CONDITION VARIABLE

CODERS SCHOOL

BARTOSZ SZURGOT

ŁUKASZ ZIOBROŃ

AGENDA

1. Thread-safe queue
2. `std::condition_variable`
3. How does it work?
4. Spurious wake-ups
5. Lost notifications
6. Recap & Homework

SOMETHING ABOUT YOU

- Your experience in C++ in years
- Have you ever used condition variable?

ŁUKASZ ZIOBROŃ

EXPERIENCE NOT ONLY IN PROGRAMMING

- Trainer and DevOps @ Coders School
- C++ and Python developer @ Nokia and Credit Suisse
- Team leader and trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C++ courses @ Coders School
- Practical Aspects of Software Engineering @ PWr & UWr
- Nokia Academy
- Internal corporate training

EXPERIENCE IN PUBLIC SPEAKING

- code::dive conference
- code::dive community
- Polish Academic Championship in Team Programming

INTERESTS

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Marketing
- Andragogy

RULES

- 🗣️ Ask whenever you wish
- ⌨️ Be active
- ⌚ Stay till the end

PRE-TEST 🤯

```cpp
// assume all necessary includes are here

int main() {
    std::mutex m;
    std::condition_variable cv;
    std::vector<int> v;
    std::vector<std::thread> producers;
    std::vector<std::thread> consumers;

    auto consume = [&] {
        std::unique_lock<std::mutex> ul(m);
        cv.wait(ul);
        std::cout << v.back();
        v.pop_back();
    };
    for (int i = 0; i < 10; i++) consumers.emplace_back(consume);

    auto produce = [&](int i) {
        {
            std::lock_guard<std::mutex> lg(m);
            v.push_back(i);
        }
        cv.notify_all();
    };
    for (int i = 0; i < 10; i++) producers.emplace_back(produce, i);

    for (auto && p : producers) p.join();
    for (auto && c : consumers) c.join();
}
```

1. there may be an Undefined Behavior in this code
2. the output is guaranteed to always be `0123456789`
3. `v` is always an empty vector at the end of this program
4. if some producers threads started before some consumers, we would have a deadlock because of lost notifications
5. this code can be improved by providing a predicate to `wait()` to disallow getting elements when the vector is empty
6. a change from `notify_all()` to `notify_one()` + `wait()` with predicate will guarantee that each consumer thread will receive a different number

THREAD-SAFE QUEUE

CODERS SCHOOL

EXERCISE: FIFO QUEUE

```cpp
template <typename T>
class ThreadsafeQueue {
    deque<T> queue_;
    // TODO: Make it thread-safe

public:
    void push(const T & element) {
        queue_.push_front(element);
    }
    T pop() {
        auto top = queue_.back();
        queue_.pop_back();
        return top;
    }
    bool empty() const {
        return queue_.empty();
    }
};
```

- In `exercises/01_threadsafe_queue.cpp` there is a FIFO queue implementation
- Fix the `textProducer` thread to generate correct strings:
  - This is random text number 0
  - This is random text number 1
  - ...
  - This is random text number n
- Prevent queue operations from being accessed by multiple threads at the same time (make it thread-safe 🙂)
- What kind of problem can you see?

PRODUCETEXT - PROBLEM

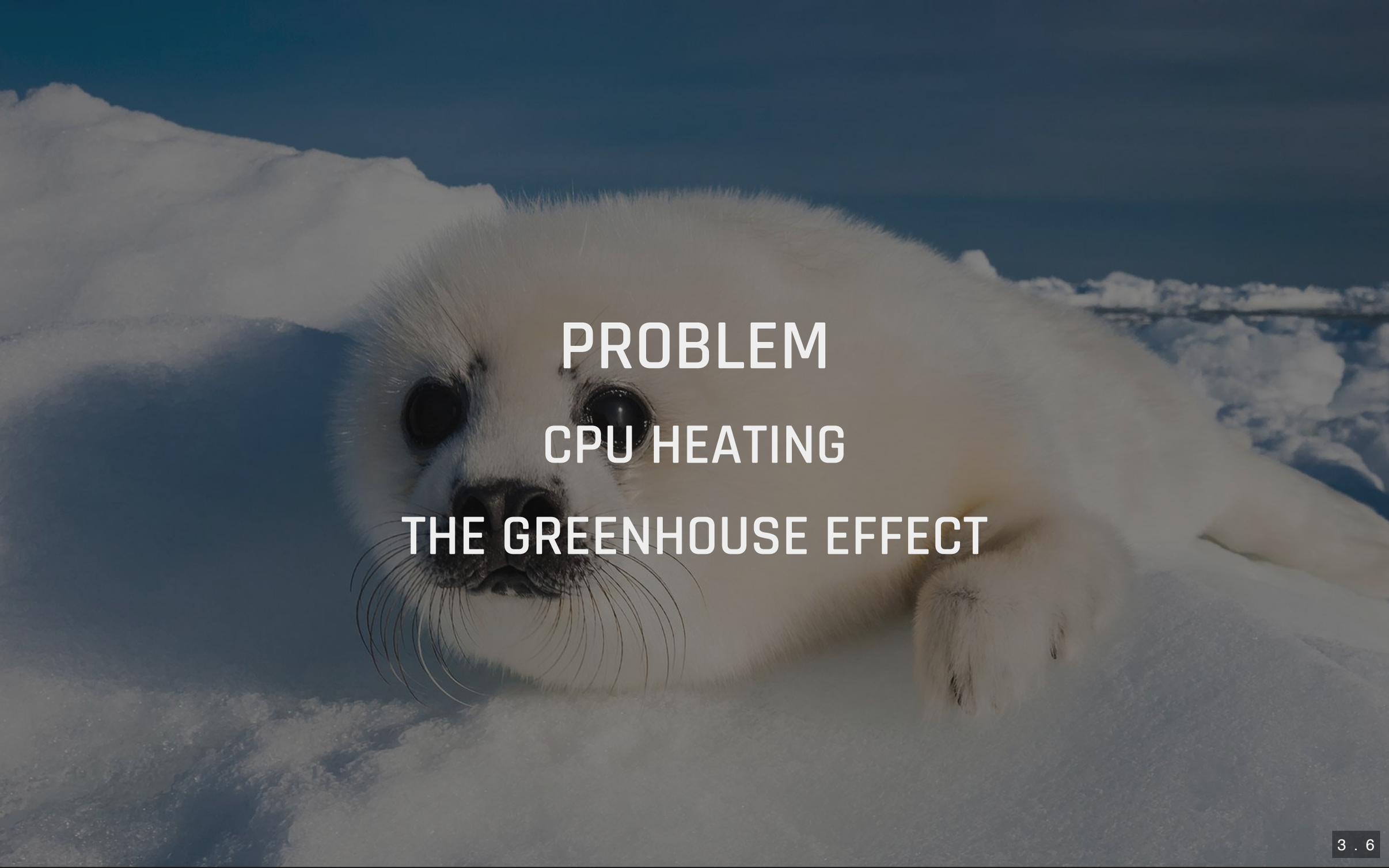
```
1 void produceText(StringQueue & sq, int number) {
2     for (int i = 0; i < number; i++)
3         sq.push("This is random text number " + i);
4 }
```

PRODUCETEXT - SOLUTION

```
1 void produceText(StringQueue & sq, int number) {
2     for (int i = 0; i < number; i++)
3         sq.push("This is random text number " + to_string(i));
4 }
```

SOLUTION: THREAD-SAFE QUEUE

```
1  template <typename T>
2  class ThreadsafeQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      using Lock = lock_guard<mutex>;
6
7  public:
8      void push(const T & element) {
9          Lock l(m_);
10         queue_.push_front(element);
11     }
12     T pop() {
13         Lock l(m_);
14         auto top = queue_.back();
15         queue_.pop_back();
16         return top;
17     }
18     bool empty() const {
19         Lock l(m_);
20         return queue_.empty();
21     }
22 };
```



PROBLEM

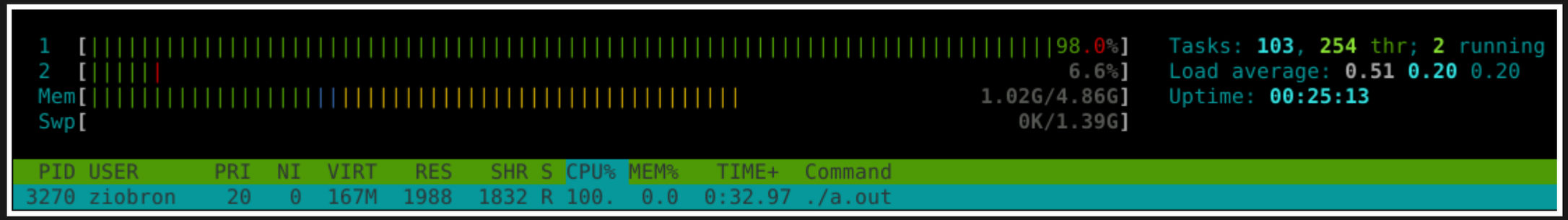
CPU HEATING

THE GREENHOUSE EFFECT

SPINLOCK

```
1 void saveToFile(StringQueue & sq) {
2     ofstream file("/tmp/sth.txt");
3     while (file) {
4         while (sq.empty()) { /* nop */ }
5         file << sq.pop() << endl;
6     }
7 }
```

- Spinlock is a state in which the thread is repeatedly checking if a certain condition has been met
- Another name for this problem is "busy waiting"
- The problem can be solved by the condition variable



CONDITION VARIABLE

CODERS SCHOOL

CONDITION VARIABLE

- `#include <condition_variable>`
- `std::condition_variable`
- Key operations
  - `wait()` - waiting for a change - blocks the current thread until it receives a notification
  - `notify_one()` - awakens one of the threads waiting to change. We have no control over which thread is notified.
  - `notify_all()` - awakens all threads waiting for change. These threads may compete for resources.

EXERCISE: ARCTIC FRIENDLY FIFO QUEUE

Correct the code from `exercises/02_wait_queue.cpp` file. A condition variable should be used instead of busy waiting.

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      using Lock = lock_guard<mutex>;
6  public:
7      void push(const T & element) {
8          Lock l(m_);
9          queue_.push_front(element);
10     }
11     T pop() {
12         Lock l(m_);
13         auto top = queue_.back();
14         queue_.pop_back();
15         return top;
16     }
17     bool empty() const {
18         Lock l(m_);
19         return queue_.empty();
20     }
21 };
```

EXERCISE - SOLUTION

```
1  // includes
2  template <typename T>
3  class WaitQueue {
4      deque<T> queue_;
5      mutable mutex m_;
6      condition_variable nonEmpty_;
7      using Lock = unique_lock<mutex>;
8
9  public:
10     void push(const T & element) {
11         Lock l(m_);
12         queue_.push_front(element);
13         nonEmpty_.notify_all();
14     }
15     T pop() {
16         Lock l(m_);
17         auto hasData = [&]{ return not queue_
18         nonEmpty_.wait(l, hasData);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```

```
1  using StringQueue = WaitQueue<string>;
2
3  void provideData(StringQueue & sq) {
4      string txt;
5      while (getline(cin, txt))
6          sq.push(txt);
7  }
8
9  void saveToFile(StringQueue & sq) {
10     ofstream file("/tmp/sth.txt");
11     while (file)
12         file << sq.pop() << endl;
13 }
14
15 void produceText(StringQueue & sq, int number
16     for (int i = 0; i < number; i++)
17         sq.push("This is random text number "
18                 to_string(i));
19 }
20
21 int main() {
22     // without changes
23 }
```

SEALS ARE HAPPY 🦭

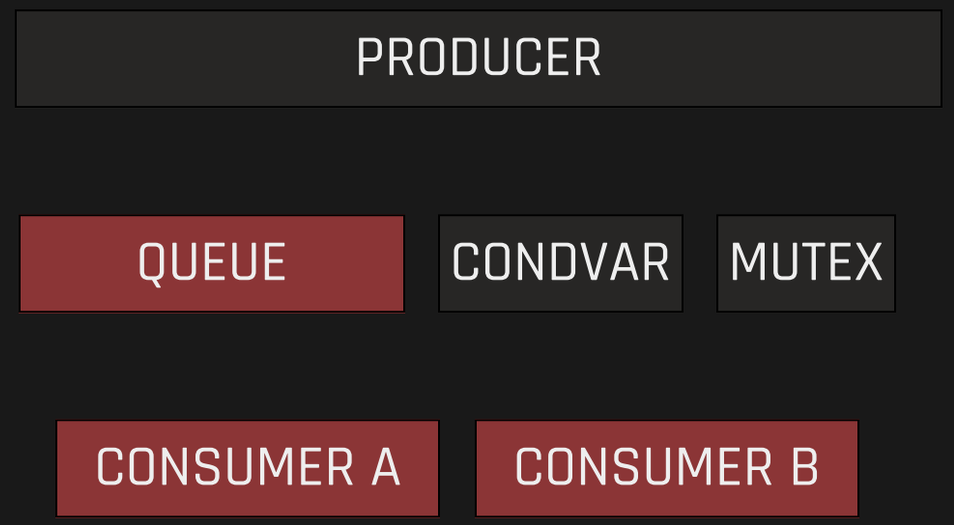
THANKS CONDVAR!

HOW EXACTLY DOES IT WORK?

CODERS SCHOOL

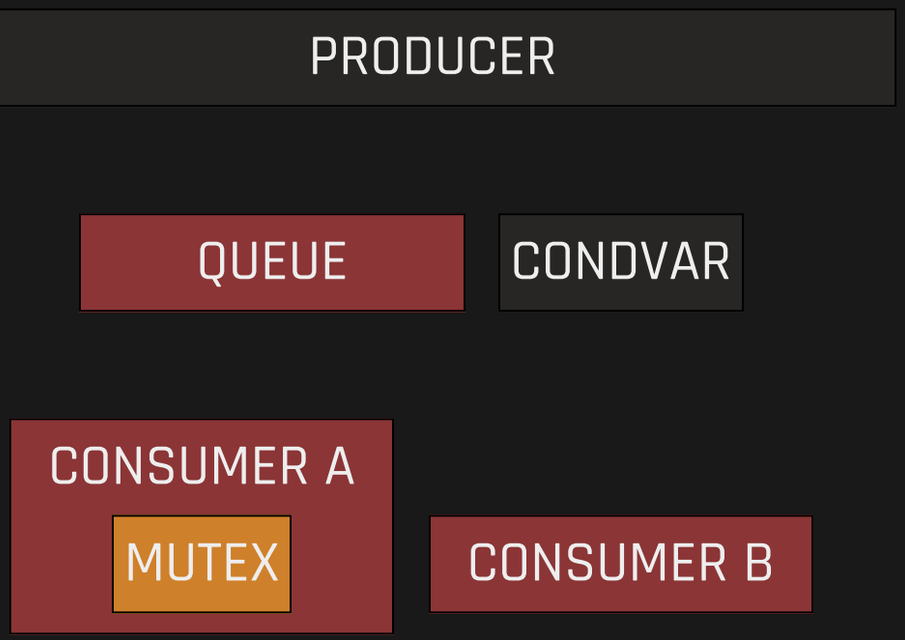
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



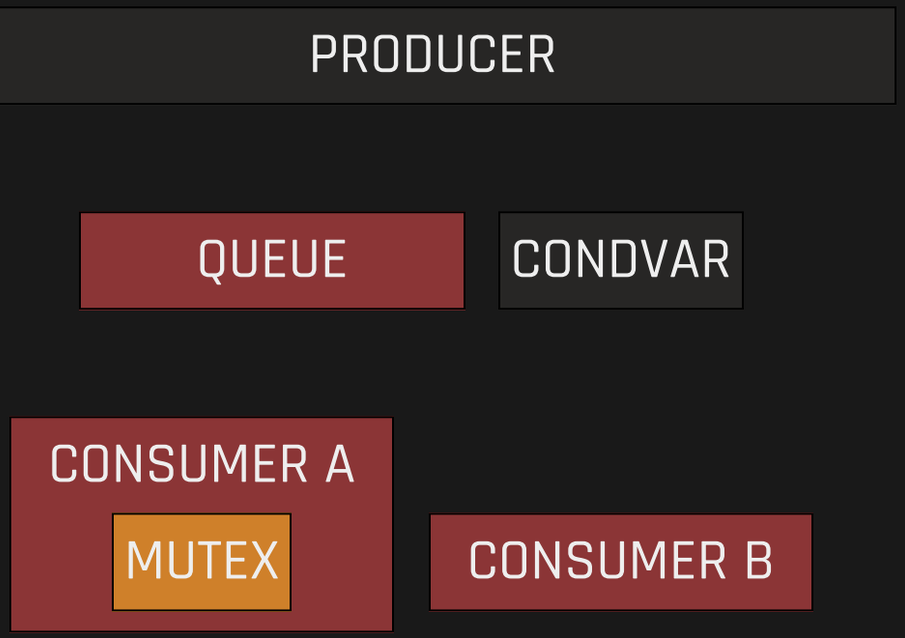
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



