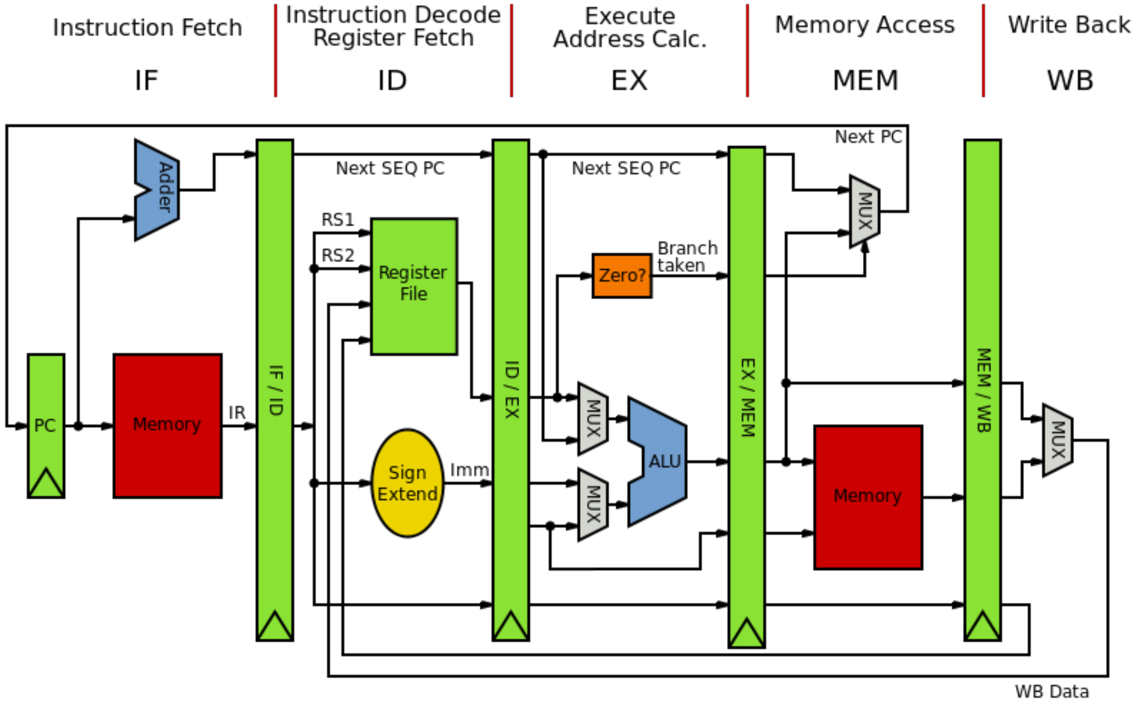
CSE113: Parallel Programming

Jan. 12, 2022

- **Topics**:
  - ILP in reduction loops
  - C++ threads



MIPS pipeline image from:
https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware)

Announcements

- Office hours and tutors are available this week!
  - Announcements on Canvas and Piazza with zoom links for tutors
  - Reese posted his
  - I post mine around noon on Thursday (along with a sign up sheet)

- Homework 1 is released
  - You can get started setting up the docker
  - After today you can do part 2 and 3

- Sign up for Piazza and ask questions on there

Let us know about any typos in the homework!

Today’s Quiz

- We’ll continue having them given after lecture for now

- Make sure to do it!

Previous quiz

How many elements of type double can be stored in a cache line?

Previous quiz

Instructions with the following property should be placed as far apart as possible in machine code:

Instructions that compute floating point values
Instructions that load from memory
Instructions that depend on each other
Instructions that perform the same operation

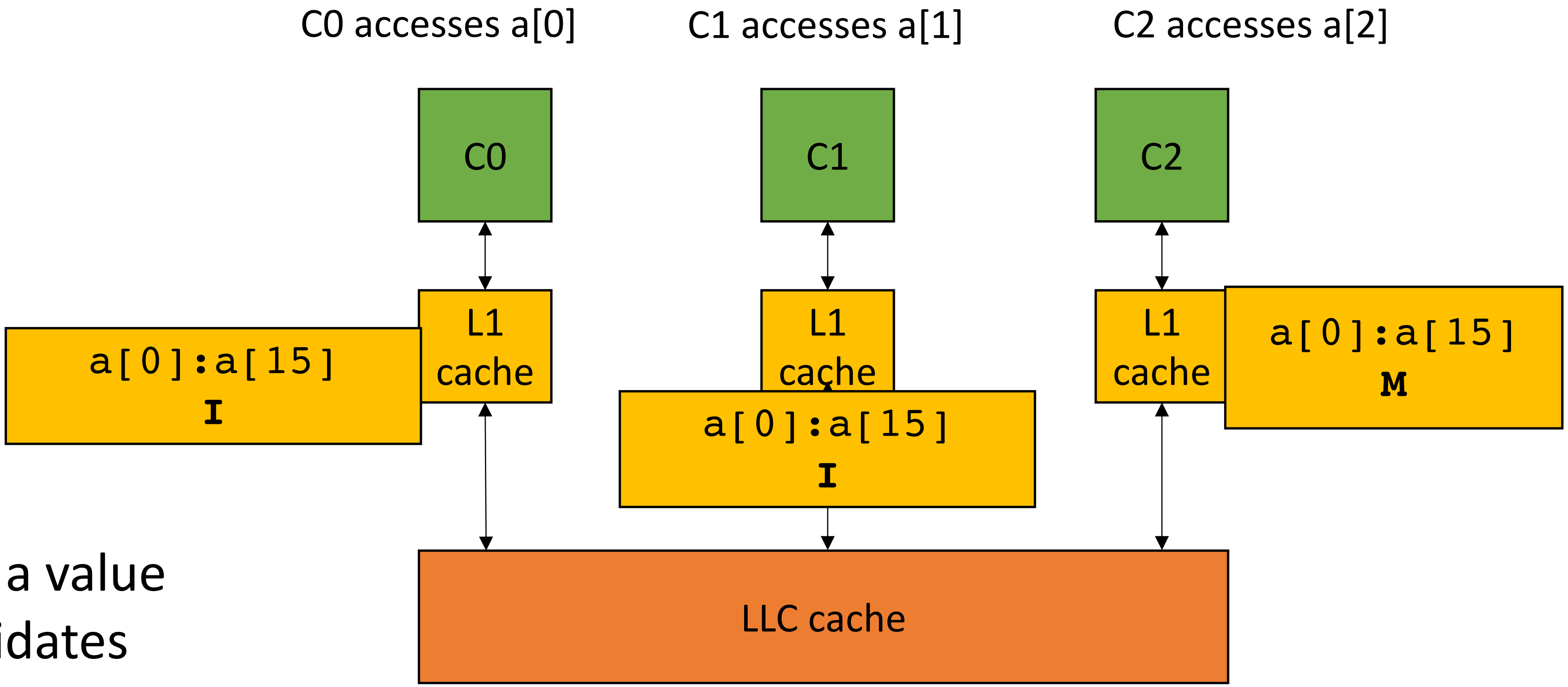
Previous quiz

What does ILP stand for?

- **Interleaved Language Program**
- Instruction Level Parallelism
- Interpreted Latency Pipeline

False sharing

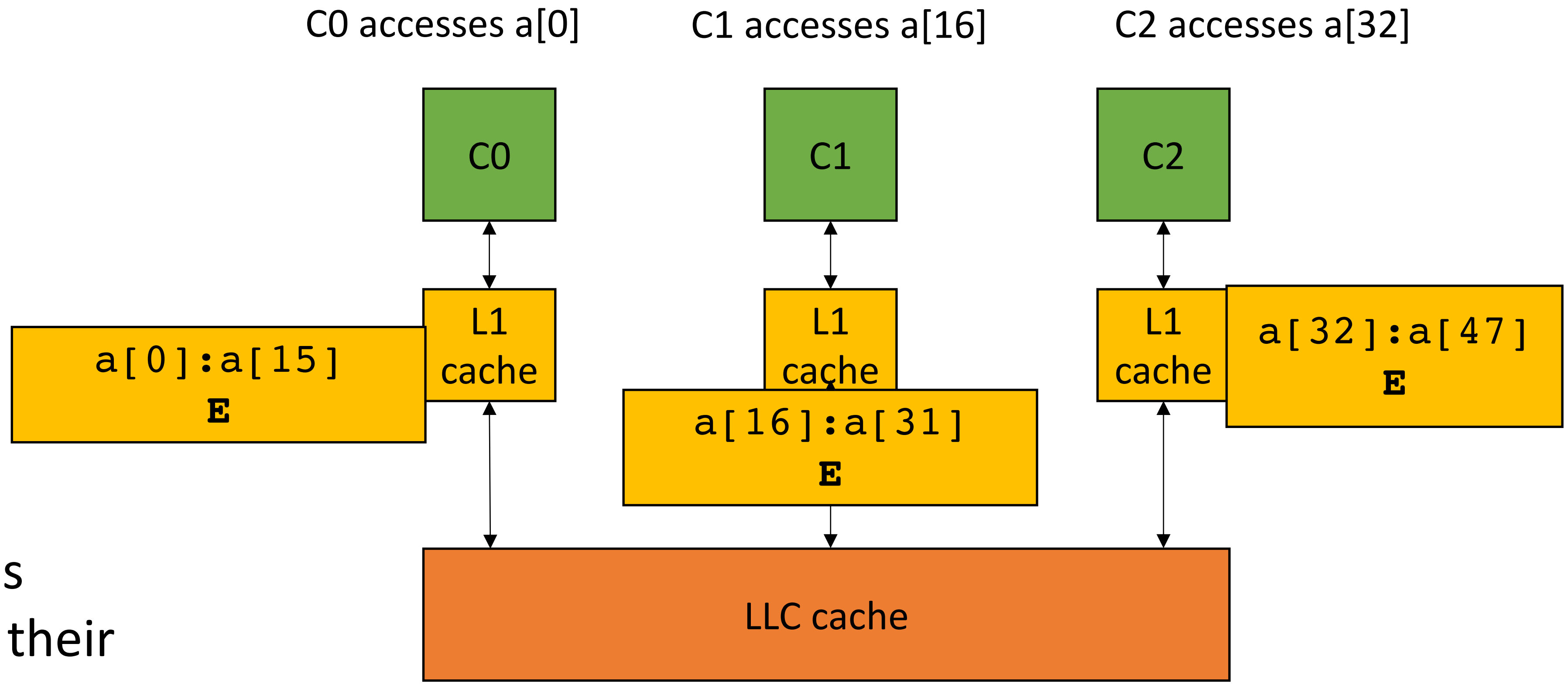
False Sharing



when one core modifies a value in the cache line, it invalidates everyone else's cache line.

This is called ***False Sharing***

Avoid false sharing with padding



With padding, all threads have exclusive access to their lines! No need to trigger invalidations or write-back each operation

Thanks!

