Monitors
Monitor

Purpose: To consolidate the wait and signal operations in a single class.

Instead of having semaphores and critical sections spread throughout the code of different processes, put the critical sections into methods of the monitor class.
Algorithm 7.1

n is an attribute of the monitor instead of being a global variable.

Solves the critical section problem.

Monitor methods are guaranteed to execute atomically.
Algorithm 7.1: Atomicity of monitor operations

monitor CS
    integer n ← 0

    operation increment
        integer temp
        temp ← n
        n ← temp + 1

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>CS.increment</td>
</tr>
</tbody>
</table>
Executing a Monitor Operation
Condition variable

A special monitor variable that has a queue (FIFO) of blocked processes.

A monitor can have more than one condition variable. There is a queue of blocked processes for each condition variable.
Condition variable

There are three operations on condition variable $cond$. 
Condition variable

There are three operations on condition variable \textit{cond}.

\texttt{waitC(\textit{cond})}
- append \textit{p} to \textit{cond} queue
- \textit{p.state} $\leftarrow$ blocked
- \texttt{monitor.lock} $\leftarrow$ released
Condition variable

There are three operations on condition variable $cond$.

waitC($cond$)
- append $p$ to $cond$ queue
- $p.state \leftarrow$ blocked
- $monitor.lock \leftarrow$ released

signalC($cond$)
- if $cond$ queue $\neq \emptyset$
  - remove head of $cond$ queue and assign to $q$
  - $q.state \leftarrow$ ready
Condition variable

There are three operations on condition variable $cond$.

$\text{waitC}(cond)$
- append $p$ to $cond$ queue
- $p.state \leftarrow \text{blocked}$
- $\text{monitor.lock} \leftarrow \text{released}$

$\text{signalC}(cond)$
- if $cond$ queue $\neq \emptyset$
  - remove head of $cond$ queue and assign to $q$
  - $q.state \leftarrow \text{ready}$

$\text{empty}(cond)$
- return $cond$ queue is Empty
<table>
<thead>
<tr>
<th>Semaphore</th>
<th>Ben-Ari monitor</th>
</tr>
</thead>
</table>

**Semaphore**

**Ben-Ari monitor**
<table>
<thead>
<tr>
<th>Semaphore</th>
<th>Ben-Ari monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. wait((S)) may or may not block.</td>
<td>1. wait(C(cond)) always blocks.</td>
</tr>
</tbody>
</table>
### Semaphore

1. `wait(S)` may or may not block.
2. `signal(S)` always has an effect.

### Ben-Ari monitor

1. `waitC(cond)` always blocks.
2. `signalC(cond)` has no effect if `cond` queue is empty.
<table>
<thead>
<tr>
<th>Semaphore</th>
<th>Ben-Ari monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. wait($S$) may or may not block.</td>
<td>1. waitC($cond$) always blocks.</td>
</tr>
<tr>
<td>2. signal($S$) always has an effect.</td>
<td>2. signalC($cond$) has no effect if $cond$ queue is empty.</td>
</tr>
<tr>
<td>3. Process unblocked by signal($S$) might not resume execution immediately.</td>
<td>3. Process unblocked by signalC($cond$) resumes executing immediately.</td>
</tr>
</tbody>
</table>
Semaphore / monitor equivalence

Semaphores and monitors have equivalent capabilities.

You can construct a semaphore with a monitor.

You can construct a monitor with a semaphore.
Algorithm 7.2: Semaphore simulated with a monitor

Modified (Warford) monitor Sem

```
monitor Sem
    integer s ← k
    condition notZero
    operation wait
        if s = 0
            waitC(notZero)
        s ← s - 1
    operation signal
        s ← s + 1
    signalC(notZero)
```

