Concurrency

CS 442: Mobile App Development

Michael Lee <lee@iit.edu>

concurrency | kənˈkərənsē |
noun

the fact of two or more events or circumstances happening or existing at the same time

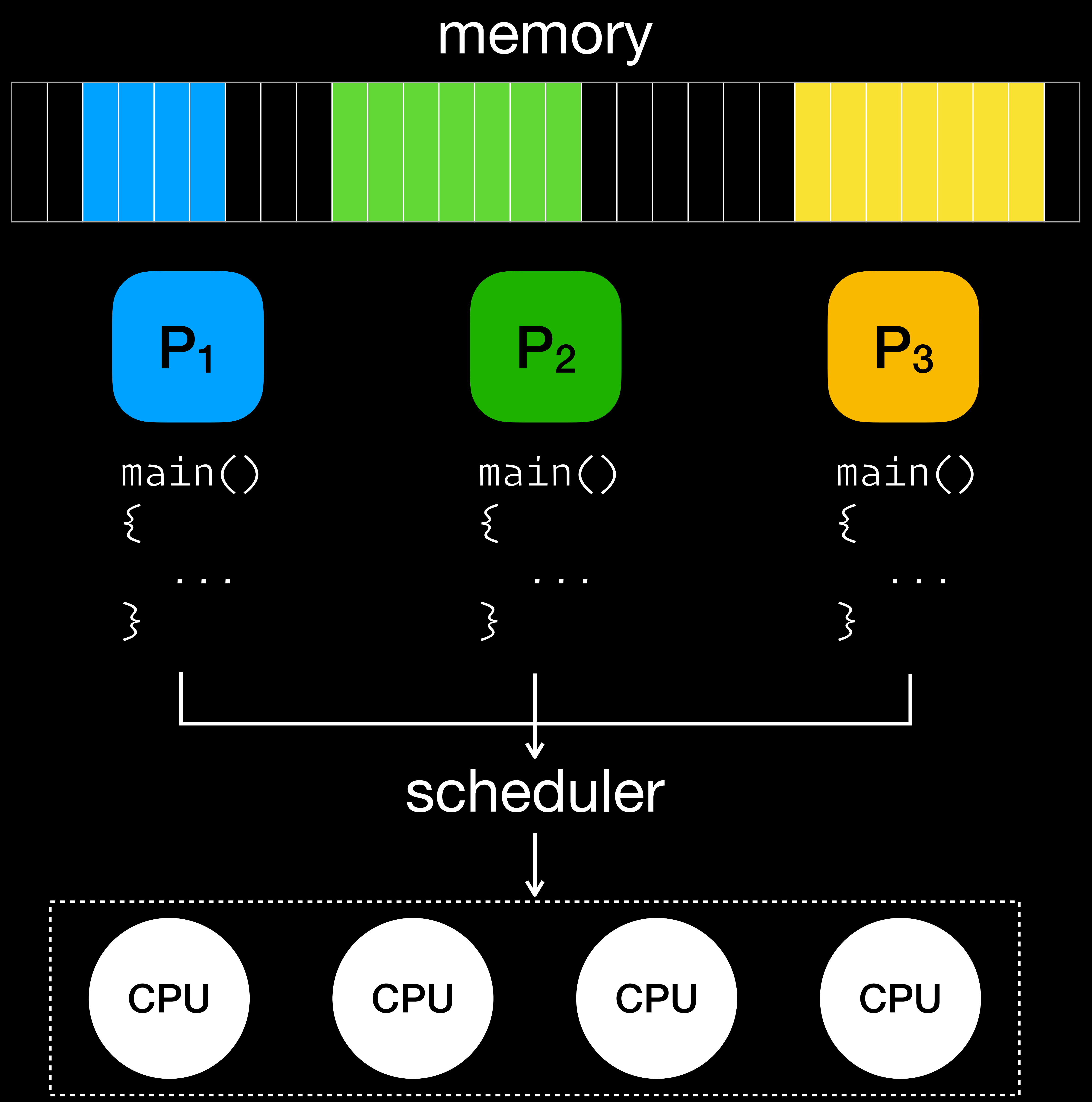
New Oxford American Dictionary

Concurrency in computing

- Multi-processing
- Multi-threading
- Parallelism
- Asynchronous programming

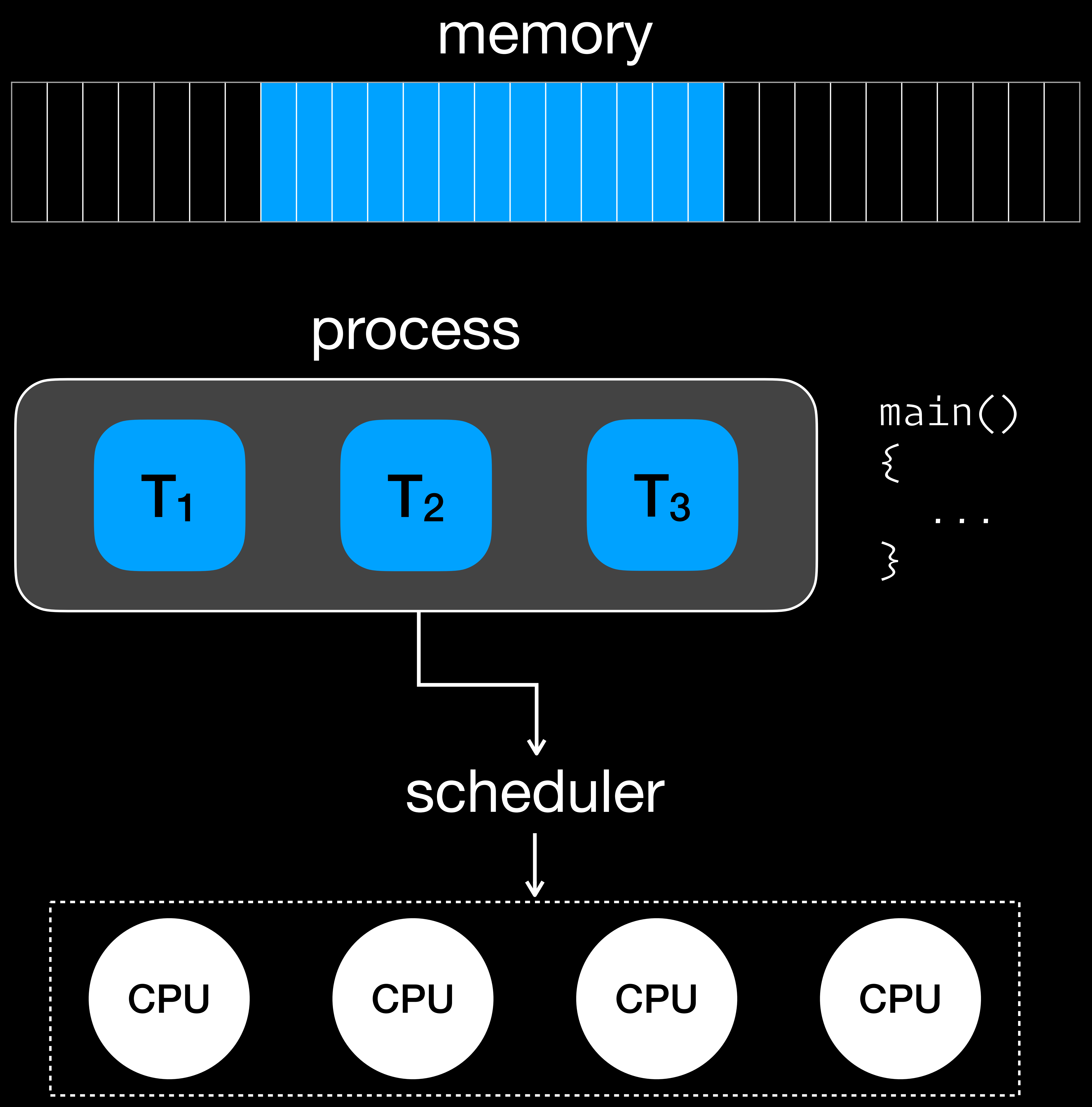
Multi-processing

- Based on the operating system unit of execution: the **process**
