Web Scraping

HTML, XPath e Iteração

Fluxo do web scraping

1. Imitar

- Na aba Network seu navegador, investigue as requisições.
- Tente imitá-las no R, copiando os caminhos e parâmetros utilizados.

2. Coletar

- Baixar todas as páginas HTML (ou outro formato).
- Boa prática: salvar arquivos brutos com `httr::write_disk()`.

3. Parsear

- Transformar os dados brutos em uma base de dados passível de análise.
- Utilizar pacotes `{xml2}`, `{jsonlite}`, `{pdftools}`, dependendo do arquivo.

Pacotes

- Utilizar `{httr}` para imitar/coletar.
- Utilizar `{xml2}` para parsear.
- Utilização massiva do `{tidyverse}`.

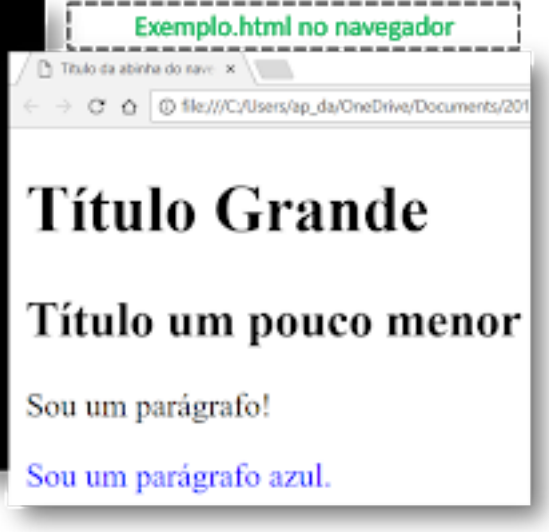
HTML

- HTML (Hypertext Markup Language) é uma linguagem de marcação cujo uso é a criação de páginas web.
- Por trás de todo site há pelo menos um arquivo .html.

Exemplo.html no editor de texto

```
<!DOCTYPE html>

  <head>
    <meta charset = latin1>
    <title>Título da abinha do navegador</title>
  </head>

  <body>
    <h1>Título Grande</h1>

    <h2>Título um pouco menor</h2>

    <p>Sou um parágrafo!</p>

    <p style='color: blue;'>Sou um parágrafo azul.</p>

  </body>
</html>
```

Exemplo.html no navegador



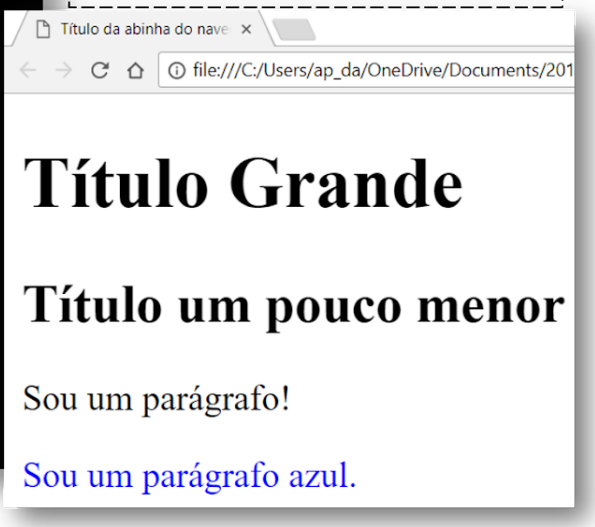
HTML

- Todo arquivo HTML pode ser dividido em seções que definirão diferentes aspectos da página.
- <head> descreve metadados, enquanto <body> é o corpo da página.