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: Sem.wait</td>
<td>q1: Sem.wait</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>p2: Sem.signal</td>
<td>q2: Sem.signal</td>
</tr>
</tbody>
</table>
Condition Variable in a Monitor

monitor Sem

notZero

s 0
Class exercise
Construct the state transition diagram.

```
monitor Sem
integer s <- k
condition notZero
operation wait
    if s > 0
        s <- s - 1
    else
        waitC(notZero)
operation signal
    if empty(notZero)
        s <- s + 1
    else
        signalC(notZero)
```

```
loop forever
    non-critical section
    p1: Sem.wait
    critical section
    p2: Sem.signal

loop forever
    non-critical section
    q1: Sem.wait
    critical section
    q2: Sem.signal

waitC(cond)
    append p to cond queue
    p.state ← blocked
    monitor.lock ← released

signalC(cond)
    if cond queue ≠ Ø
        remove head of cond queue and assign to q
    q.state ← ready
```
State Diagram for the Semaphore Simulation

- **p1**: Sem.wait, q1: Sem.wait, 1, <>
- **q1**: Sem.wait, 0, <>
- **p2**: Sem.signal, q1: Sem.wait, 0, <>
- **q2**: Sem.signal, blocked, 0, <q>
- **p2**: Sem.signal, blocked, 0, <q>
- **q2**: Sem.signal, blocked, 0, <p>
The Ben-Ari monitor

Ben-Ari defines his monitor to have “the immediate resumption requirement.”

When `signalC(cond)` executes, the blocked process, if any is blocked, immediately resumes.

The process that executed `signalC` is put in a signaling queue. (No waiting queue necessary)
The Immediate Resumption Requirement

condition 1

condition 2

monitor

waiting

signaling
Notes on monitors


Monitor Classification
Peter A. Buhr and Michel Fortier

Dept. of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Michael H. Coffin
EDS Research and Development, 901 Tower Drive, 1st Floor, Troy Michigan 48007-7019, U. S. A.
General monitor

All procedures are mutually exclusive. Each monitor has

* One entry queue
* One queue for each condition variable
* One waiting queue
* One signaler queue
The diagram in Figure 3 illustrates processes waiting to use a monitor. It shows two conditions, A and B, each with an entry queue, a monitor section (monitor variables), a waiting queue, and a signaller queue. The diagram uses solid circles to represent active tasks and open circles for waiting tasks. The figure caption reads: "Figure 3: Processes Waiting to use a Monitor."
General actions

waitC(cond)
Blocked on condition queue for cond
General actions

waitC(cond)
  Blocked on condition queue for cond

signalC(cond)
  Signaler moved to signaler queue
  Signaled moved from condition queue to wait queue
General actions

waitC(cond)
  Blocked on condition queue for cond
signalC(cond)
  Signaler moved to signaler queue
  Signaled moved from condition queue to
  wait queue
Monitor is unlocked
General actions

\texttt{waitC}(\texttt{cond})
  Blocked on condition queue for \texttt{cond}
\texttt{signalC}(\texttt{cond})
  Signaler moved to signaler queue
  Signaled moved from condition queue to
  wait queue
Monitor is unlocked
Monitor chooses from one of the queues which
process gets to enter
Action of waitC(A)

Process blocked on condition queue A
Monitor is unlocked
An unblocked process is selected to continue
Action of `signalC(A)`
Signaler to signaler queue, signaled to wait queue
Monitor is unlocked
An unblocked process is selected to continue
Types of monitors

The type of monitor is determined by how the monitor chooses which process gets to enter. Each queue has a specific precedence:

* E — entry precedence
* W — waiting precedence
* S — signaler precedence
In the monitor section, there are three queues: the entry queue, the waiting queue, and the signaller queue. Each queue is associated with a precedence rule:

- **E** — entry precedence
- **W** — waiting precedence
- **S** — signaller precedence