- No shared memory
  - Via *virtual address spaces*
- Independent control flow
  - On one or more CPU cores
  - May require *context switching*



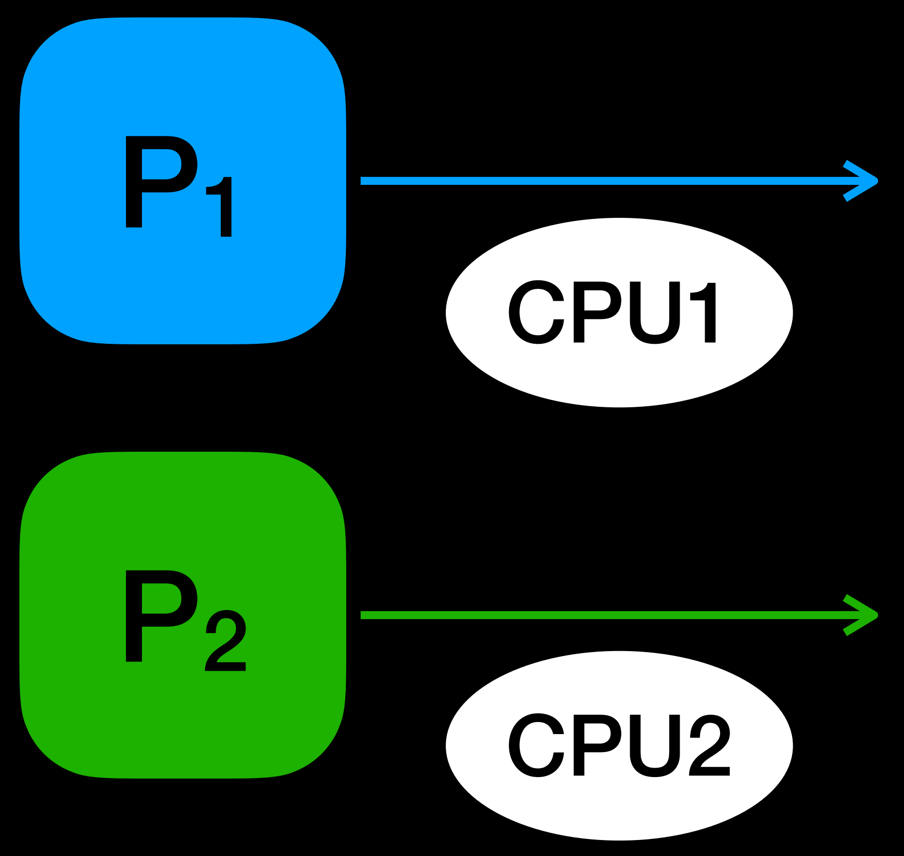
Multi-threading

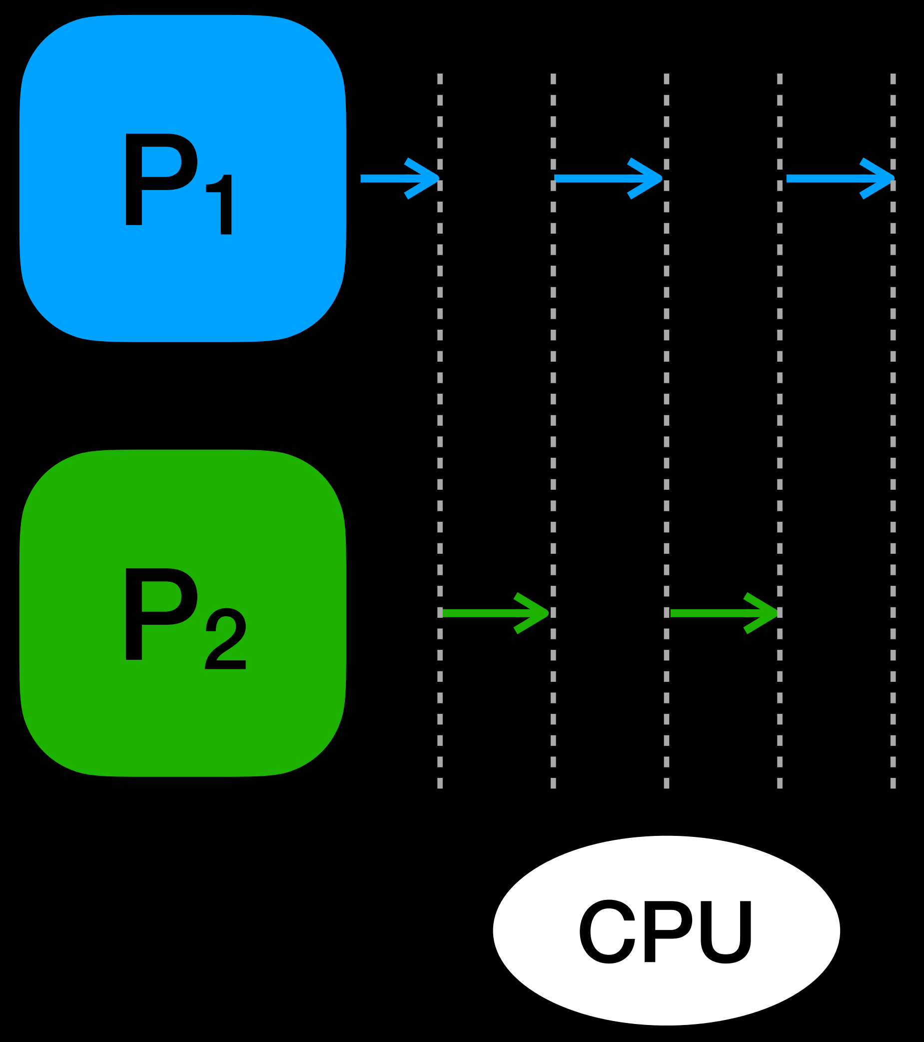
- Separate flows of control (threads) *within the same process*
- Shared program
- Shared global/heap memory
- Typically, separate stacks
- Threads may execute on one or more CPU cores