Exemplo.html no editor de texto

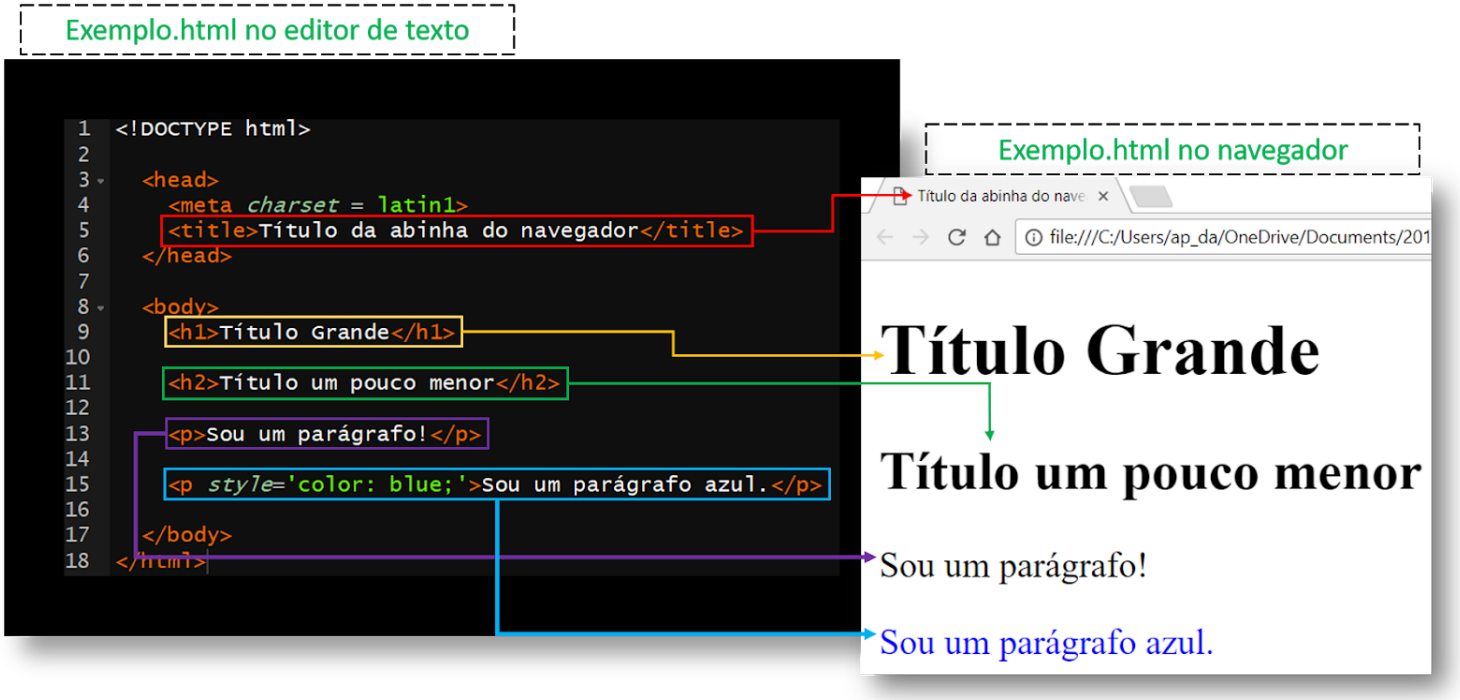
```
1  <!DOCTYPE html>
2
3  <head>
4    <meta charset = latin1>
5    <title>Título da abinha do navegador</title>
6  </head>
7
8  <body>
9    <h1>Título Grande</h1>
10
11   <h2>Título um pouco menor</h2>
12
13   <p>Sou um parágrafo!</p>
14
15   <p style='color: blue;'>Sou um parágrafo azul.</p>
16
17 </body>
18 </html>
```

Exemplo.html no navegador



HTML

- Tags (head, body, h1, p, ...) demarcam as seções e sub-seções
- enquanto atributos (charset, style, ...) mudam como essas seções são renderizadas pelo navegador.



Teoria

1) Todo HTML se transforma em um DOM (document object model) dentro do navegador.