The diagram illustrates how tasks are managed under different conditions. Tasks are categorized into sets based on these precedence rules, and the monitor variables determine which task is executed next. The diagram shows how tasks are prioritized and selected for execution, taking into account the conditions and the precedence rules.
<table>
<thead>
<tr>
<th>relative priority</th>
<th>traditional monitor name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_p = W_p = S_p$</td>
<td>Wait and Notify [Lampson and Redell 1980]</td>
</tr>
<tr>
<td>$E_p = W_p &lt; S_p$</td>
<td>Signal and Wait [Howard 1976a]</td>
</tr>
<tr>
<td>$E_p = S_p &lt; W_p$</td>
<td>Signal and Continue [Howard 1976b]</td>
</tr>
<tr>
<td>$E_p &lt; W_p = S_p$</td>
<td>Signal and Urgent Wait [Hoare 1974]</td>
</tr>
<tr>
<td>$E_p &gt; W_p = S_p$</td>
<td>(rejected)</td>
</tr>
<tr>
<td>$E_p = S_p &gt; W_p$</td>
<td>(rejected)</td>
</tr>
<tr>
<td>$S_p &gt; E_p &gt; W_p$</td>
<td>(rejected)</td>
</tr>
<tr>
<td>$E_p = W_p &gt; S_p$</td>
<td>(rejected)</td>
</tr>
<tr>
<td>$W_p &gt; E_p &gt; S_p$</td>
<td>(rejected)</td>
</tr>
<tr>
<td>$E_p &gt; S_p &gt; W_p$</td>
<td>(rejected)</td>
</tr>
<tr>
<td>$E_p &gt; W_p &gt; S_p$</td>
<td>(rejected)</td>
</tr>
</tbody>
</table>

Table 1: Relative Priorities for Internal Monitor Queues
Buhr, Signal and Continue, $E < W < S$

Diagram:
- Entry queue
- E
- Waiting queue $W$
- Signaller queue $S$
- Exit

Conditions:
- Condition A
- Condition B

Monitors based on variables, with
- Monitor for E
- Monitor for W
- Monitor for S

Each condition has its own queue and is connected to the next condition.
Buhr, Signal and Continue, \( E < W < S \)

Signaler always picked.
Signaler queue not necessary.
When w eventually picked, condition may no longer be met. May need waitC in the body of a loop instead of an if.
Ben-Ari, Signal and Urgent Wait, E < S < W

entry queue

condition A

monitor variables

condition B

waiting queue

signaller queue

exit
Ben-Ari, Signal and Urgent Wait, \( E < S < W \)

Signaled always picked
Waiting queue not necessary
Can have waitC in the body of an if statement. However, signalC should be the last statement of operation.
Java, Wait and Notify, \( E = W < S \)
Java, Wait and Notify, $E = W < S$

Signaler always picked. Signaled waits in entry queue. There are no condition variables.
Java, Wait and Notify, \( E = W < S \)

* waitC is called wait in Java.
* signalC is called notify in Java.
* notifyAll moves all processes from the waiting queue to the entry queue.
* Signaler usually executes notifyAll, and waiting processes loop on their boolean expressions.
States and State Transitions of a Thread. A thread’s transition from one state to another may be caused by a method call performed by the thread itself (shown in the monospace font), by a method call possibly performed by another thread (shown in the slanted monospace font); and by timeouts and other actions.
Java, Wait and Notify, $E = W < S$

$E = W$ criticized by Buhr:

“In all cases, the no-priority property complicates the proof rules, makes performance worse, and makes programming more difficult. ... Therefore, we have rejected all no-priority monitors from further consideration.”
The producer-consumer problem with a finite buffer

Two condition variables: notEmpty and notFull

The producer calls append(D). Only the producer can be in the notFull queue of blocked processes.

The consumer calls take(). Only the consumer can be in the notEmpty queue of blocked processes.
### Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype D</td>
<td>datatype D</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: D ← produce</td>
<td>q1: D ← PC.take</td>
</tr>
<tr>
<td>p2: PC.append(D)</td>
<td>q2: consume(D)</td>
</tr>
</tbody>
</table>
Algorithm 7.3: Producer-consumer (finite buffer, monitor)

monitor PC

  bufferType buffer ← empty
  condition notEmpty
  condition notFull

  operation append(datatype V)
    if buffer is full
      waitC(notFull)
      append(V, buffer)
      signalC(notEmpty)

  operation take()
    datatype W
    if buffer is empty
      waitC(notEmpty)
      W ← head(buffer)
      signalC(notFull)
    return W
The producer-consumer problem with a finite buffer