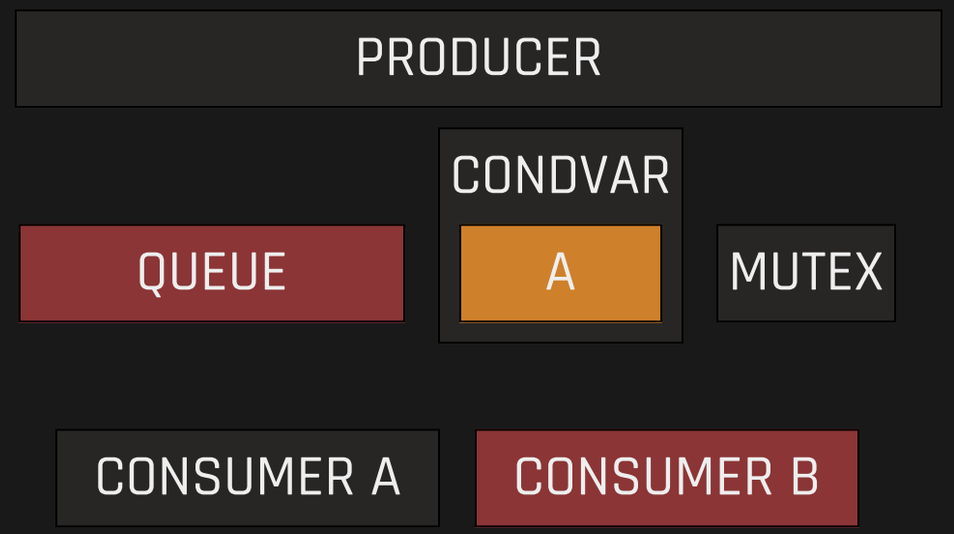
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



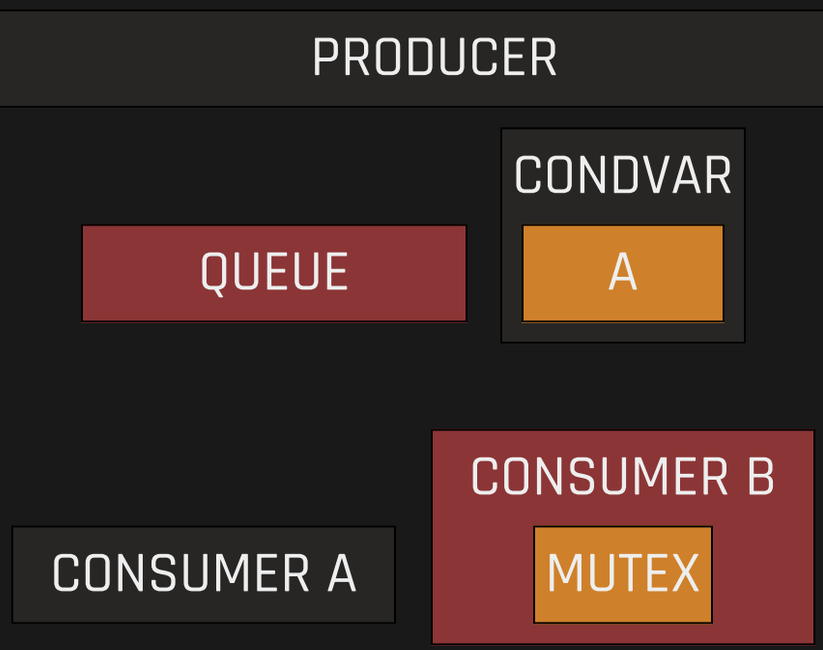
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



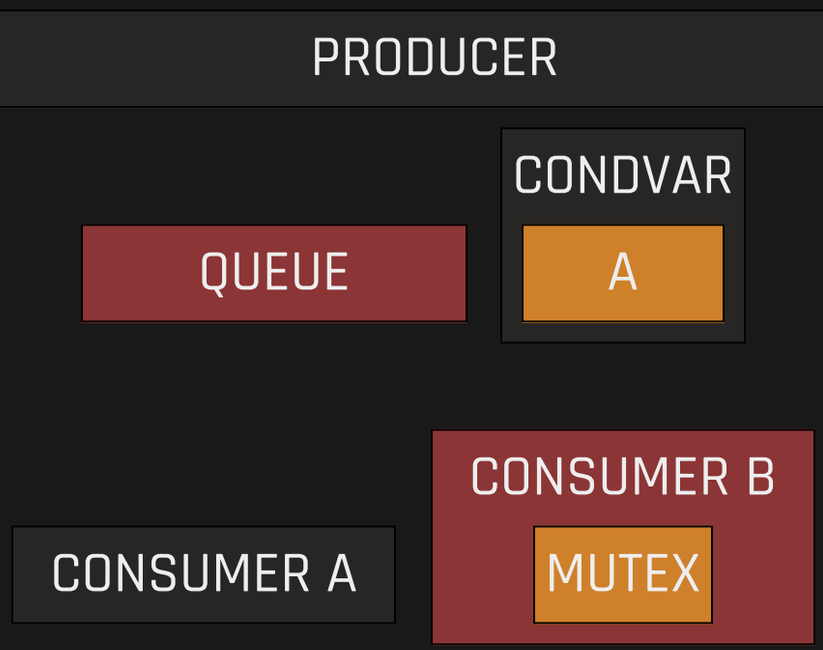
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



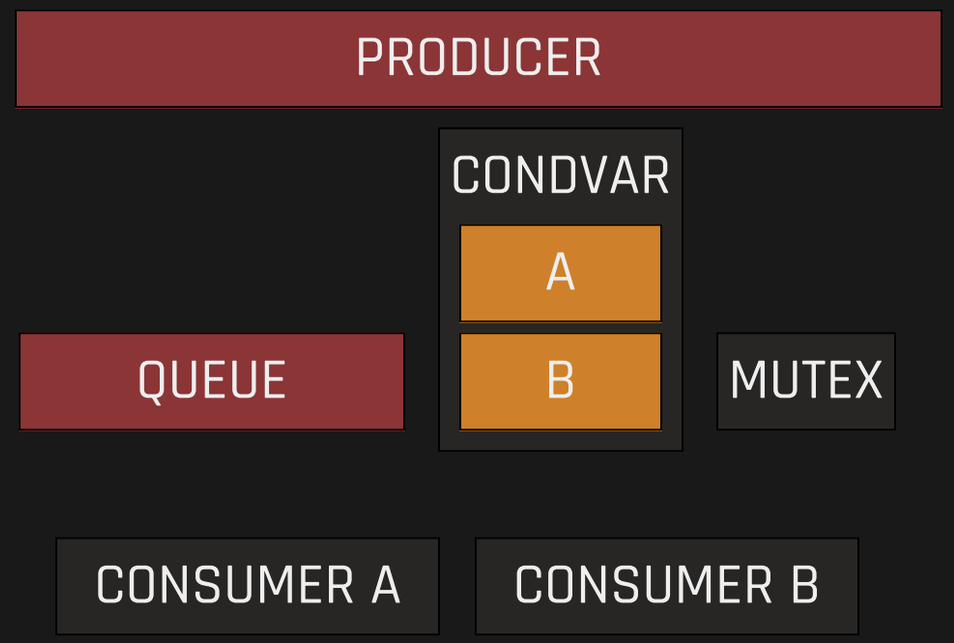
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



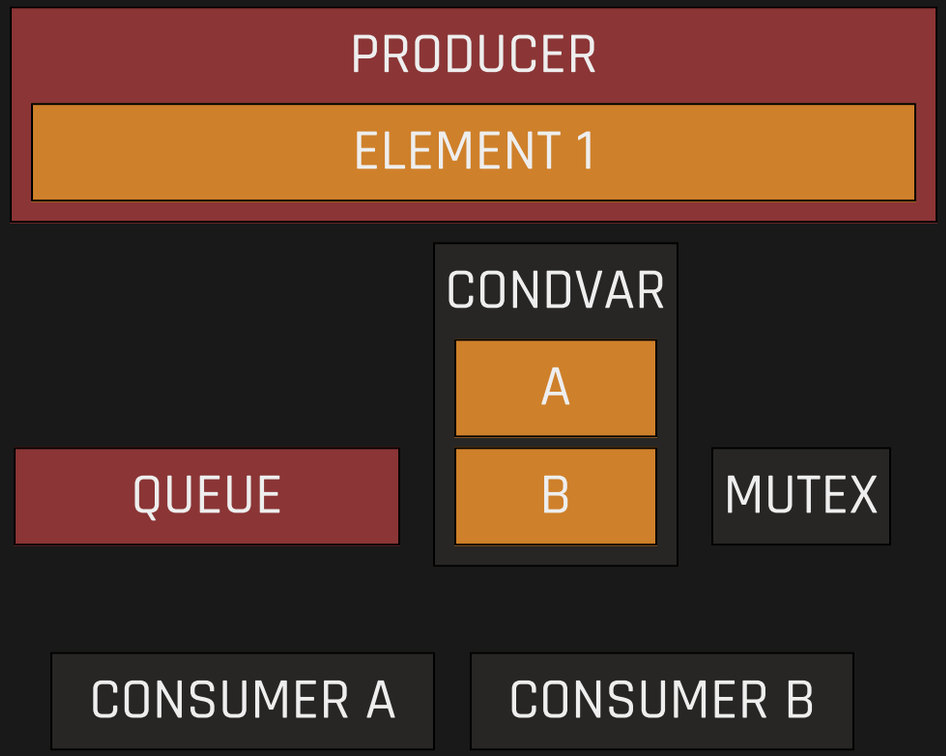
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



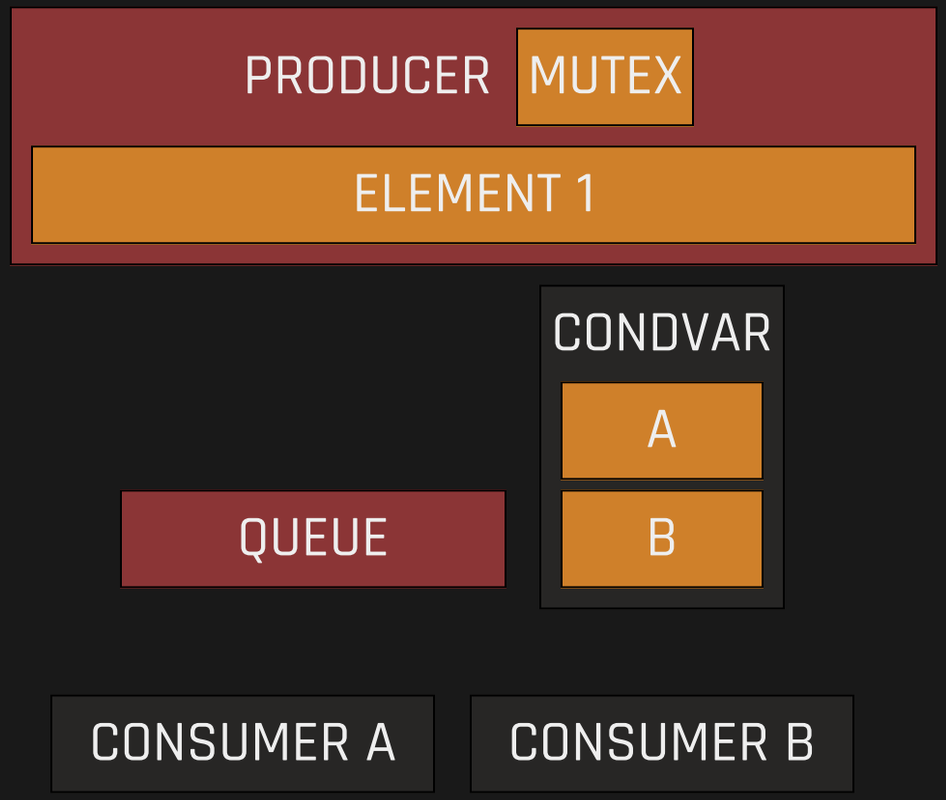
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



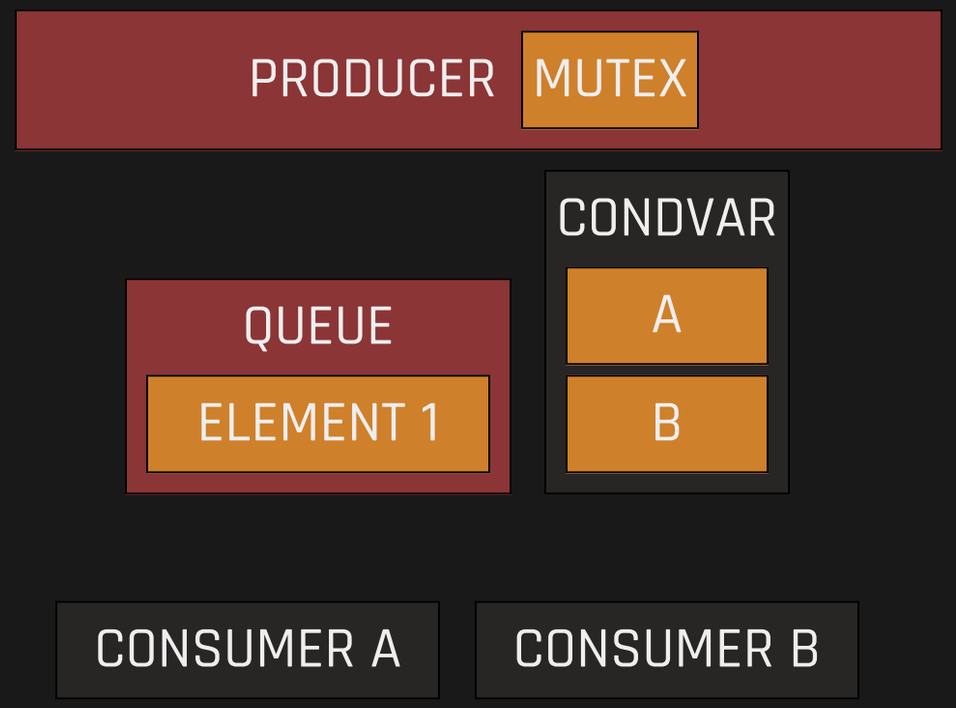
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



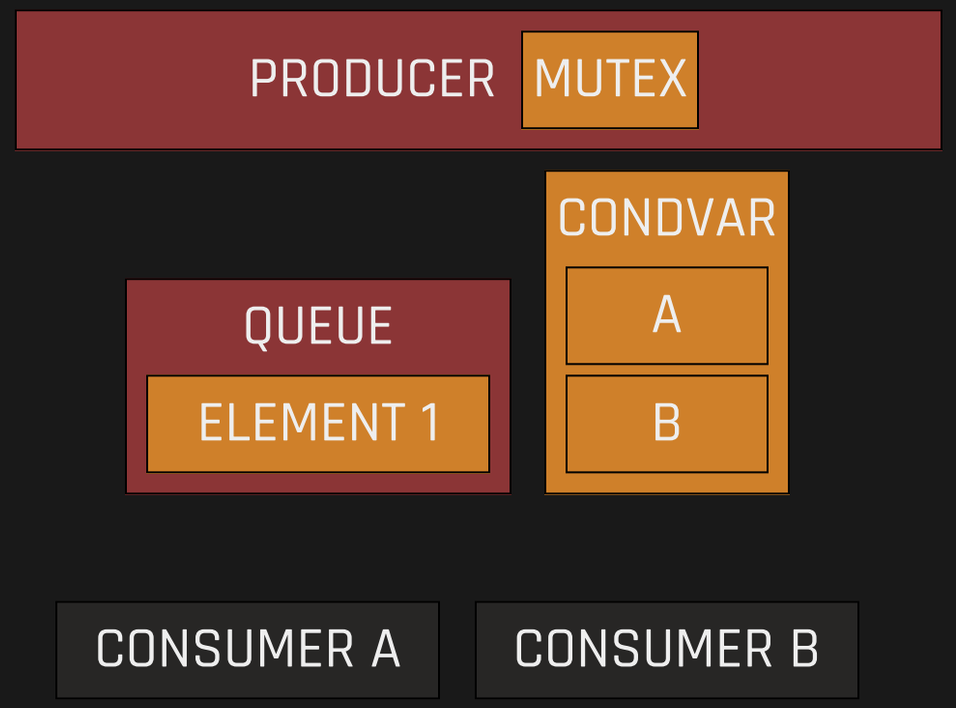
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