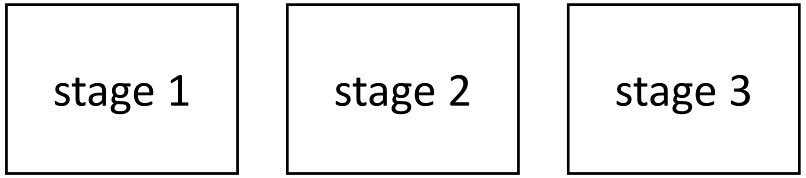
- Thanks for all the interesting answers on quizzes!
- 57 submitted: some people aren’t submitting

Review

- Instruction level parallelism

Pipeline

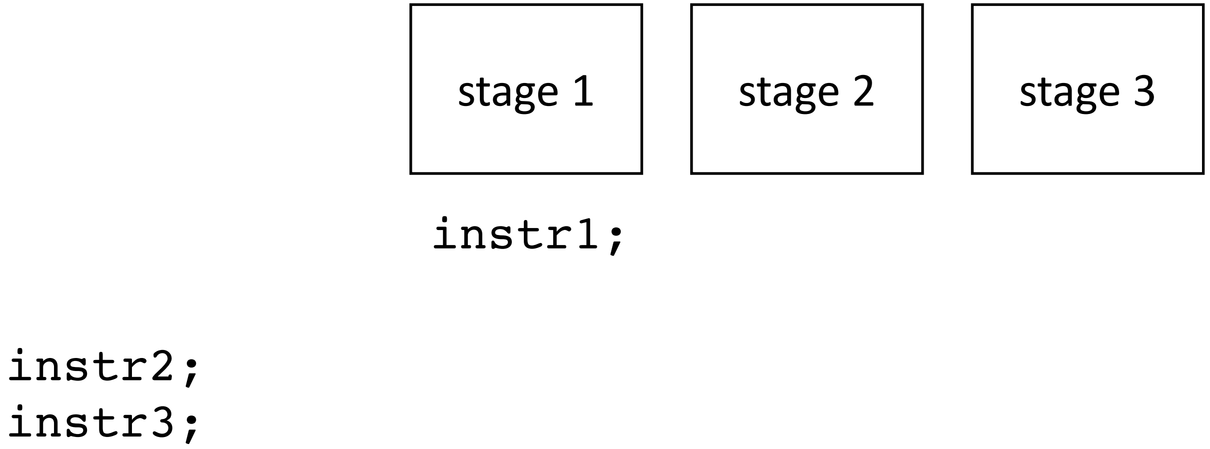
- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline




```
instr1;
instr2;
instr3;
```

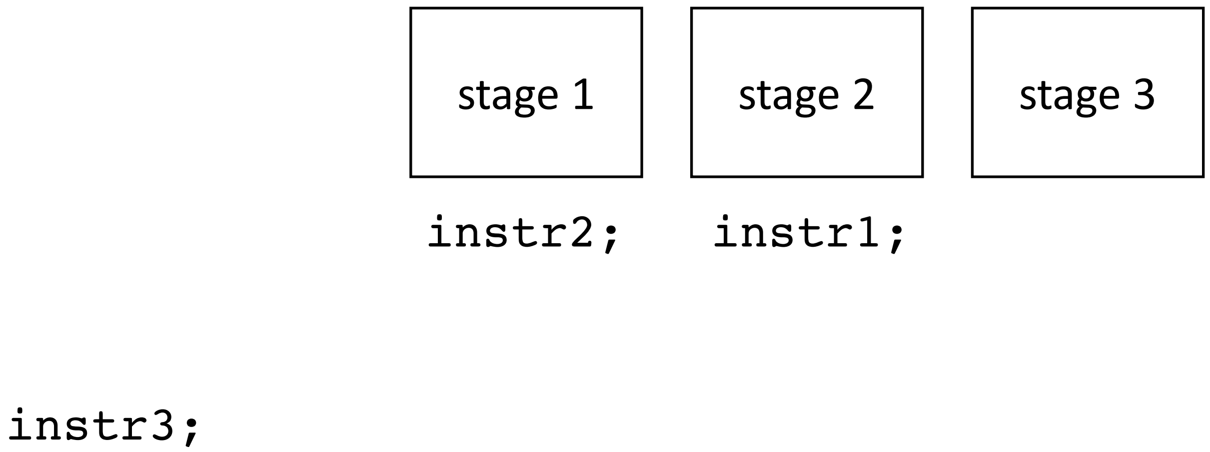
Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



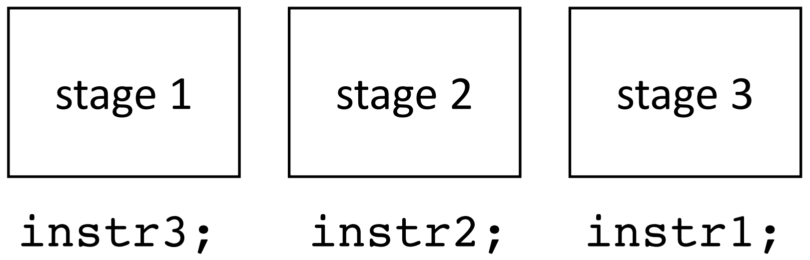
Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



Superscalar

- Executing multiple instructions at once:

- Superscalar architecture:
  - Several sequential operations are issued in parallel
  - hardware detects dependencies

issue-width is maximum number of instructions that can be issued in parallel

```
instr0;
instr1;
instr2;
```

Superscalar

- Executing multiple instructions at once:

- Superscalar architecture:
  - Several sequential operations are issued in parallel
  - hardware detects dependencies

issue-width is maximum number of instructions that can be issued in parallel

```
instr0;
instr1;
instr2;
```

if instr0 and instr1 are independent, they will be issued in parallel

Smaller instructions

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

convert to c++

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

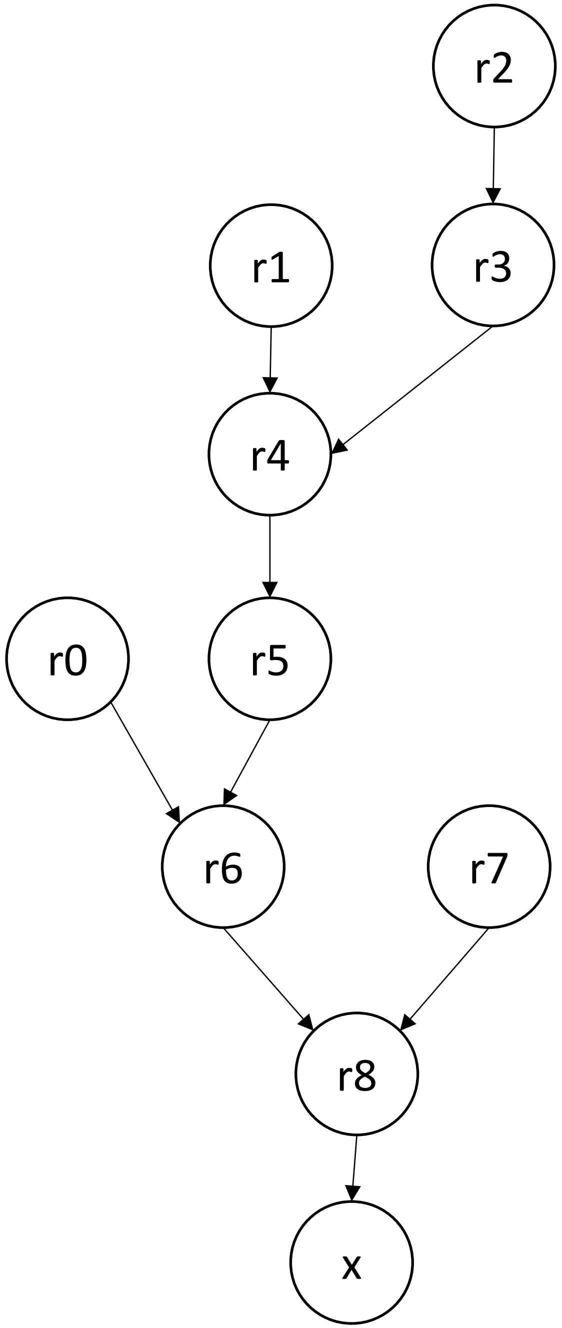
compiler

```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```

```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```

Create a data dependency graph (DDG)

Different instructions can be executed in parallel



Loop unrolling

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i);
    SEQ(i+1);
}
```

Saves one addition and one comparison per loop, but doesn’t help with ILP

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i);
    SEQ(i+1);
}
```

Let green highlights indicate instructions from iteration `i`.

Let blue highlights indicate instructions from iteration `i + 1`.

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i);
    SEQ(i+1);
}
```

Let `SEQ(i,j)` be the jth instruction of `SEQ(i)`.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i,1);
    SEQ(i,2);
    ...
    SEQ(i,N); // end iteration for i
    SEQ(i+1,1);
    SEQ(i+1,2);
    ...
    SEQ(i+1, N); // end iteration for i + 1
}
```

Let `SEQ(i,j)` be the jth instruction of `SEQ(i)`.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
     SEQ(i,1);
     SEQ(i+1,1);
     SEQ(i,2);
     SEQ(i+1,2);
     ...
     SEQ(i,N);
     SEQ(i+1, N);
}
```

They can be interleaved

On to the lecture!

Lecture Schedule

- ILP for reduction loops
- C++ threads

Lecture Schedule

- **ILP for reduction loops**
- C++ threads

Loop Unrolling for Reduction Loops

- Prior approach examined loops with independent iterations and chains of dependent computations

- Now we will look at reduction loops:
  - Entire computation is dependent
  - Typically short bodies (addition, multiplication, max, min)

1	2	3	4	5	6

addition: ?

max: ?

min: ?

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE; i++) {
    a[0] = REDUCE(a[0], a[i]);
}
```

