Parallel and Concurrent Programming

Slides by Melina O'Dell, adapted from slides by Andrew DeOrio

Agenda

- Distributed Parallel Computing: MapReduce
  - MapReduce programming model
  - MapReduce execution model
  - Fault tolerance
- Asynchronous Programming
  - Asynchronous programming introduction
  - Example: JavaScript event-driven programming
  - Example: JavaScript Promises
- Summary

Parallelism vs Concurrency

- Concurrency: Managing multiple tasks at the same time
- Parallelism: Actually running multiple tasks at the same time

- Concurrency creates the illusion of parallelism

Parallelism vs Concurrency

- Concurrency examples:
  - JavaScript Promises
  - Python asyncio
  - C++ std::async

- Parallelism examples:
  - Multithreading
  - Multiprocessing
  - MapReduce

Agenda

- **Distributed Parallel Computing: MapReduce**
  - MapReduce programming model
  - MapReduce execution model
  - Fault tolerance
- Asynchronous Programming
  - Asynchronous programming introduction
  - Example: JavaScript event-driven programming
  - Example: JavaScript Promises
- Summary

Why learn about parallel programming?

- Distributed system: multiple computers cooperating on a task

- Distributed systems are the solution for dealing with the scale of the web
- Parallel programs are very easy to run on distributed systems

Big program examples

- What kinds of programs are too big to run on one computer?

- Count the number of requests to each web page
  - Given several TB of log files
- Build a search engine inverted index ("look up table")
  - Over several PB of HTML files
- Count the frequency of words
  - In several PB of text files
  - Part of search engine inverted index construction

Not a parallel program

```
import sys
import collections

word_count = collections.defaultdict(int)

for line in sys.stdin:
    words = line.split()
    for word in words:
        word_count[word] += 1

for word, count in word_count.items():
    print(f"{word}\t{count}")
```

`defaultdict` automatically initializes values to zero

- Only one input file
- Does this run faster with multiple computers?

```
$ cat input.txt | python3 wc.py
Hello 2
World 2
Bye 1
Hadoop 2
Goodbye 1
```

Parallel example

- Split input into two files
- Run same program on two computers

```
input01.txt
Hello World
Bye World
```

```
word_count =
 llections.defaultdict(int)
 r line in sys.stdin:
    words = line.split()
    for word in words:
        word_count[word] += 1
```

```
input02.txt
Hello Hadoop
Goodbye Hadoop
```

```
word_count =
 ollections.defaultdict(int)
 or line in sys.stdin:
    words = line.split()
    for word in words:
        word_count[word] += 1
```

- Problem: two computers can't share a data structure

Stateless AKA pure functions

- Problem: two computers can't share a data structure*
    - One computer can't access the memory of another computer
- Insight: programs that do not share data are easier to parallelize
    - Can't modify any shared data
    - *Only* read input, write output
    - Stateless AKA pure function

*A shared data structure could be implemented using a message passing system, but it would be slow. The network is a bottle neck.

MapReduce

- MapReduce: distributed system for compute
  - Run a program that would be too slow on one computer on many computers
- MapReduce Framework handles
  - Parallelization over many machines
  - Segment and distribute input
  - Fault tolerance

- MapReduce provides a clean abstraction for programmers
- Programmer writes two ***stateless*** programs
  - Map
  - Reduce

Agenda

- Distributed Parallel Computing: MapReduce
  - **MapReduce programming model**
  - MapReduce execution model
  - Fault tolerance
- Asynchronous Programming
  - Asynchronous programming introduction
  - Example: JavaScript event-driven programming
  - Example: JavaScript Promises
- Summary

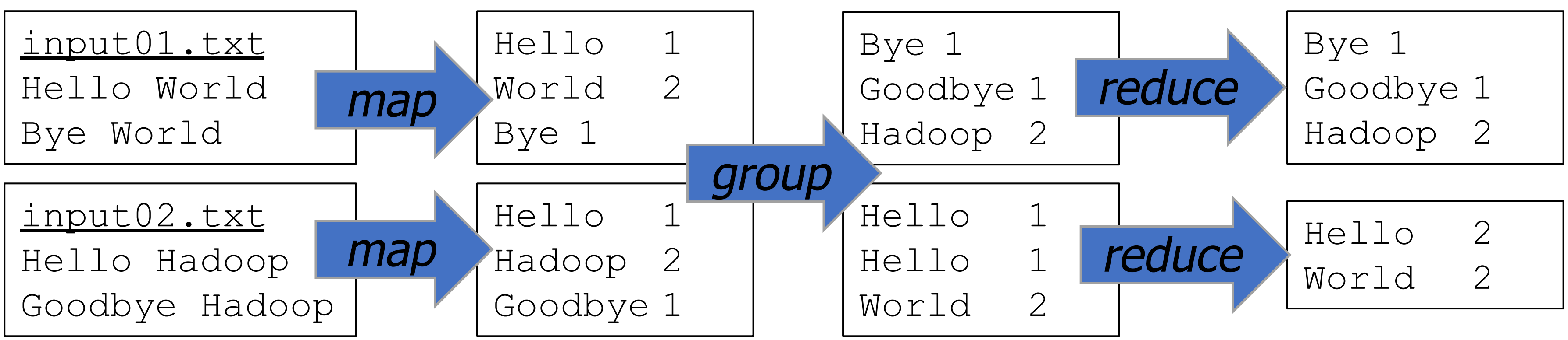
MapReduce programming model

- Our examples will use the Hadoop MapReduce streaming interface

- Map is a program
  - Each line of input is from an input file
  - Each line of output is `<key>\t<value>`
- Group is provided by MapReduce framework
- Reduce is a program
  - Each line of input is `<key>\t<value>`
  - Each line of output is up to the programmer

Parallel example: Word count

- Two computers, two inputs, "one" output
- Count the number of occurrences of each word in a collection of documents