C-- implementation implements bufferIsFull and bufferIsEmpty functions with an int count variable.
// File: alg-7-3.cm
// Stan Warford
// Ben-Ari, Algorithm 7.3

monitor PC {
    const int N = 5;
    int oldest, newest;
    int count;
    int buffer[N];
    condition notEmpty;
    condition notFull;

    int bufferIsFull() {
        return count == N;
    }

    int bufferIsEmpty() {
        return count == 0;
    }
}
void append(int v, int prodID) {
    if (bufferIsFull()) {
        waitc(notFull);    // waitc in the body of an if
    }
    buffer[newest] = v;
    newest = (newest + 1) % N;
    count++;
    cout << "Producer " << prodID << " put " << v << endl;
    signalc(notEmpty);    // signalc the last statement of the operation
}

int take(int consID) {
    int w;
    if (bufferIsEmpty()) {
        waitc(notEmpty);    // waitc in the body of an if
    }
    w = buffer[oldest];
    oldest = (oldest + 1) % N;
    count--;
    cout << "Consumer " << consID << " got " << w << endl;
    signalc(notFull);    // signalc the last possible statement of the operation
    return w;
}
init {
    oldest = 0;
    newest = 0;
    count = 0;
}

void producer(int producerID) {
    int i, d;
    for (i = 0; i < 10; i++) {
        d = 10 * producerID + i;
        append(d, producerID);
    }
    cout << "Producer " << producerID << " finished." << endl;
}

void consumer(int consumerID) {
    int i, d;
    for (i = 0; i < 10; i++) {
        d = take(consumerID);
    }
    cout << "Consumer " << consumerID << " finished." << endl;
}

void main() {
    cobegin { producer(0); producer(1); consumer(0); consumer(1); }
}
Java monitors

There is no special monitor type.

Any class can be a monitor.

The keyword `synchronized` makes a method atomic.
The producer-consumer problem
with a finite buffer

Java implementation has four classes/files:

* BAListing0703.java for main program
* PCMonitor.java for the monitor
* Producer.java for the producer
* Consumer.java for the consumer
The main program:

* Allocates the monitor
* Allocates the consumer, passing it a pointer to the monitor, so the consumer can access the monitor
* Allocates the producer, passing it a pointer to the monitor, so the producer can assess the monitor
class BAListing0703 {

    public static void main(String[] args) {
        PCMonitor pCMonitor = new PCMmonitor();
        Consumer consumer = new Consumer(pCMonitor);
        consumer.start();
        Producer producer = new Producer(pCMonitor);
        producer.start();
        try {
            consumer.join();
            producer.join();
        } catch (InterruptedException e) {
        }
    }
}
PCMonitor.java

/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
// Modified by Stan Warford
// Added System.out in monitor to prevent interleaving of output trace statements.

class PCMonitor {
    final int N = 5;
    int Oldest = 0, Newest = 0;
    volatile int Count = 0;
    int Buffer[] = new int[N];

    synchronized void Append(int V) {
        while (Count == N) // Waiting processes loop on their conditions
            try {
                wait();
            } catch (InterruptedException e) {
            }
        Buffer[Newest] = V;
        Newest = (Newest + 1) % N;
        Count = Count + 1;
        System.out.println("Producer put " + V);
        notifyAll(); // Signaller executes notifyAll
    }
}
synchronized int Take() {
    int temp;
    while (Count == 0) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    temp = Buffer[Oldest];
    Oldest = (Oldest + 1) % N;
    Count = Count - 1;
    System.out.println("Consumer got " + temp);
    notifyAll();
    return temp;
}

Waiting processes loop on their conditions

Signaller executes notifyAll
// Stan Warford
// BAListing00703
// File: Producer.java

class Producer extends Thread {
    private PCMonitor pCMonitor;