1	2	3	4	5	6

1 + 2 + 3 + 4 + 5 + 6

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE; i++) {
    a[0] = REDUCE(a[0], a[i]);
}
```

What is associativity?

1	2	3	4	5	6

1 + 2 + 3 + 4 + 5 + 6

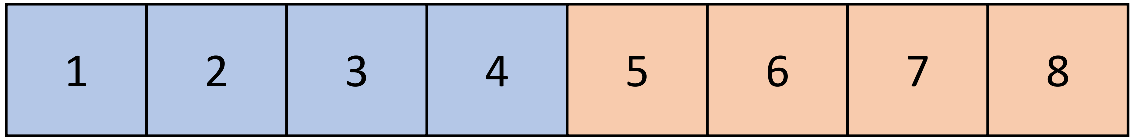
Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:

1	2	3	4	5	6	7	8

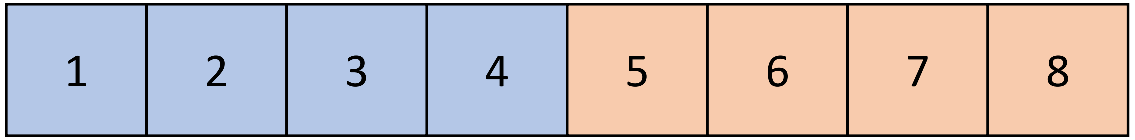
Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Do addition reduction in base memory location

Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:

10	2	3	4	26	6	7	8

Do addition reduction in base memory location

Loop Unrolling for Reduction Loops

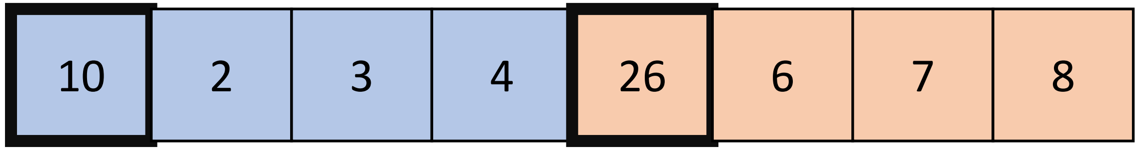
- chunk array in equal sized partitions and do local reductions
- Consider size 2:

10	2	3	4	26	6	7	8

Add together base locations

Loop Unrolling for Reduction Loops

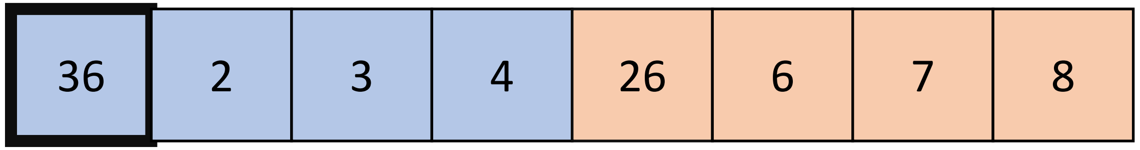
- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Add together base locations

Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Add together base locations

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {
    a[0] = REDUCE(a[0], a[i]);
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);
}

a[0] = REDUCE(a[0], a[SIZE/2])
```

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {
    a[0] = REDUCE(a[0], a[i]);
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);
}

a[0] = REDUCE(a[0], a[SIZE/2])
```

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {
    a[0] = REDUCE(a[0], a[i]);
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);
}

a[0] = REDUCE(a[0], a[SIZE/2])
```

independent instructions can be done in parallel!

Watch out!

- Our abstraction: separate dependent instructions as far as possible

- Pros:
  - Simple

- Cons:
  - Can lead to register spilling, causing expensive loads

consider `instr1` and `instr2` have a data dependence, and `instrX's` are independent

```
instr1;
instrX0;
instrX1;
...
instr2;
```

independent instructions. If they overwrite the register storing `instr1's` result, then it will have to be stored to memory and retrieved before `instr2`

Watch out!

- Our abstraction: separate dependent instructions as far as possible

- Pros:
  - Simple

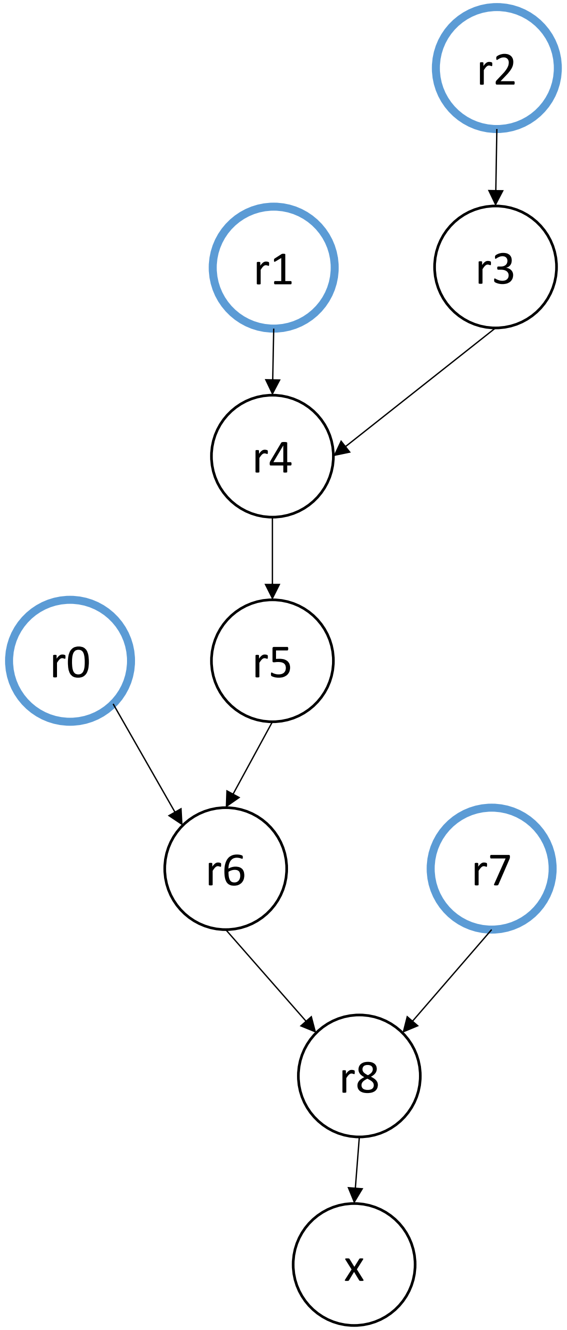
- Cons:
  - Can lead to register spilling, causing expensive loads

Solutions include using a **resource model** to guide the topological ordering. Highly architecture dependent. Algorithms become more expensive

Consider timing the compile time in your homework assignment

Priority Topological Ordering of DDGs

```
r7 = 2 * a;
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r8 = r6 / r7;
x  = r8;
```



Lecture Schedule

- ILP for reduction loops
- **C++ threads**

C++ Threads

- Introduction
  - Learn as needed throughout class

- Multi-threading officially introduced in C++11
  - only widely available after ~2014
  - official specification
  - cross-platform

- Before C++ threads
  - pthreads

C++ Threads

- Introduction
  - Learn as needed throughout class

- Multi-threading officially introduced in C++11
  - only widely available after ~2014
  - official specification
  - cross-platform

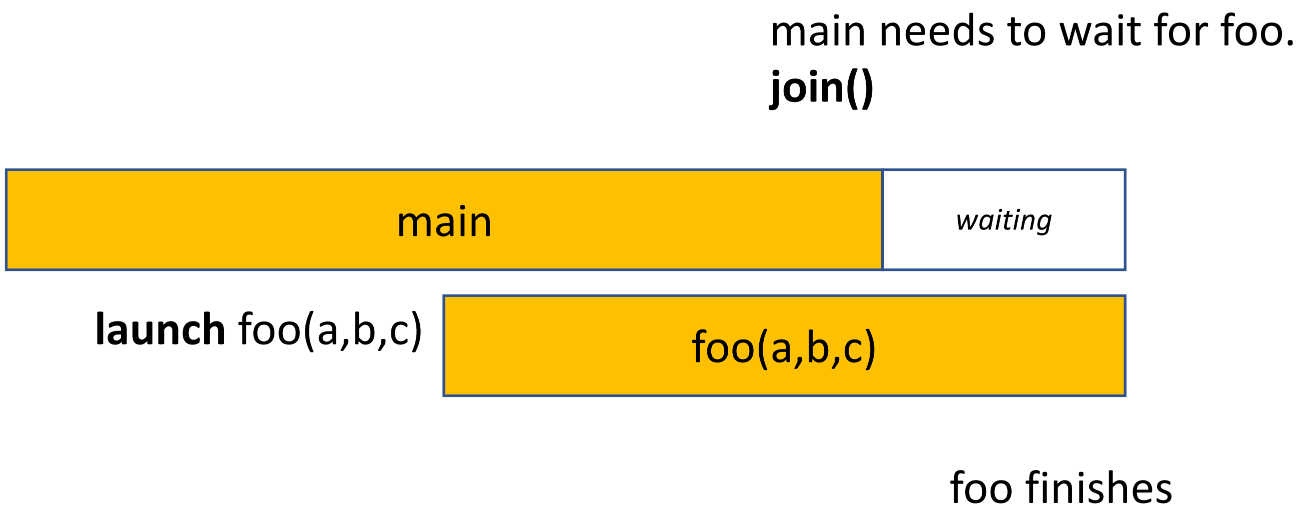
- Before C++ threads
  - pthreads
  - volatile