Map

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    words = line.split()
    for word in words:
        word_count[word] += 1
for word, count in word_count.items():
    print(f"{word}\t{count}")
```

- Mapper counts the words in each document
- No shared data structures between computers
- Problem: need to "put together" the outputs

```
input01.txt
Hello World
Bye World
```

map

```
Hello     1
World     2
Bye    1
```

```
input02.txt
Hello Hadoop
Goodbye Hadoop
```

map

```
Hello     1
Hadoop2
Goodbye   1
```

Intermediate key-value pairs

- Map output is in the form of *key-value pairs*
- Separated by a TAB (`\t`) in the Hadoop streaming interface
- What to put here is up to the programmer

```
input01.txt
Hello World
Bye World
```

map →

```
Hello     1
World     2
Bye   1
```

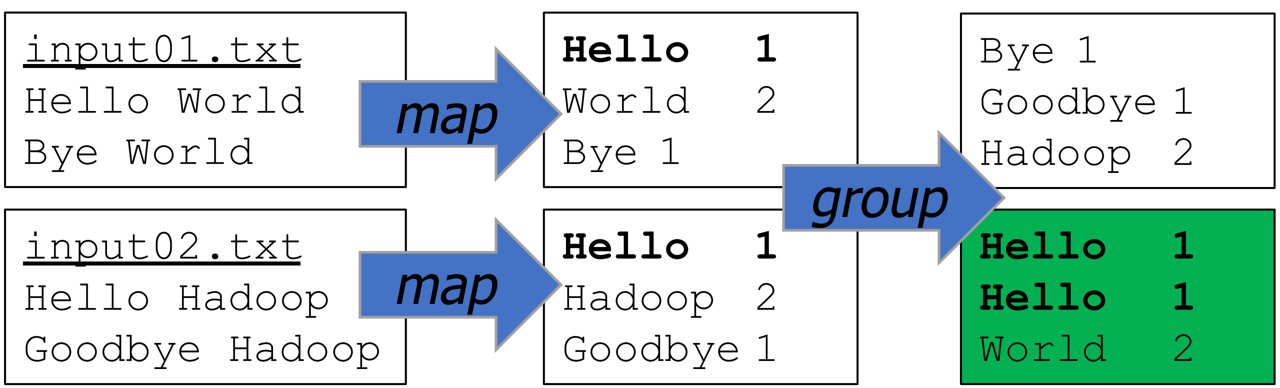
```
input02.txt
Hello Hadoop
Goodbye Hadoop
```

map →

```
Hello     1
Hadoop2
Goodbye   1
```

Group

- Problem: need to "put together" the output in the end
- Insight: output of first program(s) is in the form <*key*>*t*<*value*>
- Solution: every line with the **same key** is in the **same group**
    - Could a group have more than 1 key? **Yes!** (With Hadoop)
- Grouping is provided by the MapReduce framework



Reduce

- After we have groups, we can "put together the output"
- No shared data structures, runs in parallel like map
- Write reduce program assuming all lines with the same key are present

```
Bye 1
Goodbye 1
Hadoop  2
```

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    word, count = line.split()
    word_count[word] += count
for word, count in word_count.items():
    print(f"{word}\t{count}")
```

```
Bye 1
Goodbye 1
Hadoop  2
```

```
Hello   1
Hello   1
World   2
```

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    word, count = line.split()
    word_count[word] += count
for word, count in word_count.items():
    print(f"{word}\t{count}")
```

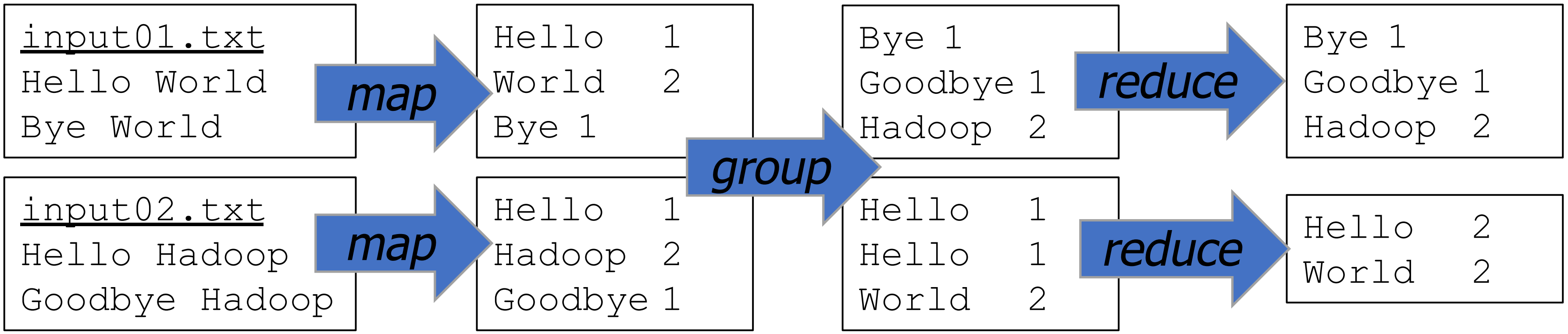
```
Hello   2
World   2
```

Reduce

```
counts = collections.defaultdict(int)

for line in sys.stdin:
    line = line.strip()
    word, count = line.split("\t")
    counts[word] += int(count)

