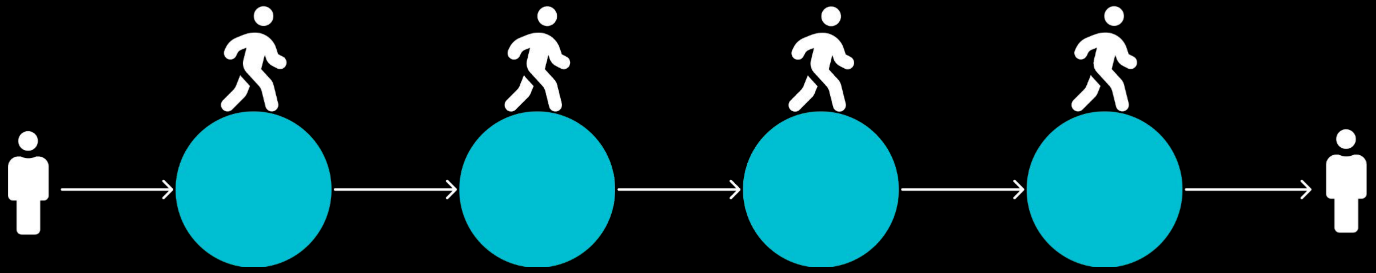
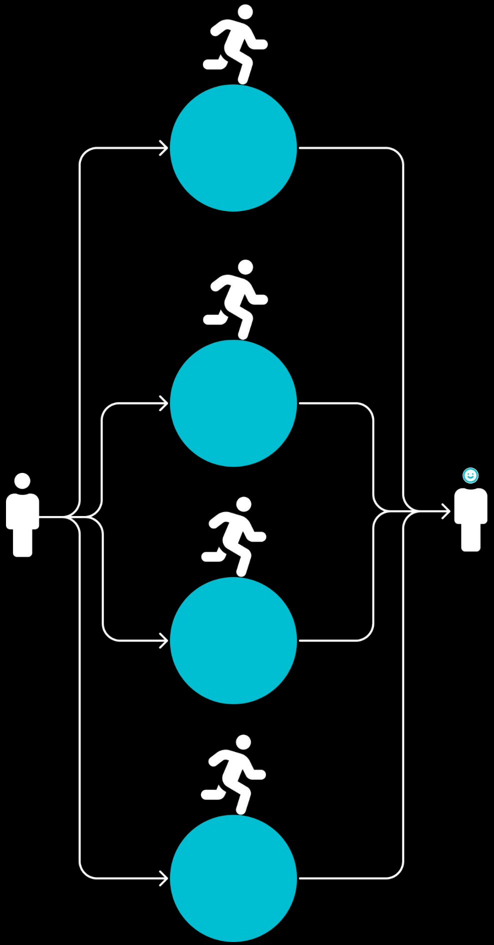
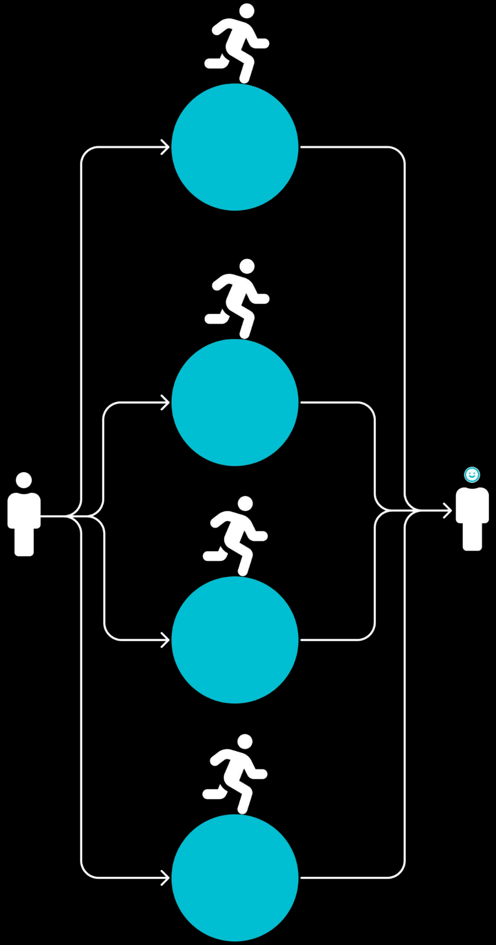
Untangled

Caleb Ledi
Duru Ugurlu
Luke Deen Taylor
Nick Doan
Chloe Lam

Introduction & Motivation

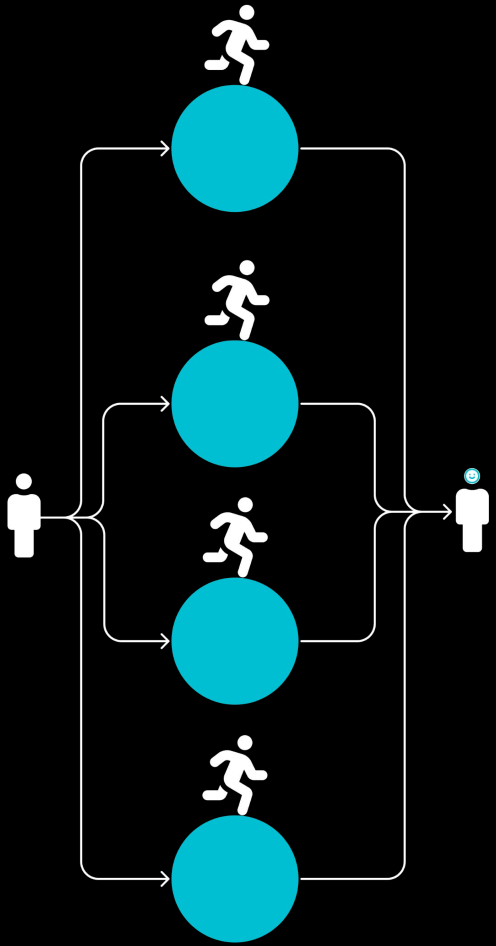






Other Coding Language

```
Unnecessarily long code
Unnecessarily long
codeUnnecessarily long
codeUnnecessarily long
codeUnnecessarily long
codeUnnecessarily long code
Unnecessarily long
codeUnnecessarily long
codeUnnecessarily long
codeUnnecessarily long code
Unnecessarily long
codeUnnecessarily long code
Unnecessarily long
codeUnnecessarily long
codeUnnecessarily long
codeUnnecessarily long code
Unnecessarily long
codeUnnecessarily long code
```