C++ Threads

- Main idea:
  - run functions concurrently

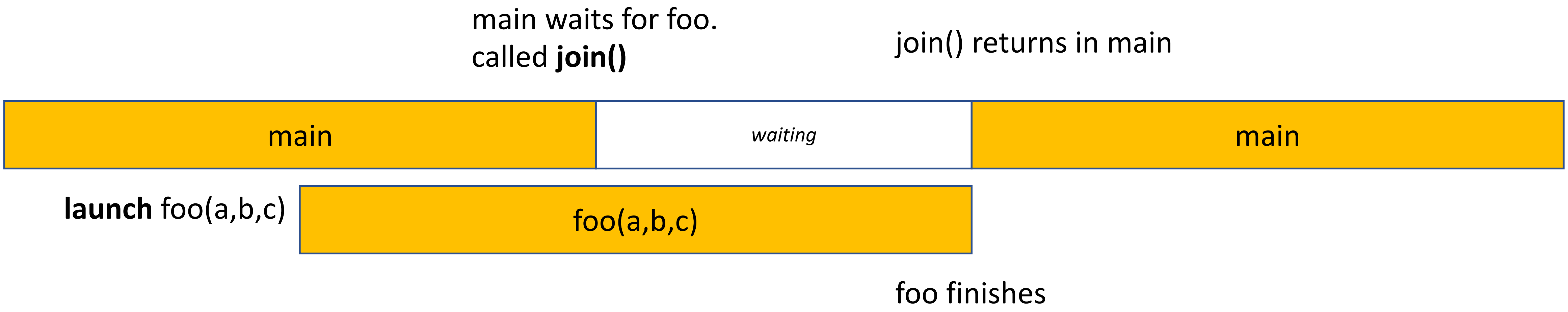
C++ Threads

- Main idea:
  - run functions concurrently

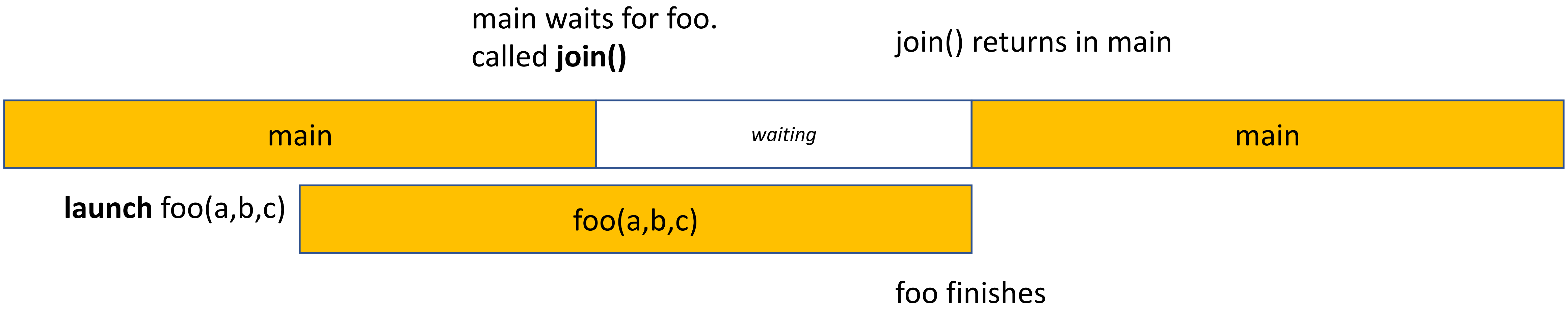


C++ Threads

- Main idea:
  - run functions concurrently

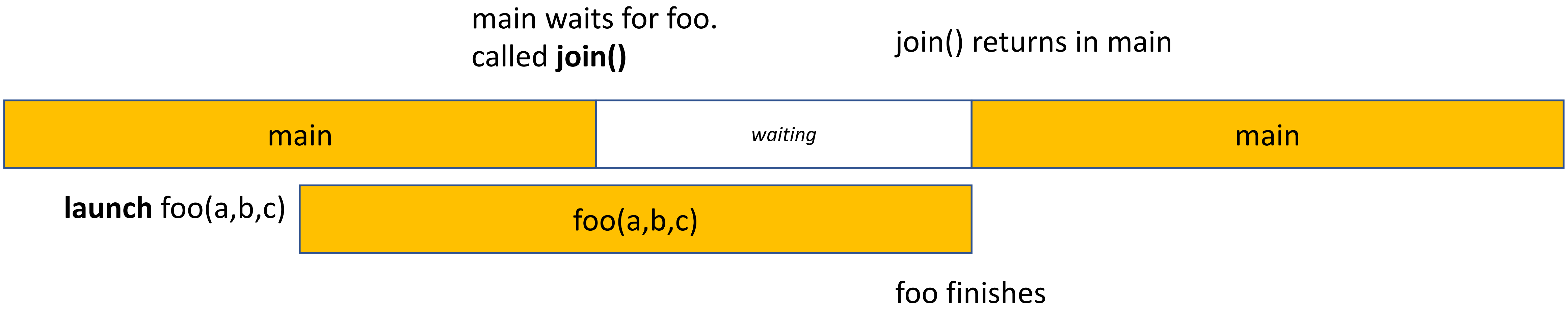


```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```



```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
   // some foo code
}

int main() {
   // some main code
   thread thread_handle (foo,1,2,3);
   // code here runs concurrently with foo
   thread_handle.join();
   return 0;
}
```

header and namespace

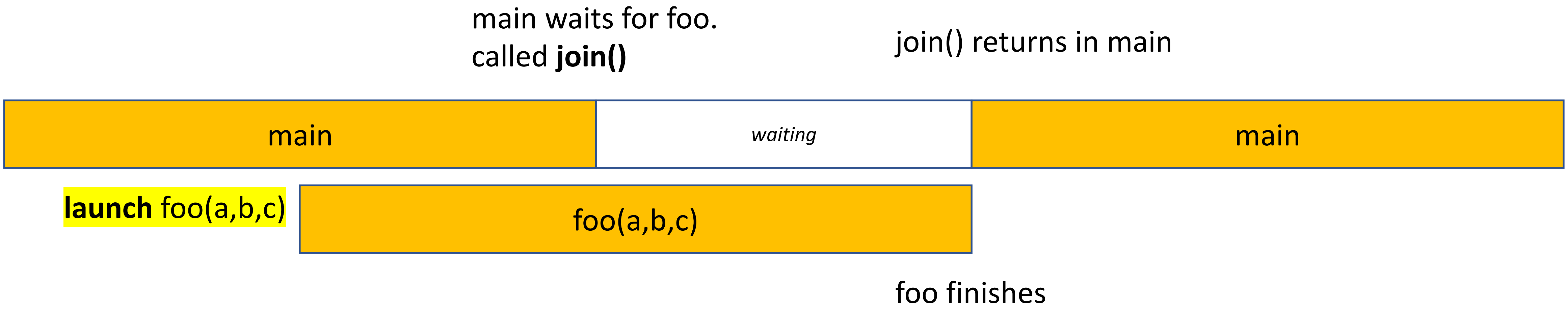


```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

Launches a concurrent thread that executes foo

Stores a handle in thread_handle (don’t lose the handle!)

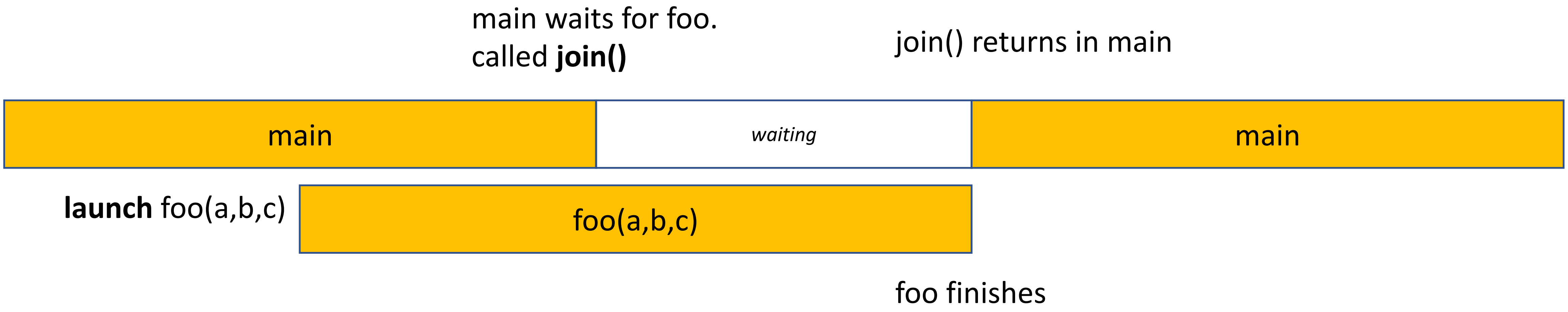
constructor takes in the function, and all arguments



```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
   // some foo code
}

int main() {
   // some main code
   thread thread_handle (foo,1,2,3);
   // code here runs concurrently with foo
   thread_handle.join();
   return 0;
}
```

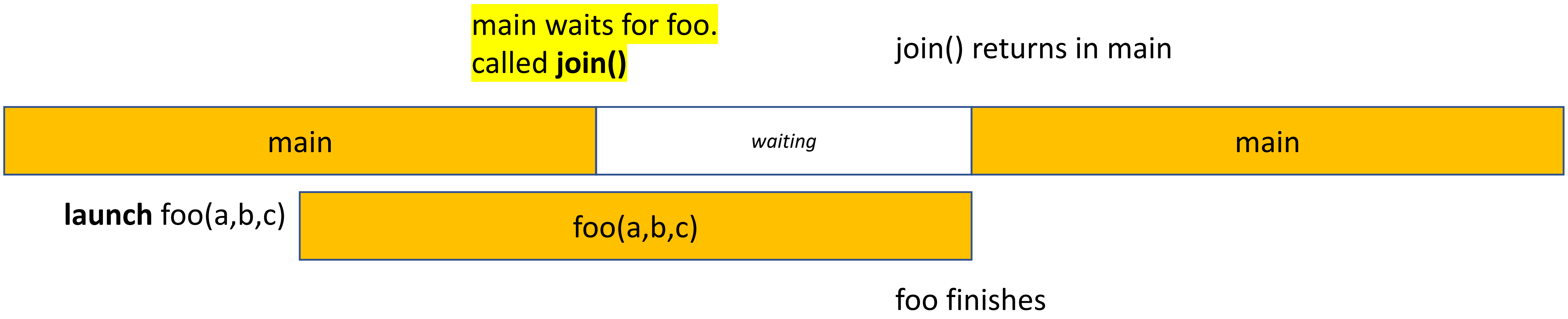
Requires C++14

clang++ -std=c++14 main.cpp



```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

calling join() on the thread handle will cause main to wait for the thread launched with thread_handle to finish.



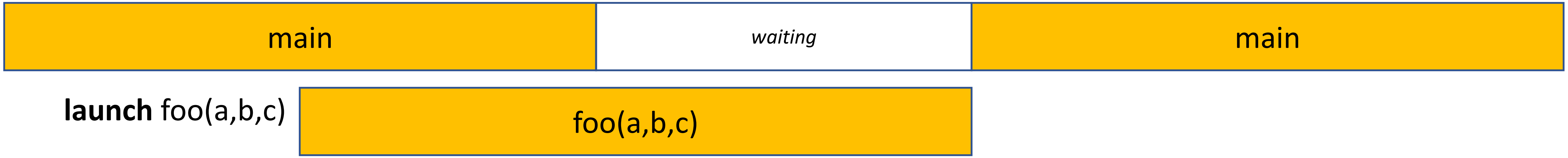
```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

After foo finishes,
main starts executing again

main waits for foo.
called **join()**

join() returns in main

launch foo(a,b,c)



foo finishes

```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

What happens if you don't join your threads?

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

What happens if you don't join your threads?

```
libc++abi.dylib: terminating
Abort trap: 6
```

JOIN YOUR THREADS!!!

```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

return value?

Doesn’t have to be void, but it is ignored

how to get values back from threads?

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int &c) {
  // return a + b;
  c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, ref(ret));
  // code here runs concurrently with foo
  thread_handle.join();
  cout << ret << endl;
  return 0;
}
```

Options

pass by reference (C++)

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
   // return a + b;
   *c = a + b;
}

int main() {
   // some main code
   int ret = 0;
   thread thread_handle (foo,1,2, &ret);
   // code here runs concurrently with foo
   thread_handle.join();
   cout << ret << endl;
   return 0;
}
```

Options

pass by address (C++ or C)

```cpp
#include <thread>
#include <iostream>
using namespace std;

int c;
void foo(int a, int b) {
  // return a + b;
  c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2);
  // code here runs concurrently with foo
  thread_handle.join();
  cout << c << endl;
  return 0;
}
```

Options

global variable
(don't do this!)

```cpp
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if....

```cpp
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if....

Undefined behavior!
Cannot access the same values concurrently without protection!

Next module we will talk protection (locks)

SPMD programming model

- Same program, multiple data

- Main idea: many threads execute the same function, but they operate on different data.