for word, count in counts.items():
    print(f"{word}\t{count}")
```



Key observations

- Map and reduce functions (programs) inspired by functions of the same name in Lisp programming language
- Functional programming
    - Computation as the evaluation of mathematical functions
- Functions have no side effects
    - AKA "pure" functions
    - AKA stateless
    - Does not change state outside itself
- Easy to parallelize!

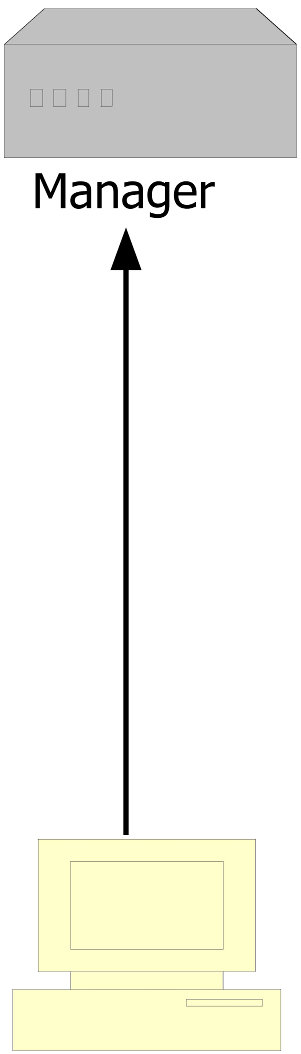
Agenda

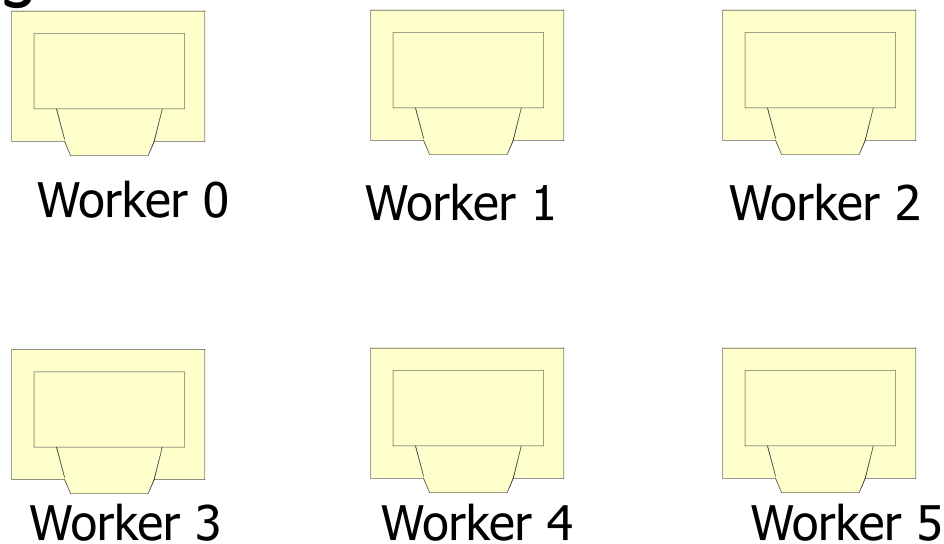
- Distributed Parallel Computing: MapReduce
    - MapReduce programming model
    - **MapReduce execution model**
    - Fault tolerance
- Asynchronous Programming
    - Asynchronous programming introduction
    - Example: JavaScript event-driven programming
    - Example: JavaScript Promises
- Summary

MapReduce execution model

1. Segment input (AKA partition)
2. Map stage
3. Group stage
4. Reduce stage

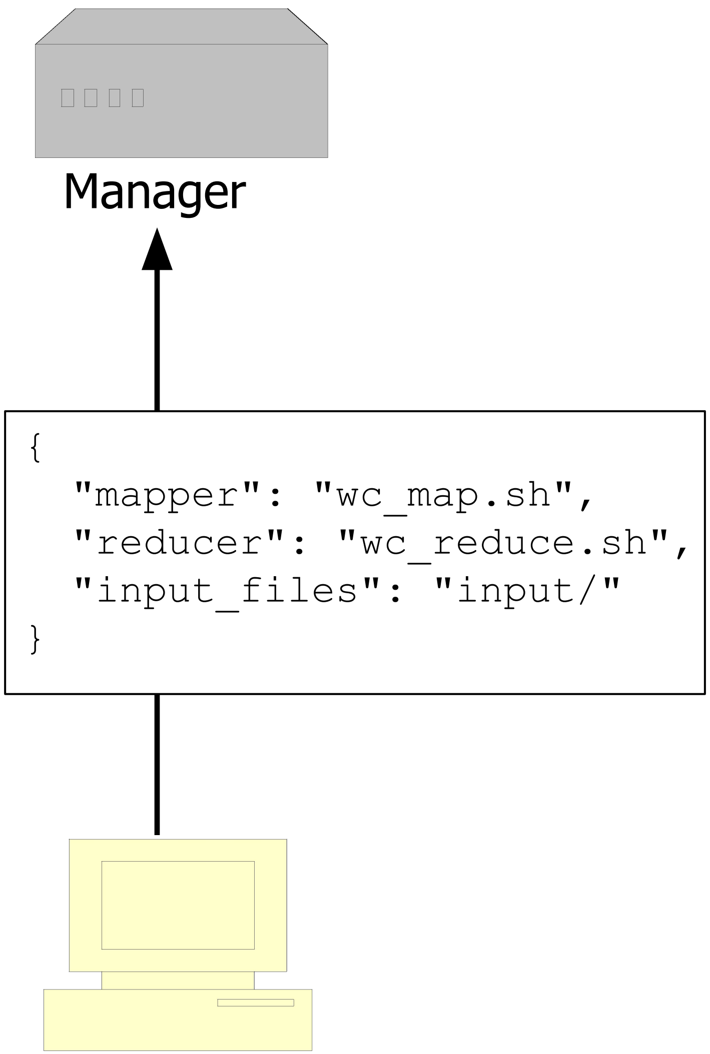
MapReduce machines





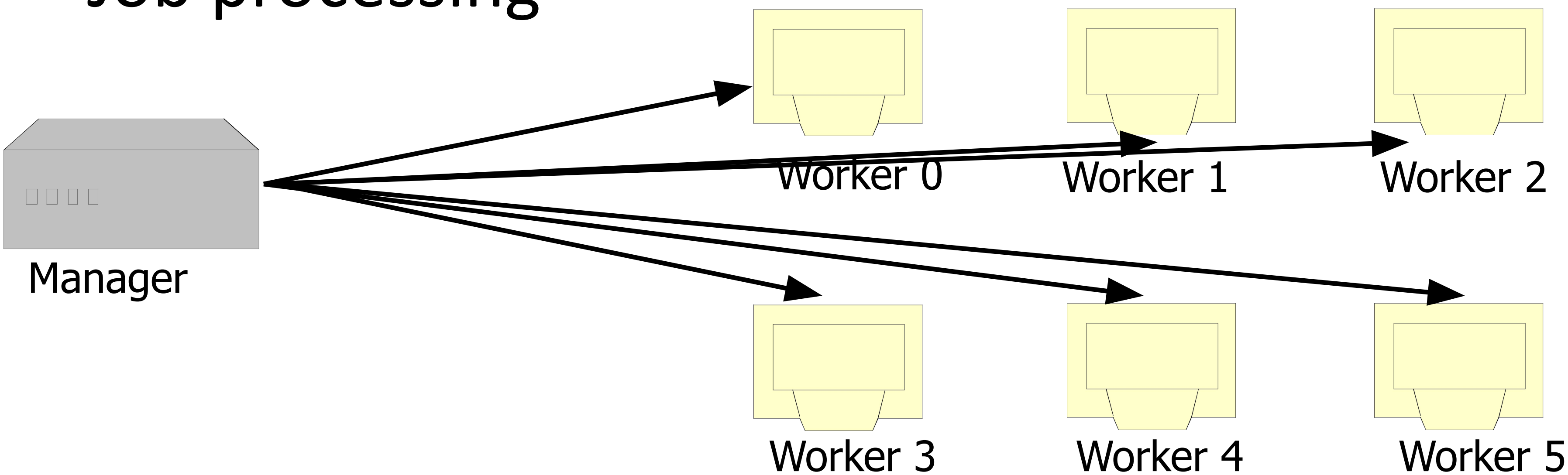
- Client is the programmer
- **Manager** coordinates workers
- **Workers** do actual computation

Job processing

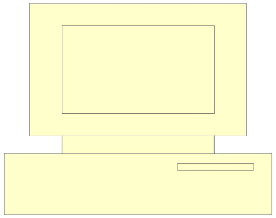


1. Client submits word count job, indicating map code, reduce code and input files

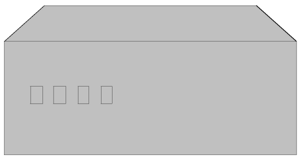
Job processing



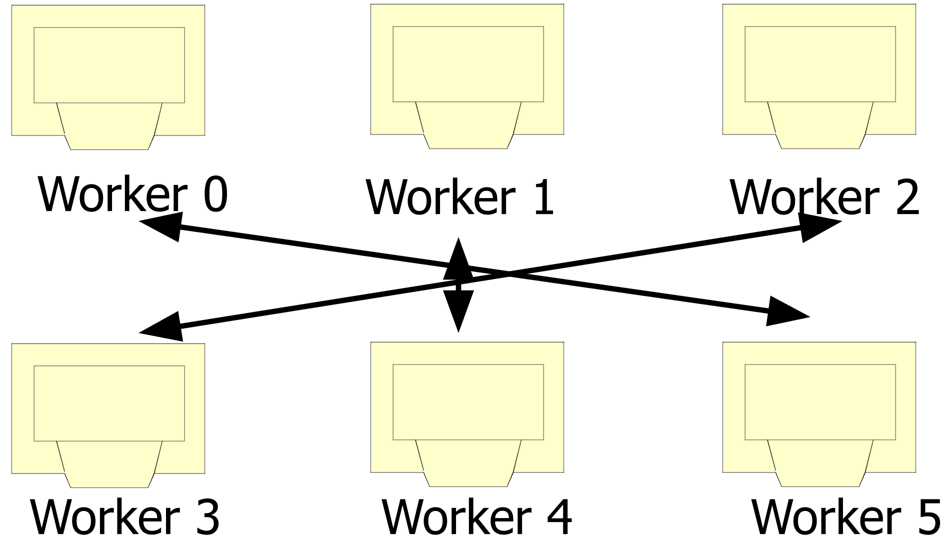
1. Client submits word count job, indicating map code, reduce code and input files
2. Manager breaks input file into *k* chunks, (in this case 6). Assigns work to workers.



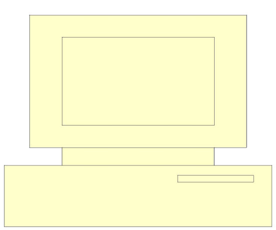
Job processing



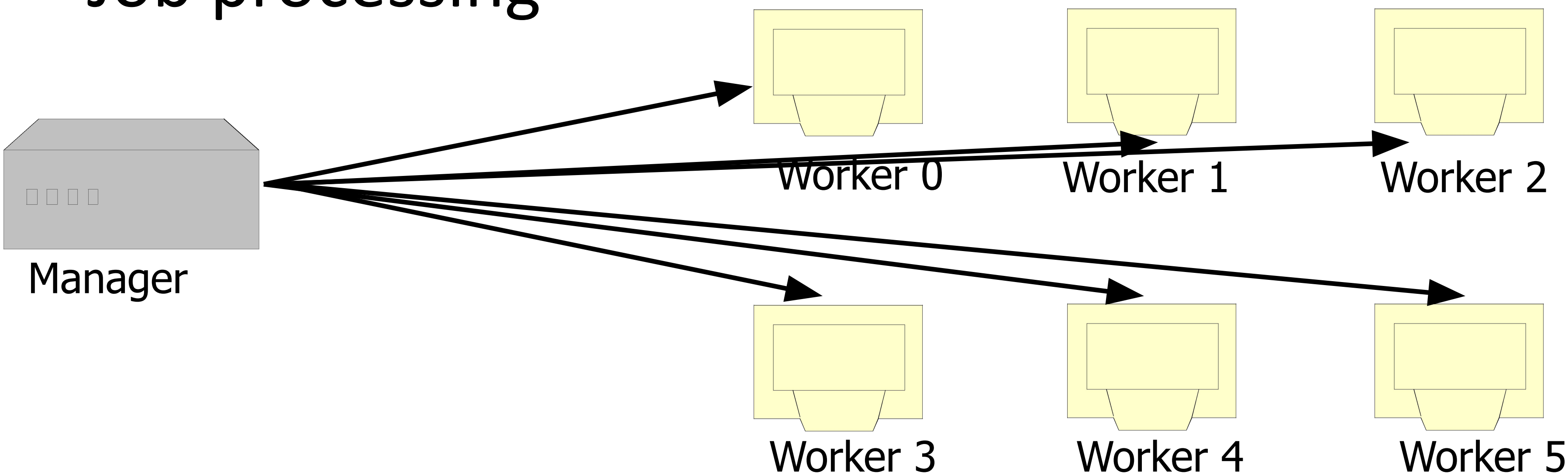
Manager



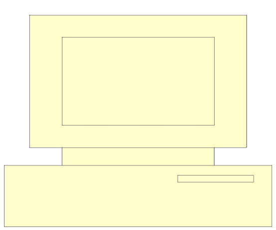
1. Client submits word count job, indicating map code, reduce code and input files
2. Manager breaks input file into *k* chunks, (in this case 6). Assigns work to workers.