Untangled

Easy to understand code
Simple syntax
Flexibility

Tutorial to Untangled

First Untangled Program!!!

```
thread_def Main {
  print("Hello, world!");
}
```

```
thread_def Main {
  print("Hello, world!");
}
```

```
./untangled.exe hello.unt -o hello
```

```
thread_def Main {
  print("Hello, world!");
}
```

```
./untangled.exe hello.unt -o hello
```

```
./hello
# you should see "Hello, world!"
```

```
thread_def Main {
  print("Hello, world!");
}
```

```
./untangled.exe hello.unt -o hello
```

```
./hello
# you should see “Hello, world!”
```

First Untangled Program

Something a little more complicated…

```
thread_def Main {
  // Control flow (for loops, conditionals) look like C/C++
  for (int num = 2; num <= 20; num++) {
    // Variables are statically typed and mutable
    bool is_prime = true;

    /* Check whether `num` is a prime number:
     * If any number 2..num-1 divides `num` evenly, then it’s not prime */
    for (int divisor = 2; divisor < num; divisor++) {
      if (num % divisor == 0) is_prime = false;
    }

    /* Report our results */
    if (is_prime) {
      // “print” only takes strings, but the standard library includes
      // common type conversion functions
      print(string_of_int(num) + " is prime\n");
    } else {
      // Strings support concatenation using “+”
      print(string_of_int(num) + " is not prime\n");
    }
  }
}
```

```
./untangled.exe primes.unt -o primes
./primes

# 2 is prime
# 3 is prime
# 4 is not prime
# ...
# 19 is prime
# 20 is not prime
```

Introduction to Multithreading

```
thread_def MyThread {
  print("Hello from MyThread\n");
}
```

```
thread_def MyThread {
  print("Hello from MyThread\n");
}

thread_def Main {
  // Spawn a thread to run the MyThread procedure
  spawn MyThread;
}
```

```
thread_def MyThread {
  print("Hello from MyThread\n");
}

thread_def Main {
  // "t" is the reference to the thread we spawned
  thread t = spawn MyThread;
  // The << operator is used to send a message to the thread
  t << "Hello from Main\n";
}
```

```
thread_def MyThread {
  receive {
    // should be "Hello from Main"
    string msg → print(msg);
    _ → exit(1);
  }
}

thread_def Main {
  // "t" is the reference to the thread we spawned
  thread t = spawn MyThread;
  // The << operator is used to send a message to the thread
  t << "Hello from Main\n";
}
```

```
thread_def MyThread {
  receive {
    string s → print("I got a string: " + s + "\n");
    int i → print("I got an int: " + string_of_int(i) + "\n");
    _ → exit(1);
  }
}

thread_def Main {
  thread t1 = spawn MyThread;
  thread t2 = spawn MyThread;
  t1 << "hello";
  t2 << 3;
}
```

Multithreaded Primes Program

```
thread_def PrimeCalculator {
  // A message will tell us which number we should check
  int num;
  receive {
    int n -> num = n;
    _ -> exit(1);
  }

  // Do our calculation (checking primality)
  bool is_prime = true;
  for (int divisor = 2; divisor < num; divisor++) {
    if (num % divisor == 0) is_prime = false;
  }

  // Print the results
  if (is_prime) {
    print(string_of_int(num) + " is prime\n");
  } else {
    print(string_of_int(num) + " is not prime\n");
  }
}

thread_def Main {
  // Spawn 20 threads to check numbers 1 to 20
  for (int num = 2; num <= 20; num++) {
    thread t = spawn PrimeCalculator;
    t << num;
  }
}
```

```
2 is prime
5 is prime
4 is not prime
7 is prime
3 is prime
8 is not prime
9 is not prime
6 is not prime
...
```

Tuples

```
// Define a tuple
(int, (string, bool)) x = (5, ("hi", true));
// Unpack a tuple’s values
(int i, (string s, bool b)) = x;
// Show the unpacked values
print(string_of_int(i) + " " + s + " " + string_of_bool(b));
// prints “5 hi true”
```

```
thread_def PrimeCalculator {
  int num;
  receive {
    int n -> num = n;
    _ -> exit(1);
  }

  bool is_prime = true;
  for (int divisor = 2; divisor < num; divisor++) {
    if (num % divisor == 0) is_prime = false;
  }

  // the "parent" keyword automatically refers to the thread that spawned
  // this one
  parent << (num, is_prime);
}
```

```
thread_def Main {
  // Spawn 20 threads to check numbers 1 to 20
  for (int num = 2; num <= 20; num++) {
    thread t = spawn PrimeCalculator;
    t << num;
  }

  // Receive and print the results
  for (int num = 2; num <= 20; num++) {
    receive {
      (int n, bool is_prime) -> {
        if (is_prime) {
          print(string_of_int(n) + " is prime\n");
        } else {
          print(string_of_int(n) + " is not prime\n");
        }
      }
      _ -> exit(1);
    }
  }
}
```

Arrays

```
thread_def Main {
  // Create the array to hold the results
  bool[21] results_arr;

  // Spawn 20 threads to check numbers 1 to 20
  for (int num = 2; num ≤ 20; num++) {
    thread t = spawn PrimeCalculator;
    t << num;
    // Give the thread the results array
    t << results_arr;
  }

  // Wait for the threads to tell us they're finished
  for (int num = 2; num ≤ 20; num++) {
    receive { string x → {} _ → exit(1); }
  }

  // Print the results
  for (int num = 2; num ≤ 20; num++) {
    print(string_of_int(num) + ": " + string_of_bool(
      results_arr[num]
    ) + "\n");
  }
}
```

