Question **1**

Not yet answered

Marked out of 1.00

I hereby agree that I have not, and will not, receive any help from anyone for all the duration of the exam, with respect to the subject of the exam.

(you should answer True to take the exam)

Select one:

- (●) True
- ( ) False

Question **2**

Not yet answered

Marked out of 3.00

Consider the following code for a queue with multiple producers and consumers. The *close()* function is guaranteed to be called exactly once by the user code, and *enqueue()* will not be called after that. dequeue() is supposed to block if the queue is empty and to return an empty optional if the queue is closed and all the elements have been dequeued.

```
template<typename T>
class ProducerConsumerQueue {
    list<T> items;
    bool isClosed = false;
    condition_variable cv;
    mutex mtx;
public:
    void enqueue(T v) {
        unique_lock<mutex> lck(mtx);
        items.push_back(v);
        cv.notify_one();
    }
    optional<T> dequeue() {
        unique_lock<mutex> lck(mtx); // statement 1
        while(items.empty() && !isClosed) {
              // place 1
            cv.wait(lck);
              // place 2
        }
        lck.unlock(); // statement 2
        if(!items.empty()) {
            optional<T> ret(items.front());
            items.pop_front();
              // place 3
            return ret;
        }
          // place 4
        return optional<T>();
    }
    void close() {
        unique_lock<mutex> lck(mtx);
        isClosed = true;
        cv.notify_all();
    }
};
```

or the Java equivalent.

```
class ProducerConsumerQueue<T> {
    ArrayList<T> items;
    boolean isClosed = false;
    Lock mtx;
    Condition cv = mtx.newCondition();
public:
    void enqueue(T v) throws InterruptedException {
        mtx.lock();
        items.add(v);
        cv.signal();
        mtx.unlock();
    }
    T dequeue() throws InterruptedException {
        mtx.lock(); // statement 1
        while(items.empty() && !isClosed) {
              // place 1
            cv.await(lck);
              // place 2
        }
        mtx.unlock(); // statement 2
        if(!items.isEmpty()) {
            T ret = items.get(0);
            items.remove(0);
              // place 3
            return ret;
        }
          // place 4
        return null;
    }
    void close() throws InterruptedException {
        mtx.lock();
        isClosed = true;
        cv.signalAll();
        mtx.unlock();
    }
};
```

What concurrency issues does it present? How to fix them?

Select one or more:

- [ ] [fix] insert a statement unlocking the mutex in the place marked *place 1* and lock it back in *place 2*
- [ ] [fix] remove line marked *statement 2* and insert copies of it in placed marked *place 3* and *place 4*
- [ ] [issue] two simultaneous calls to *dequeue()* may deadlock
- [ ] [issue] two simultaneous calls to *enqueue()* may result in corrupted *items* list
- [ ] [issue] a call to *dequeue()* can deadlock if simultaneous with the call to *enqueue()*
- [ ] [fix] move line marked *statement 1* in the place marked *place 1* and *statement 2* in *place 2*

- [ ] [fix] eliminate lines marked *statement 1* and *statement 2*

- [ ] [issue] a call to *dequeue()* can result data corruption or undefined behavior if simultaneous with the call to *enqueue()*

- [ ] [fix] remove line marked *statement 2* and insert copies of it in placed marked *place 3* and *place 4*, and then move *statement 1* in the place where *statement 2* was