3. After map(), workers exchange map-output to produce groups



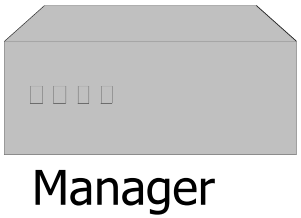
Job processing

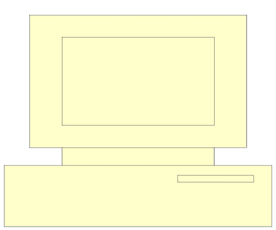


1. Client submits word count job, indicating map code, reduce code and input files
2. Manager breaks input file into *k* chunks, (in this case 6). Assigns work to workers.
3. After map(), workers exchange map-output to produce groups
4. Manager reduce() keyspace into *m* chunks (in this case 6). Assigns work.



Job processing





1. Client submits word count job, indicating map code, reduce code and input files
2. Manager breaks input file into *k* chunks, (in this case 6). Assigns work to workers.
3. After map(), workers exchange map-output to produce groups
4. Manager breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to shared fs

Agenda

- Distributed Parallel Computing: MapReduce
  - MapReduce programming model
  - MapReduce execution model
  - **Fault tolerance**
- Asynchronous Programming
  - Asynchronous programming introduction
  - Example: JavaScript event-driven programming
  - Example: JavaScript Promises
- Summary

Fault tolerance

- How do we know if a machine goes down?
- Workers send periodic heartbeat messages to Manager
- Manager keeps track of which workers are up

Fault tolerance

- What happens when a machine dies?
- Without MapReduce
    - Program is restarted
    - Not so hot if your job is in hour 23
- With MapReduce
    - If worker dies
        - Just restart that task on a different machine
        - You lose a piece the overall work, but no big deal
        - If Reducing, can reuse output of Map/Group stage

Further reading

- Nice explanation from UC Berkeley
  - http://inst.eecs.berkeley.edu/~cs61a/book/chapters/streams.html#distributed-data-processing
- Some researchers disagree with MapReduce's popularity: “MapReduce: A Major Step Backwards”