Parallelism

- Parallelism = **simultaneous execution** of two or more processes/threads
    - Requires multiple CPU/GPU cores
- Concurrency **does not imply** parallelism!
    - Concurrency can be achieved by **time-multiplexing** (aka time-slicing)
- Parallelism is *one form* of concurrency





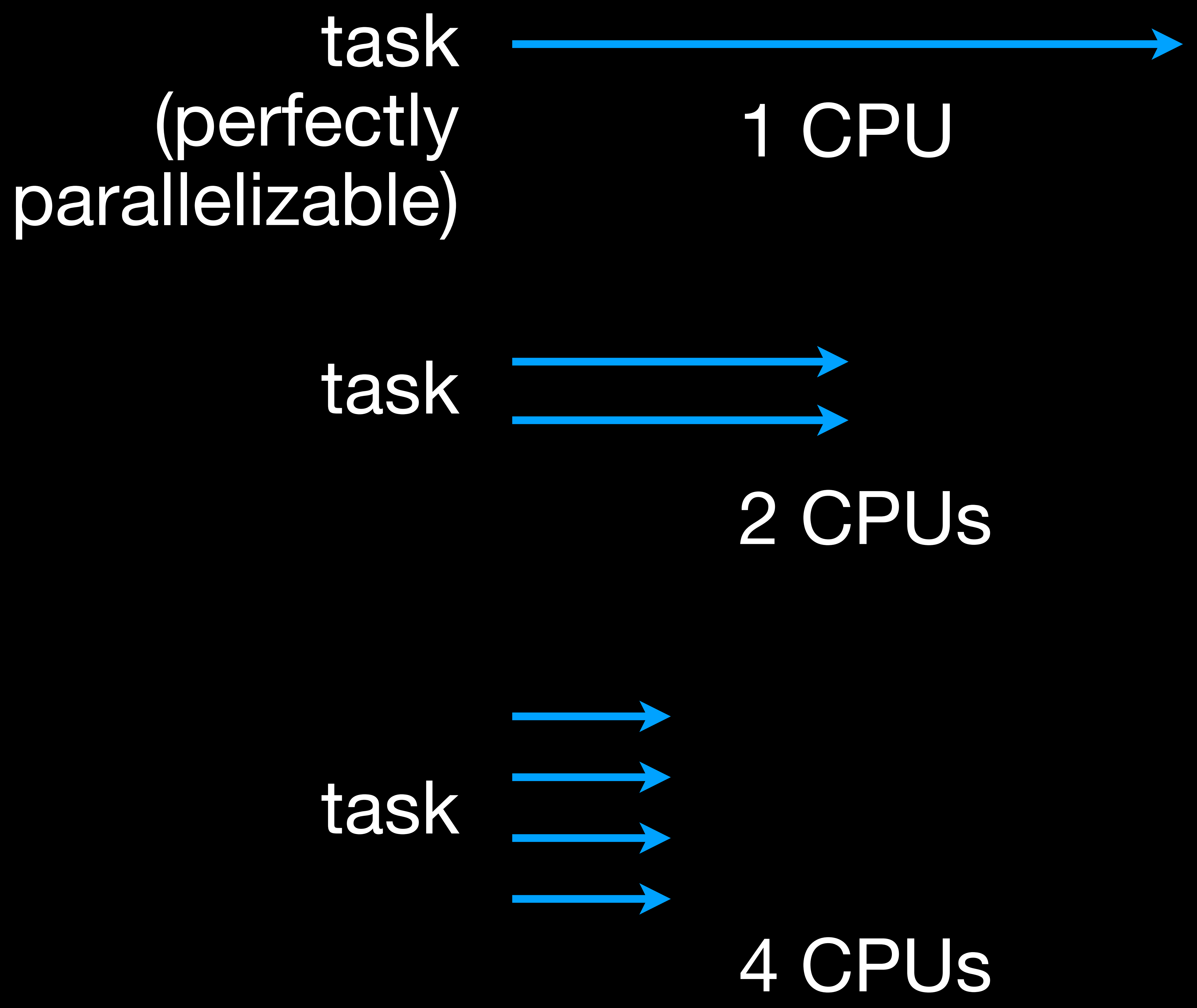
Benefits/Limits of Parallelism

- Parallelism may allow some programs to complete faster
  - By running *parallelizable* portions simultaneously
  - This is a big draw!
- But not all programs are easily parallelized
  - E.g., there may be *serial dependencies*
- Two formulae for estimating speed-up via parallelization:
  - Amdahl's law
  - Gustafson's law

Amdahl's law

$$S_A(n) = \frac{1}{\frac{p}{n} + (1 - p)}$$

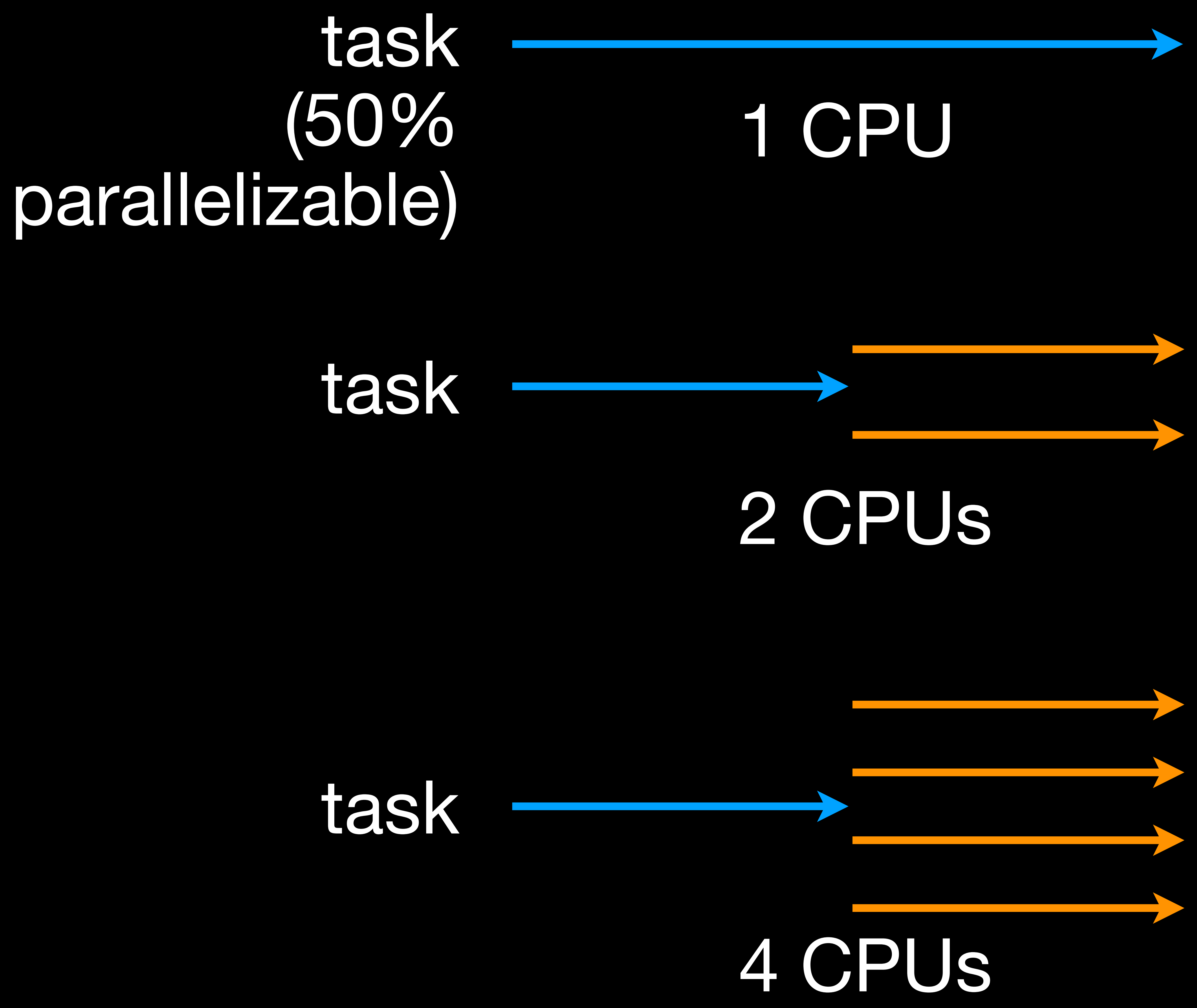
- n is the # of CPUs and p is the parallelizable fraction of the program
- Assumption: **fixed problem size**
  - Completed in less time