- How do they get different data?
  - each thread can access their own thread id, a contiguous integer starting at 0 up to the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size) {
    for (int i = 0; i < a_size; i++) {
        a[i]++;
    }
}
```

lets do this in parallel!
each thread increments different elements in the array

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = 0; i < a_size; i++) {
        a[i]++;
    }
}
```

The function gets a thread id and the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = 0; i < a_size; i++) {
        a[i]++;
    }
}
```

A few options on how to split up the work

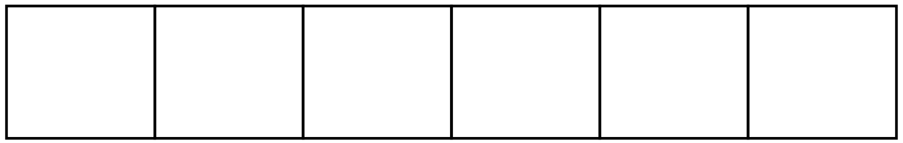
lets do round robin

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```



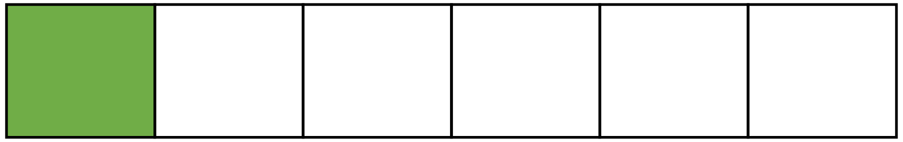
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

iteration 1 computes index 0



array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

iteration 2 computes index 2

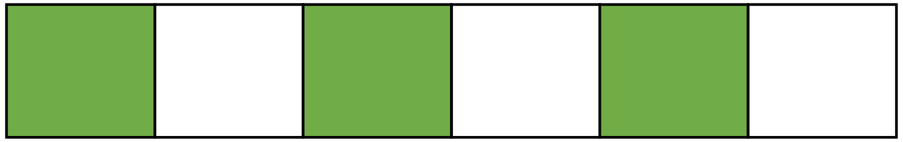
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

iteration 3 computes index 4



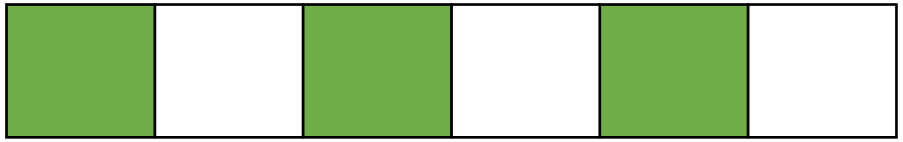
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1



array a

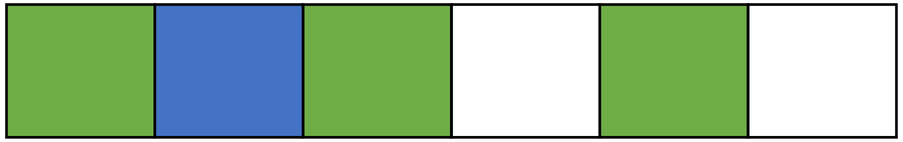
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

switch to thread 1

iteration 1 computes index 1



array a

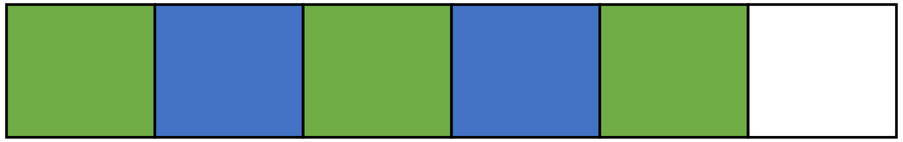
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

switch to thread 1

iteration 2 computes index 3



array a

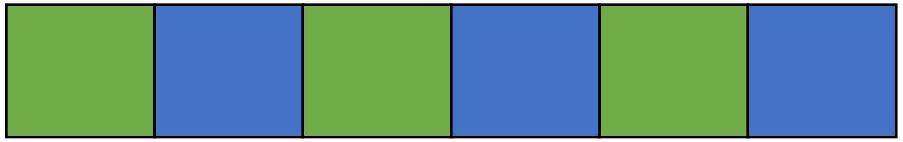
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 3 computes index 5



array a

Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads);


#define THREADS 8
#define A_SIZE 1024
int main() {
  int *a = new int[A_SIZE];
  // initialize a
  thread thread_ar[THREADS];
  for (int i = 0; i < THREADS; i++) {
    thread_ar[i] = thread(increment_array, a, A_SIZE, i, THREADS);
  }
  for (int i = 0; i < THREADS; i++) {
    thread_ar[i].join();
  }
  delete[] a;
  return 0;
}
```

Thank you!

- Remember to do the quiz today!

- Get started on homework
  - Should be able to do all parts now

- Start on module 2 on Friday

Extra if time

Concurrency vs. Parallelism

- Abstract tasks:
  - In the abstract: a sequence of computation
  - *Given an input, produces an output*

Concurrency vs. Parallelism

- Abstract tasks:
  - In the abstract: a sequence of computation
  - *Given an input, produces an output*

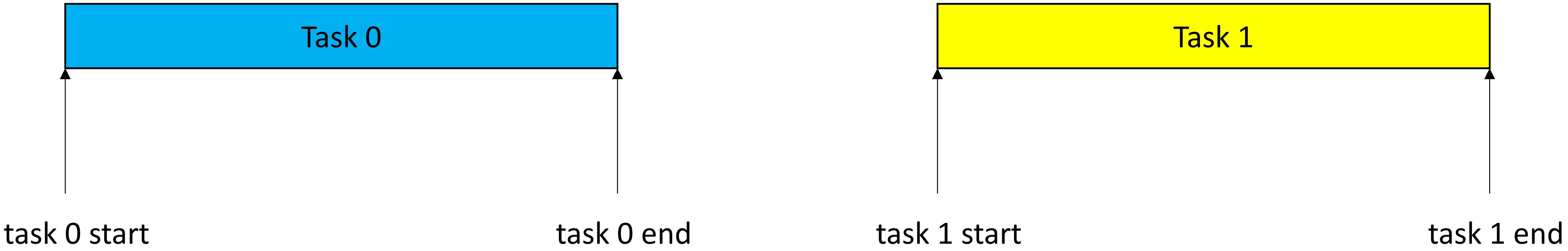
- Concrete tasks:
  - Application (e.g. Spotify and Chrome)
  - Function
  - Loop iterations
  - Individual instructions
  - Circuit level?

coarse

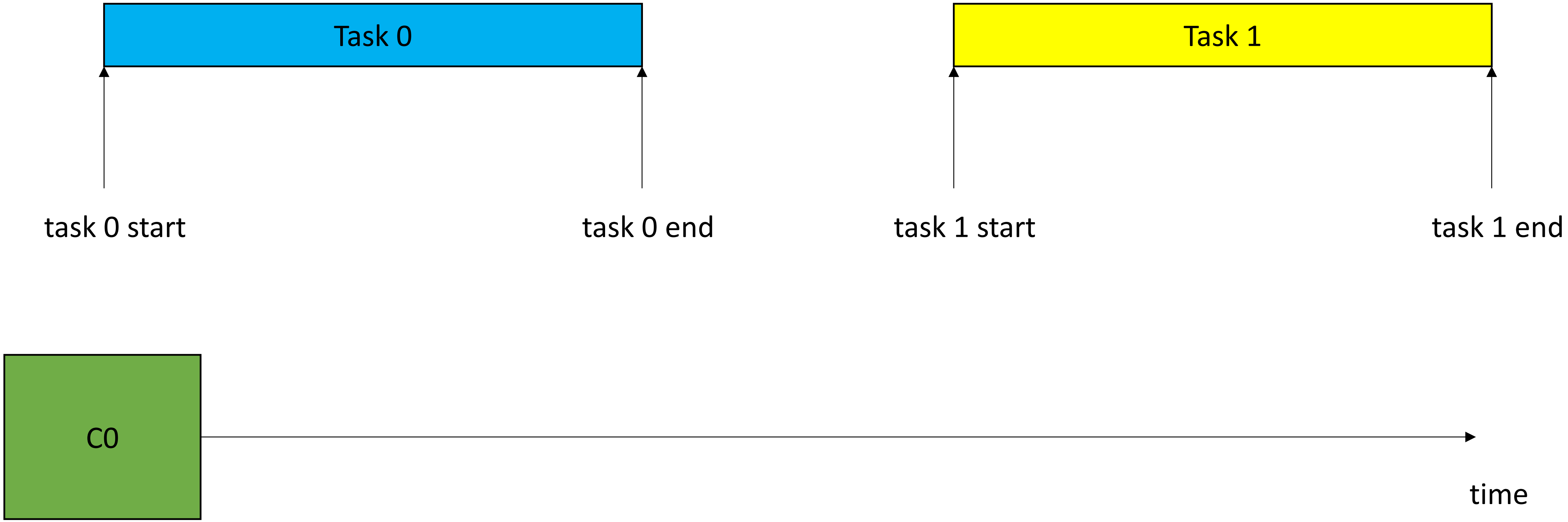
granularity

fine

Concurrency vs. Parallelism

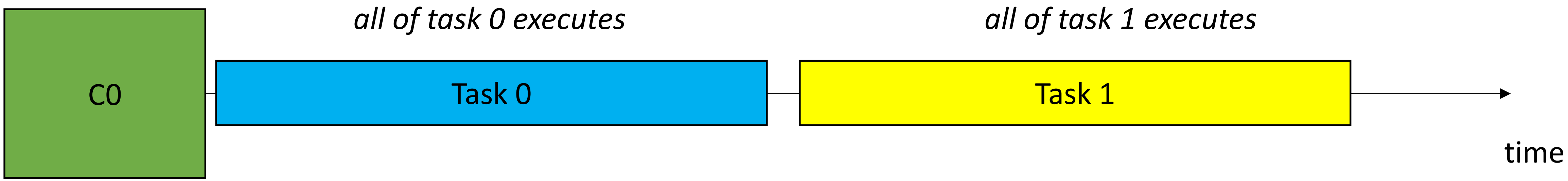


Concurrency vs. Parallelism



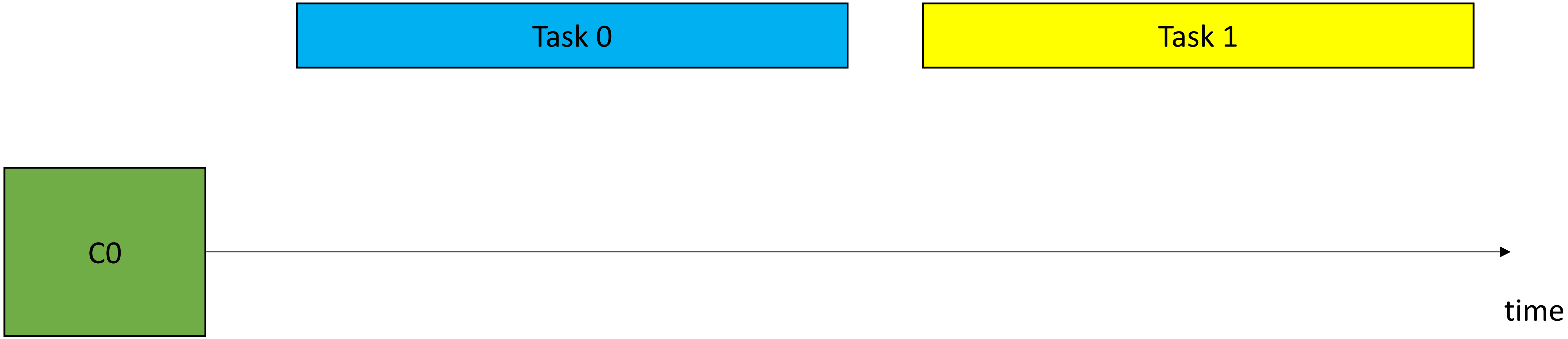
Concurrency vs. Parallelism

Sequential execution
Not concurrent or parallel



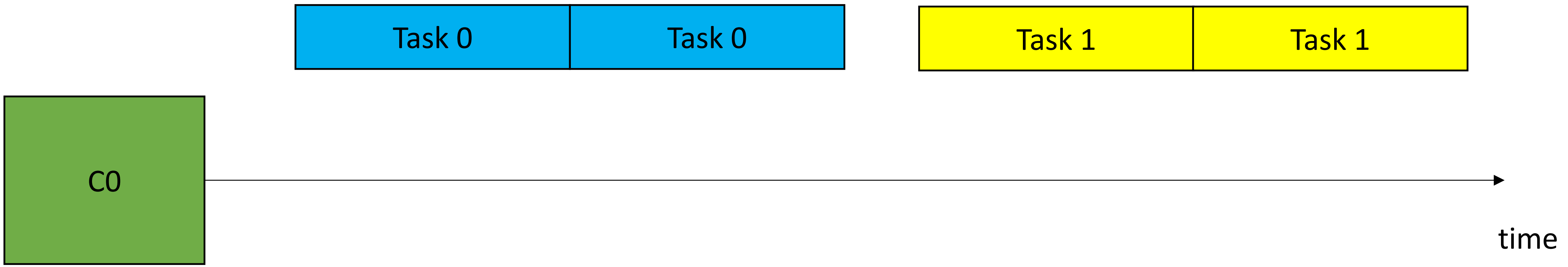
Concurrency vs. Parallelism

The OS can preempt a thread
(remove it from the hardware resource)