  - https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html
- Paper on Google's MapReduce framework "MapReduce: Simplified Data Processing on Large Clusters" by Jeffrey Dean and Sanjay Ghemawat
  - https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Agenda

- Distributed Parallel Computing: MapReduce
  - MapReduce programming model
  - MapReduce execution model
  - Fault tolerance
- **Asynchronous Programming**
  - Asynchronous programming introduction
  - Example: JavaScript event-driven programming
  - Example: JavaScript Promises
- Summary

Asynchronous is not…

- Asynchronous programming is not a single-thread blocking program
- Blocking: wait for one task to finish before executing the next

- Examples of tasks:
    - fetch(): a GET request to a REST API
    - json(): parse JSON string
    - Respond to user clicking a button on UI and update UI

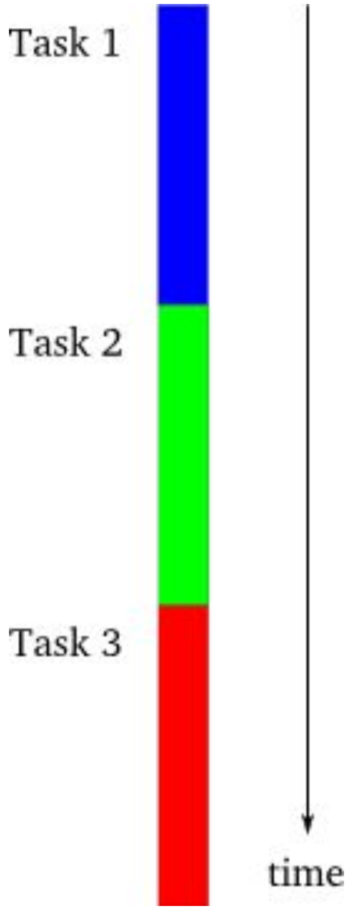


Image credit:
http://krondo.com/

Asynchronous is not…

- Asynchronous programming is not a multi-thread blocking program
- Modern OS threads "take turns" on one processor

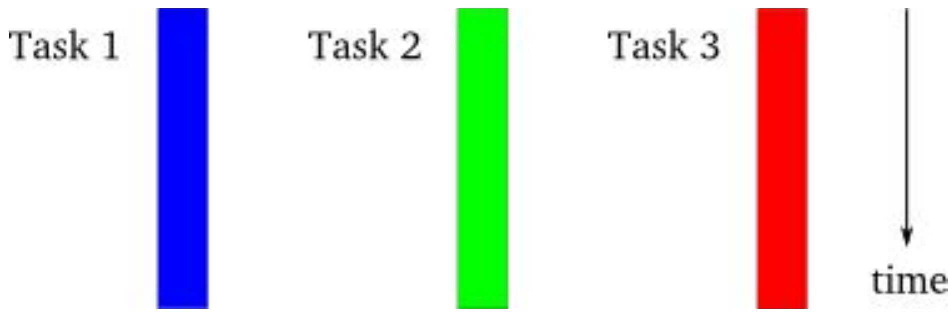


Image credit:
http://krondo.com/

Asynchronous is…

- Asynchronous programming is tasks interleaved with one another, in a single thread of control
- Programmer controls when tasks "take turns"

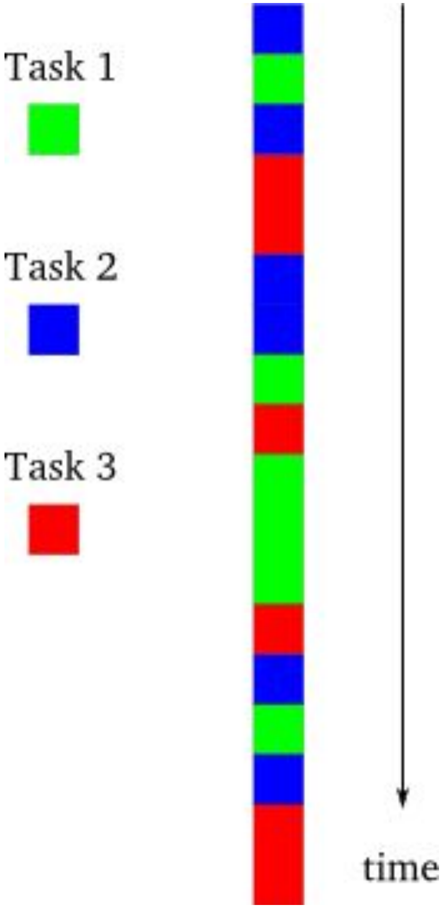


Image credit:
http://krondo.com/

Why asynchronous?

- Why use asynchronous programming?
- UIs: by interleaving the tasks, system is responsive to user input while still performing other work in the "background"
- Waiting for I/O: do "other useful things" while waiting for I/O, like a network or disk
  - Synchronous programs are bad at this

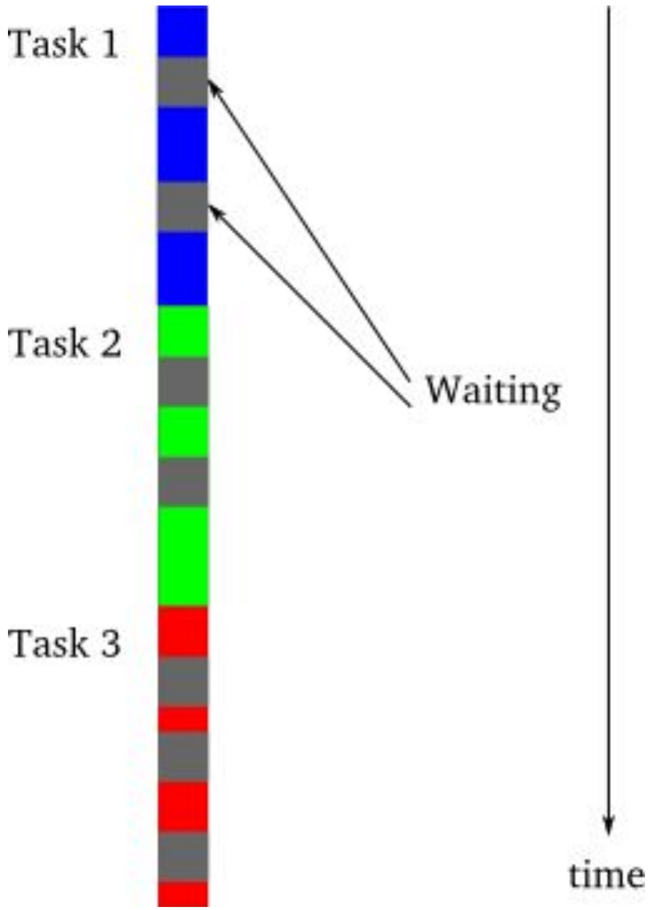


Image credit: http://krondo.com/

Why asynchronous?