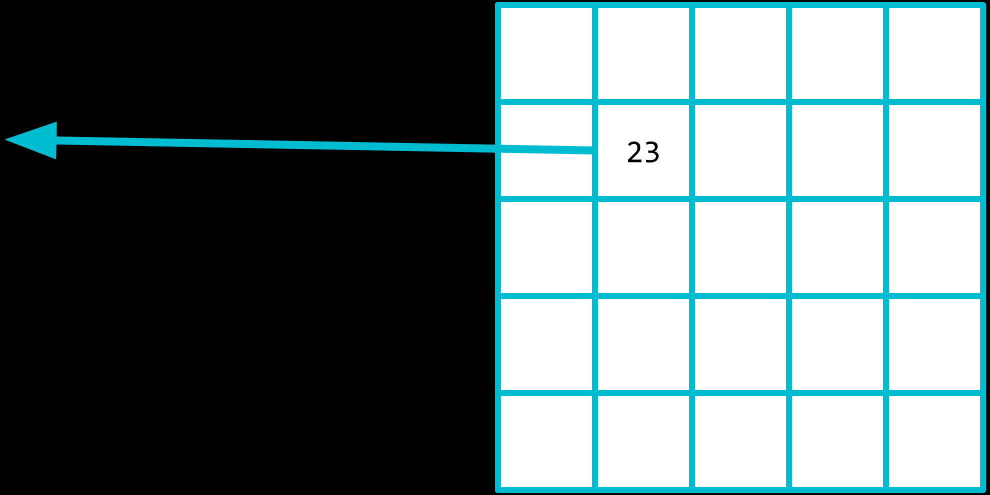
```
thread_def PrimeCalculator {
  int num;
  receive { int n -> num = n; _ -> exit(1); }

  bool is_prime = true;
  for (int divisor = 2; divisor < num; divisor++) {
    if (num % divisor == 0) is_prime = false;
  }

  // Wait to receive the "results" array, and store the result we found
  receive { bool[21] results -> results[num] = is_prime; _ -> exit(1); }
  // Signal our success to the parent thread, so that it can know when
  // it's safe to read the results array.
  parent << "done";
}
```

```
thread_def Main {
  // Create the array to hold the results
  bool[21] results_arr;

  // Spawn 20 threads to check numbers 1 to 20
  for (int num = 2; num <= 20; num++) {
    thread t = spawn PrimeCalculator;
    t << num;
    // Give the thread the results array
    t << results_arr;
  }

  // Wait for the threads to tell us they're finished
  for (int num = 2; num <= 20; num++) {
    receive { string x -> {} _ -> exit(1); }
  }

  // Print the results
  for (int num = 2; num <= 20; num++) {
    print(string_of_int(num) + ": " + string_of_bool(
      results_arr[num]
    ) + "\n");
  }
}
```

Semaphores

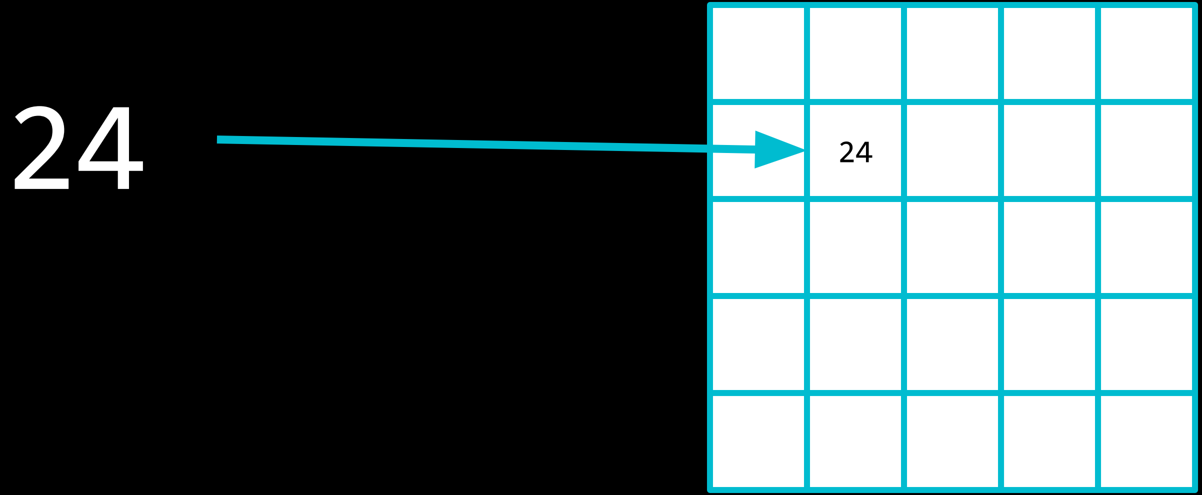
```
thread_def Incrementor {
  // Receive an array (shared memory)
  receive {
    // Mutate the shared memory
    int[1] a -> a[0] += 1;
    _ -> exit(1);
  }
  // Signal we're done
  parent << "done";
}

thread_def Main {
  // Create a block of shared memory
  int[1] arr = [0];
  // Spawn 10000 threads that try to increment the shared memory simultaneously
  for (int i = 0; i < 10000; i++) {
    thread t = spawn Incrementor;
    t << arr;
  }
  // Wait for all threads to signal done
  for (int i = 0; i < 10000; i++) {
    receive { string s -> {} _ -> exit(1); }
  }
  // See what result we got
  print(string_of_int(arr[0]) + "\n");
}
```