Concurrency vs. Parallelism

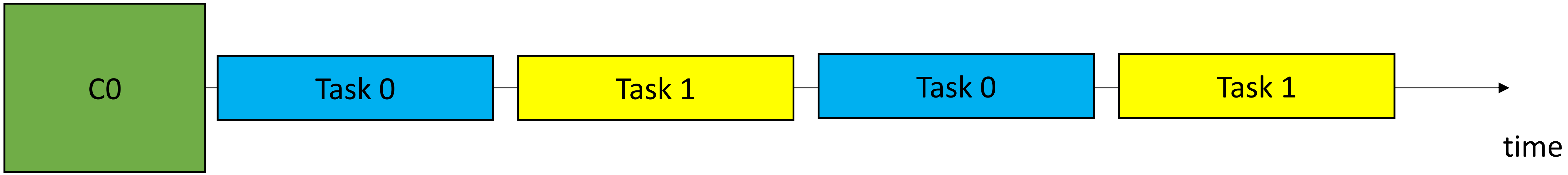
The OS can preempt a thread
(remove it from the hardware resource)



Concurrency vs. Parallelism

The OS can preempt a thread
(remove it from the hardware resource)

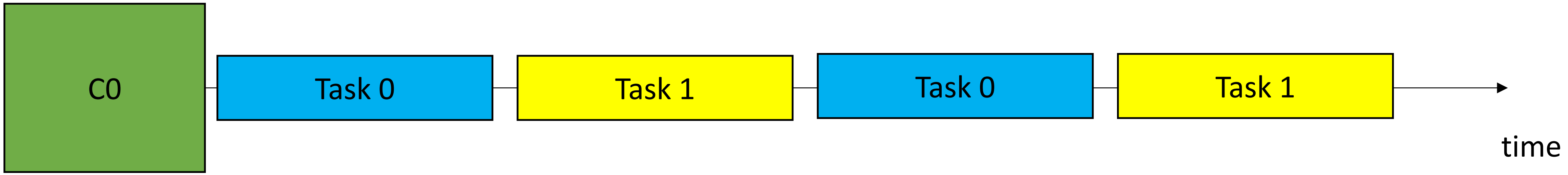
tasks are interleaved on the same processor



Concurrency vs. Parallelism

- Definition:
  - 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

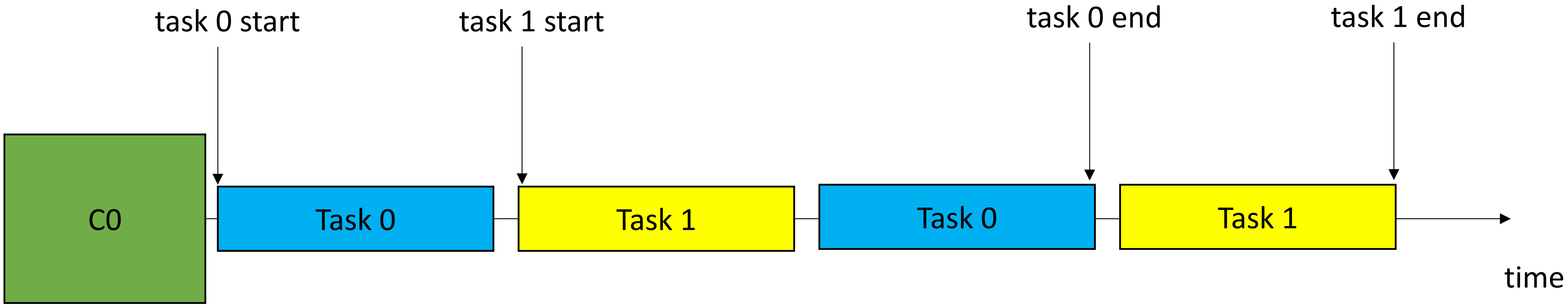
The OS can preempt a thread
(remove it from the hardware resource)



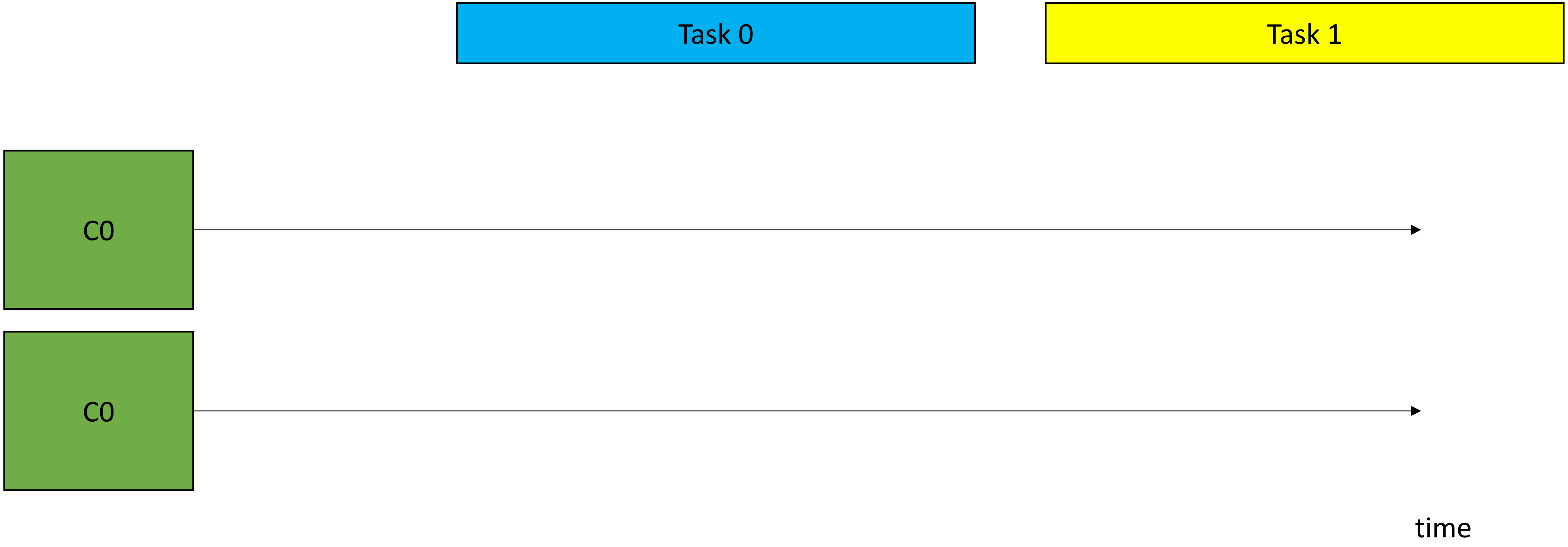
Concurrency vs. Parallelism

- Definition:
    - 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

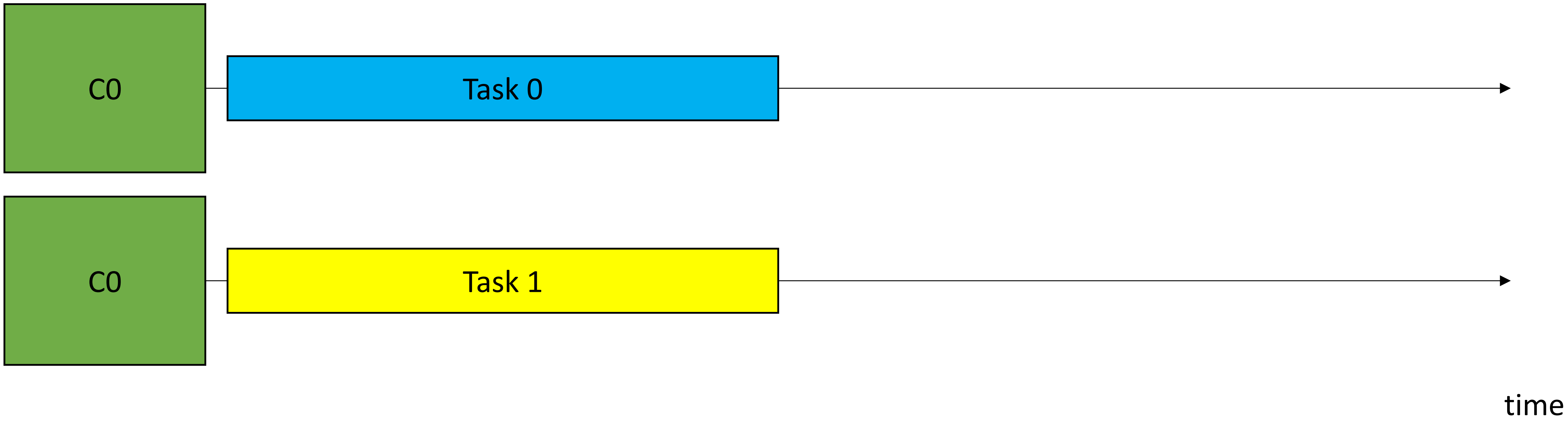
The OS can preempt a thread
(remove it from the hardware resource)