Gustafson's law

$$S_G(n) = 1 - p + np$$

- Assumption: problem size can be scaled up to take advantage of computing power
    - Same completion time, but more work done (e.g., at higher resolution)



Is Concurrency useful *without* Parallelism?

- Yes! How?
  - Simulating multitasking
    - e.g., many tasks on OS with 1 CPU
  - Improving hardware utilization
    - e.g., let another task use CPU while one performs I/O
  - Software design tool
    - e.g., separate logical flows of control vs. a single monolithic one

addressed by asynchronous programming!

Asynchronous programming

- Paradigm that allows tasks to execute independently of the original/main control flow
- Different supporting mechanisms:
  - Callback functions
  - Promises/Futures
  - await/async semantics

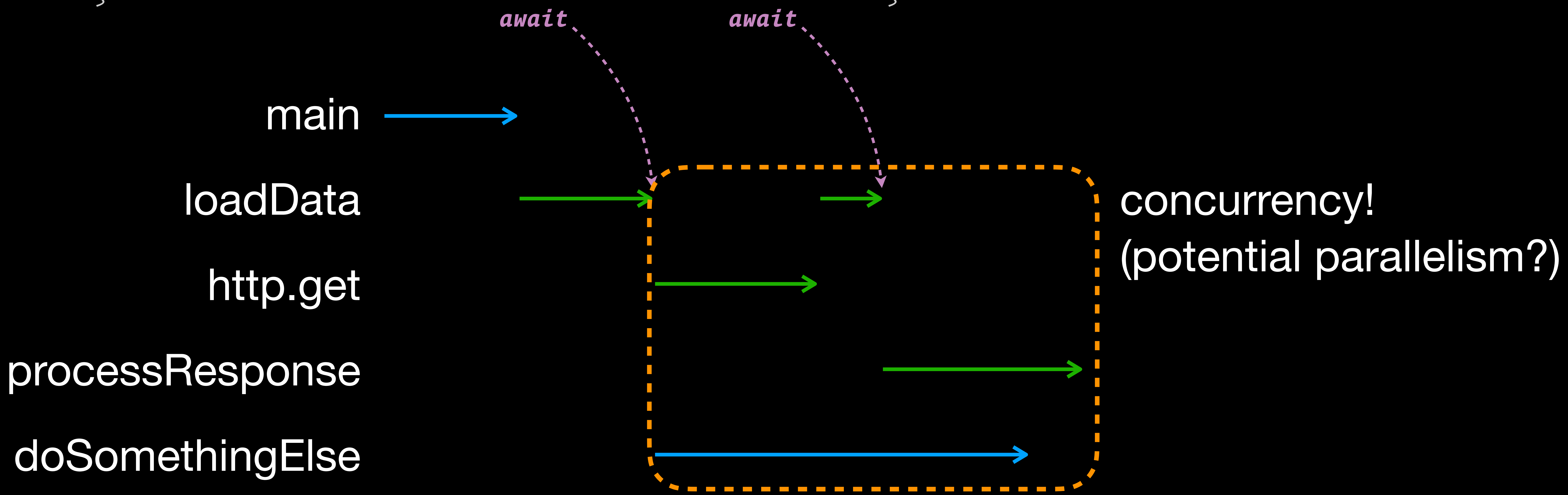
```
Data loadData(Uri url) {
  Future<Response> response = http.get(url);
  response.then((res) {
    Future<Data> data = processResponse(res.body);
    data.then((value) {
      return value;
    });
  });
}
```

```
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

Where is the concurrency?

```
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

```
void main() {
  Future<Data> data = loadData('https://...');
  doSomethingElse();
  data.then((value) => print('Loaded: $value'));
}
```



(Potential) Problems with Concurrency

- When multi-threading, shared memory can lead to **race conditions**
- Simple example: concurrent increment of shared variable
  - Final counter value?
    - 1 or 2 (unpredictable!)