2) Um DOM pode ser representado como uma árvore em que cada node é:

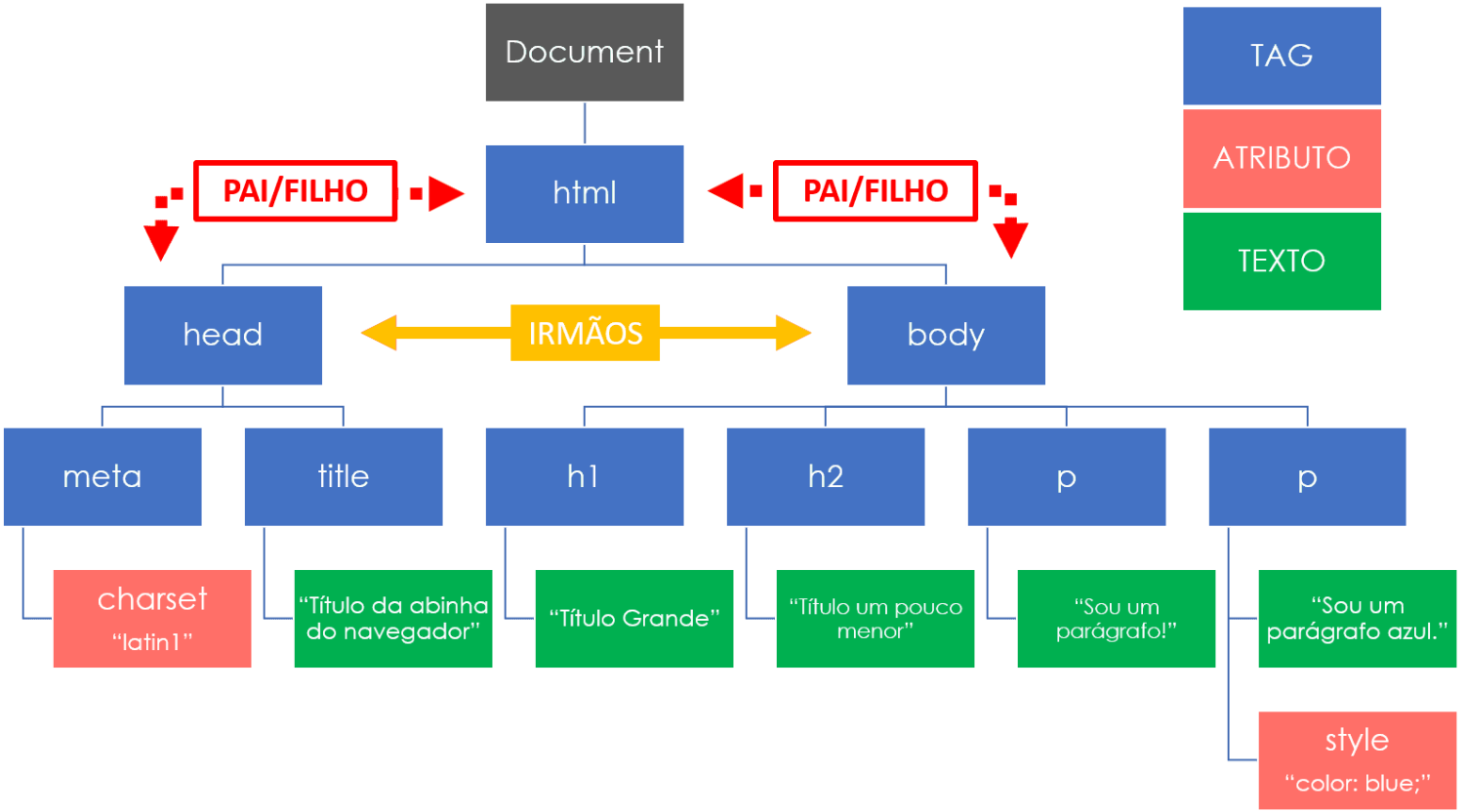
- ou um atributo
- ou um texto
- ou uma tag
- ou um comentário

3) Utiliza-se a relação de pai/filho/irmão entre os nós.

4) Para descrever a estrutura de um DOM, usamos uma linguagem de markup chamada XML (Extensible Markup Language) que define regras para a codificação de um documento.

