SEMINAR 3

C++
std::call_once
std::once_flag

Producer:
```
v = ...
result.setValue(v)
```

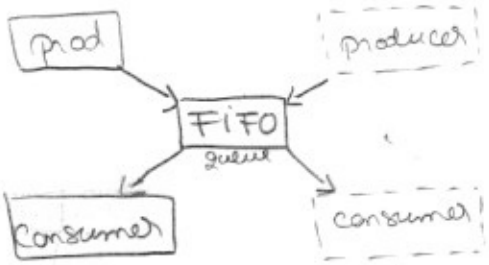
Consumer1:
```
x = result.getValue()
use(x)
```

Consumer2:
```
x = result.getValue()
use(x)
```

```
template<typename T>
class Future{
  public:
    void setValue(T v);
    T getValue();
  private:
    T value;
    condition_variable cond;
    mutex mtx;
    bool hasValue; // value can be used instead, if a special "null" value exists
}

setValue(T v){
    unique_lock<mutex> lock(mtx);
    if(hasValue) return; //dacă îi deja setată valoarea
    value = v;
    hasValue = true;
    cond.notify_all();
} //the destructor of the function destroys the mutex mtx (when the block ends)
//wait/unlocks the mutex, otherwise there will be a deadlock.

T getValue(){
    unique_lock<mutex> lock(mtx);
    while(!hasValue){
        cond.wait(lock);
    }
    return value;
}
```



Multiple producers, multiple consumers.
When a producer closes, everything closes
(simplest approach)

```
template <typename T>
class Queue {
public:
    void enqueue(T v);
    bool dequeue(T&);
    void close();
private:
    std::queue<T> q;
    mutex mtx;
    condition_variable condEmpty;
    condition_variable condFull;
    bool isClosed;
    size_t maxSize;
};
```

- if ∃ at least 1 element, returns the top element + return true and removes it.
- otherwise, if the queue is closed, returns false,
- otherwise, it blocks

producer:
```
for(int i=0; i<1000; ++i){
    q.enqueue(i);
}
q.close();
```

consumer:
```
int s=0;
while(...){  !q.isClosed() varianta
    s+=q.dequeue();
}
cout<<s;
```
⇓
```
int s=0;
int tmp;
while(dequeue(tmp)){
    s+=tmp;
}
cout<<s;
```

```
bool dequeue(T& v){
    unique_lock<mutex> lock(mtx);
    while(true){
        if(!q.empty()){
            v=q.front();
            q.pop();
            return false;
        }
        if(isClosed) return false;
        condEmpty.wait(lock);
    }
}
```
condFull.notify_one(); (for version 2)

```
void close(){
    unique_lock<mutex> lock(mtx);
    isClosed=true;
    condEmpty.notify_all();
}
```

Varianta 1:
```
void enqueue(T v){
    unique_lock<mutex> lock(mtx);
    q.push(v);
    condEmpty.notify_one();
}
```

Varianta 2:
```
void enqueue(T v){
    unique_lock<mutex> lock(mtx);
    while(q.size()>=maxSize){
        condFull.wait(lock);
    }
    q.push(v);
    condEmpty.notify_one();
}
```