23

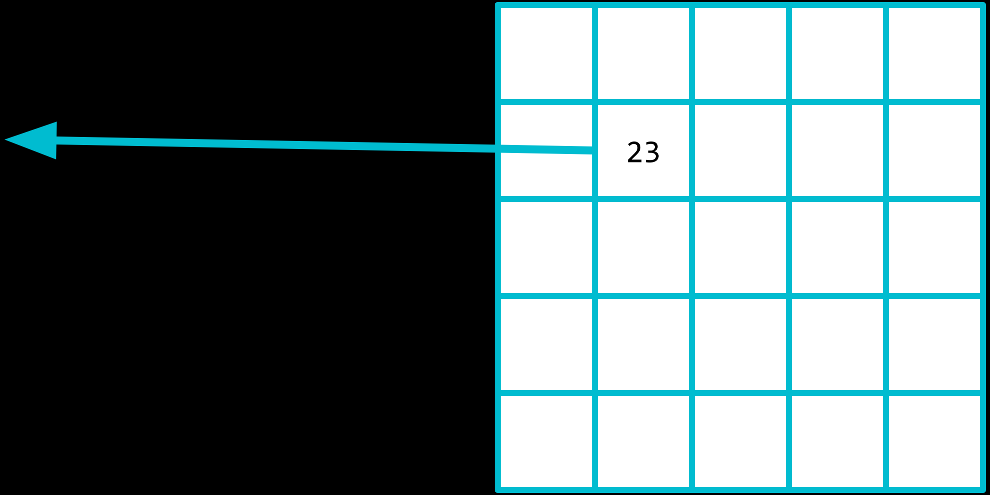


24

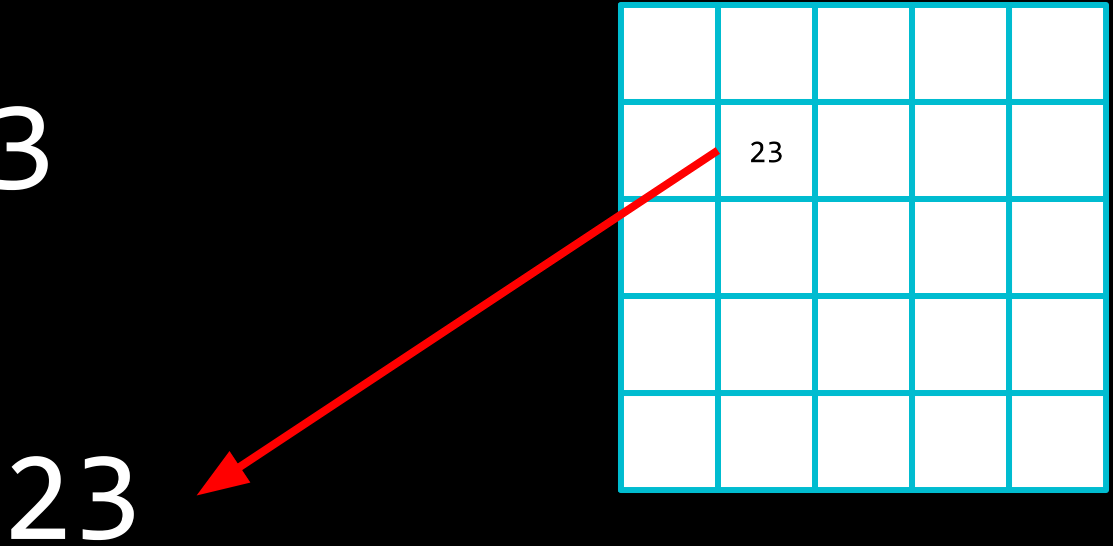
	23			


24
24

23



23



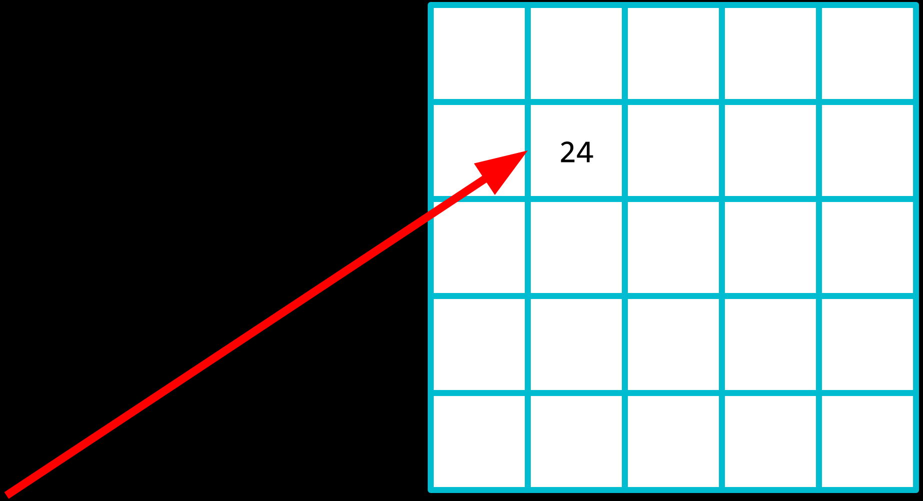
23

24

	23			

23

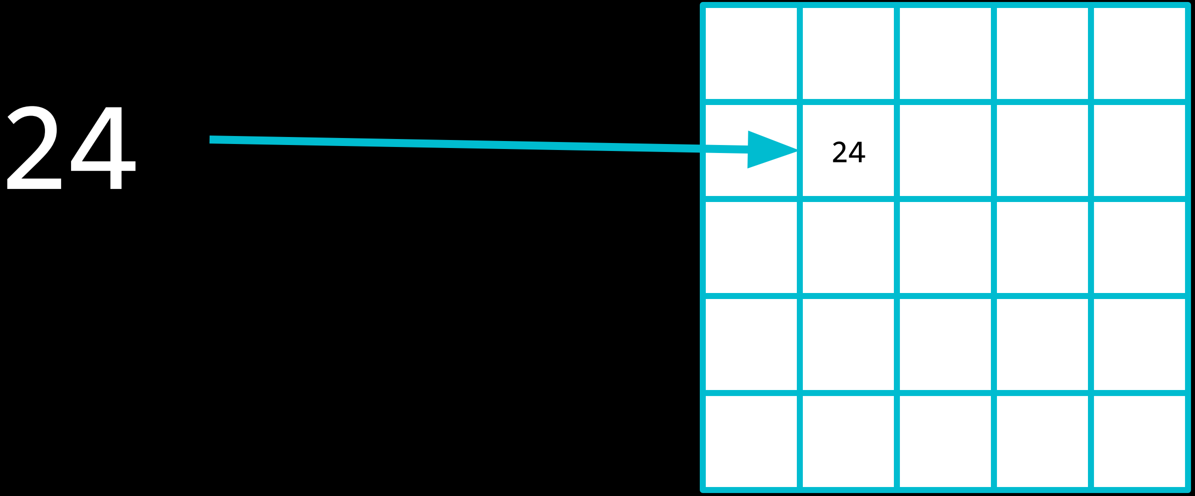
24



24

	24			

24

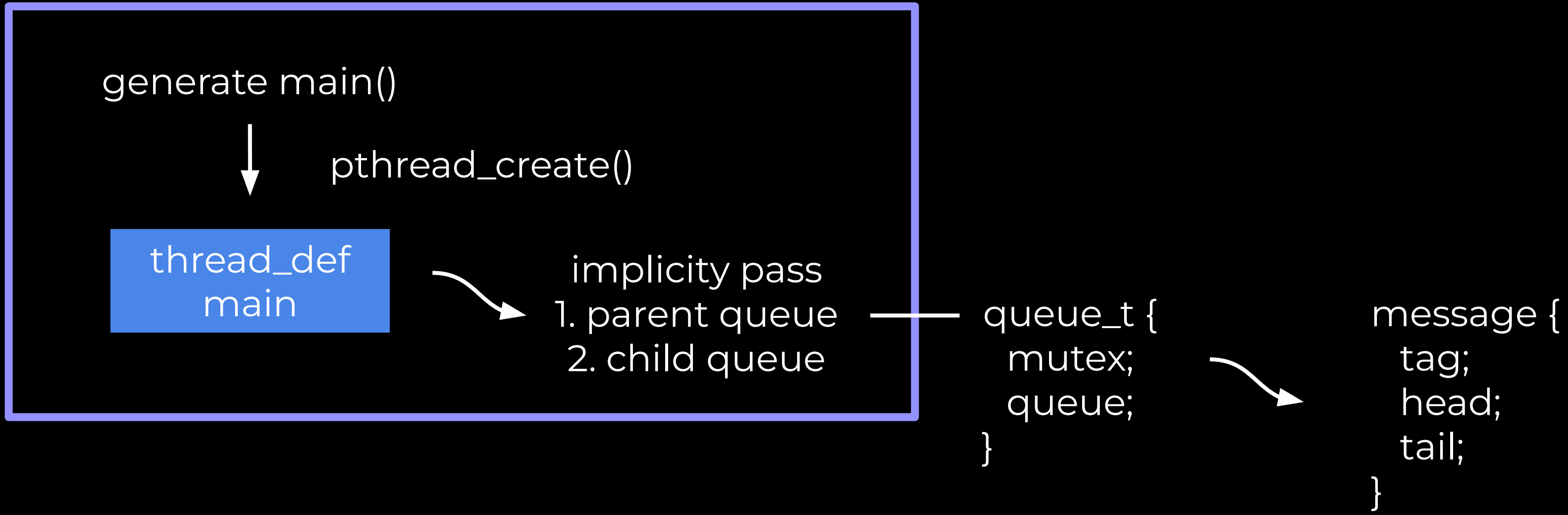


```
thread_def Main {
  // Create a block of shared memory
  int[1] arr = [0];
  // Create the semaphore that will guard access
  semaphore shared_sem = make_semaphore(1);

  for (int i = 0; i < 10000; i++) {
    thread t = spawn Incrementor;
    t << shared_sem;
    t << arr;
  }

  // Wait for all threads to signal done
  for (int i = 0; i < 10000; i++) {
    receive { string s -> {} _ -> exit(1); }
  }
  // See what result we got
  print(string_of_int(arr[0]) + "\n");
}
```

```
thread_def Incrementor {
  // Receive the semaphore
  semaphore sem;
  receive {
    semaphore s -> sem = s;
    _ -> exit(1);
  }
  // Receive the array (shared memory)
  receive {
    int[1] a -> {
      // “lock” the semaphore
      // if another thread gets there first, we’ll wait
      sem--;
      // Access shared memory only once we get past the “lock”
      a[0] += 1;
      // Allow another thread to enter the protected section of code
      sem++;
    }
    _ -> {}
  }
  // Signal we’re done
  parent << "done";
}
```

```
thread_def Main {
  // Create a block of shared memory
  int[1] arr = [0];
  // Create the semaphore that will guard access
  semaphore shared_sem = make_semaphore(1);

  for (int i = 0; i < 10000; i++) {
    thread t = spawn Incrementor;
    t << shared_sem;
    t << arr;
  }

  // Wait for all threads to signal done
  for (int i = 0; i < 10000; i++) {
    receive { string s -> {} _ -> exit(1); }
  }
  // See what result we got
  print(string_of_int(arr[0]) + "\n");
}
```