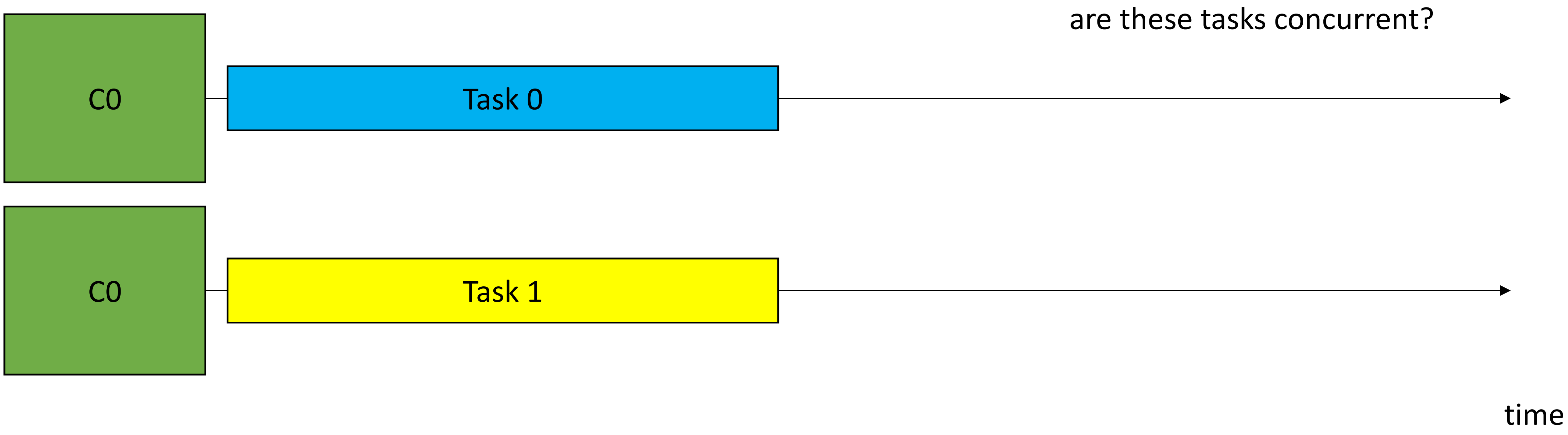
Concurrency vs. Parallelism



Concurrency vs. Parallelism

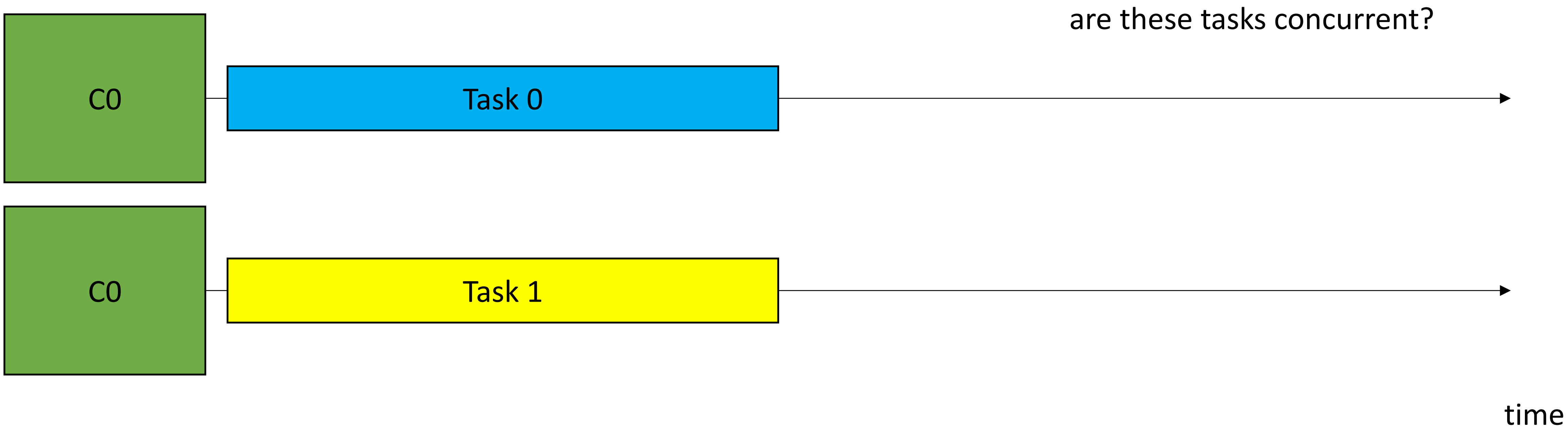


Concurrency vs. Parallelism

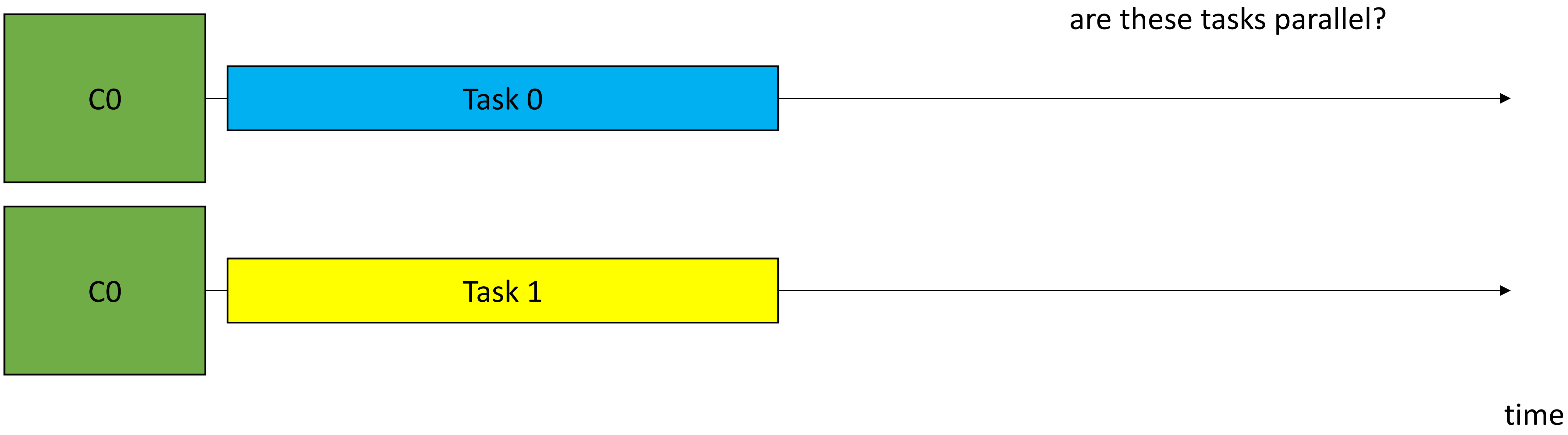


Concurrency vs. Parallelism

- 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

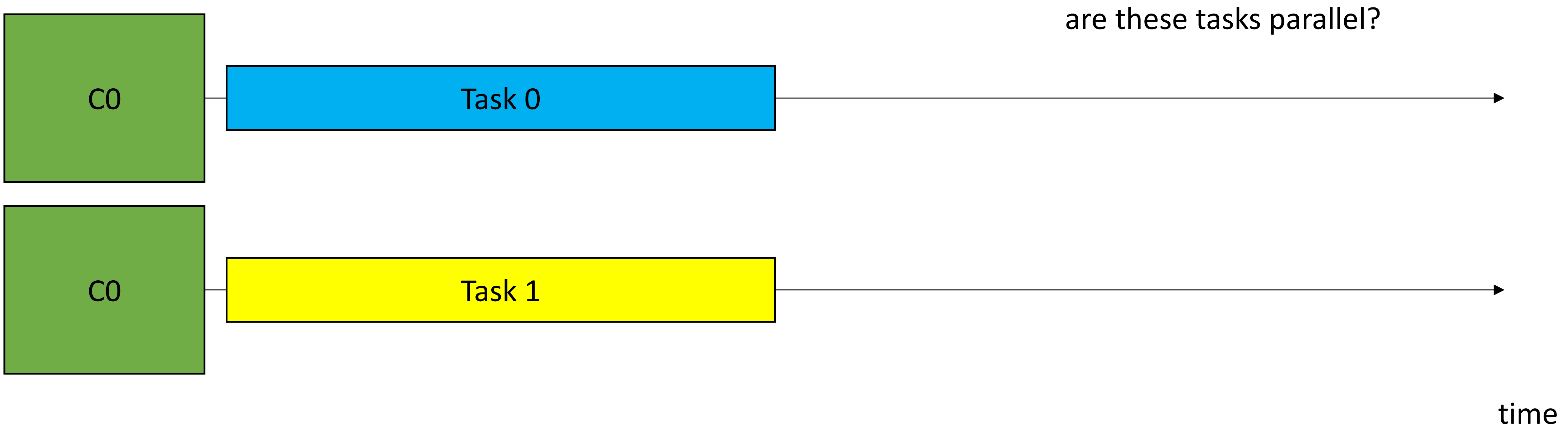


Concurrency vs. Parallelism



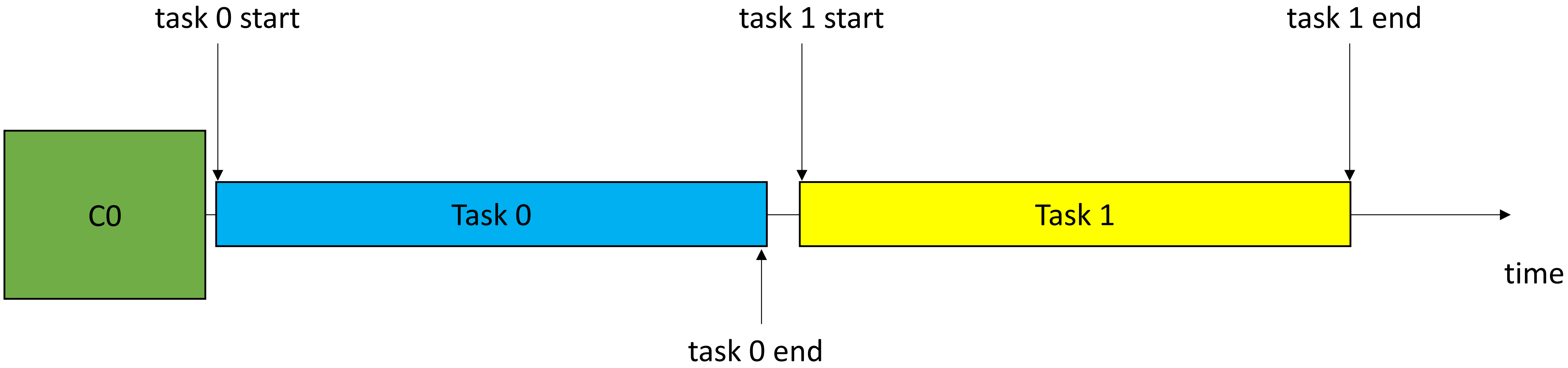
Concurrency vs. Parallelism

- Definition:
  - An execution is **parallel** if there is a point in the execution where computation is happening simultaneously