    Producer(PCMonitor pCMonitor) {
        this.pCMonitor = pCMonitor;
    }

    public void run() {
        int delay, d;
        System.out.println("Producer started.");
        for (int i = 0; i < 15; i++) {
            try {
                delay = (int) (60 * Math.random());
                Thread.sleep(delay);
                d = 10 * i;
                pCMonitor.Append(d);
            } catch (InterruptedException e) {
                System.out.println("Producer interrupted.");
            }
        }
        System.out.println("Producer finished.");
    }
}
// Stan Warford
// BAListing0703
// File: Consumer.java

class Consumer extends Thread {
    private PCMonitor pCMonitor;

    Consumer(PCMonitor pCMonitor) {
        this.pCMonitor = pCMonitor;
    }

    public void run() {
        int delay, d;
        System.out.println("Consumer started.");
        for (int i = 0; i < 15; i++) {
            try {
                delay = (int) (100 * Math.random());
                Thread.sleep(delay);
                d = pCMonitor.Take(); // Ignore returned value
            } catch (InterruptedException e) {
                System.out.println("Consumer interrupted.");
            }
        }
        System.out.println("Consumer finished.");
    }
}
The readers and writers problem

* There is a shared database with many readers and writers.

* There can be many readers at one time.

* But there can only be one writer.

* Following solution is starvation-free.
Algorithm 7.4: Readers and writers with a monitor

monitor RW
  integer readers ← 0
  integer writers ← 0
  condition OKtoRead, OKtoWrite
operation StartRead
  if writers ≠ 0 or not empty(OKtoWrite)
    waitC(OKtoRead)  \[\text{waitc in the body of an if}\]
  readers ← readers + 1
  signalC(OKtoRead)  \[\text{signalc the last statement of the operation}\]
operation EndRead
  readers ← readers − 1
  if readers = 0
    signalC(OKtoWrite)  \[\text{signalc the last statement of the operation}\]
Algorithm 7.4: Readers and writers with a monitor (continued)

**operation StartWrite**
- if writers ≠ 0 or readers ≠ 0
  - waitC(OKtoWrite)
  - writers ← writers + 1

**operation EndWrite**
- writers ← writers − 1
  - if empty(OKtoRead)
    - then signalC(OKtoWrite)
    - else signalC(OKtoRead)

<table>
<thead>
<tr>
<th>reader</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: RW.StartRead</td>
<td>q1: RW.StartWrite</td>
</tr>
<tr>
<td>p2: read the database</td>
<td>q2: write to the database</td>
</tr>
<tr>
<td>p3: RW.EndRead</td>
<td>q3: RW.EndWrite</td>
</tr>
</tbody>
</table>
Readers and Writers

* New readers are blocked if a writer is active or a writer is blocked.

* When one reader enters, it unblocks any other reader trying to enter.

* Consequently, when one unblocked reader enters, all other blocked readers enter.

* The last reader unblocks any blocked writer.
Readers and Writers

* New writers are blocked if any readers are active or if a writer is active.

* An exiting writer unblocks any blocked reader rather than a blocked writer. Consequently, all blocked readers enter.

* Any subsequent incoming readers will be blocked by the blocked writer, who will eventually enter when the last active reader exits.
The dining philosopher’s problem

* fork[i] is how many forks are available to philosopher[i]. Initialized to 2 because two forks are initially available.
The dining philosopher’s problem

* fork[i] is how many forks are available to philosopher[i]. Initialized to 2 because two forks are initially available.

