System and Device Programming
Standard Exam
22.02.2024

Ex 1 (1.5 points)

Analyze the following code snippet in C++. Indicate the possible output or outputs obtained by executing the program.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const int N = 3;

int main() {
  vector<string> v = {"this", "is", "a", "c++", "vector", "of", "strings", "!!!"};
  int n = count_if(v.begin(), v.end(), [](string s) {
    return (s.length()>N);
  });
  cout << n;
  return 0;
}
```

Choose one or more options:

1. ⬜ The program displays the value 4.
2. ⬜ The program displays the value 2.
3. ⬜ The program does not run as it contains a bug.
4. ✅ The program displays the value 3.
5. ⬜ The variable `s` in the lambda function is not defined.
6. ⬜ We must define the lambda function before the main program.

Ex 2 (1.5 points)

Analyze the following code snippet in C++. When the main is executed, indicate how many (standard) constructors, copy assignment operators, and destructors are called.

```
class C {
   private:
      ...
   public:
     ...
};

void f1(C e) { ... }
void f2(C &e) { ... }

int main() {
  C e1, e2;
  e2 = e1;
  C e3 = *new C;
  e2 = e3;
  return 0;
}
```

Solution
{1}[C][C]{2}[CAO]{3}[C][CC]{4}[CAO]{6}[D][D][D]

Choose one or more options:

1. ✅ 3 constructors, 2 copy assignment operators, and 3 destructors.
2. ⬜ 3 constructors, 3 copy assignment operators, and 3 destructors.
3. ⬜ 3 constructors, 3 copy assignment operators, and 4 destructors.
4. ⬜ 2 constructors, 3 copy assignment operators, and 3 destructors.
5. ⬜ 2 constructors, 2 copy assignment operators, and 3 destructors.
6. ⬜ 3 constructors, 2 copy assignment operators, and 4 destructors.

Ex 3 (1.5 points)

Analyze the following code snippet. Indicate which of the following statements are correct. Note that wrong answers imply a penalty in the final score.

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main() {
  map<string, int> mp;

  mp["one"] = 1;
  mp["two"] = 2;
  mp["three"] = 3;
  map<string, int>::iterator it = mp.begin();
  while (it != mp.end()) {
    cout << it->first << " " << it->second << " ";
    ++it;
  }

  return 0;
}
```

Choose one or more options:

1. ✅ The program displays the following sequence: one 1 three 3 two 2
2. ⬜ The program displays the following sequence: one 1 two 2 three 3
3. ⬜ The program displays the following sequence: 1 one 3 three 2 two
4. ⬜ The program displays the following sequence: 1 one 2 two 3 three
5. ⬜ The program displays the following sequence: three 3 twp 2 one 1
6. ⬜ The program displays the following sequence: 3 three 2 two 1 one
7. ⬜ The program displays a sequence different from all the reported ones.

Ex 4 (2.0 points)

Illustrate the limitations of multi-threading in C++ and describe how to perform task processing. Indicate how to run a task and describe the main running policies. Report one example to illustrate the use of futures and promises, describing the meaning of the main construct and the main features of the strategy.

Solution

TO BE DONE

Ex 5 (3.25 points)

The following function computes the Fibonacci number of position `num` recursively:

```
int Fibonacci(int num){
  if (num < 2)
    return 1;
  else
    return Fibonacci(num-1) + Fibonacci(num-2);
}
```

Write a complete concurrent C++ program, producing the same result, using tasks, futures, and promises. In particular, implement a program properly replacing the recursive calls to the Fibonacci function with new task activations.