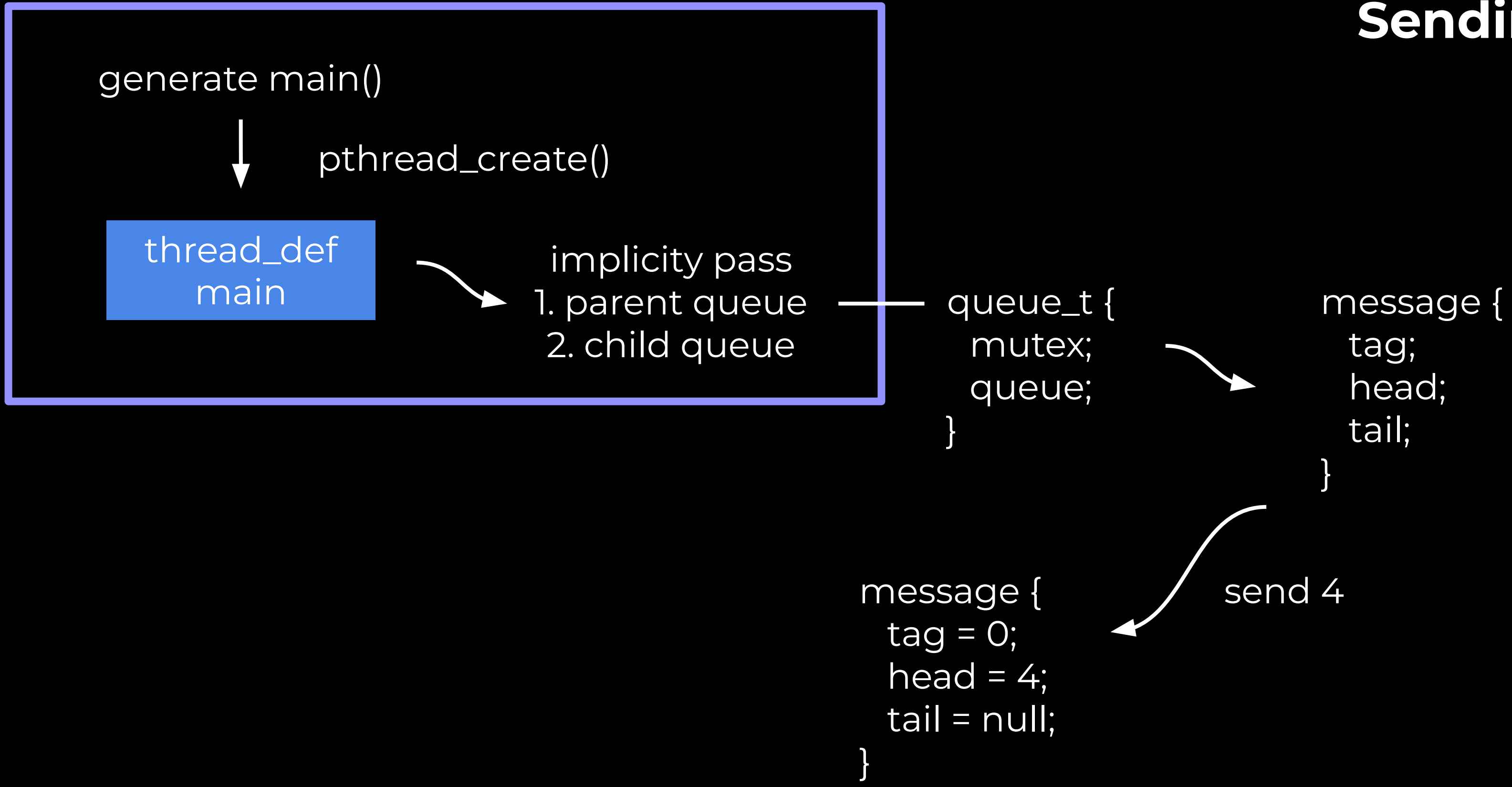
Implementation



```
queue_t {
  mutex;
  queue;
}
```

```
message {
  tag;
  head;
  tail;
}
```

Sending 4

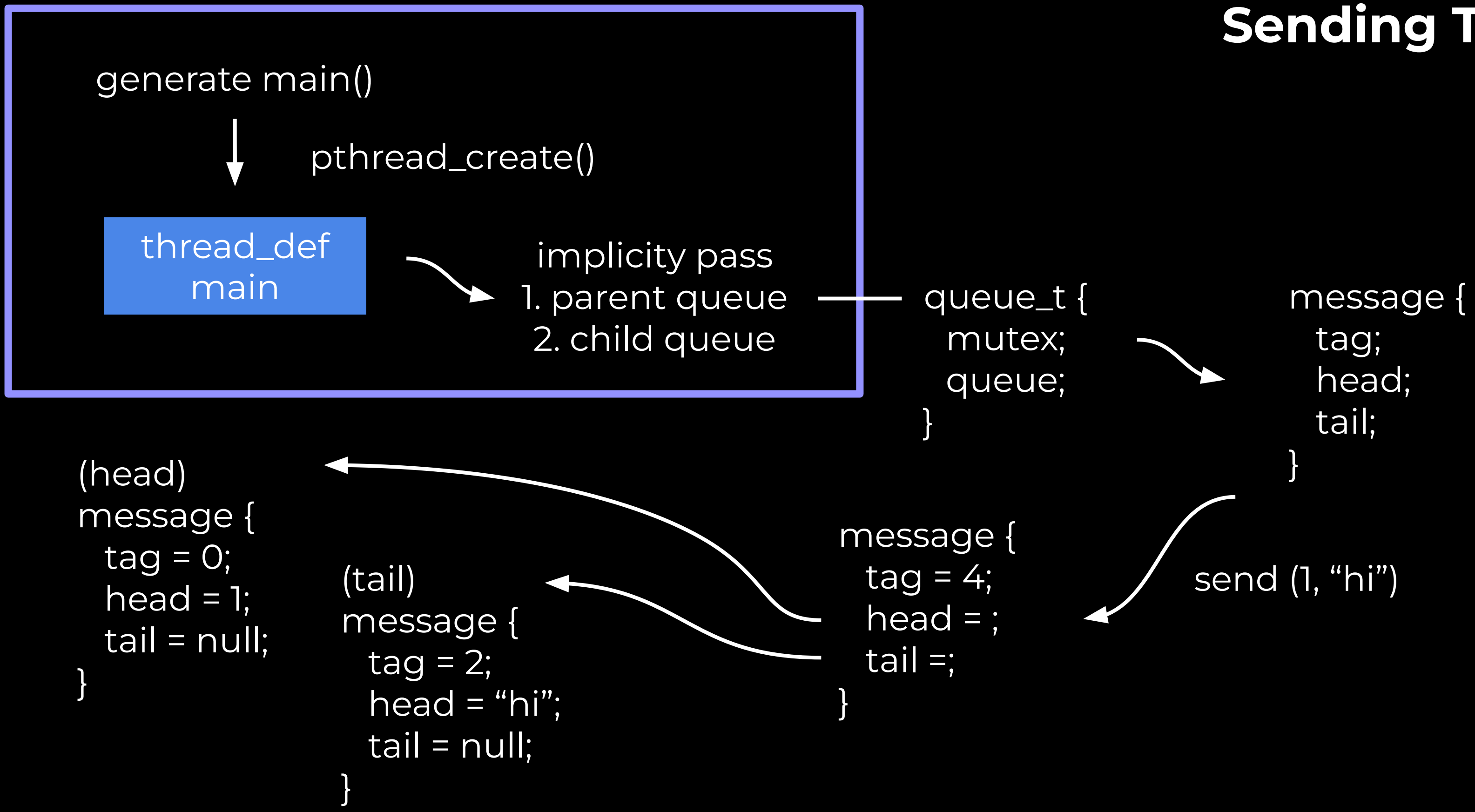


```
queue_t {
  mutex;
  queue;
}
```

```
message {
  tag;
  head;
  tail;
}
```

```
message {
  tag = 0;
  head = 4;
  tail = null;
}
```