- What are "other useful things" to do while waiting in a web app?
  - Respond to user mouse hover event
  - Respond to user clicking a button
  - Respond to use filling in a form, e.g., validate input
  - Check for new mail (Gmail)
  - Check for new posts (Instagram)

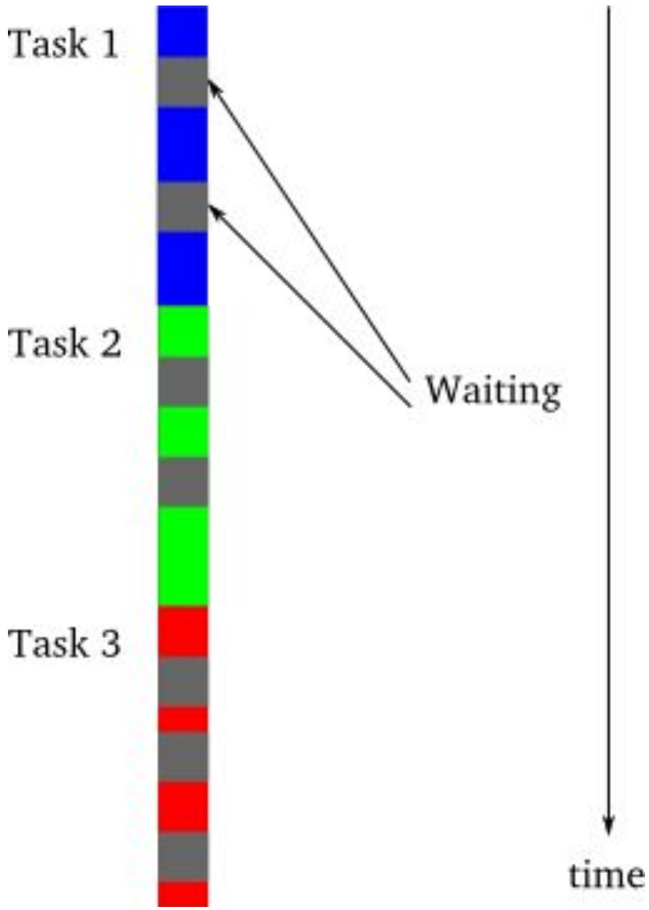


Image credit: http://krondo.com/

When asynchronous?

- When to use asynchronous programming?

- There are a large number of tasks so there is likely always at least one task that can make progress
- The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running
- The tasks are largely independent from one another so there is little need for inter-task communication (and thus for one task to wait upon another)

JavaScript

- Programming language commonly used on the web for GUIs
- Single-threaded
- Two examples of asynchronous programming:
  - Event-driven (event table, event loop, event queue)
  - Using Promises (microtask queue)

Agenda

- Distributed Parallel Computing: MapReduce
    - MapReduce programming model
    - MapReduce execution model
    - Fault tolerance
- Asynchronous Programming
    - Asynchronous programming introduction
    - **Example: JavaScript event-driven programming**
    - Example: JavaScript Promises
- Summary

Event-driven programming

- In event-driven programming, the flow of the program is determined by *events*

- A few examples of events built into the browser:
  - **`click`**: user clicks a button
  - **`mouseover`**: The user moves the mouse over an HTML element
  - **`keydown`**: The user pushes a keyboard key
  - **`load`**: The browser has finished loading the page

Callback functions

- A main loop (***event loop***) listens for events and triggers a callback function
- A *callback function* is a function waiting to be executed
  - Current example: `hello` is a callback