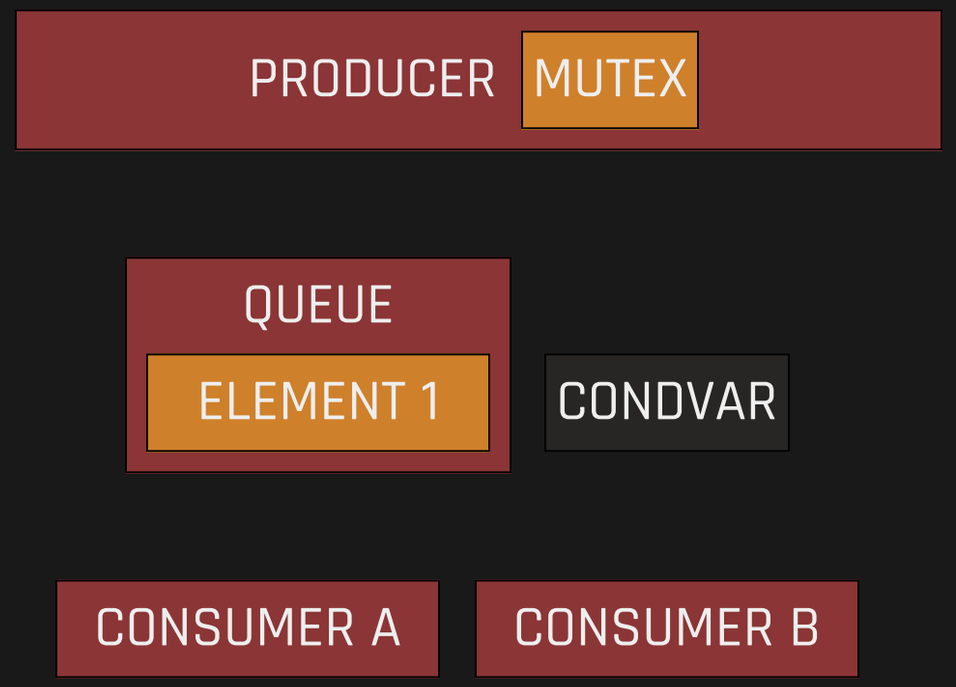
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



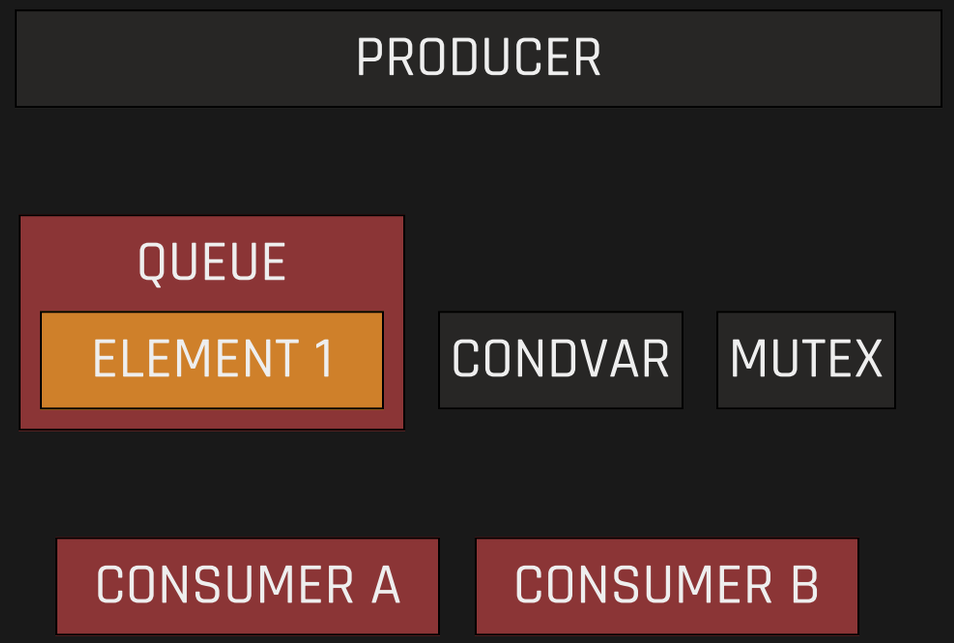
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



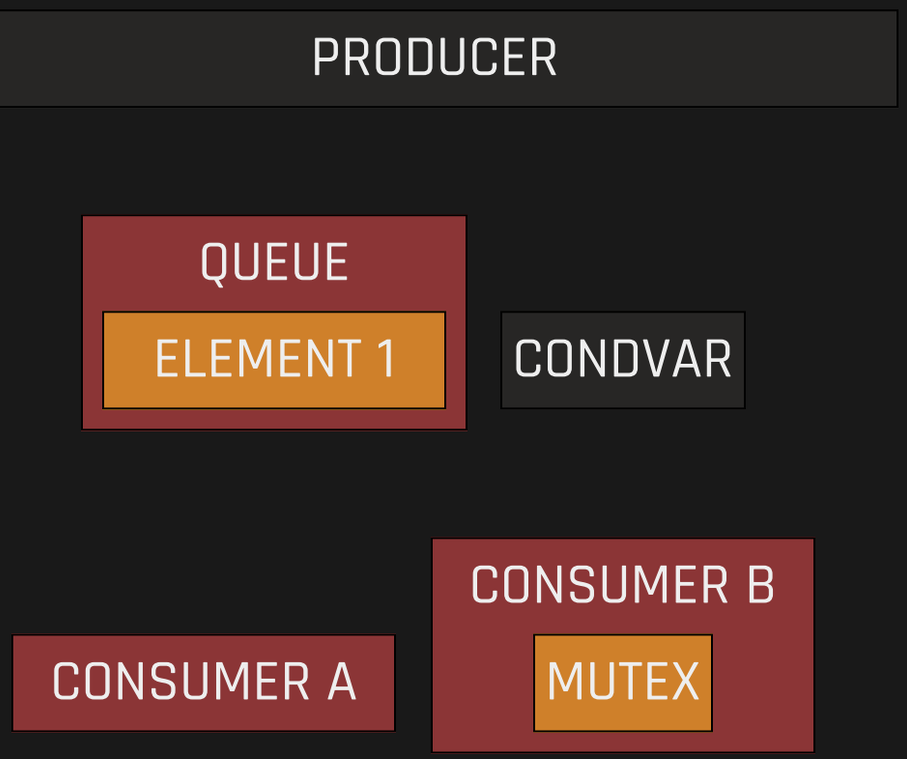
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



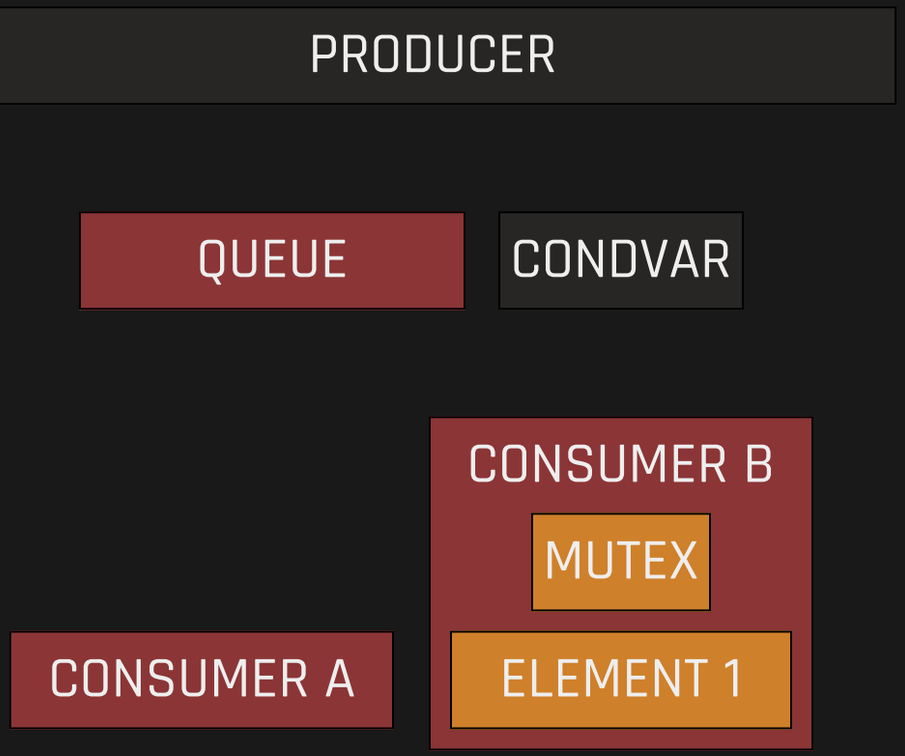
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



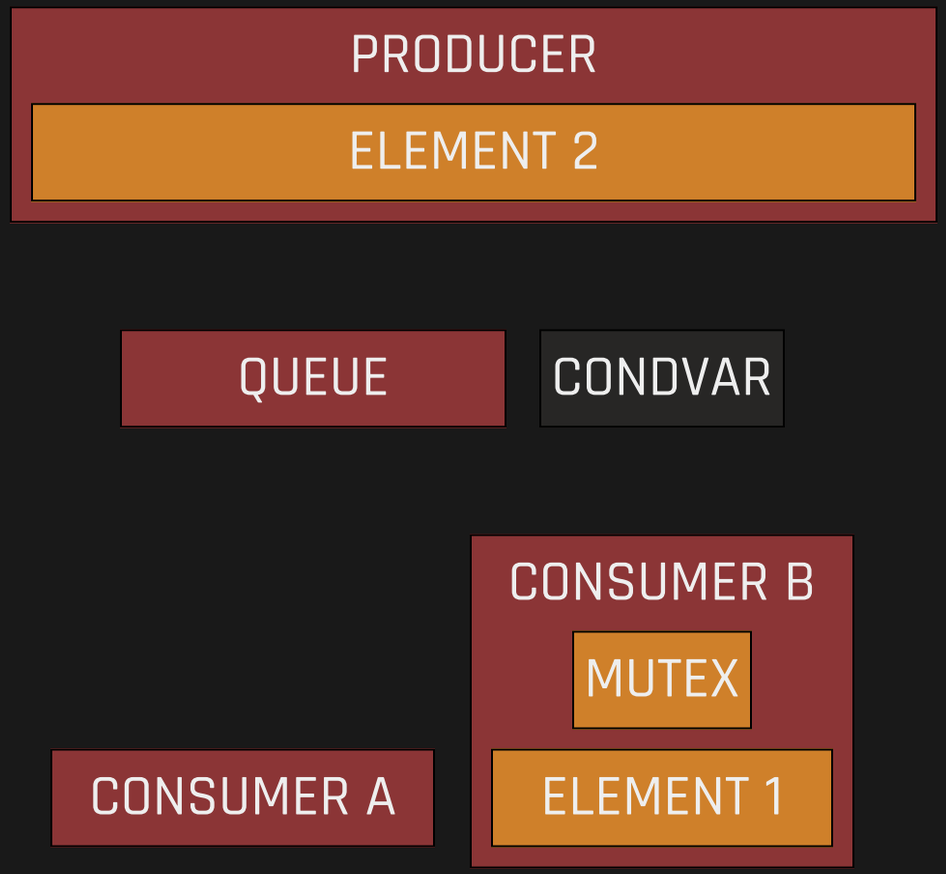
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



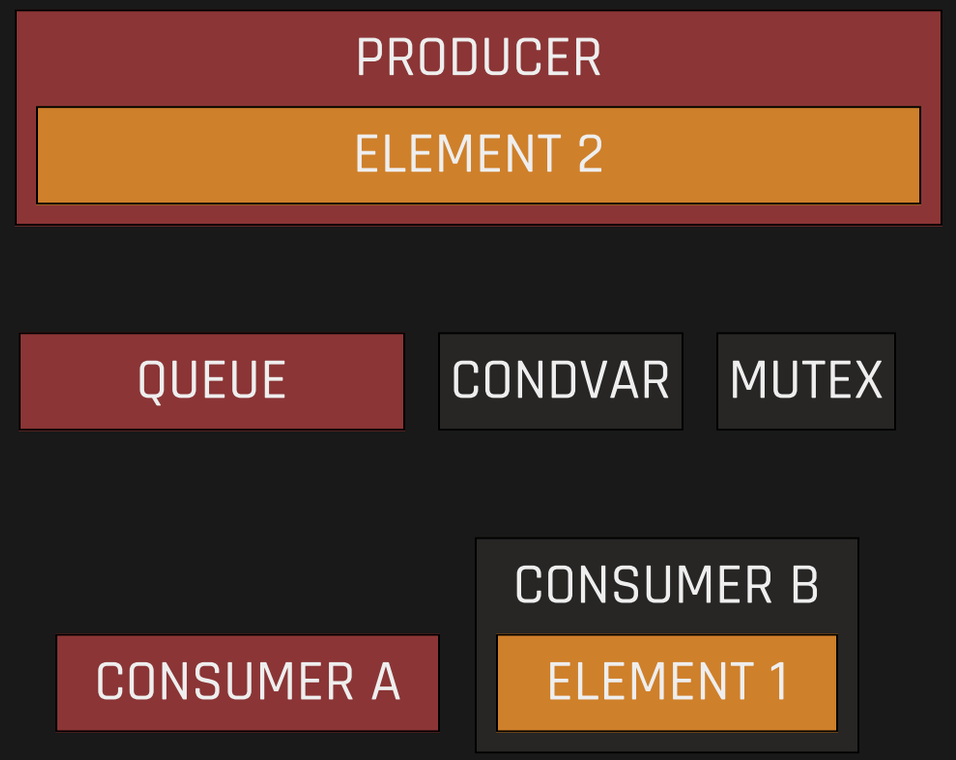
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



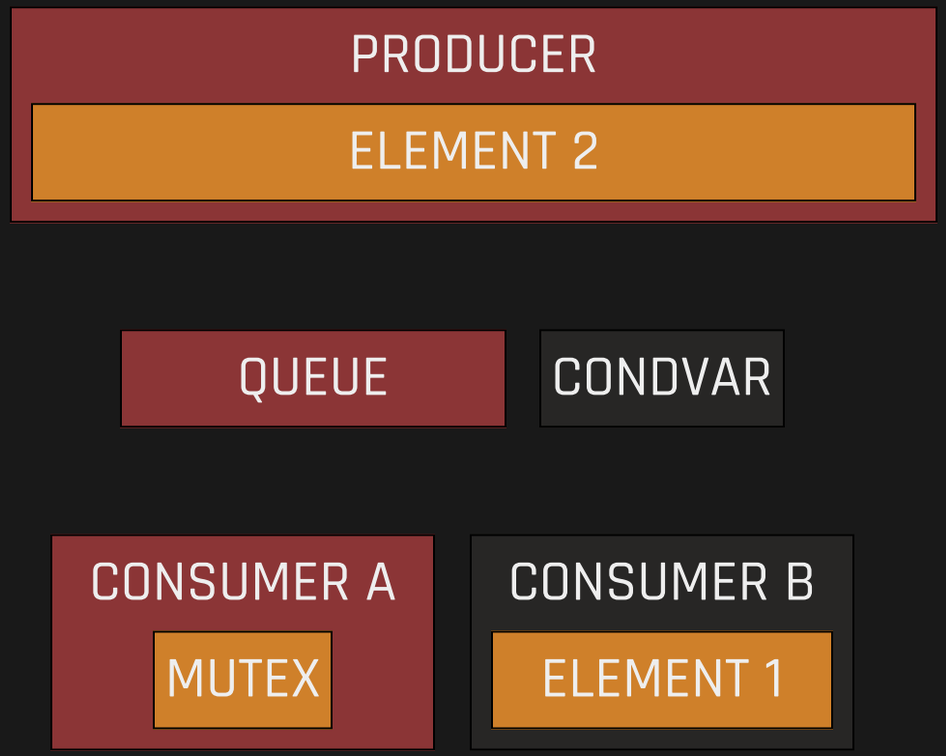
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