Sending Tuple



```
queue_t {
  mutex;
  queue;
}
```

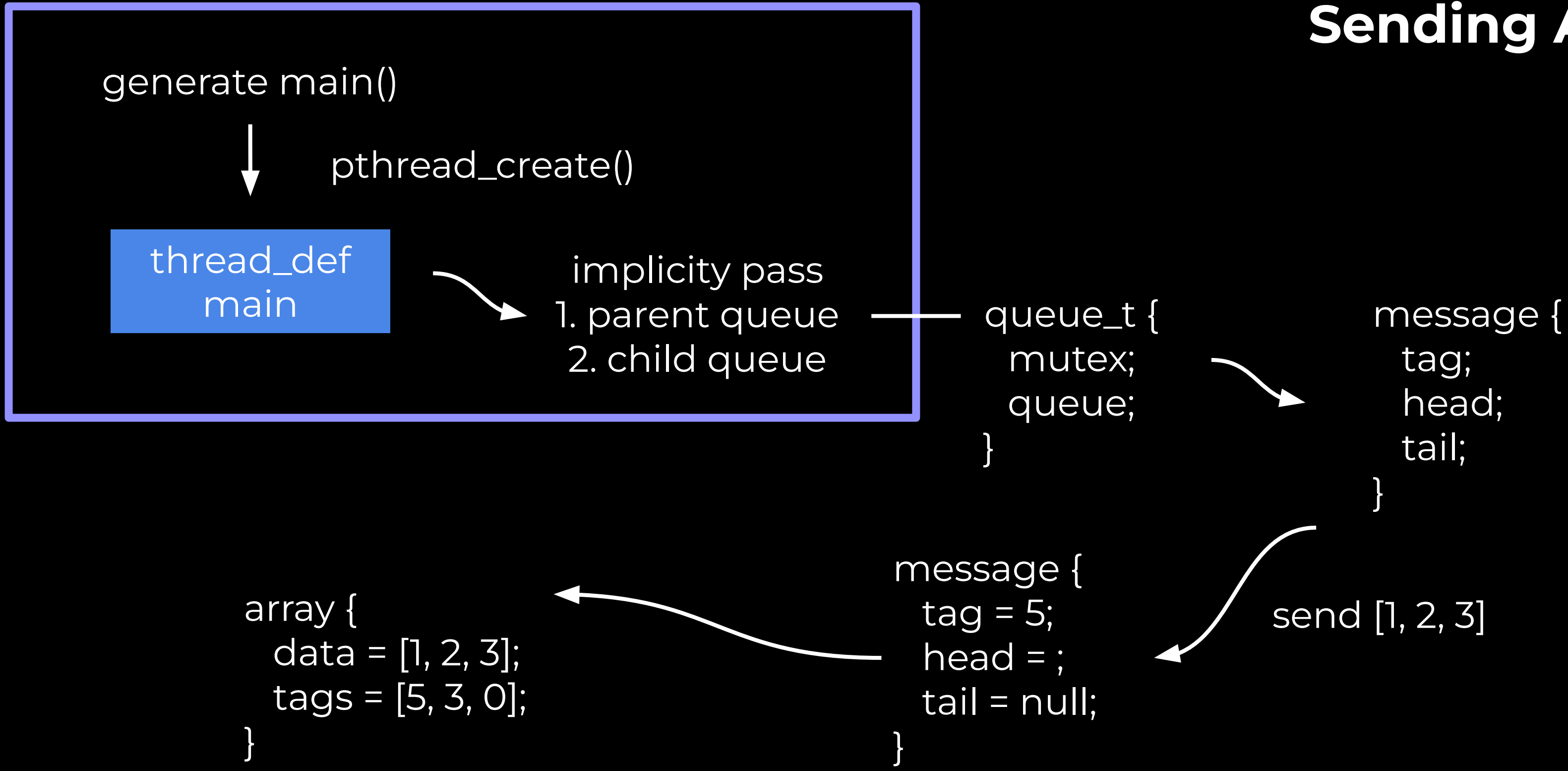
```
message {
  tag;
  head;
  tail;
}
```

```
message {
  tag = 4;
  head = ;
  tail =;
}
```

```
(head)
message {
  tag = 0;
  head = 1;
  tail = null;
}
```

```
(tail)
message {
  tag = 2;
  head = “hi”;
  tail = null;
}
```

Sending Array



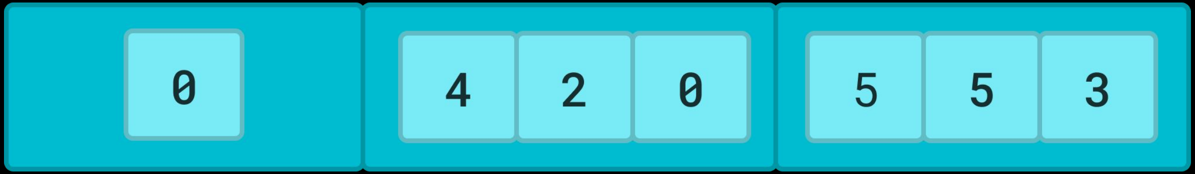
```
queue_t {
  mutex;
  queue;
}
```

```
message {
  tag;
  head;
  tail;
}
```

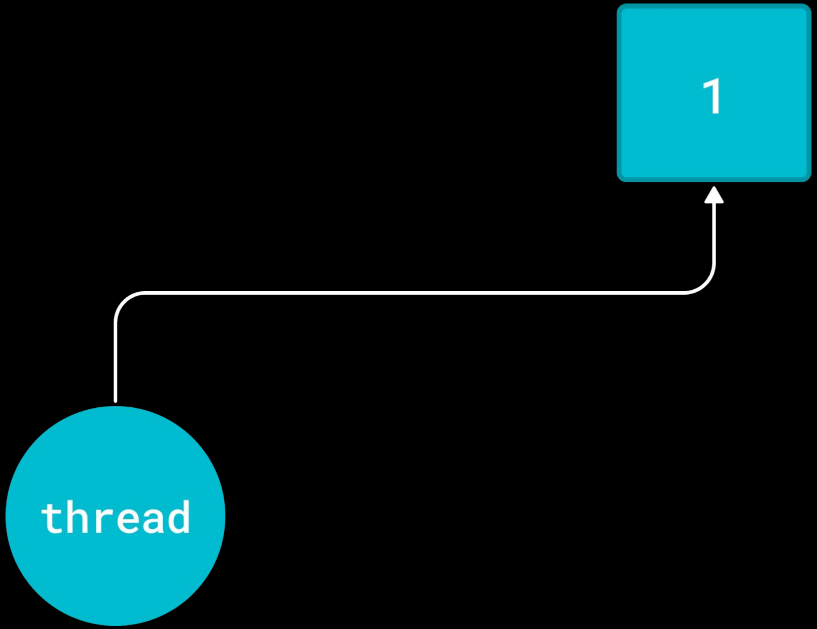
```
message {
  tag = 5;
  head = ;
  tail = null;
}
```

