*home page -> teaching -> parallel and distributed programming -> Sample subjects for the written exam*

# Sample subjects for the written exam

## Requirements

1. Write a parallellized and distributed version of the QuickSort algorithm using MPI.

2. Consider the following code. Find and fix the concurrency issue.

```
struct Account {
    unsigned id;
    unsigned balance;
    mutex mtx;
};

bool transfer(Account& from, Account& to, unsigned amount)
{
    unique_lock lck1(from.mtx);
    unique_lock lck2(to.mtx);
    if(from.balance < amount) return false;
    from.balance -= amount;
    to.balance += amount;
}
```

3. Give an algorithm for solving the following variant of the consensus problem:

- Each process *i* has an input value $v_i$;
- All processes must decide some output value; the output value must be the same for all processes and, if all inputs are equal, then the common output must be equal to that input;
- We have synchronous rounds;
- The processes are all correct;
- Messages may be lost; however, if a party sends infinitely many messages, then infinitely many will be delivered;
- Each process must decide after a finite amount of time; however, it can continue the algorithm forever.

## Solutions

1. See implementation in quicksort-mpi.cpp.

2. There is a potential deadlock. If two instances of transfer() run concurrently between the same accounts but in reverse directions (say transfer(a1, a2, amount1) and transfer(a2, a1, amount2) ) it is possible that the first locks the mutex for account a1 and the second locks the account a2, and then the first transfer waits for a2 and the second for a1 to become available.

One solution is to always lock the mutexes in increasing order of the account IDs. Possible implementation:

```
bool transferAssumingLocked(Account& from, Account& to, unsigned amount)
{
    if(from.balance < amount) return false;
    from.balance -= amount;
    to.balance += amount;
}
bool transfer(Account& from, Account& to, unsigned amount)
{
    if(from.id < to.id) {
        unique_lock lck1(from.mtx);
        unique_lock lck2(to.mtx);
        return transferAssumingLocked(from, to, amount);
    }
    if(from.id > to.id) {
        unique_lock lck2(to.mtx);
        unique_lock lck1(from.mtx);
        return transferAssumingLocked(from, to, amount);
    }
    return true; // transfer from an account to itself
}
```

3. Since no process is faulty, it is enough to make the first process send its value to all other processes and have all processes agree on that value. There is no risk that the first process sends distinct values to distinct processes; however, we must make sure that the value sent by the first process arrives at all other processes.

So, the solution is the following:

- process 1 decides $y_1=x_1$; Then, forwever, it waits for requests from other processes and answer each request by sending back $x_1$ to the requester.
- each of the other processes repeatedly sends requests to process 1 until it receives an answer. At that moment, it decides the values read from that answer.

*Radu-Lucian LUPŞA*
*2017-01-08*