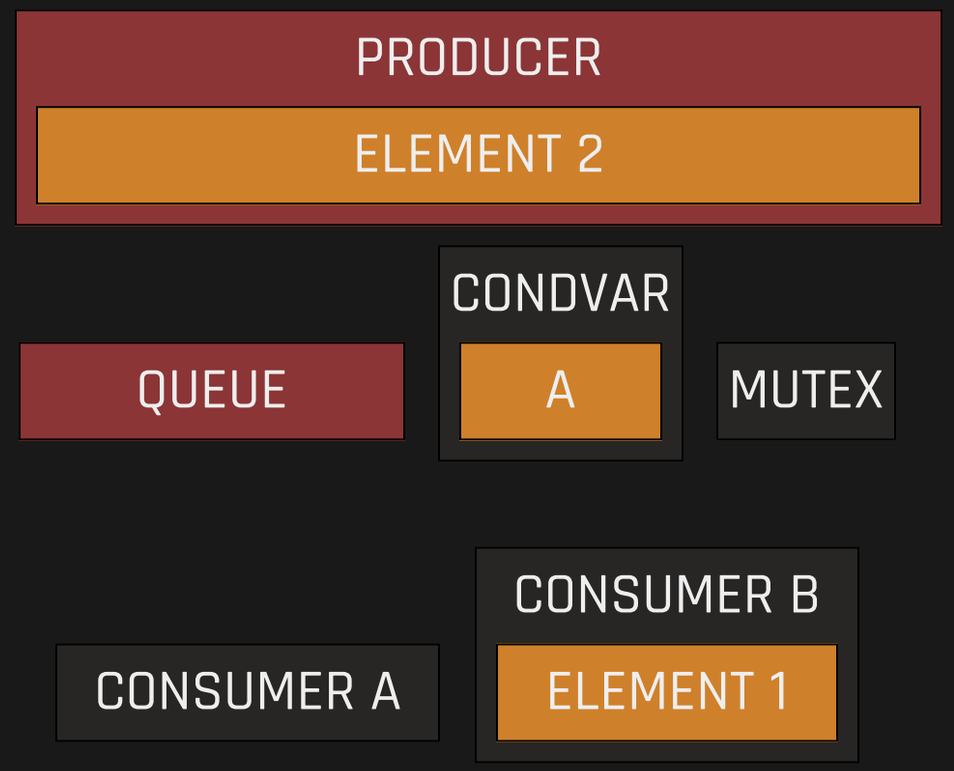
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



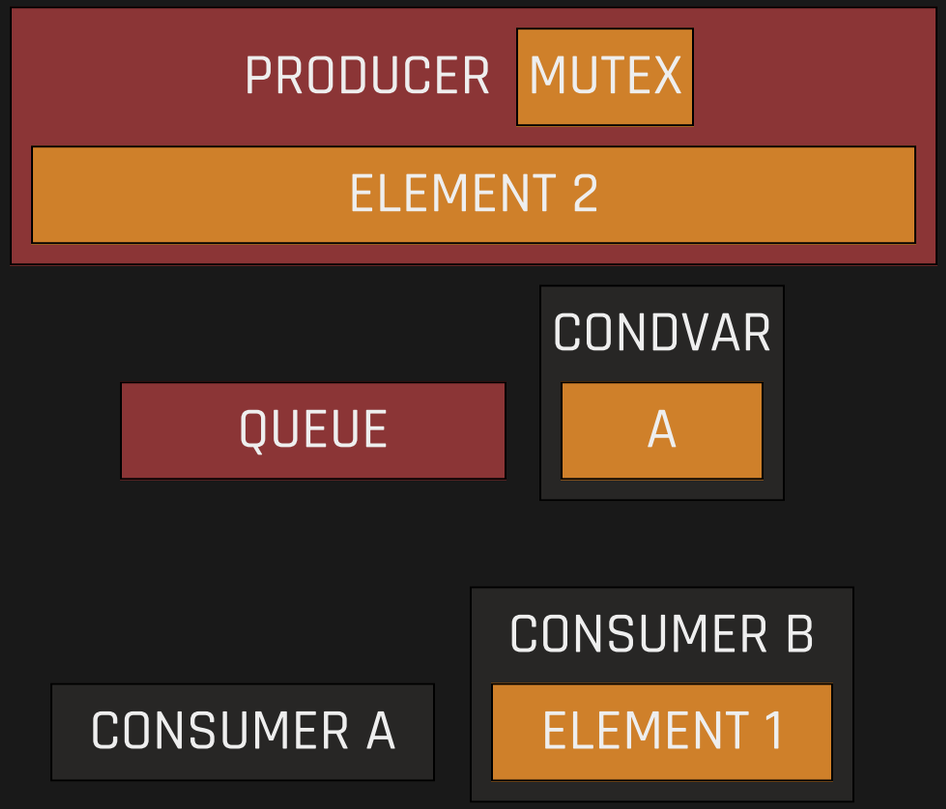
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



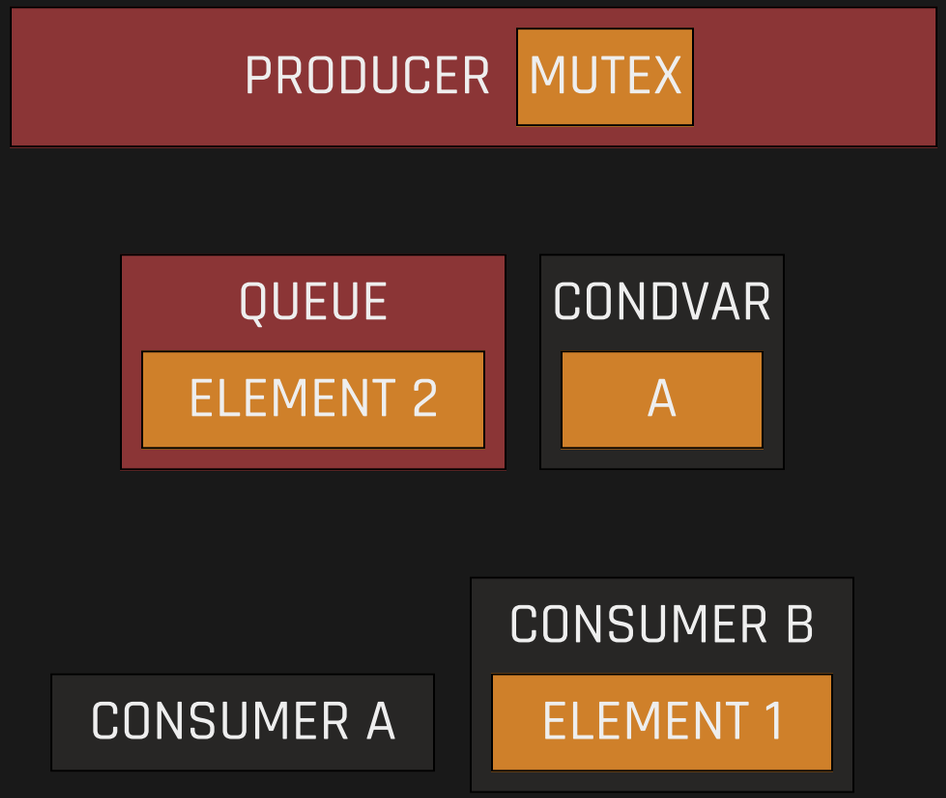
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



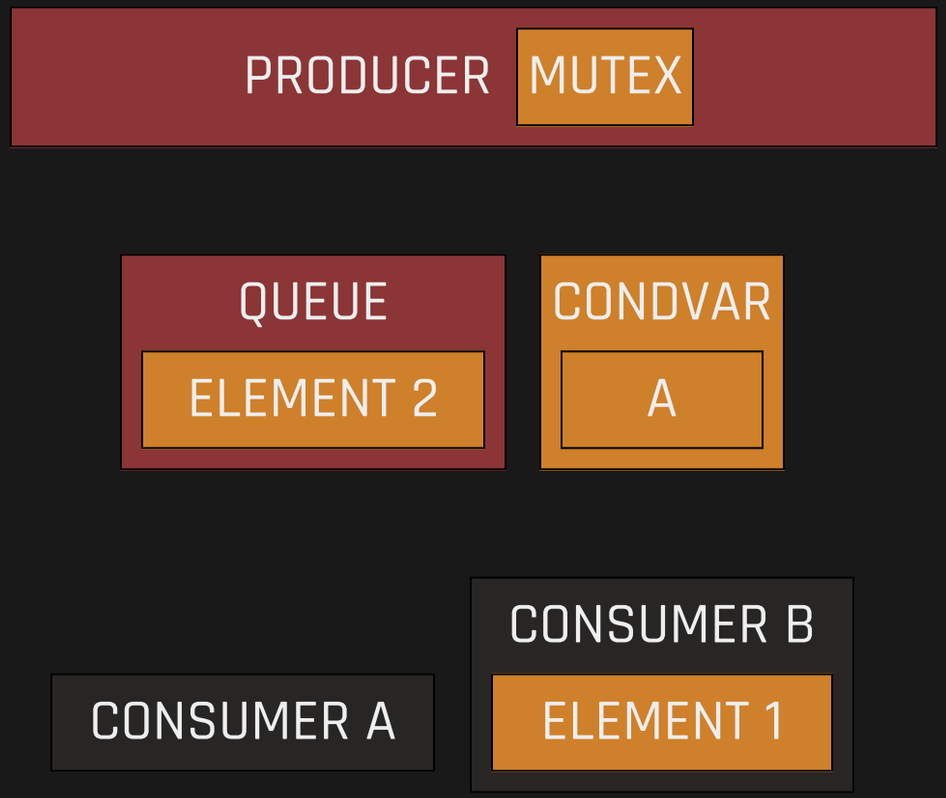
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



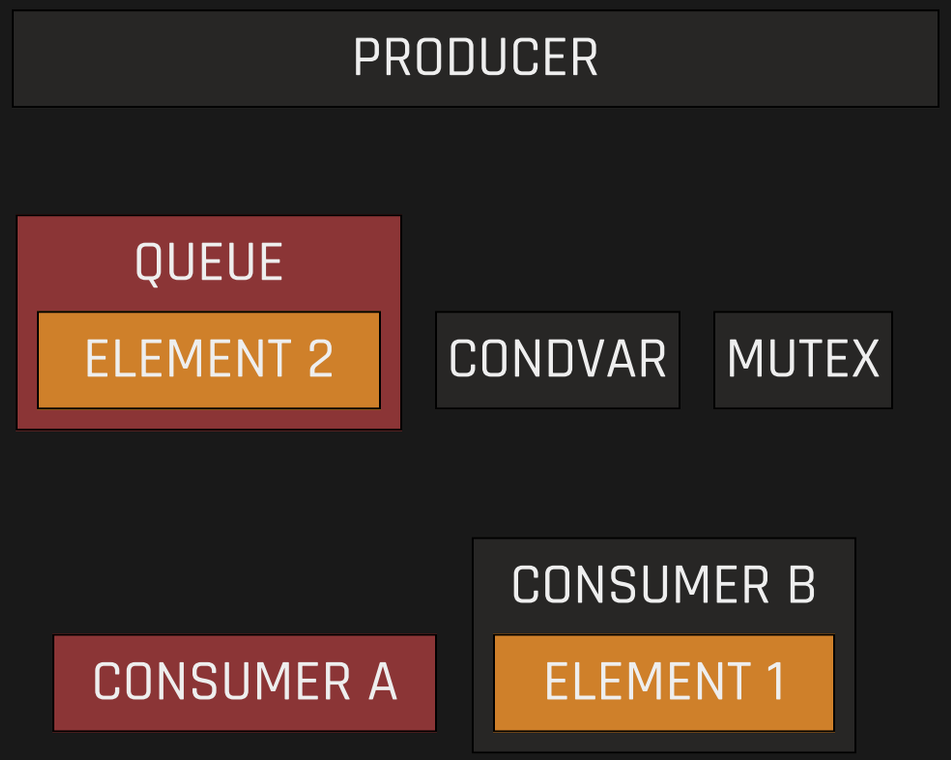
HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



HOW EXACTLY DOES IT WORK?

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         Lock l(m_);
11         queue_.push_front(element);
12         nonEmpty_.notify_all();
13     }
14     T pop() {
15         Lock l(m_);
16         auto hasData = [&]{ return not queue_.empty(); };
17         nonEmpty_.wait(l, hasData);
18         auto top = queue_.back();
19         queue_.pop_back();
20         return top;
21     }
22 };
```



CONDITION VARIABLE - DETAILS

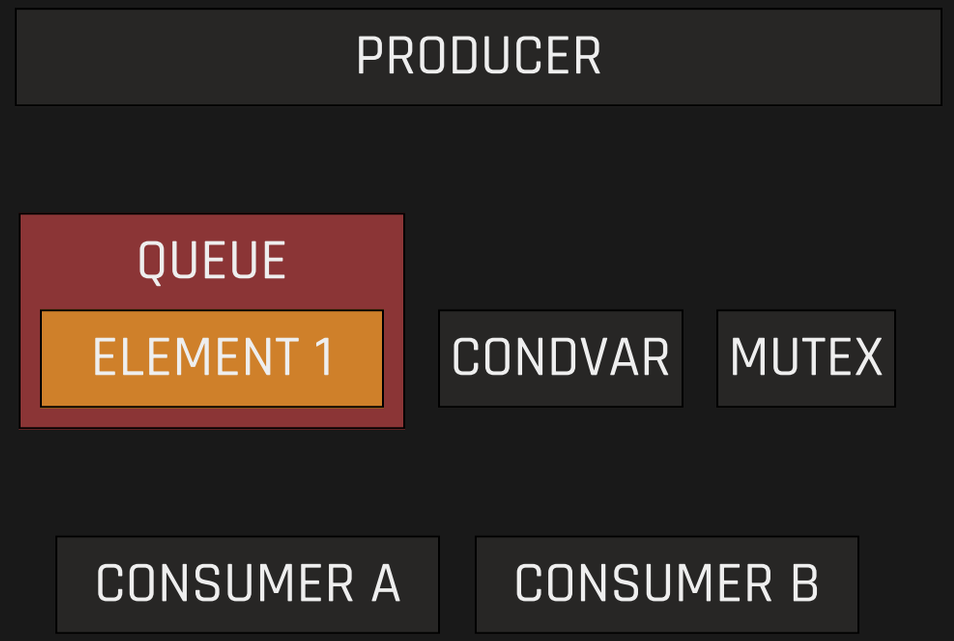
- `std::condition_variable` works only with exclusive locks (`unique_lock`)
- `std::condition_variable_any` works with any type of lock (eg. `shared_lock`)
- They are non-copyable, but movable
- The `wait()` method takes a lock and optionally a predicate.
- All threads waiting on the condition variable must have the same mutex locked. Otherwise, the behavior is undefined.
- The `wait_for()` and `wait_until()` methods take respectively a period of time or a point in time, until which the threads wait to wake up. After that, threads will wake up.
- If several threads are waiting on the condition variable and each has a different predicate, using `notify_one()` may cause a deadlock. A thread for which the condition has not been met may wake up, and if no other thread calls `notify_one()` or `notify_all()` they all will be waiting.

SPURIOUS WAKE-UP

CODERS SCHOOL

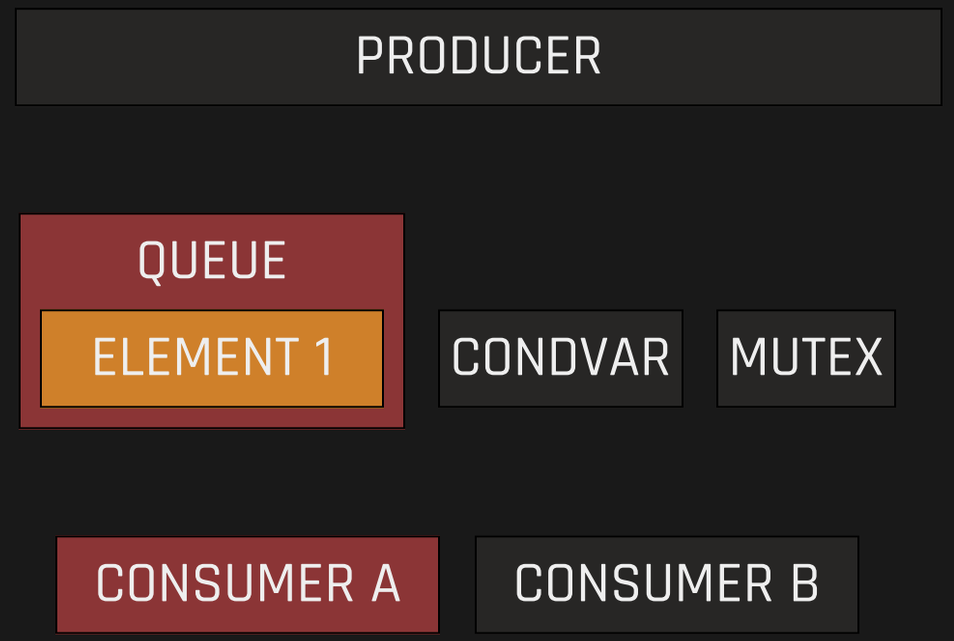
SPURIOUS WAKE-UP

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



SPURIOUS WAKE-UP

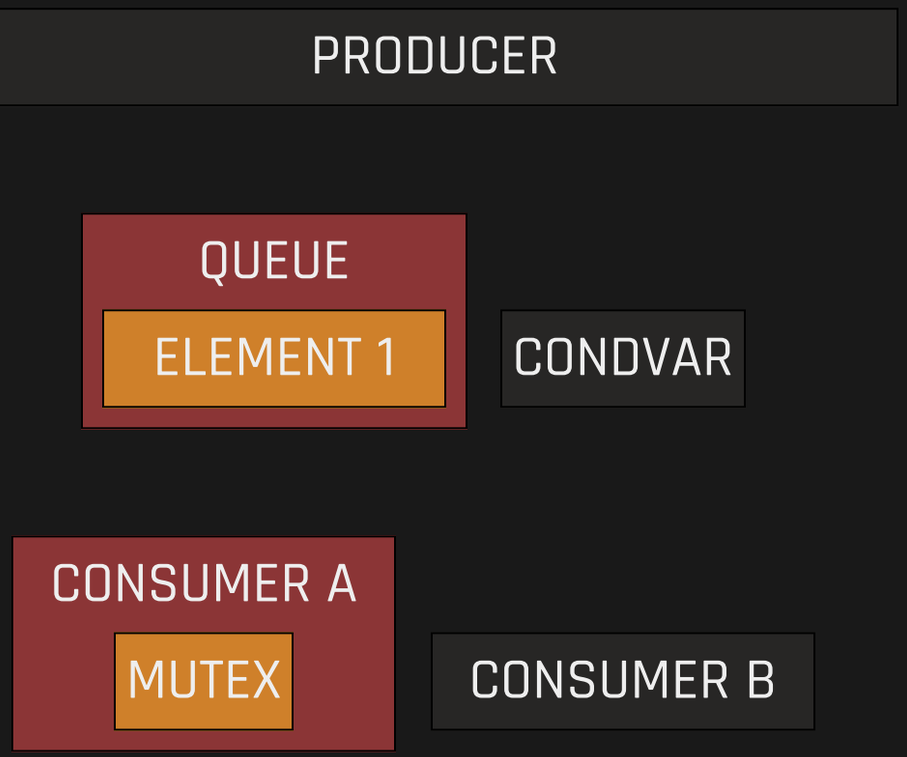
```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



