```
1  #include <iostream>
2  #include <shared_mutex>
3  #include <vector>
4  #include <thread>
5  #include <chrono>
6
7  #include <condition_variable>
8
9  /*4. Un sistema embedded riceve su due porte seriali sequenze di dati provenienti da
   due diversi sensori.
10  * Ciascun sensore genera i dati con cadenze variabili nel tempo e non predicibili,
   in quanto il processo di
11  * digitalizzazione al suo interno può richiedere più o meno tempo in funzione del
   dato letto. Ogni volta che
12  * il sistema riceve un nuovo valore su una delle due porte seriali, deve
   accoppiarlo con il dato più recente
13  * ricevuto sull'altra (se già presente) e inviarlo ad una fase successiva di
   computazione. Il sistema al
14  * proprio interno utilizza due thread differenti per leggere dalle due porte
   seriali e richiede l'uso di un
15  * oggetto di sincronizzazione in grado di implementare la logica descritta sopra.
   Tale oggetto offre la
16  * seguente interfaccia pubblica:
17
18 class Synchronizer {
19 public:
20  Synchronizer(std::function<void(float d1, float d2)> process);
21  void dataFromFirstPort(float d1);
22  void dataFromSecondPort(float d2);
23 }
24
25 All'atto della costruzione, viene fornita la funzione process(...) che rappresenta
   la fase successiva della
26  computazione. Quando vengono invocati i metodi dataFromFirstPort(...) o
   dataFromSecondPort(...), se non è
27  ancora presente il dato dalla porta opposta, questi si bloccano al proprio interno
   senza consumare CPU,
28  in attesa del valore corrispondente. Al suo arrivo, viene invocata una sola volta
   la funzione process(...).
29  Si implementi tale classe utilizzando le funzionalità offerte dallo standard C++.
30 */
31
32 class Synchronizer{
33
34    std::mutex m;
35    std::condition_variable cv;
36    std::function<void(float, float) f;
37
38
39    float dataPort1;
40    bool isPresentPort1;
41    float dataPort2;
42    bool isPresentPort2;
43
44
45 public:
46    Synchronizer(std::function<void(float d1, float d2)> process):f(process),
   isPresentPort1(false), isPresentPort2(false){};
47
48    void dataFromFirstPort(float d1){
```

```
49          std::unique_lock<std::mutex> l(m);
50          cv.wait(l, [this](){return !isPresentPort1;});
51
52          dataPort1 = d1;
53          isPresentPort1 = true;
54          cv.notify_all();
55
56          cv.wait(l, [this](){return isPresentPort2;});
57          isPresentPort1 = false; isPresentPort2 = false;
58          cv.notify_all(); //si dice cosi a port2 di rinizializzare ispresport2 a false
59          l.unlock(); //unlock qui e non a chiusura scope ()})
60          ff(dataPort1,dataPort2); //richiamo funziona (consumer)
61      }
62
63      void dataFromSecondPort(float d2){
64          std::unique_lock<std::mutex> l(m);
65          cv.wait(l, [this](){return !isPresentPort2;};
66          dataPort2 = d2;
67          isPresentPort2 = true;
68          cv.notify_all();
69          cv.wait(l, [this](){return isPresentPort1;};
70          isPresentPort2 = false;
71          cv.notify_all();
72      }
73
74  }
75
76
77  int main(){
78
79      using namespace std::chrono_literals;
80
81      Synchronizer sync([&](float d1,float d2){
82          std::cout << "Elaboro i dati d1: " << d1 << " d2: " << d2 << std::endl;
83      });
84
85      std::thread t1([&](){
86
87          for( int i = 1 ; i < 5 ; i++) {
88
89              sync.dataFromFirstPort(static_cast<float>(i) * 2.5);
90              std::this_thread::sleep_for(2000ms);
91
92
93          }
94      });
95
96      std::thread t2([&](){
97
98          for( int i = 1 ; i < 5 ; i++)
99              sync.dataFromSecondPort( static_cast<float>(i) );
100
101     });
102
103
104
105     if ( t1.joinable()) t1.join();
106     if ( t2.joinable()) t2.join();
107
108     return 0;
```

```
109 }
```