shared var:

```
int counter = 0;
```

thread 1:

```
counter = counter + 1;
```

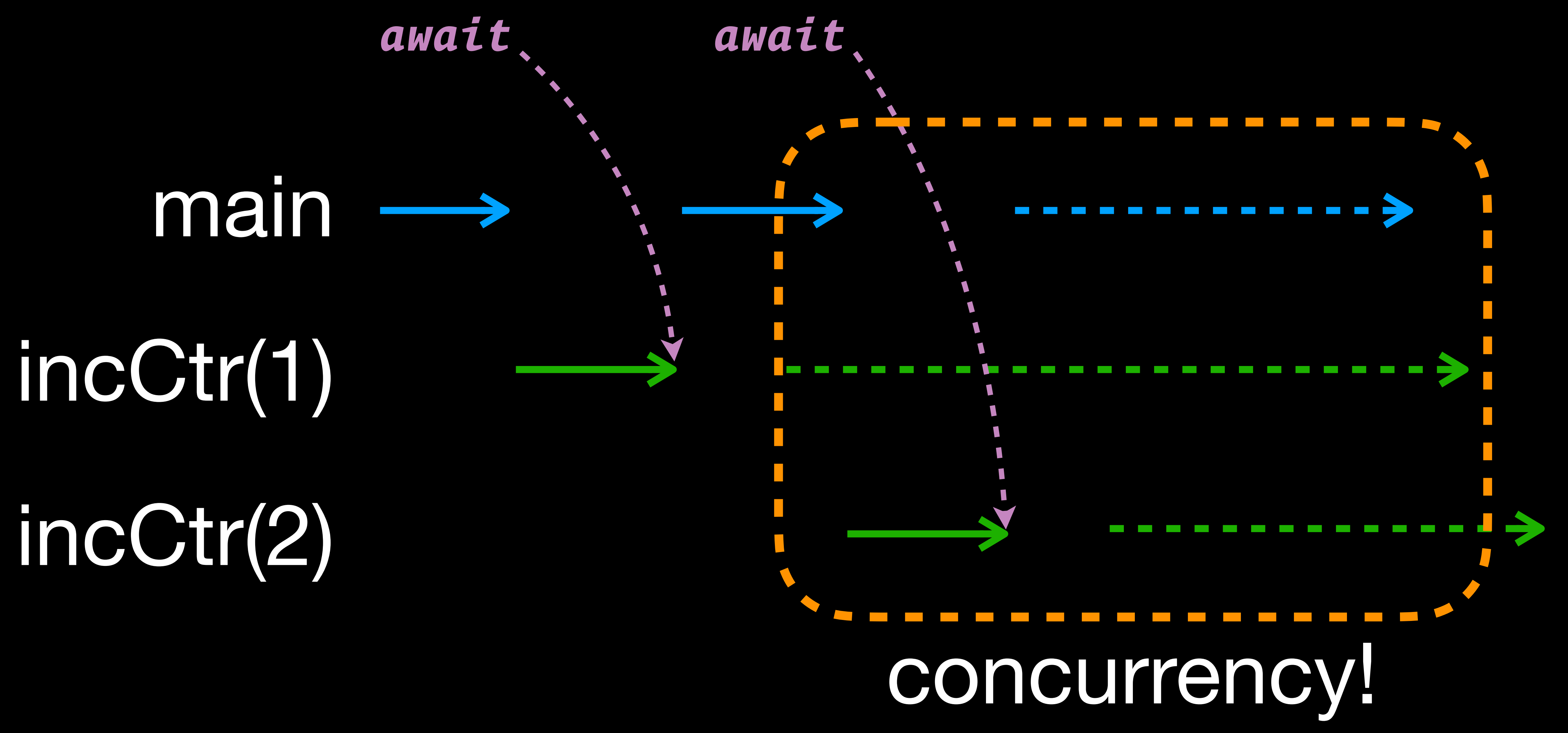
thread 2:

```
counter = counter + 1;
```

Can asynchronous code → race conditions?

```
Future<void> incrementCounter() async {
  for (int i = 0; i < 1000; i++) {
    int temp = counter;
    await ...;
    counter = temp + 1;
  }
}
```

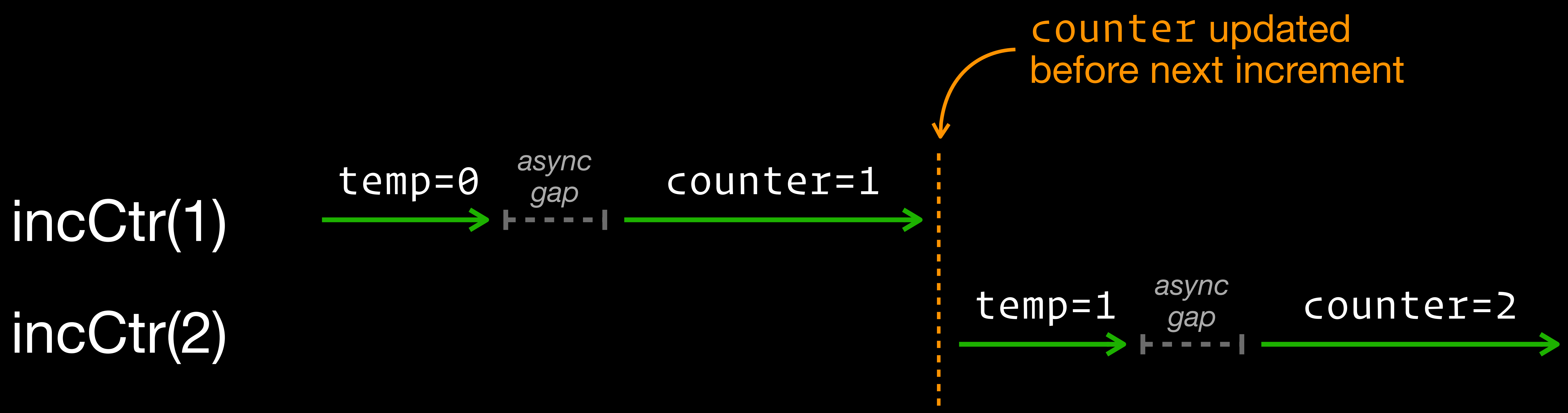
```
void main() {
  counter = 0;
  incrementCounter();
  incrementCounter();
  ...
}
```



Can asynchronous code → race conditions?

```
Future<void> incrementCounter() async {
  for (int i = 0; i < 1000; i++) {
    int temp = counter;
    await ...;
    counter = temp + 1;
  }
}
```

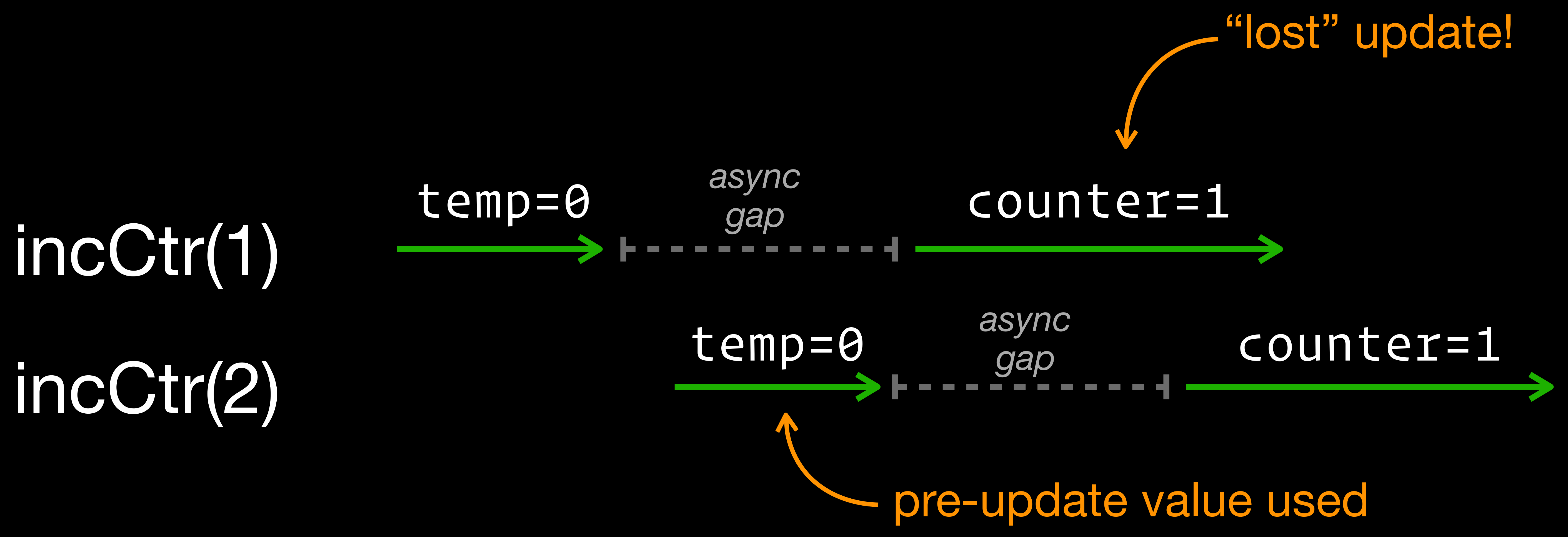
```
void main() {
  counter = 0;
  incrementCounter();
  incrementCounter();
  ...
}
```