SPURIOUS WAKE-UP

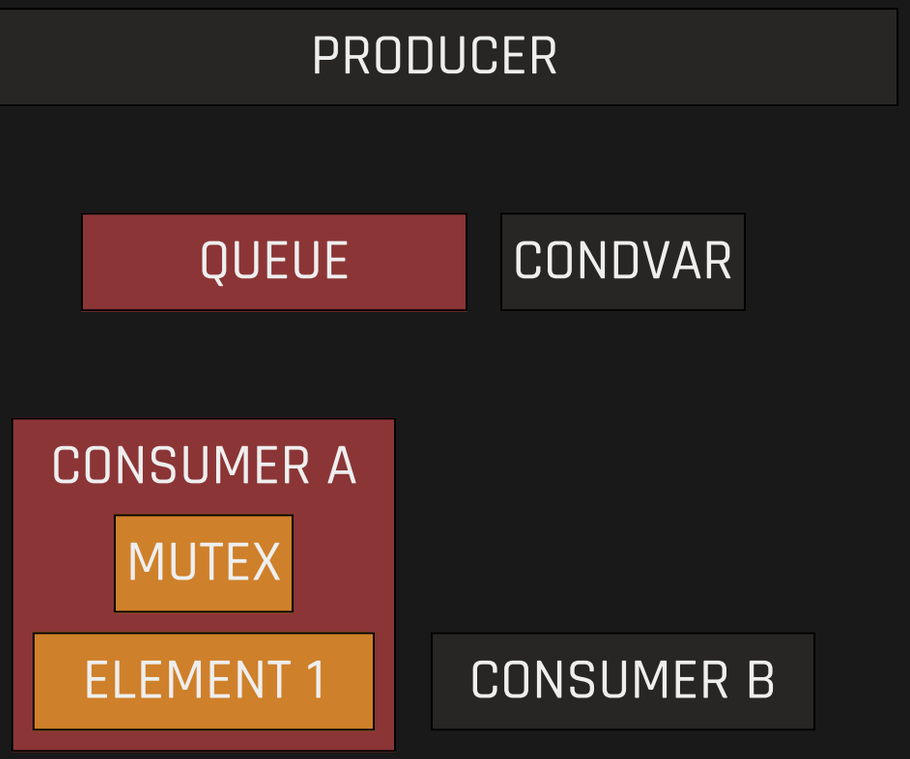
```cpp
template <typename T>
class WaitQueue {
    deque<T> queue_;
    mutable mutex m_;
    condition_variable nonEmpty_;
    using Lock = unique_lock<mutex>;

public:
    void push(const T & element) {
        {
            Lock l(m_);
            queue_.push_front(element);
        }
        nonEmpty_.notify_all();
    }
    T pop() {
        Lock l(m_);
        auto hasData = [&]{ return not queue_.empty(); };
        nonEmpty_.wait(l, hasData);
        auto top = queue_.back();
        queue_.pop_back();
        return top;
    }
};
```