```
const hello = function (event) {
    console.log("Hello World!");
};
```

Event handlers

- We can register the callback function as an *event handler*
- AKA "Please run this function when the click event occurs"

```
const button = document.querySelector("#button");
button.addEventListener("click", hello);
```

- The JavaScript interpreter maintains a table of events that map to references to functions

Event	Function
`click`	`hello`

The event queue

- In JavaScript, **function calls** live on the stack, **objects** live on the heap, and ***messages*** *live on the queue*
- The function on the top of the stack executes
- *When the stack is empty, a message is taken out of the queue and processed*
- Each message is a function

- An ***event*** adds a message to the *queue* from the *event table*

Example: Adding events to the queue

- You can schedule an event on the queue for a later time
- This function will run approximately 1s in the future
- `callback1` is added to the event table, which maps events to callbacks

```
function callback1() {
    console.log("this is a msg from callback1");
}
setTimeout(callback1, 1000);
```

Example: Adding events to the queue

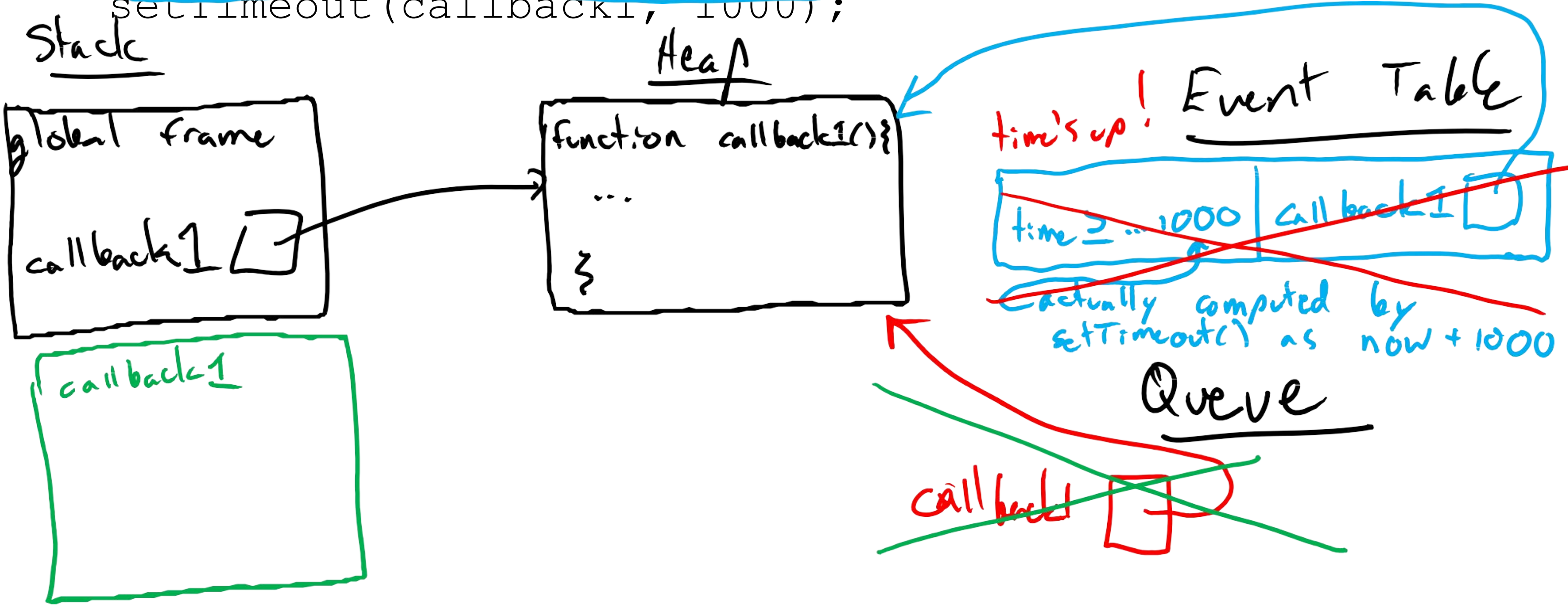
```
function callback1() {
    console.log("this is a msg from callback1");
}
setTimeout(callback1, 1000);
```

output: 'this is a msg from callback1'

Review: adding events to the queue

1000 ms later...