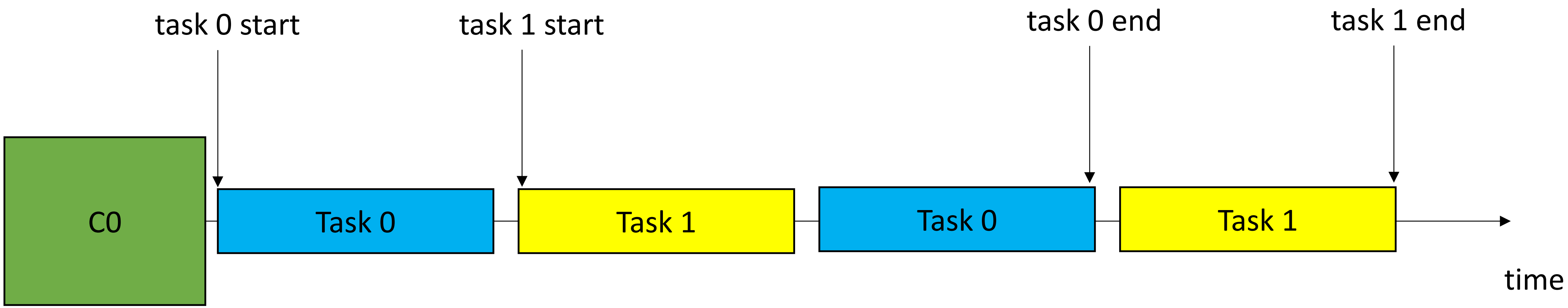
Concurrency vs. Parallelism

- Examples:
  - Neither concurrent or parallel (sequential)



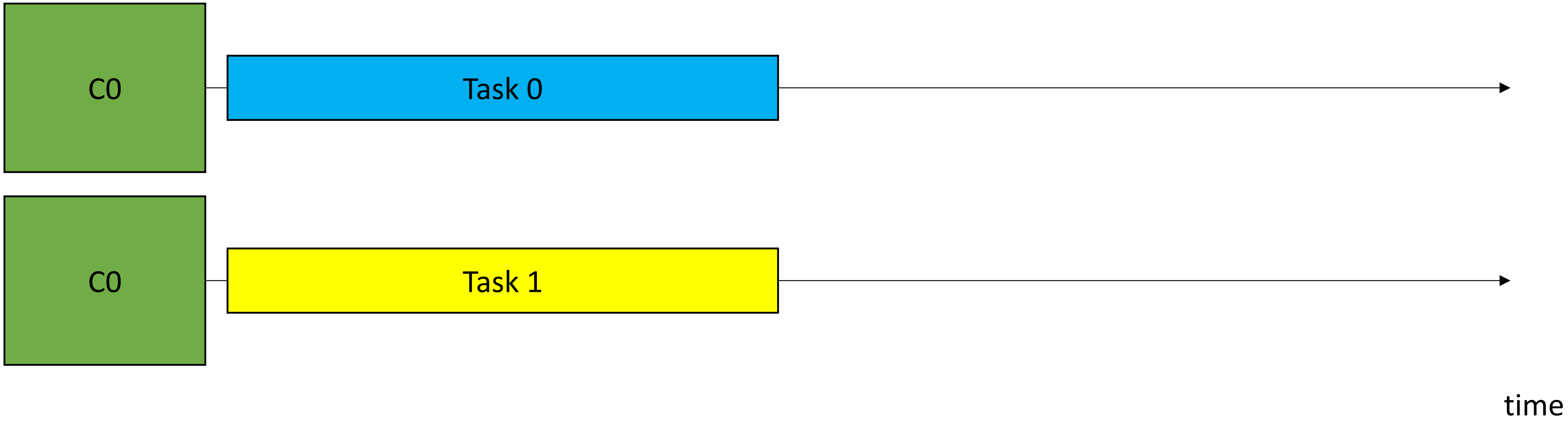
Concurrency vs. Parallelism

- Examples:
  - Concurrent but not parallel



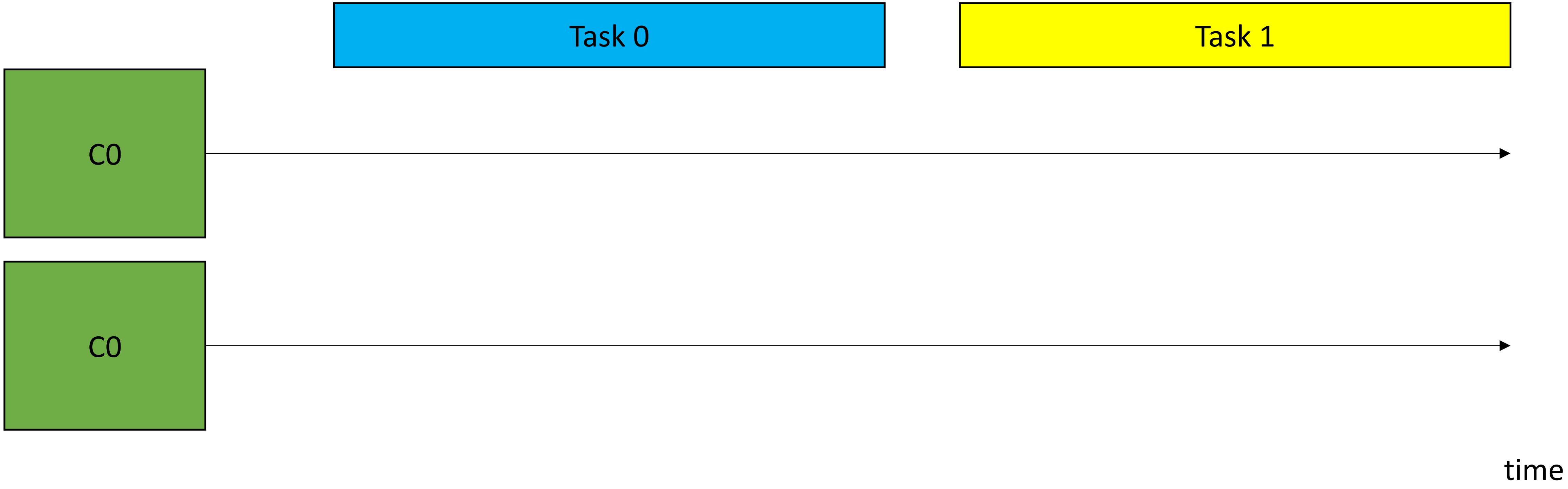
Concurrency vs. Parallelism

- Examples:
  - Parallel and Concurrent



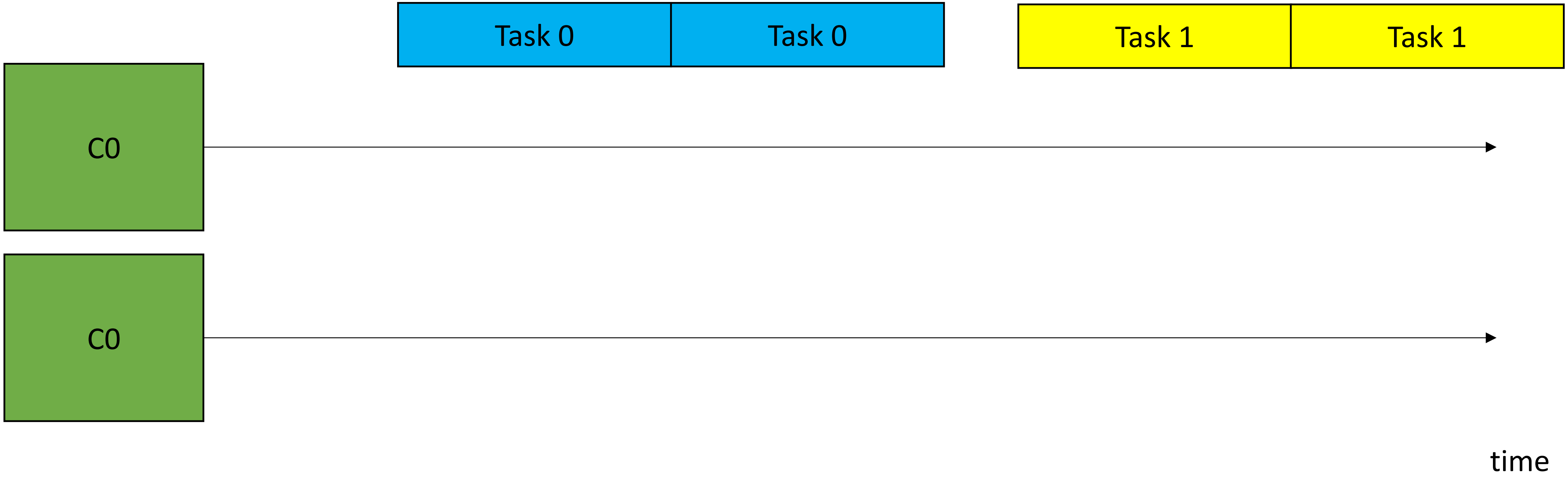
Concurrency vs. Parallelism

- Examples:
  - Parallel but not concurrent?



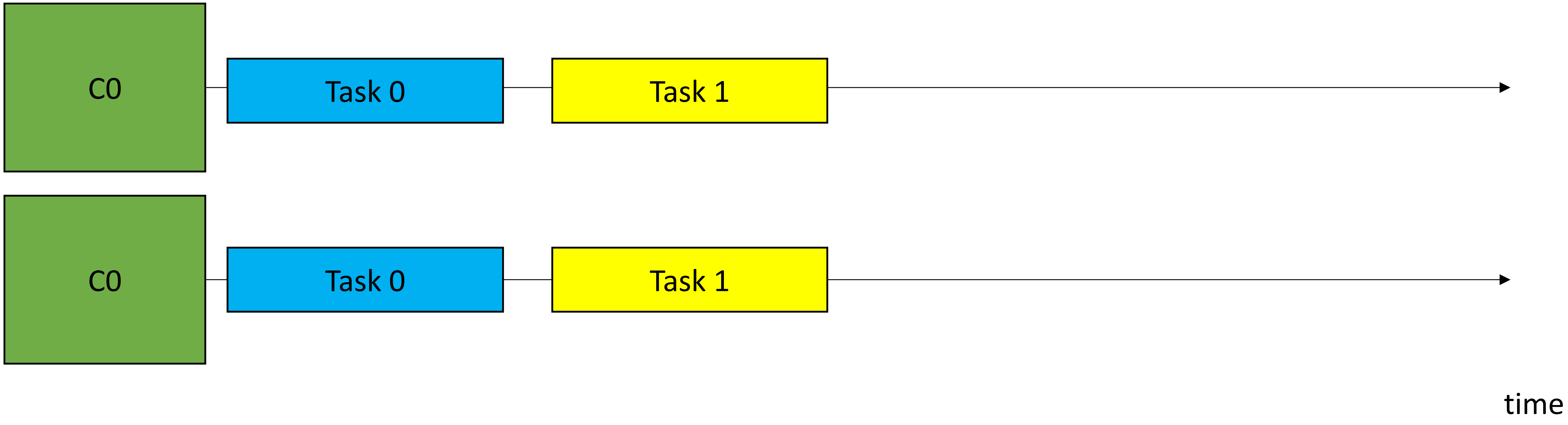
Concurrency vs. Parallelism

- Examples:
  - Parallel but not concurrent?



Concurrency vs. Parallelism

- Examples:
  - Parallel execution but task 0 and task 1 are not concurrent?



Concurrency vs. Parallelism

- In practice:
  - Terms are often used interchangeably.
  - *Parallel programming* is often used by high performance engineers when discussing using parallelism to accelerate things
  - *Concurrent programming* is used more by interactive applications, e.g. event driven interfaces.