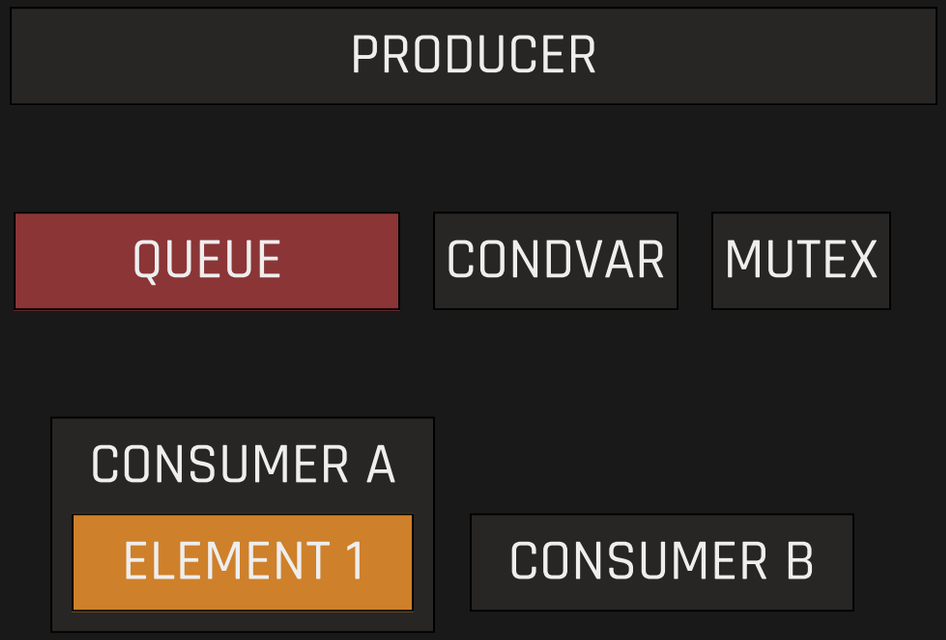
SPURIOUS WAKE-UP

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



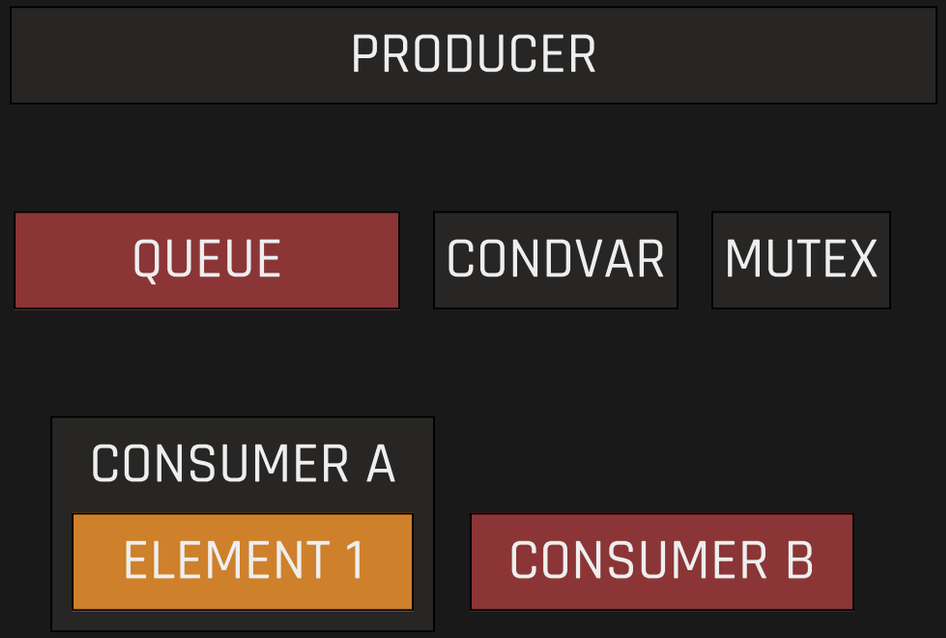
SPURIOUS WAKE-UP

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



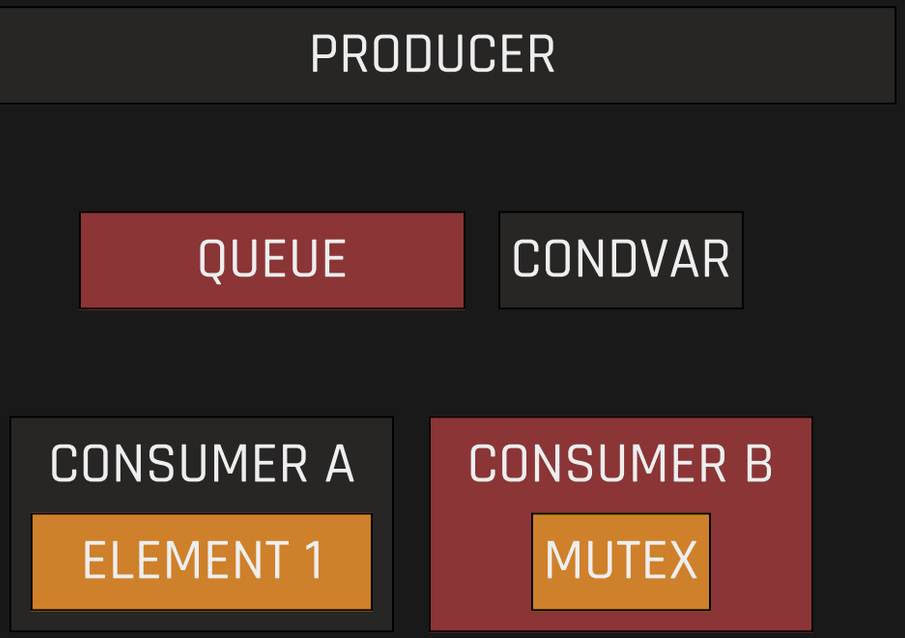
SPURIOUS WAKE-UP

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



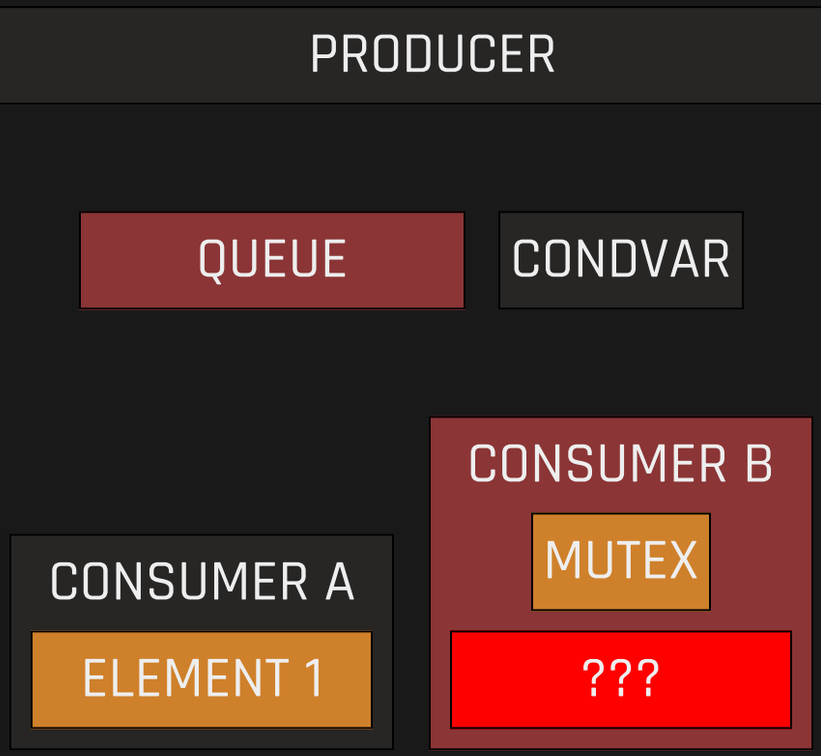
SPURIOUS WAKE-UP

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



SPURIOUS WAKE-UP

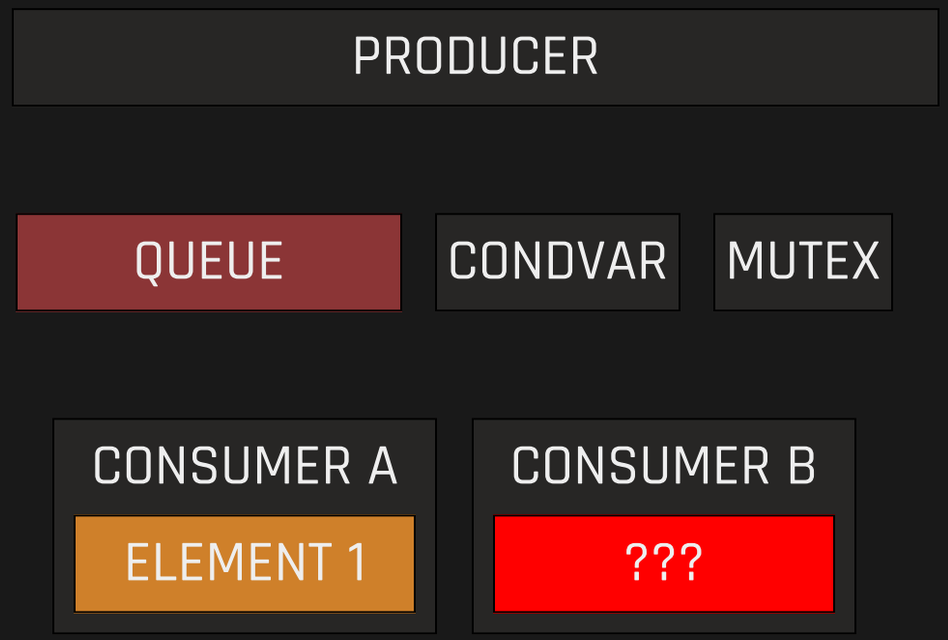
```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



UNDEFINED BEHAVIOR

SPURIOUS WAKE-UP

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



UNDEFINED BEHAVIOR

SPURIOUS WAKE-UP

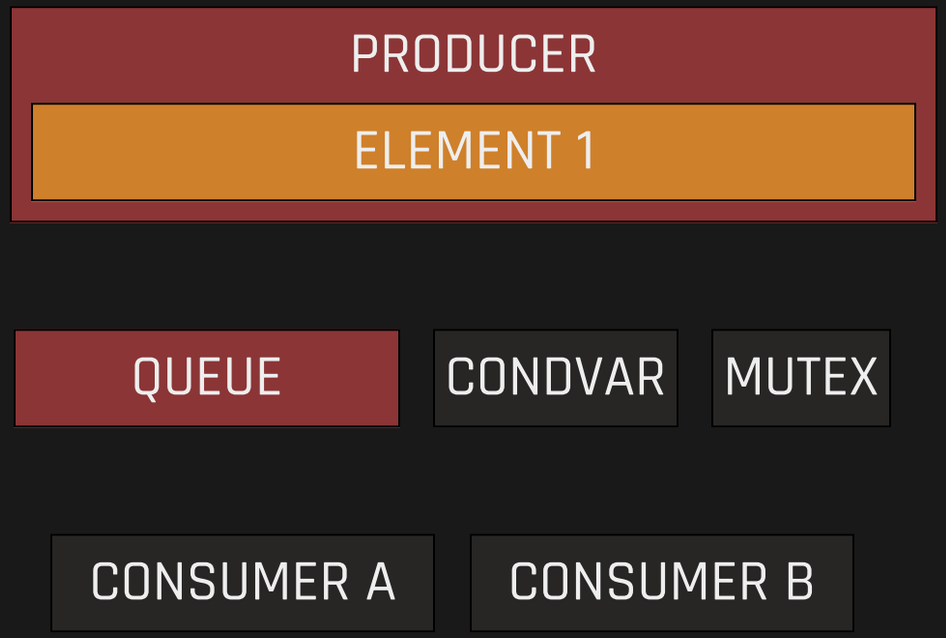
- The thread waiting on the `condition_variable` periodically wakes up from time to time.
- A `unique_lock` is required to wait on a condition variable. During sleep, the thread unlocks it, and when it wakes up, it locks it again.
- Predicate added to the `wait()` function makes spurious wake-ups not dangerous.
  - Only when the condition is met, the thread will continue its work.
  - If the condition is not met, the thread releases the lock and goes to sleep again.
- **ALWAYS** use `wait()` with predicate to avoid potential problems.

LOST NOTIFICATION

CODERS SCHOOL

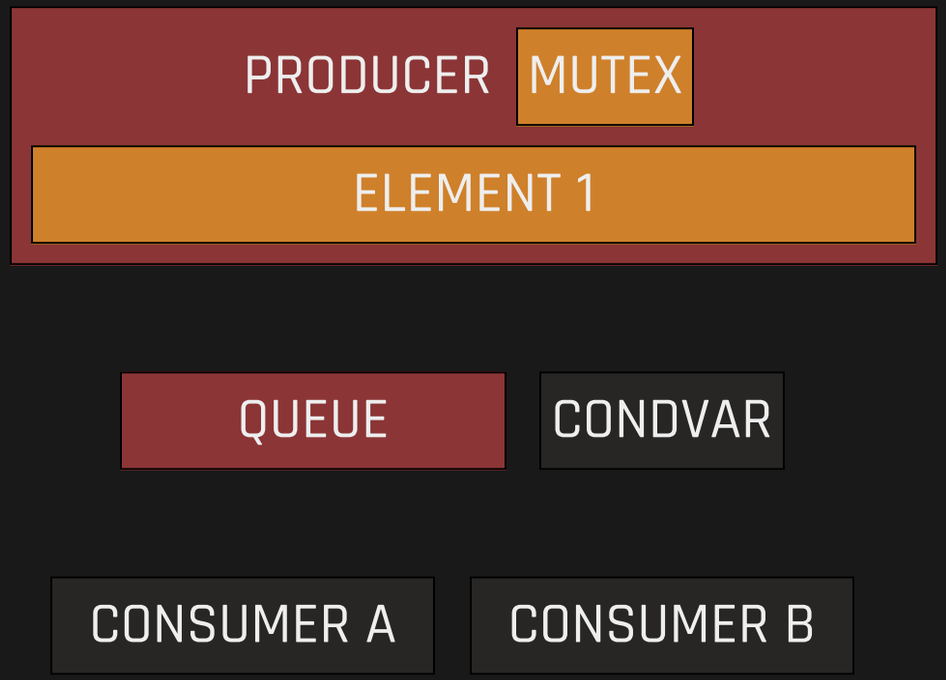
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



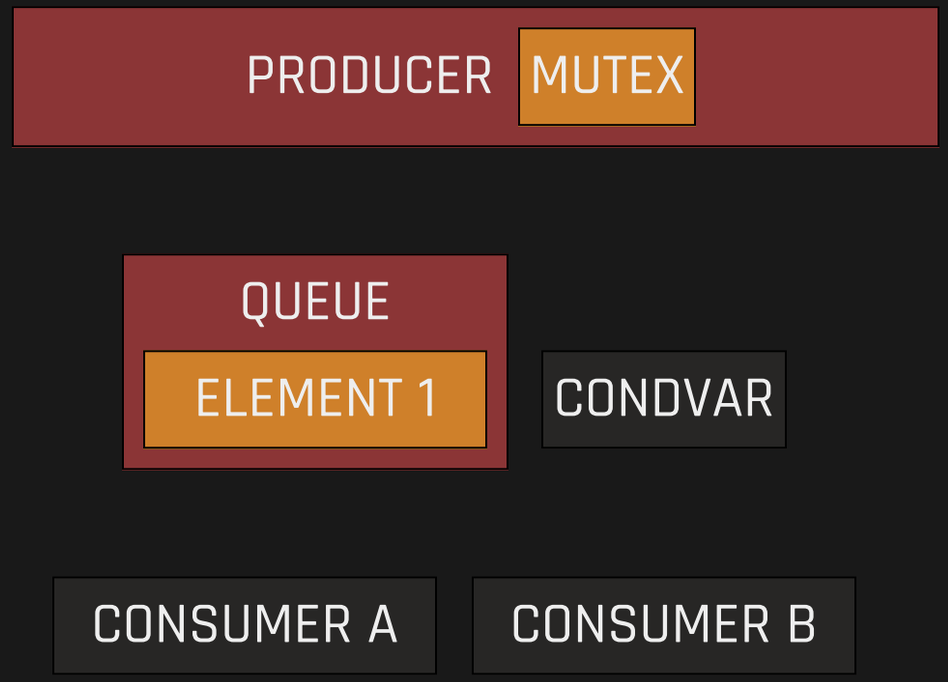
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



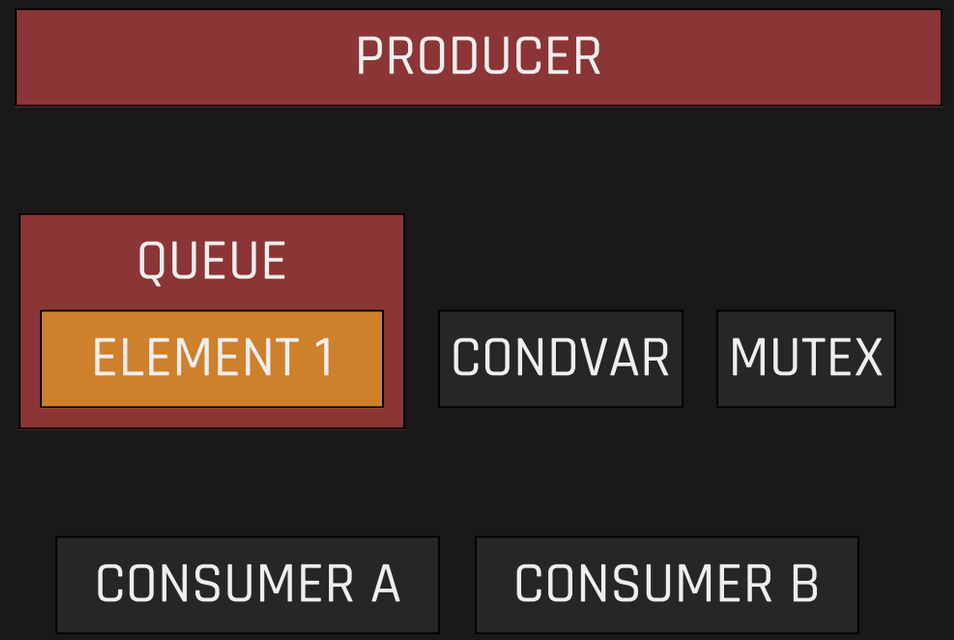
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



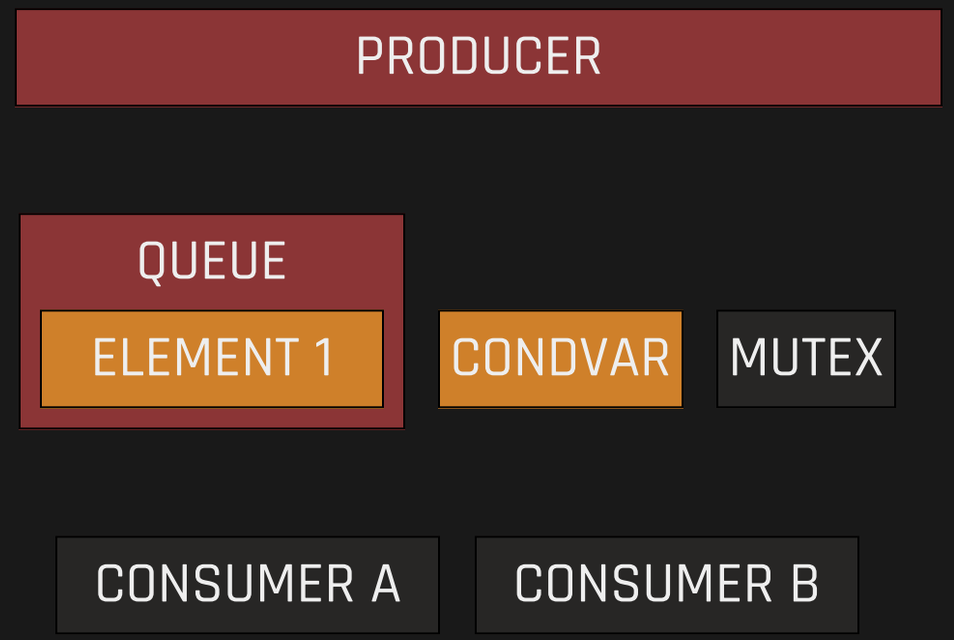
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



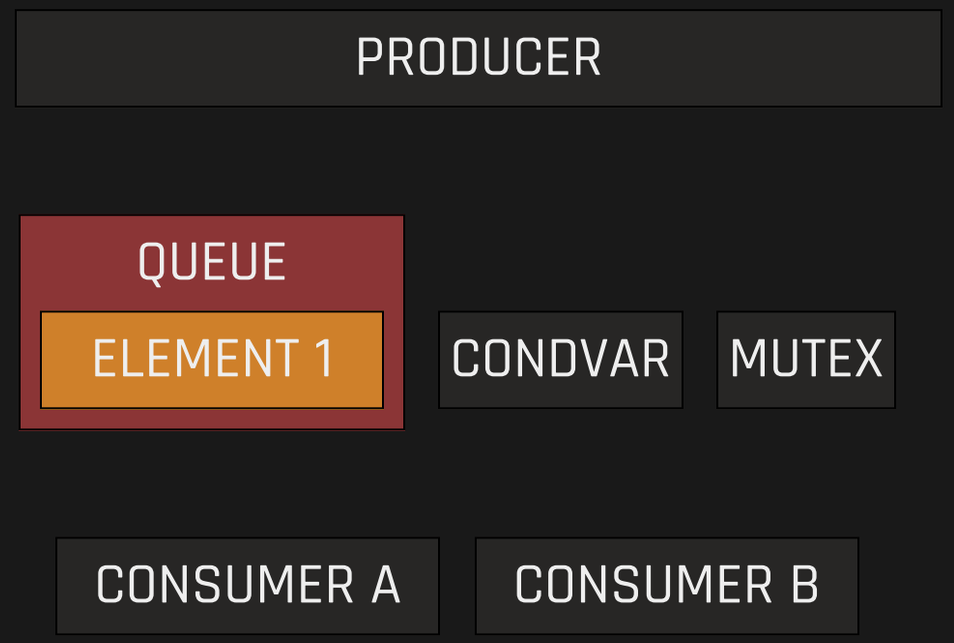
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



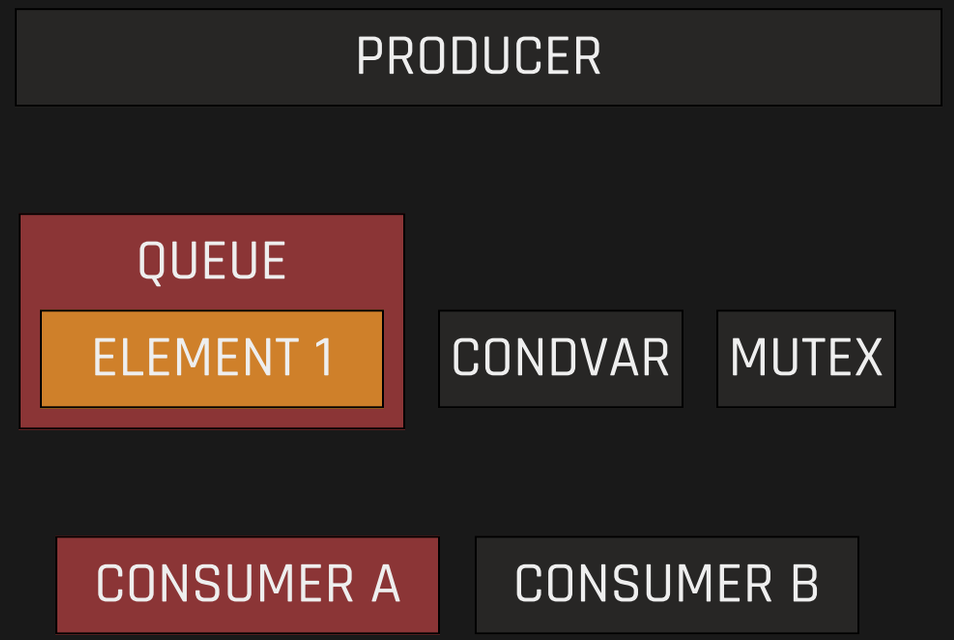
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



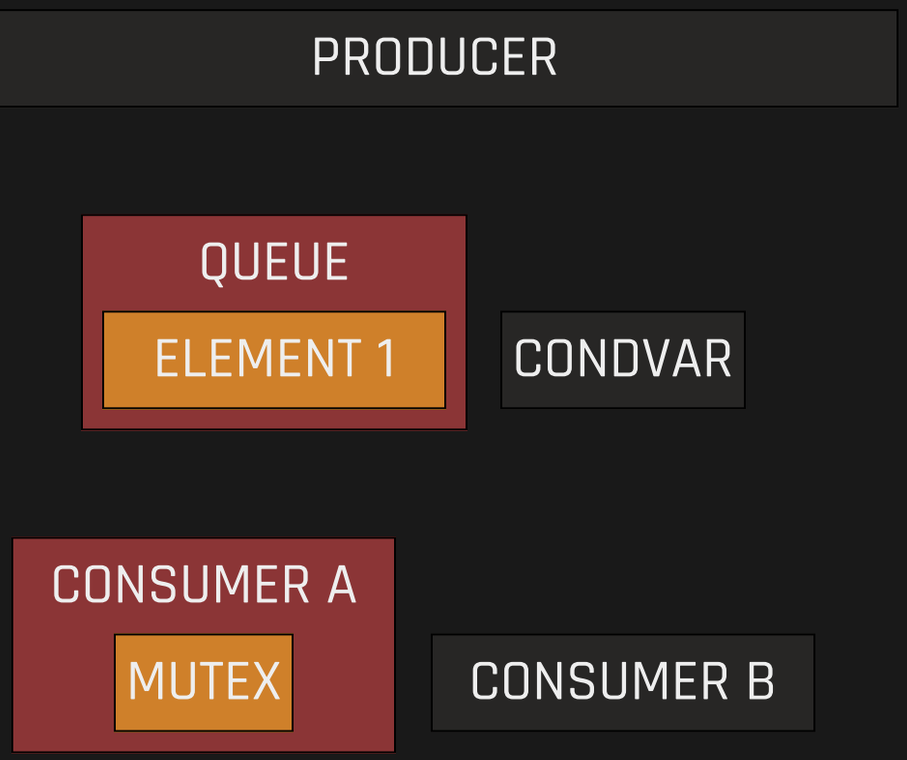
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



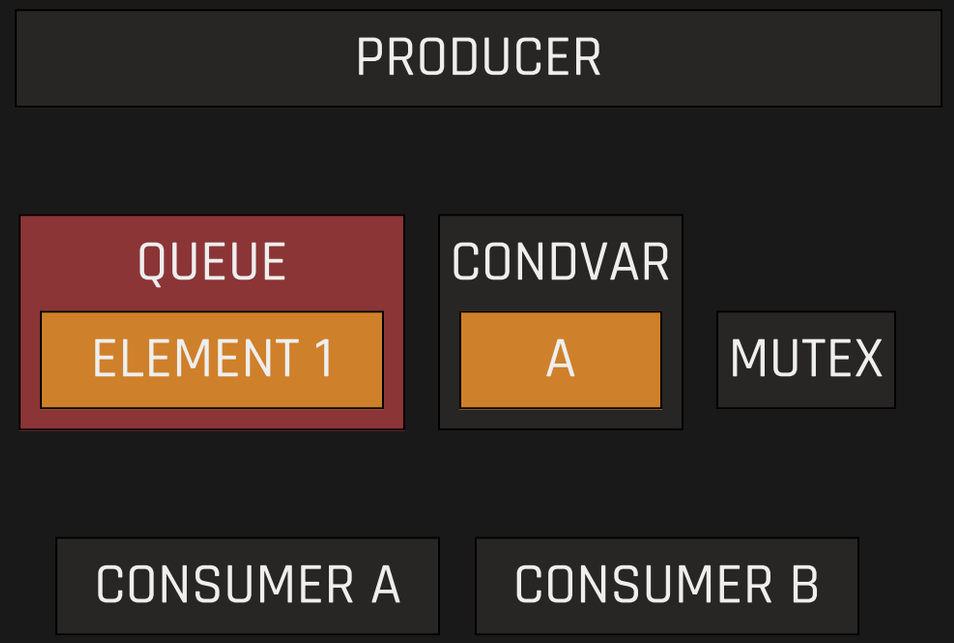
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



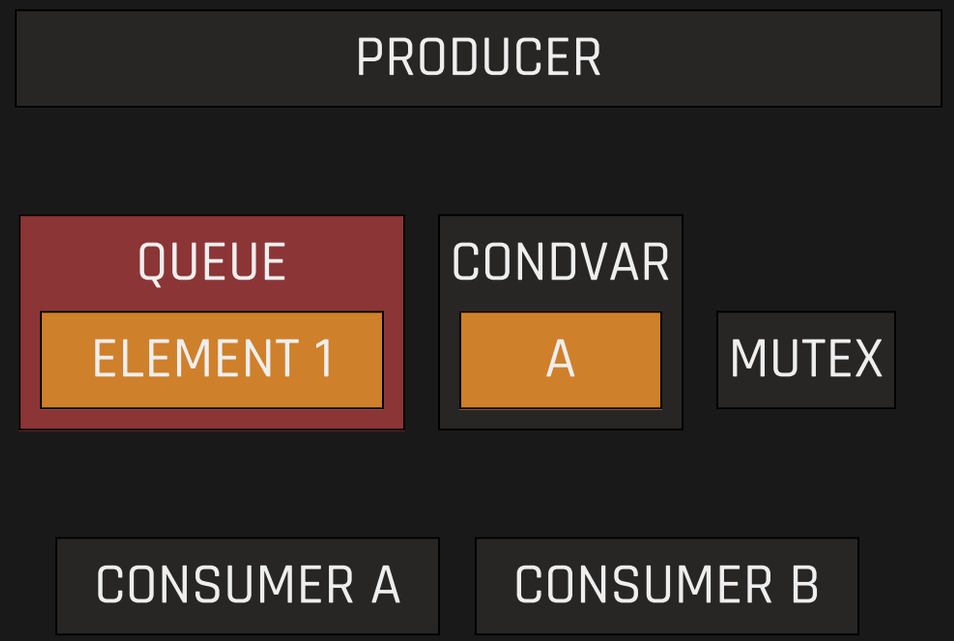
LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



LOST NOTIFICATION

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         nonEmpty_.wait(l);
19         auto top = queue_.back();
20         queue_.pop_back();
21         return top;
22     }
23 };
```