* Before eating, decrement number of forks available to neighbor by 1 each. No interleaving.
Algorithm 7.5: Dining philosophers with a monitor

monitor ForkMonitor
    integer array[0..4] fork ← [2, . . . , 2]
    condition array[0..4] OKtoEat
operation takeForks(integer i)
    if fork[i] ≠ 2
        waitC(OKtoEat[i])
    fork[i+1] ← fork[i+1] − 1
    fork[i−1] ← fork[i−1] − 1

operation releaseForks(integer i)
    fork[i+1] ← fork[i+1] + 1
    fork[i−1] ← fork[i−1] + 1
    if fork[i+1] = 2
        signalC(OKtoEat[i+1])
    if fork[i−1] = 2
        signalC(OKtoEat[i−1])
Algorithm 7.5: Dining philosophers with a monitor (continued)

<table>
<thead>
<tr>
<th>philosopher i</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: think</td>
</tr>
<tr>
<td>p2: takeForks(i)</td>
</tr>
<tr>
<td>p3: eat</td>
</tr>
<tr>
<td>p4: releaseForks(i)</td>
</tr>
</tbody>
</table>
The dining philosopher’s problem

* This solution has mutual exclusion and is deadlock-free but can starve.
### Scenario for starvation of Philosopher 2

<table>
<thead>
<tr>
<th>n</th>
<th>phil1</th>
<th>phil2</th>
<th>phil3</th>
<th>$f_0$</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>take(1)</strong></td>
<td>take(2)</td>
<td>take(3)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>release(1)</td>
<td>take(2)</td>
<td><strong>take(3)</strong></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>release(1)</td>
<td><strong>take(2) and waitC(OK[2])</strong></td>
<td>release(3)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td><strong>release(1)</strong></td>
<td>(blocked)</td>
<td>release(3)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td><strong>take(1)</strong></td>
<td>(blocked)</td>
<td>release(3)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>release(1)</td>
<td>(blocked)</td>
<td>release(3)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>release(1)</td>
<td>(blocked)</td>
<td><strong>take(3)</strong></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
C++11

Latest standard set in 2011.
C++11

Latest standard set in 2011.
“Hello World” thread

#include <iostream>
#include <thread>

void hello() {
    std::cout<<"Hello Concurrent World\n";
}

int main() {
    std::thread t(hello);
    t.join();
}
```cpp
#include <cstdlib>
#include <iostream>
#include <thread>

void pRun() {
    std::cout << "Hello Concurrent World from p\n";
}

void qRun() {
    std::cout << "Hello Concurrent World from q\n";
}

int main(int argc, char** argv) {
    std::thread p(pRun);
    std::thread q(qRun);
    p.join();
    q.join();
    return 0;
}
```

**Output**
Hello Concurrent World from p
Hello Concurrent World from q
C++11 types for concurrency

mutex, a binary semaphore with lock() for wait and unlock() for signal.
std::mutex okToWrite;

void pRun() {
    okToWrite.lock();
    std::cout << "Hello Concurrent World from p\n";
    okToWrite.unlock();
}

void qRun() {
    okToWrite.lock();
    std::cout << "Hello Concurrent World from q\n";
    okToWrite.unlock();
}

int main(int argc, char** argv) {
    std::thread p(pRun);
    std::thread q(qRun);
    p.join();
    q.join();
    return 0;
}

Output
Hello Concurrent World from p
Hello Concurrent World from q
C++11 types for concurrency

mutex, a binary semaphore with lock() for wait and unlock() for signal.

lambda expressions for passing unnamed functions as actual parameters.
#include <cstdlib>
#include <iostream>
#include <thread>
#include <functional>

int main(int argc, char** argv) {
    std::thread t([](){std::cout << "Hello Concurrent World\n";});
    t.join();
    return 0;
}
C++11 types for concurrency

mutex, a binary semaphore with lock() for wait and unlock() for signal.

lambda expressions for passing unnamed functions as actual parameters to wait().

atomic<>, specialized for integer types to provide atomic integer operations.
#include <atomic>

struct AtomicCounter {
    std::atomic<int> value;

    void increment(){
        ++value;
    }

    void decrement(){
        --value;
    }

    int get(){
        return value.load();
    }
};
mutex types for concurrency

mutex, a binary semaphore with lock() for wait and unlock() for signal.

lambda expressions for passing unnamed functions as actual parameters to wait().

atomic<> specialized for integer types to provide atomic integer operations.

condition_variable associated with a mutex and wait() for waitC and notify_one() for signalC.