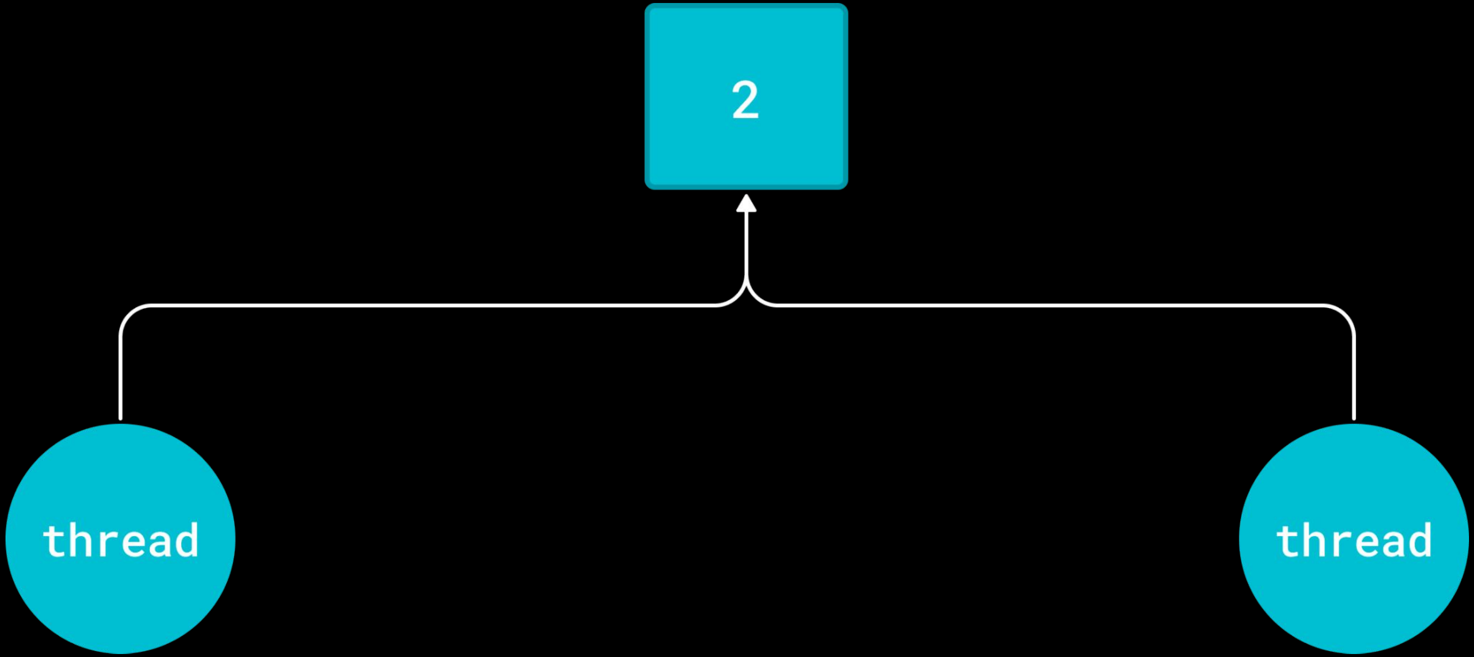
```
array {
  data = [1, 2, 3];
  tags = [5, 3, 0];
}
```

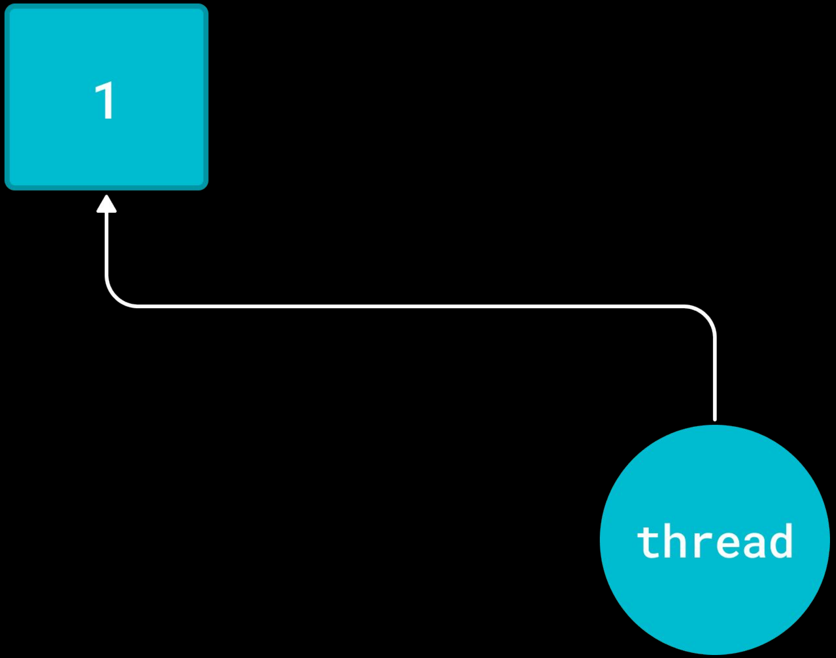
```
receive {
  int x -> print("it's an integer!");
  (string x, int y) -> print("it's a (string, int) tuple!");
  bool[5] arr -> print("it's a bool array!");
  _ -> print("it's none of these");
}
```



0


1
thread


2
thread
thread


1
thread

0

What we learned

- Working on a big project
- Communicating about implementation ideas
- Delegating tasks
- Power of friendship!
- What’s possible at compile time
- Debugging nonsensical error messages

Thank you!