DEADLOCK...

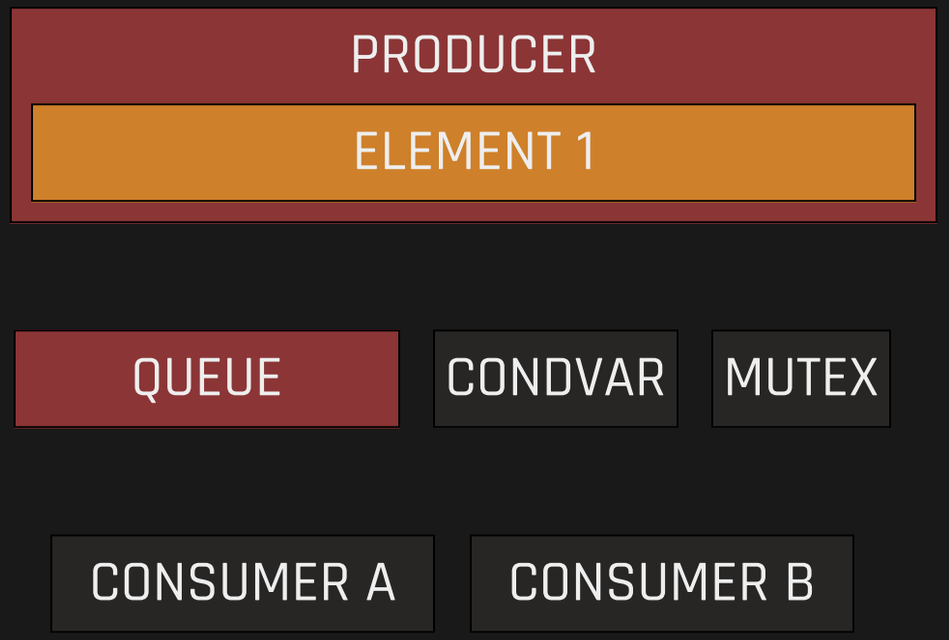
OR...

SPURIOUS WAKE-UP TO THE RESCUE!

`wait()` WITH PREDICATE SCENARIO

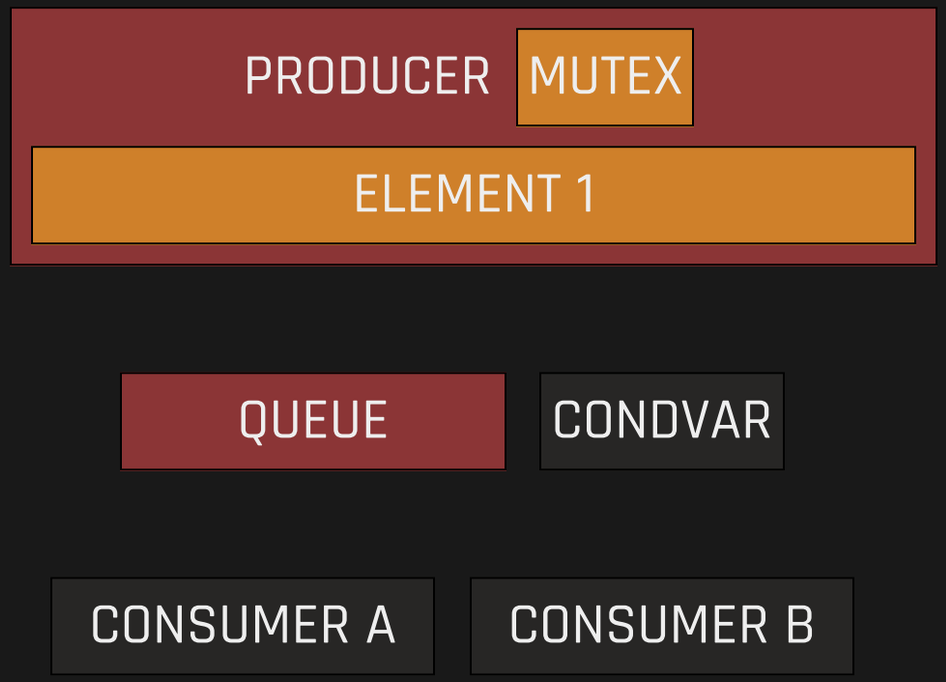
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



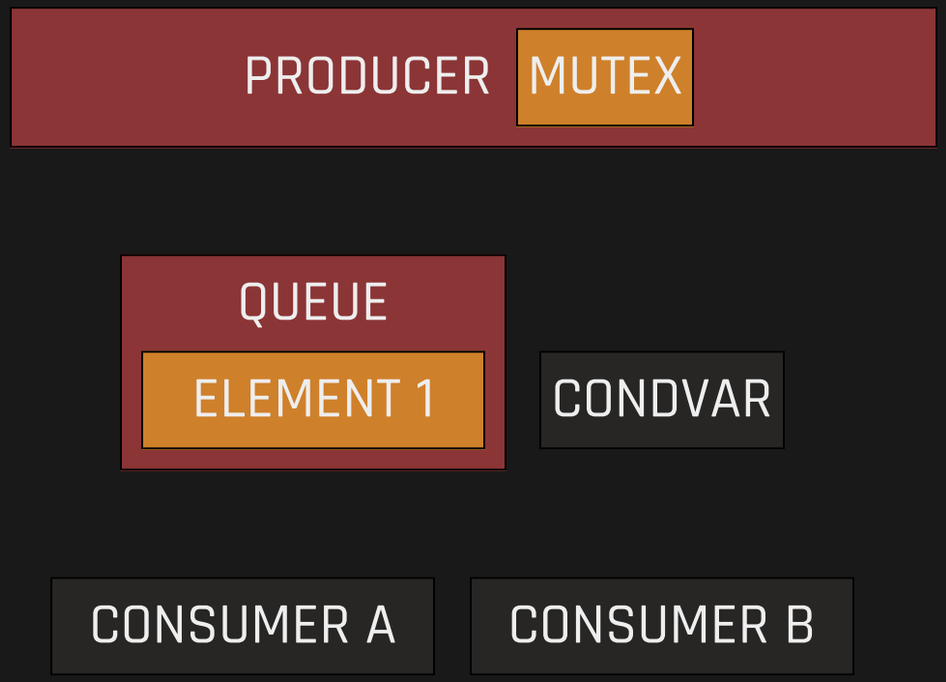
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



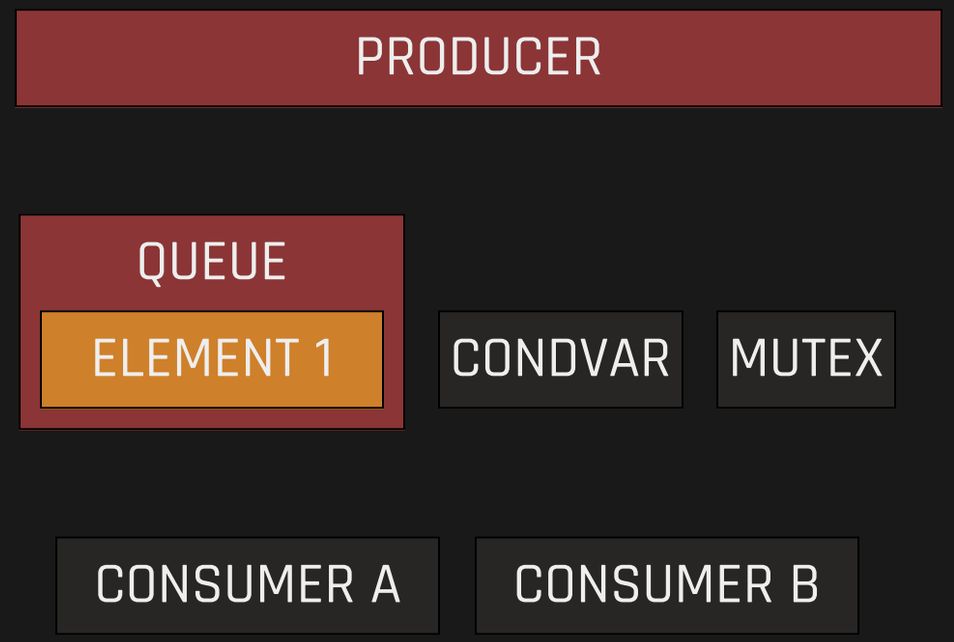
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



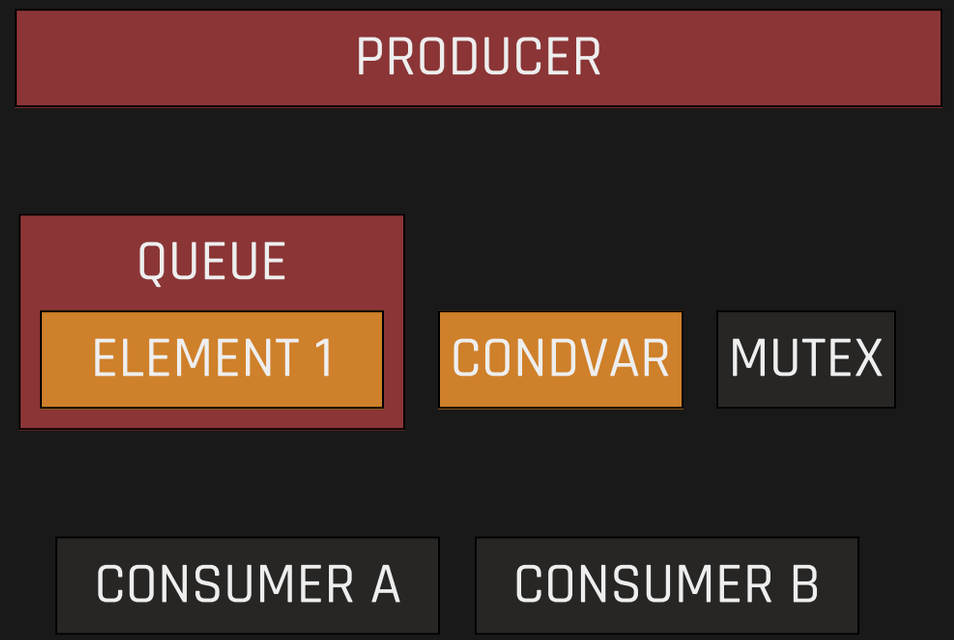
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



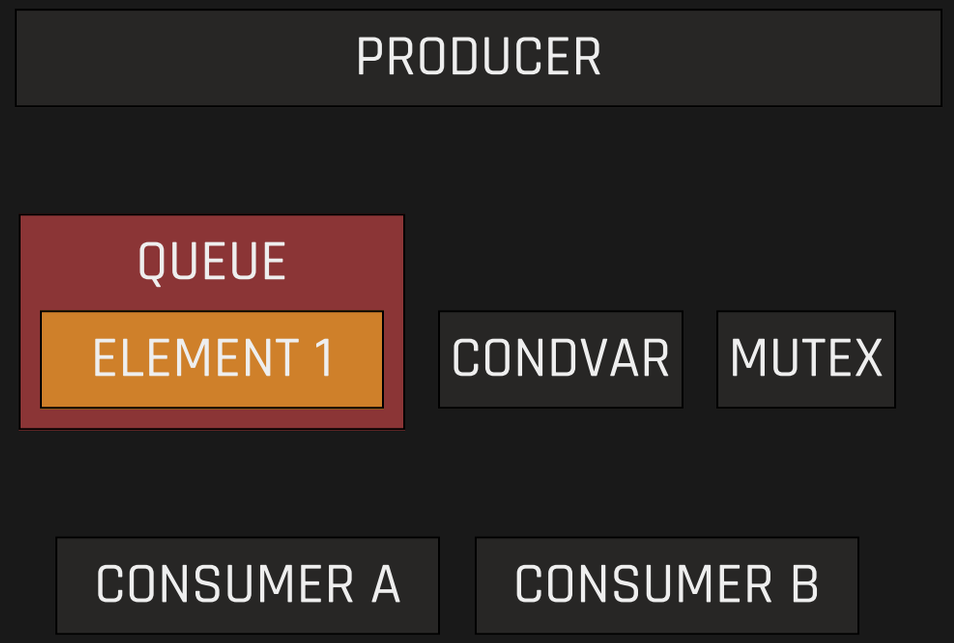
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