Solution

```
#include <iostream>
#include <future>

using namespace std;

// Define the function for computing the Fibonacci number
int Fibonacci(int num) {
    if (num < 2)
        return 1;
    else {
        // Create two new async tasks to compute the Fibonacci numbers
        auto future1 = async(launch::async, Fibonacci, num-1);
        auto future2 = async(launch::async, Fibonacci, num-2);
        // Wait for the results of the async tasks and return their sum
        return future1.get() + future2.get();
    }
}

int main(int argc, char *argv[]) {
    int num = atoi(argv[1]);
    int result = Fibonacci(num-1);
    cout << "The Fibonacci number of position " << num << " is "
         << result << endl;
    return 0;
}
```

Ex 6 (3.25 points)

Implement a program in C++ running three threads:

- The first thread loops indefinitely and, for each iteration, generates an atom of Chlorine (Cl); a new atom is created randomly in a time range varying from 0 to 5 seconds.
- The second thread loops indefinitely and, for each iteration, generates an atom of Sodium (Na); a new atom is created randomly in a time range varying from 0 to 5 seconds.
- The third thread produces a sodium chloride molecule (NaCl) whenever a Na atom and a Cl atom are available.

Once a molecule of NaCl is created, the entire process restarts. Use semaphores, mutex, and, eventually, condition variables, to synchronize the three threads:

Solution

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <random>
#include <condition_variable>

using namespace std;
```

```
// Define the maximum time range for atom generation (in seconds)
const int MAX_TIME_RANGE = 5;

// Define the semaphores and mutexes for the atoms and molecules
mutex sodium_mutex, chlorine_mutex, molecule_mutex;
condition_variable molecule_cv;
int sodium_count = 0, chlorine_count = 0;

// Define the functions for generating atoms and molecules
void generate_sodium(mt19937 &rng) {
    std::uniform_int_distribution<unsigned int> dist(0, MAX_TIME_RANGE);

    while (true) {
        // Generate a new sodium atom after a random time interval
        this_thread::sleep_for(chrono::seconds(dist(rng)));
        {
            lock_guard<mutex> lock(sodium_mutex);
            sodium_count++;
            cout << "Generated a sodium atom" << endl;
        }
        // Check if a molecule can be created
        molecule_cv.notify_all();
        // Wait for the molecule to be created
        unique_lock<mutex> lock(molecule_mutex);
        molecule_cv.wait(lock, []{ return sodium_count < 1; });
    }
}

void generate_chlorine(mt19937 &rng) {
    std::uniform_int_distribution<unsigned int> dist(0, MAX_TIME_RANGE);

    while (true) {
        // Generate a new chlorine atom after a random time interval
        this_thread::sleep_for(chrono::seconds(dist(rng)));
        {
            lock_guard<mutex> lock(chlorine_mutex);
            chlorine_count++;
            cout << "Generated a chlorine atom" << endl;
        }
        // Check if a molecule can be created
        molecule_cv.notify_all();
        // Wait for the molecule to be created
        unique_lock<mutex> lock(molecule_mutex);
        molecule_cv.wait(lock, []{ return chlorine_count < 1; });
    }
}

void create_molecule() {
    while (true) {
        // Wait until there are enough atoms to create a molecule
        unique_lock<mutex> lock(molecule_mutex);
        molecule_cv.wait(lock, []{ return sodium_count >= 1 && chlorine_count >= 1;
});
        // Create a molecule by consuming the atoms
        {
            lock_guard<mutex> lock_s(sodium_mutex);
            lock_guard<mutex> lock_c(chlorine_mutex);
            sodium_count -= 1;
            chlorine_count -= 1;
            cout << "Created a sodium chloride molecule" << endl;
        }
        // Notify the atom-generating threads that the molecule has been created
        molecule_cv.notify_all();
    }
```

```
}

int main() {
    // Seed the random number generator
    std::random_device rd;
    std::mt19937 mt(rd());
    // Create the threads for generating atoms and creating molecules
    thread t1(generate_sodium, ref(mt));
    thread t2(generate_chlorine, ref(mt));
    thread t3(create_molecule);
    // Wait for the threads to finish (which will never happen)
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

Ex 7 (2.0 points)

Indicate the main strategy to perform IO multiplexing in C++ and compare it with all possible alternatives to obtain similar implementations. Report an example describing those techniques, reporting the advantages and disadvantages of each strategy.

Solution

TO BE DONE