Can asynchronous code → race conditions?

```
Future<void> incrementCounter() async {
  for (int i = 0; i < 1000; i++) {
    int temp = counter;
    await ...;
    counter = temp + 1;
  }
}
```

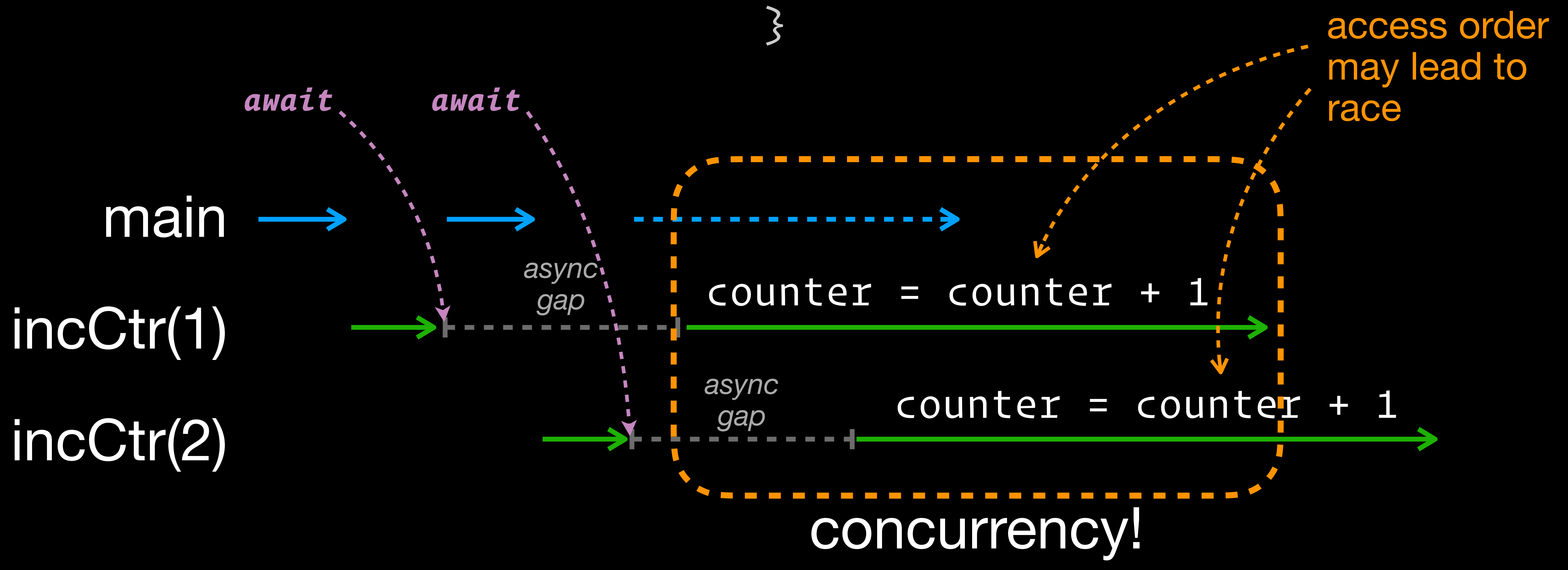
```
void main() {
  counter = 0;
  incrementCounter();
  incrementCounter();
  ...
}
```



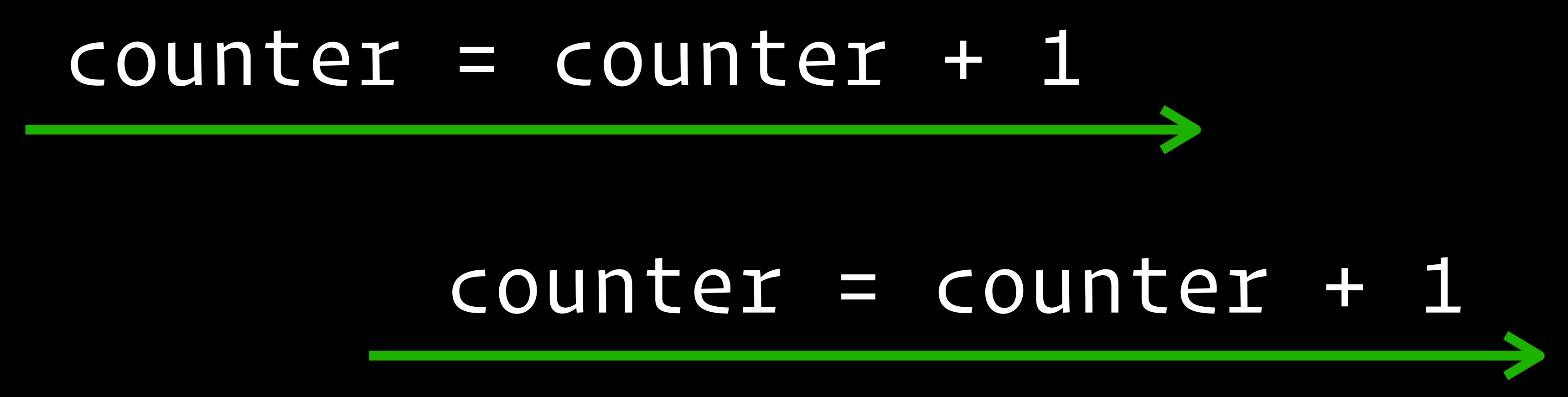
Does this fix it?

```
Future<void> incrementCounter() async {
  for (int i = 0; i < 1000; i++) {
    await ...;
    counter = counter + 1;
  }
}
```