- [ ] [issue] two simultaneous calls to *enqueue()* may deadlock

Question **3**

Not yet answered

Marked out of 3.00

Consider the following code for computing the product of two matrices (assuming the number of columns of a is equal to the number of rows of b).

```
void computeOneElement(
    std::vector<std::vector<int> > const& a,
    std::vector<std::vector<int> > const& b,
    size_t row, size_t col,
    std::vector<std::vector<int> >& rez,
    std::mutex& mtx)
{
    mtx.lock();
    int sum = 0;
    for(size_t i=0 ; i<b.size() ; ++i) {
        sum += a[row][i]*b[i][col];
    }
    rez[row][col] = sum;
    mtx.unlock();
}

std::vector<std::vector<int> > matrixProd(
    std::vector<std::vector<int> > const& a,
    std::vector<std::vector<int> > const& b,
    size_t nrThreads)
{
    std::mutex mtx;
    size_t outNrRows = a.size();
    size_t outNrCols = b[0].size();
    std::vector<std::vector<int> > rez(outNrRows);
    for(std::vector<int>& row : rez) {
        row.resize(outNrCols);
    }
    size_t begin = 0;
    size_t step = (outNrRows+nrThreads-1)/nrThreads; // statement 1
    std::vector<std::thread> threads;
    threads.reserve(nrThreads);
    for(size_t th=0 ; th<nrThreads ; ++th) {
        size_t end = begin+step; // statement 2
        threads.emplace_back([begin,end,outNrCols,&a,&b,&rez,&mtx]() {
            for(size_t i=begin ; i<end ; ++i) {
                for(size_t j=0 ; j<outNrCols ; ++j) {
                    computeOneElement(a, b, i, j, rez, mtx);
                }
            }
        });
        begin = end;
    }
    for(std::thread& th : threads) {
        th.join();
    }
    return rez;
}
```

or java equivalent

```
void computeOneElement(int[][] a, int[][] b,
    int row, int col,
    int[][] rez,
    RecursiveMutex mtx)
{
    mtx.lock();
    int sum = 0;
    for(int i=0 ; i<b.length ; ++i) {
        sum += a[row][i]*b[i][col];
    }
    rez[row][col] = sum;
    mtx.unlock();
}

int[][] matrixProd(int[][] a, int[][] b, int nrThreads)
{
    final RecursiveMutex mtx = new RecursiveMutex();
    final int outNrRows = a.length;
    final int outNrCols = b.get(0).length;
    int[][] rez = new int[][outNrRows];
    for(int i=0 ; i<rez.length : ++i) {
        rez[i] = new int[outNrCols];
    }
    int begin = 0;
    final int step = (outNrRows+nrThreads-1)/nrThreads; // statement 1
    Thread[] threads(nrThreads);
    for(int th=0 ; th<nrThreads ; ++th) {
        final int begin1 = begin;
        int end = begin+step; // statement 2
        final int end1 = end;
        threads[th] = new Thread(() -> {
            for(int i=begin1 ; i<end1 ; ++i) {
                for(int j=0 ; j<outNrCols ; ++j) {
                    computeOneElement(a, b, i, j, rez, mtx);
                }
            }
        });
        begin = end;
    }
    for(Thread th : threads) {
        th.join();
    }
    return rez;
}
```

Identify the issues with this code and how to fix them (fixes marked fix-A are to be considered only as far as the issues marked issue-A are concerned, and similarly for B)

Select one or more:

- [ ] [issue-B] The program will attempt to access non-existent elements
- [ ] [fix-A] Remove the mutex mtx and all references to it
- [ ] [issue-A] There is essentially no parallelism because no two threads can access the matrices at the same time
- [ ] [issue-B] There are elements of the output matrix that are not computed
- [ ] [fix-B] After *statement 2* add *if(end>a.size()) end=a.size()*
- [ ]

[issue-B] There are elements of the output matrix that are computed twice

- [ ] [issue-A] The result may be incorrect because of race conditions between the concurrent threads
- [ ] [fix-B] In *statement 1* put *step = outNrRows/nrThreads*
- [ ] [fix-B] In *statement 1* put *step = outNrRows/nrThreads* and after *statement 2* add *if(end>a.size()) end=a.size()*
- [ ] [fix-A] Make the output matrix *std::vector<std::vector<std::atomic<int>>>*

Question **4**

Not yet answered

Marked out of 3.00

We want a distributed program that computes the convolution of two vectors of equal length, that is, it computes a vector r with r[k] equal to the sum of all *a[i]*b[k-i]* for i from 0 to the length of the vectors minus one.

You shall implement two functions (in C++, Java or C\#):

vector<int> primes(vector<int> const& a, vector<int> const& b, int nrProcs);

that will run on the process 0 of the *MPI_COMM_WORLD* communicator, where *a* and *b* are the input vectors (assumed of equal length) and *nrProcs* is the
number of processes in the world communicator.

void worker(int myId, int nrProcs);

will run on all other processesin the world communicator, with *myId* representing the rank and *nrProcs* the number of processes.

Maximum size for new files: 2MB

Files

You can drag and drop files here to add them.

◄ Exam 2021-01-23

Jump to...