```
function callback1() {
    console.log("this is a msg from callback1");
}
setTimeout(callback1, 1000);
```



Example: Adding events to the queue

```
function callback1() {
    console.log("this is a msg from callback1");
}
setTimeout(callback1, 1000);
slow();  // How does this example change?
```

Event-driven programming is asynchronous

- Events fire outside of the regular flow of the program
- Event callback functions are added to event queue
- Callbacks are popped and executed in the background at a later time

- All of this happens asynchronously, without blocking

Agenda

- Distributed Parallel Computing: MapReduce
  - MapReduce programming model
  - MapReduce execution model
  - Fault tolerance
- Asynchronous Programming
  - Asynchronous programming introduction
  - Example: JavaScript event-driven programming
  - **Example: JavaScript Promises**
- Summary

Promises

- Control the flow of deferred and asynchronous operations
- First class representation of a value that may be made asynchronously and be available in the future

- Examples of values that will be available in the future
    - The response to a server request: `fetch()`
    - The data from parsing a JSON string: `json()`

- AKA "Futures" in other languages
- More generically “asynchronous tasks”

Promise chaining

- Promise types look like `Promise<T>`, where `T` is the type of the value that will be available later
- When future logic relies on the value from the async call, we can use *promise chains* to block the execution while we wait
  - `.then()` lets us specify a callback function to be run when the value is available
  - The output (resolved value) of one Promise is the input to the callback

- Independent logic outside of the chain still runs while we wait

GitHub API

- We'll use the GitHub API for our examples
- Example:

```
$ curl -s https://api.github.com/users/melodell
{
  "login": "melodell",
  "id": 69565683,
  ...
  "url": "https://api.github.com/users/melodell",
  ...
}
```

`fetch` API

- The `fetch` API provides an interface for HTTP requests
  - `fetch()` returns a Promise that resolves when the response comes back
  - `response.json()` returns a Promise that resolves when parsing is complete

```
function showUser() {
  function handleResponse(response) {
      return response.json();
  }

  function handleData(data) {
    // just print to console for today's examples
    console.log(data);
  }

  fetch('https://api.github.com/users/melodell')
    .then(handleResponse)
    .then(handleData)
}
```

Using a Promise

- After the value is available, the Promise calls a function provided by `.then()`

```
function showUser() {
  function handleResponse(response) {
      return response.json();
  }

  function handleData(data) {
      console.log(data);
  }

  fetch('https://api.github.com/users/melodell')
    .then(handleResponse)
    .then(handleData)
}
```

Exercise

What is the output of this code?

```
function showUser() {
  console.log("hello");
  let p1 = fetch('https://api.github.com/users/melodell');
  let p2 = p1.then(response => response.json());
  let p3 = p2.then(data => console.log(data.login));
  console.log("world");
}
```

Exercise

What is the output of this code?

```
function showUser() {
  console.log("hello");
  let p1 = fetch('https://api.github.com/users/melodell');
  let p2 = p1.then(response => response.json());
  let p3 = p2.then(data => console.log(data.login));
  console.log("world");
}
```

```
// hello
// world
// melodell
```

Promise chain diagram

- Imagine a `Promise` as a linked list of function objects

```
let p1 = fetch('https://api.github.com/users/melodell');
let p2 = p1.then(response => response.json());
let p3 = p2.then(data => console.log(data.login));
```

Promises explained again

- Functions performing asynchronous tasks return a `Promise`
- A `Promise` is an object to which you can attach a callback
  - Using `.then()`
- `Promise`s add messages to the *microtask queue*
  - When the stack is empty, the event loop checks the microtask and macrotask queues for messages

Promise chaining example

- A common need is to execute two or more asynchronous operations back-to- back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step
- Example:
  - Request
  - Parse JSON
- We accomplish this by creating a *promise chain*

```
function showUser() {
    fetch('https://api.github.com/users/melodell')
    .then((response) => {
        return response.json();
    })
    .then((data) => {
        console.log(data);
    })
}
```

Promise chaining example

- The output (resolved value) of one Promise is the input to the next callback function in the chain

```
function showUser() {
   fetch('https://api.github.com/users/melodell')
   .then((response) => {
     return response.json();
   })
   .then((data) => {
     console.log(data);
   })
}
```

Promise chaining example

- Alternate syntax with `async/await`
  - `await` is blocking within the scope of the async function

```
async function showUser() {
    const response = await fetch(...);
    const data = await response.json();
    console.log(data);
}
```

Promises

- A Promise is in one of these states:
  - ***pending***: initial state, neither fulfilled nor rejected
  - ***fulfilled***: meaning that the operation completed successfully
  - ***rejected***: meaning that the operation failed

- On success, the method provided by `.then()` runs
- On failure, the method provided by `.catch()` runs

Error handling example

```
function showUser() {
    fetch('https://api.github.com/users/melodell')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch(error => console.log(error))
}
```

Creating Promises

- Can write your own functions that return Promises
- Example: `wait` returns a Promise that resolves in `ms` milliseconds

```
function wait(ms) {
  return new Promise((resolve) => {
     setTimeout(resolve, ms);
   }
};
```

Relation to synchronous methods

- Asynchronous methods return values like synchronous methods
- Instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future

```
wait(1000)
  .then(() => console.log('1 second passed'))
  .catch(error => console.log(error))
```

Further Reading

- Asynchronous tasks (course notes!)
  - https://eecs390.github.io/notes/concurrent.html#asynchronous-tasks
- JavaScript Event loop: microtasks and macrotasks
  - https://javascript.info/event-loop
- JavaScript Promises
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Summary

- Concurrency: Managing multiple tasks at the same time
- Parallelism: Actually running multiple tasks at the same time

- MapReduce is a distributed system for compute
- Programs that do not share data are easier to parallelize
- MapReduce framework
  - Easy to run a parallel program
  - Fault tolerance

- Asynchronous programming: Interleave tasks
- Event-driven programming: Function calls triggered by events
- A Promise represents a value that will be available in the future
  - Controls the flow of asynchronous operations
  - Chained promises form a sequence of asynchronous events