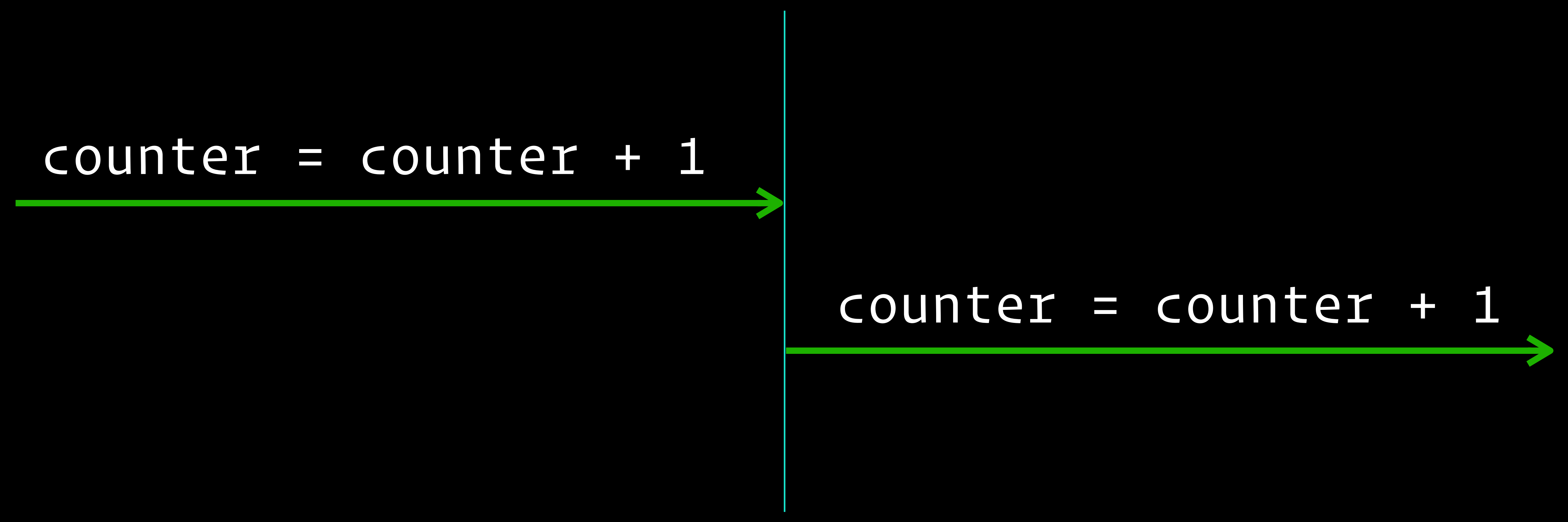
```
void main() {
  counter = 0;
  incrementCounter();
  incrementCounter();
  ...
}
```



How to “cancel” the race?



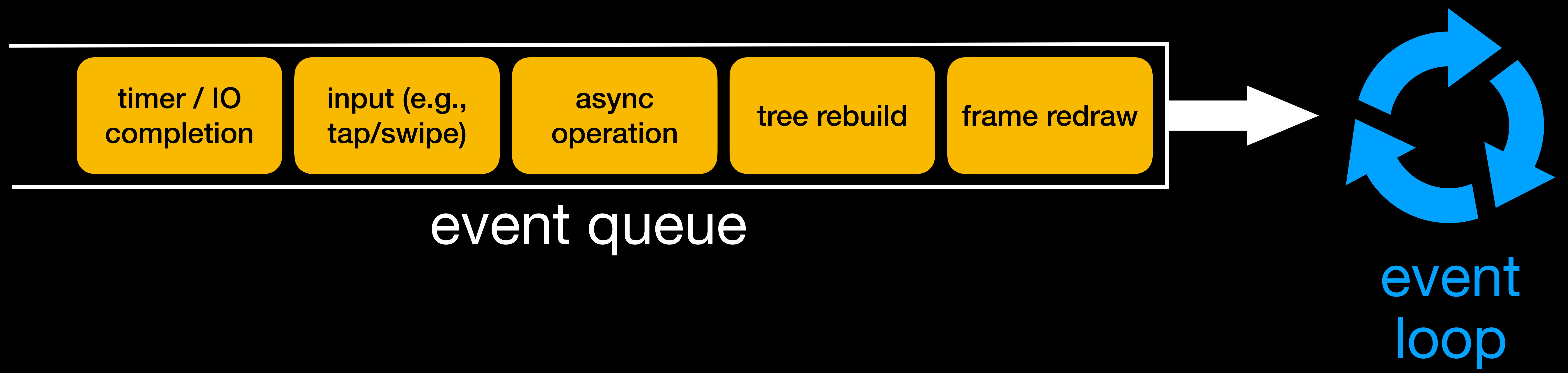
Enforce serial execution!



Single-threaded model

- Many asynchronous programming platforms execute *all* tasks — including the “main” flow of control and asynchronous code — in a **single thread**
  - Avoids overlapping execution, and helps mitigate race conditions
- Central mechanism is the **event loop**
  - Draws from a queue of tasks that are ready to run
  - Executes them sequentially

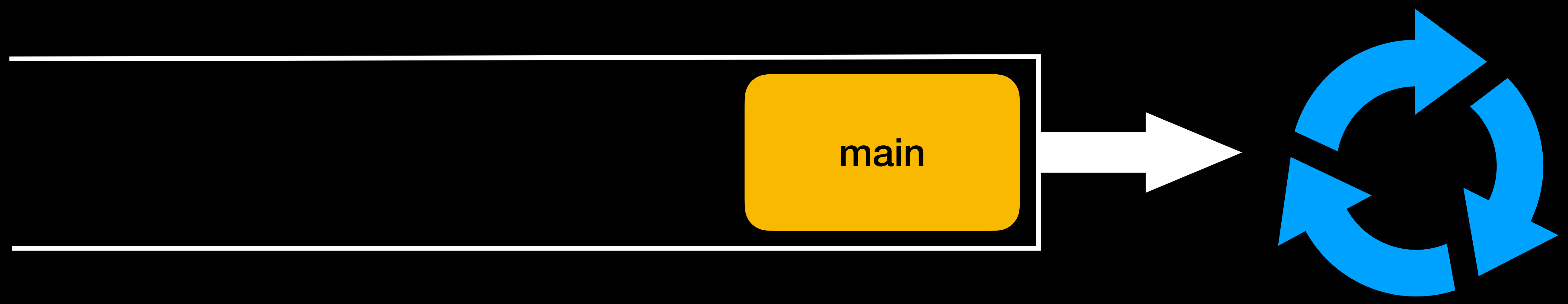
The Event Loop



How does this run on the event loop?

```
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

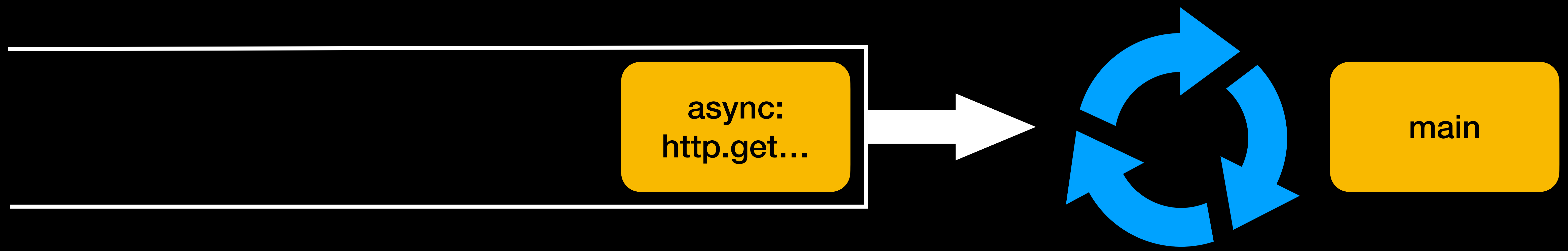
```
void main() {
  Future<Data> data = loadData('https://...');
  doSomethingElse();
  data.then((value) => print('Loaded: $value'));
}
```



How does this run on the event loop?

