 2.
Node 1 could not decide on the leader and moves to the round 2.
Node 2 could not decide on the leader and moves to the round 2.
Node 0 proposes value 2 for round 2.
Node 1 proposes value 3 for round 2.
Node 2 proposes value 0 for round 2.
Node 3 could not decide on the leader and moves to the round 2.
Node 3 proposes value 2 for round 2.
Node 1 decided 1 as the leader.
Node 2 decided 1 as the leader.
Node 0 decided 1 as the leader.
```

```
Node 3 decided 1 as the leader.
```

4 Submission Guidelines

You need to submit at least two PYTHON files and a report file:

- `ConSet.py` contains the class with the same name that implements the concurrent multi-set representing a linked list described in the first part of this document.
- `Leader.py` contains the distributed leader election algorithm described in the second section.
- `Report.pdf` contains your explanations on your multi-set implementation, modifications on the linked-list and your arguments on linearizability of your implementation. For the leader election, you should explain how you handle messages from different rounds.

For this PA, you are not allowed to work in groups. So, every one needs to submit his/her own implementation. Required files explained above should be put in a single zip file named as `CS_403-534_PA01_name_surname.zip` and submitted to PA1 under assignments in SUCOURSE+. Submissions will end on 23:55 on the day of the deadline. There is no late submission but you can use a grace day if necessary (see the syllabus, don't wait to learn about it on the last homework, grace days are given for your own good).

5 Grading

- Concurrent Multi-set Implementation (50 Points): Your class *ConSet* must implement `add` and `remove` methods. You are not allowed to use existing sequential or concurrent data type implementations like sets, lists, vectors, queues, stacks etc. Except this, your *ConSet* can only have synchronization variables like locks, semaphores and condition variables. You are advised to use `Threading` library of PYTHON for this purpose. You are allowed to use other native libraries that provide shared memory synchronization mechanisms, however, then you might not get partial credit if your code does not work correctly or fail to run in our machines. Grading of your implementation will be done with respect to the following criteria:
  - (20 Points) Your implementation works as a correct sequential set i.e., its methods are not called by concurrent threads.
  - (10 Points) Your implementation is linearizable. Your set behaves correctly when called by concurrent threads.
  - (10 Points) Your set implementation does not allow deadlocks. Please be careful about how your code is synchronized.
- Distributed Leader Election (50 Points): We will check this part with our own correct set implementation. So, if your first part is not totally correct, it will not affect grading of this part. We will check a single condition for this part. We will initiate leader election with different n values and see if

eventually all nodes agree on a leader. For this part, we again advise using `Threading` library of PYTHON. If you want to use another library, you are free to do so but you take the responsibility described in the previous part.

- Report (10 Points) : You will explain in a report your concurrent multi-set implementation (either indicate the algorithm name in the book and your modifications on it or explain your own algorithm), and how it preserves linearizability if there are any. You can write it in a PDF file and put it in the .zip along the other files.
- Bonus (10 Points): We will give bonus to the top 10 fastest (when executed in our computer and of course, correctly working) implementations.