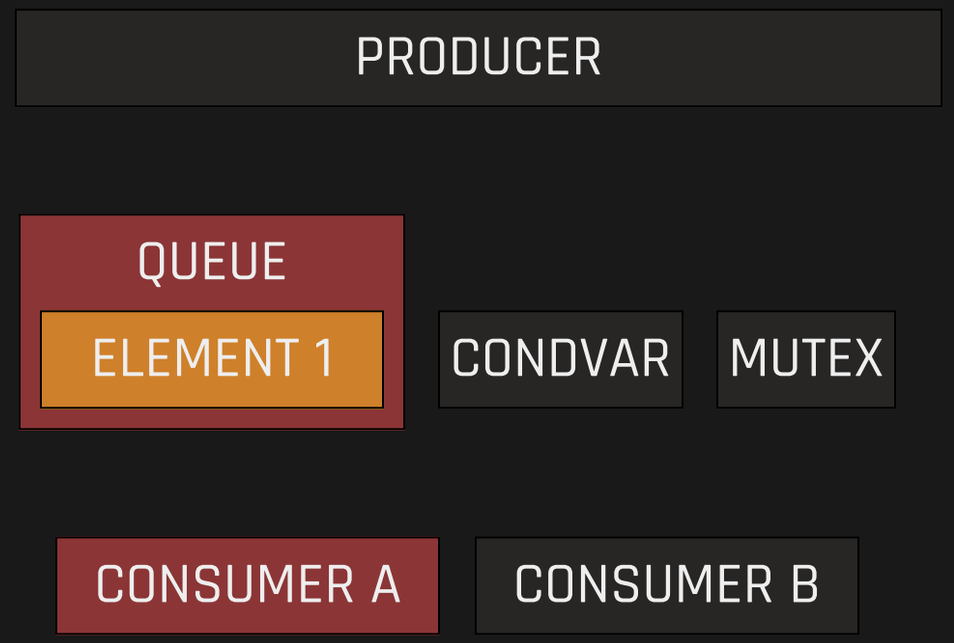
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



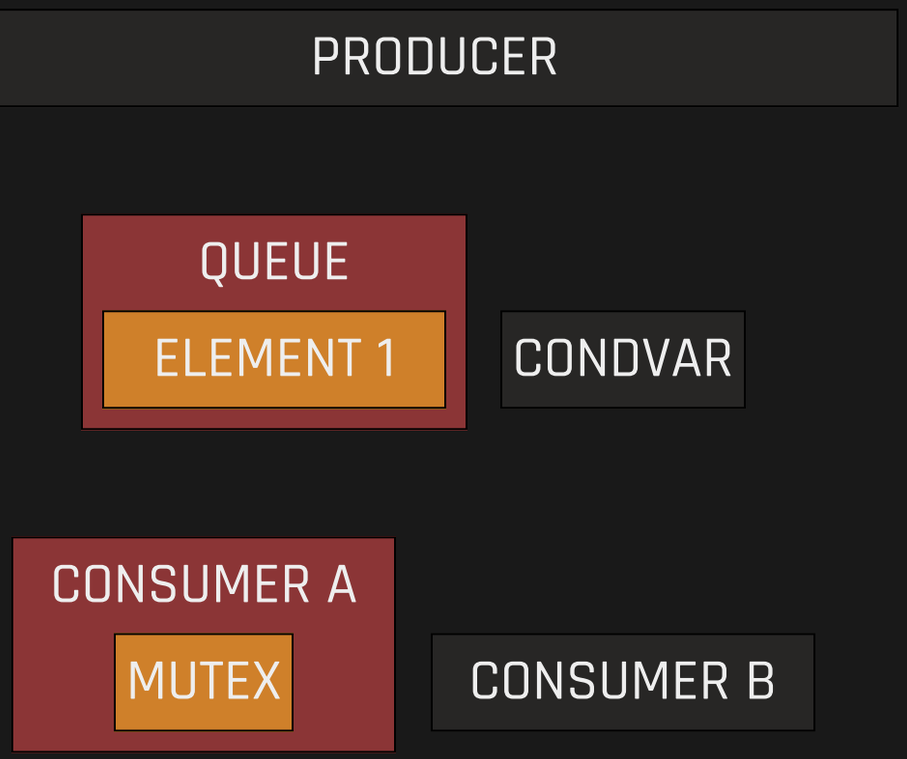
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



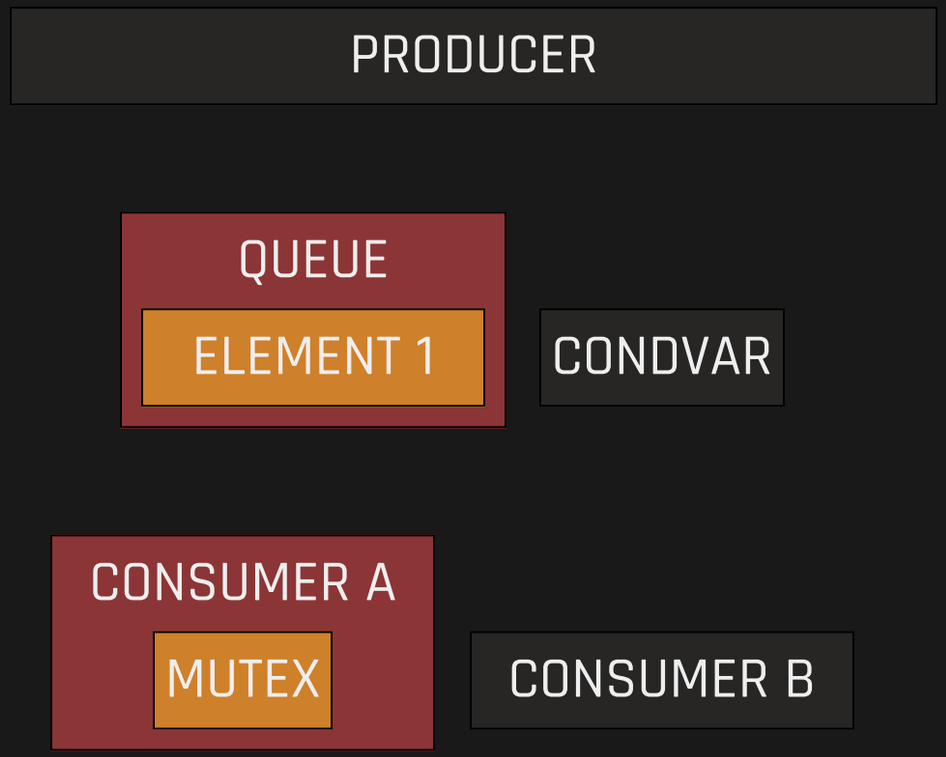
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```

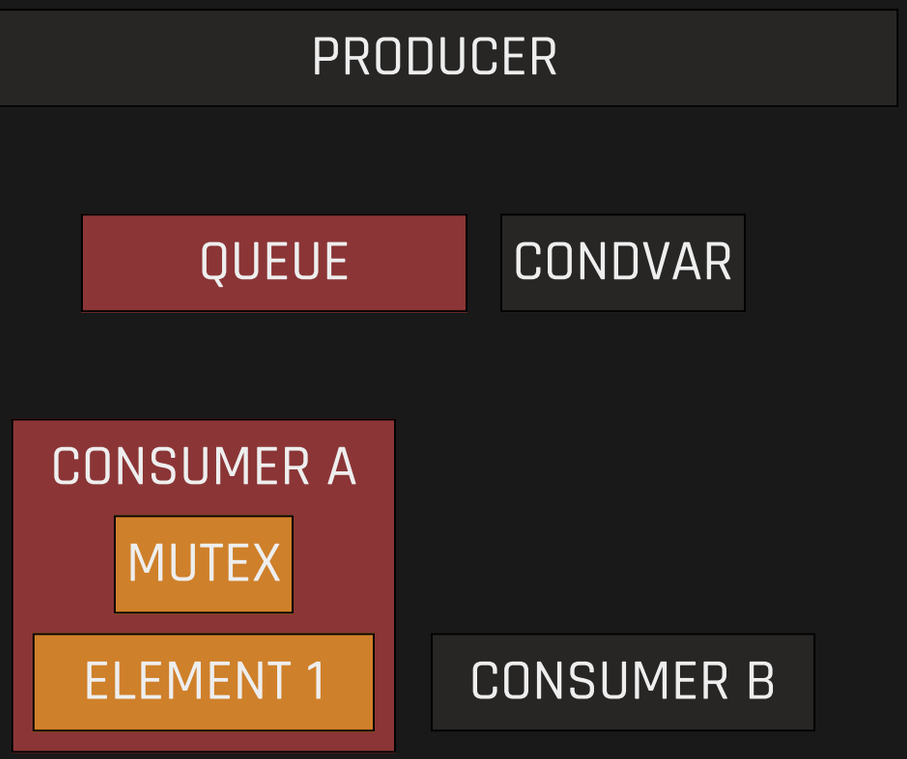


hasData == true

NO WAITING

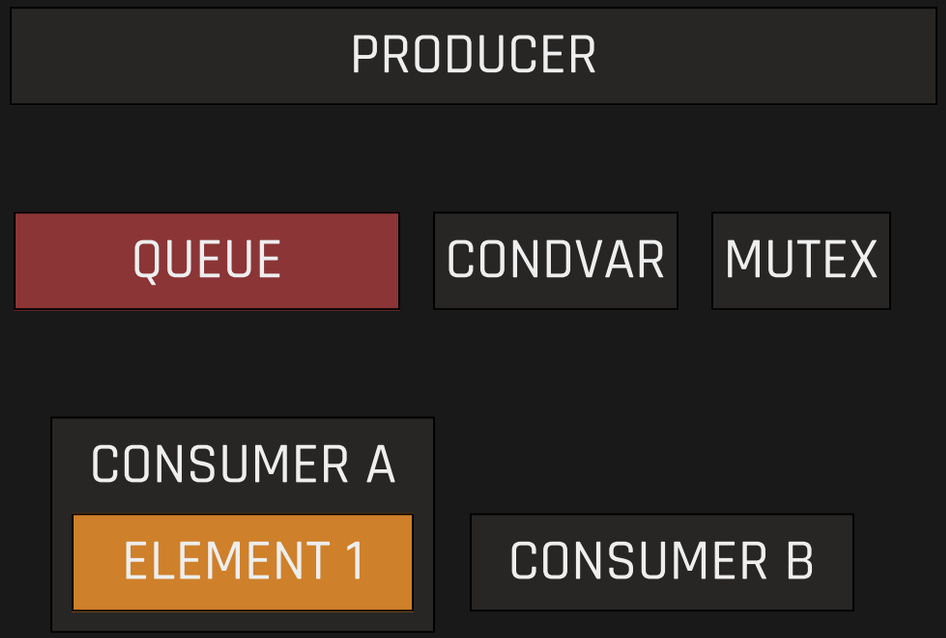
wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



wait() WITH PREDICATE SCENARIO

```
1  template <typename T>
2  class WaitQueue {
3      deque<T> queue_;
4      mutable mutex m_;
5      condition_variable nonEmpty_;
6      using Lock = unique_lock<mutex>;
7
8  public:
9      void push(const T & element) {
10         {
11             Lock l(m_);
12             queue_.push_front(element);
13         }
14         nonEmpty_.notify_all();
15     }
16     T pop() {
17         Lock l(m_);
18         auto hasData = [&]{ return not queue_.empty(); };
19         nonEmpty_.wait(l, hasData);
20         auto top = queue_.back();
21         queue_.pop_back();
22         return top;
23     }
24 };
```



LOST NOTIFICATION

- If the notification was sent before the thread started waiting on the `condition_variable`, it is lost and will not wake up the thread
- A spurious wake-up may help to unlock such thread
- If the thread was waiting on a `condition_variable` with a predicate, then the predicate must be met to resume the threads' job
- In case the predicate is true, the receiver is able to continue its work independently on the notification of the sender
  - (condition == true) => notification was delivered

SUMMARY

- Condition variables are used to sychronize threads multiple times. The future/promise mechanism is for one-time synchronization.
- Condition variables are more difficult to use. There are many details that you need to be aware of.
- Condition variables do not support exception forwarding. The future/promise mechanism allows it.
- General advice - use future/promise whenever possible (better: async/future). Use `condition_variable` only if multiple synchronization is required.
- Spurious wakeup and lost wakeup - article on modernescpp.com
- Condition variables vs tasks - article on modernescpp.com

RECAP

CODERS SCHOOL

1. WHAT WAS THE MOST SURPRISING FOR YOU? 😯

2. WHAT WAS THE MOST OBVIOUS FOR YOU? 🤭

POINTS TO REMEMBER

- Always use `wait()` with a predicate!
- Remember about spurious wake-ups and lost notifications
- Remember about a fight for locking a mutex when you use `notify_all()`
- You can't choose which thread should wake up when you use `notify_one()`

PRE-TEST 🤯

```cpp
// assume all necessary includes are here

int main() {
    std::mutex m;
    std::condition_variable cv;
    std::vector<int> v;
    std::vector<std::thread> producers;
    std::vector<std::thread> consumers;

    auto consume = [&] {
        std::unique_lock<std::mutex> ul(m);
        cv.wait(ul);
        std::cout << v.back();
        v.pop_back();
    };
    for (int i = 0; i < 10; i++) consumers.emplace_back(consume);

    auto produce = [&](int i) {
        {
            std::lock_guard<std::mutex> lg(m);
            v.push_back(i);
        }
        cv.notify_all();
    };
    for (int i = 0; i < 10; i++) producers.emplace_back(produce, i);

    for (auto && p : producers) p.join();
    for (auto && c : consumers) c.join();
}
```

1. there may be an Undefined Behavior in this code
2. the output is guaranteed to always be `0123456789`
3. `v` is always an empty vector at the end of this program
4. if some producers threads started before some consumers, we would have a deadlock because of lost notifications
5. this code can be improved by providing a predicate to `wait()` to disallow getting elements when the vector is empty
6. a change from `notify_all()` to `notify_one()` + `wait()` with predicate will guarantee that each consumer thread will receive a different number

POST-WORK

- Ping-pong
  - difficult version - homework/ping_pong.cpp
  - easier version - homework/ping_pong_easier.cpp

HOMEWORK: PING-PONG

- Thread A prints "ping" and the consecutive number
- Thread B prints "pong" and the consecutive number
- Ping always starts. Pong always ends.
- Threads must work in turns. There may not be 2 consecutive pings or pongs. The program cannot end with a ping without a respective pong.
- The program must be terminated either after the given number of repetitions or after the time limit, whichever occurs first. The reason for termination should be displayed on the screen.
- Program parameters:
  - number of repetitions
  - time limit (in seconds)

```
$> g++ 03_ping_pong.cpp -lpthread
-std=c++17 -fsanitize=thread
$> ./a.out 1 10
ping 0
pong 0
Ping reached repetitions limit
Pong reached repetitions limit
$> ./a.out 12 1
ping 0
pong 0
ping 1
pong 1
ping 2
pong 2
Timeout
```

TIPS

If you got stuck:

- You need a mutex and a condition variable in your `PingPong` class
- Wait for a condition variable with `wait_for()` in `stop()` function
- Check the number of repetitions in ping and pong threads
- Use an additional `std::atomic` variable which will tell all threads to end, when the required conditions are met
- Ping and pong threads should be using `wait()` to check if it's their turn to work. Use an additional variable that will be used in the predicate passed to `wait()`
- The pong thread should end the program after reaching the repetition limit

REPO ON GITHUB

https://github.com/coders-school/condition-variable

- Download a PDF
- Browse presentation content
- Find code examples
- Do the homework!

CODERS SCHOOL

KURS C++ ONLINE

CODERS SCHOOL

HTTPS://CODERS.SCHOOL

START KURSU W POŁOWIE KWIETNIA!

HTTPS://CODERS.SCHOOL