```dart
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

```dart
void main() {
  Future<Data> data = loadData('https://...');
  doSomethingElse();
  data.then((value) => print('Loaded: $value'));
}
```



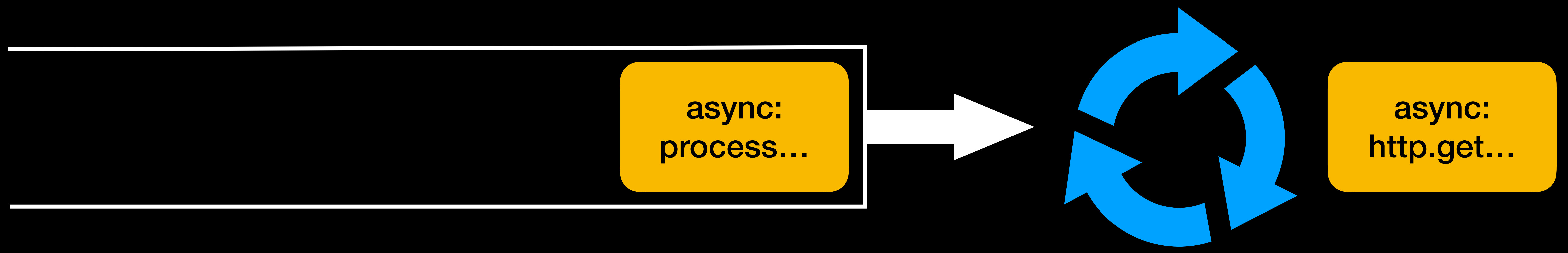

```dart
Future<Data> data = loadData('https://...');
await http.get(url)
doSomethingElse();
data.then((value) => print('Loaded: $value'));
```

registers callback

How does this run on the event loop?

```
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

```
void main() {
  Future<Data> data = loadData('https://...');
  doSomethingElse();
  data.then((value) => print('Loaded: $value'));
}
```

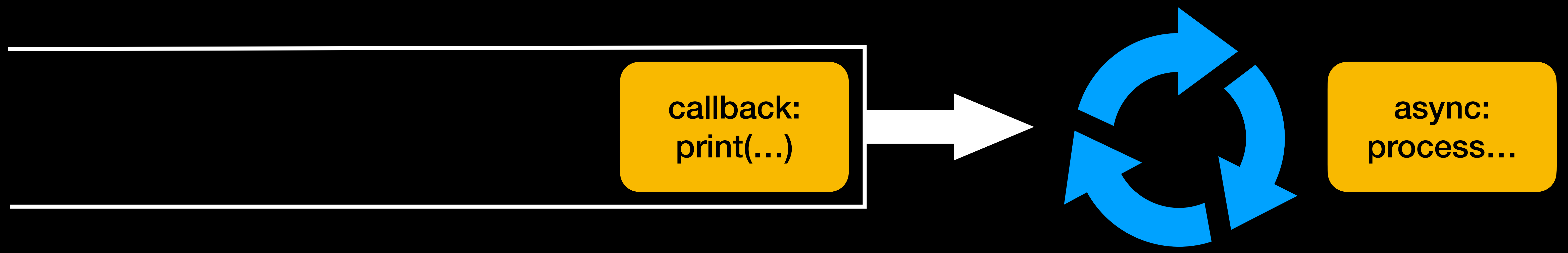



```
http.get(url)
// http.get implementation
var response = result;
await processResponse(response.body)
```

How does this run on the event loop?

```
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

```
void main() {
  Future<Data> data = loadData('https://...');
  doSomethingElse();
  data.then((value) => print('Loaded: $value'));
}
```

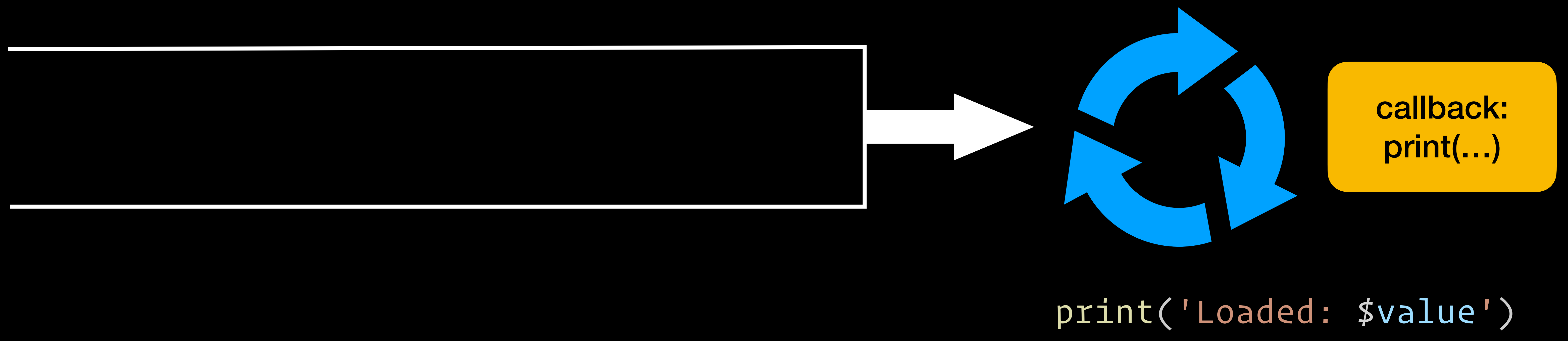



```
processResponse(response.body)
// processResponse implementation
var result = result;
return result;
```

How does this run on the event loop?

```dart
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

```dart
void main() {
  Future<Data> data = loadData('https://...');
  doSomethingElse();
  data.then((value) => print('Loaded: $value'));
}
```



Flutter uses a single-threaded event loop!

- (So does in-browser JavaScript, Node.js, iOS, and many more)
- All widget builds are serialized, and cannot happen while other operations (e.g., state changes) are taking place
- Pros/Cons?
  - Mitigates some (all?) race conditions
  - Potential for UI lag (aka stutter/jank)

Is UI lag possible here?

```
Future<Data> loadData(Uri url) async {
  var response = await http.get(url);
  var result = await processResponse(response.body);
  return result;
}
```

```
Widget build(BuildContext context) {
  return ElevatedButton(
    onPressed: () => loadData('https://...'),
    child: const Text('Load data')
  );
}
```

Dart/Flutter solution: Isolates

- Can run functions in separate, quasi-sandboxed threads: **isolates**
  - Communicate through “message-passing”