HTML

O HTML do exemplo, na verdade, é isso aqui:



XPath - XML Path Language

- Exemplo: coletando todas as tags <p> (parágrafos)

```
library(xml2)

# Ler o HTML
html <- read_html("img/html_exemplo.html")

# Coletar todos os nodes com a tag <p>
nodes <- xml_find_all(html, "//p")

# Extrair o texto contido em cada um dos nodes
text <- xml_text(nodes)
text
```

```
## [1] "Sou um parágrafo!"      "Sou um parágrafo azul."
```

XPath - XML Path Language

- Com `xml_attrs()` podemos extrair todos os atributos de um node:

```
xml_attrs(nodes)
```

```
## [[1]]
## named character(0)
##
## [[2]]
##          style
## "color: blue;"
```

```
xml_attr(nodes, "style")
```

```
## [1] NA              "color: blue;"
```

XPath - XML Path Language

- Já com `xml_children()`, `xml_parents()` e `xml_siblings()` podemos acessar a estrutura de parentesco dos nós:

```
heads <- xml_find_all(html, "head")
xml_siblings(heads)
```

```
## {xml_nodeset (1)}
## [1] <body>\r\n    <h1>Título Grande</h1>\r\n    \r\n    <h2>Título
```

```
xml_children(heads)
```

```
## {xml_nodeset (3)}
## [1] <meta http-equiv="Content-Type" content="text/html; charset=UT
## [2] <meta charset="utf-8">\n
## [3] <title>Título da abinha do navegador</title>
```

{rvest}

- Pacote construído sobre `{xml2}` e `{httr}`
- Busca facilitar a vida com alguns helpers
- Permite utilização de CSS path, uma alternativa ao XPath
- Na prática, no entanto, pode ser improdutivo utilizá-lo
- No nosso curso, só vamos utilizar a função `rvest::html_table()`, que transforma o conteúdo de uma tag `<table>` em um `data.frame`.

CSS

- CSS (Cascading Style Sheets) descrevem como os elementos HTML devem se apresentar na tela. Ele é responsável pela aparência da página.

```
<p style='color: blue;'>Sou um parágrafo azul.</p>
```

- O atributo `style` é uma das maneiras de mexer na aparência utilizando CSS. No exemplo,
- `color` é uma **property** do CSS e
- `blue` é um **value** do CSS.
- Para associar esses pares **properties/values** aos elementos de um DOM, existe uma ferramenta chamada **CSS selectors**. Assim como fazemos com XML, podemos usar esses seletores (através do pacote `rvest`) para extrair os nós de uma página HTML.

CSS

- Abaixo vemos um `.html` e um `.css` que é usado para estilizar o primeiro. Se os nós indicados forem encontrados pelos seletores do CSS, então eles sofrerão as mudanças indicadas.

```
9     <p>Sou um parágrafo normal.</p>
10
11    <p class='azul'>Sou um parágrafo azul.
12
13    <p id="meu-p-favorito" class = 'azul'>S
14
```

No HTML	No CSS
<p>	p
class = ‘azul’	.azul
Id = ‘meu-p-favorito’	#meu-p-favorito

css_exemplo.css

```
1  p {
2    font-size: 25px;
3  }
4
5  .azul {
6    color: blue;
7  }
8
9  #meu-p-favorito {
10   font-weight: bold;
11 }
```

Seletores CSS vs. XPath

- A grande vantagem do XPath é permitir que acessemos os filhos, pais e irmãos de um nó. De fato os seletores CSS são mais simples, mas eles também são mais limitados.
- O bom é que se tivermos os seletores CSS, podemos transformá-los sem muita dificuldade em um query XPath:
- Seletor de tag: `p` = `//p`
- Seletor de classe: `.azul` = `//*[@class='azul']`
- Seletor de id: `#meu-p-favorito` = `//*[@id='meu-p-favorito']`
- Além disso, a maior parte das ferramentas que utilizaremos ao longo do processo trabalham preferencialmente com XPath.

Seletores CSS vs. XPath

```
html <- read_html("img/html_exemplo_css_a_parte.html")
xml_find_all(html, "//p")
```

```
## {xml_nodeset (3)}
## [1] <p>Sou um par?grafo normal.</p>
## [2] <p class="azul">Sou um par?grafo azul.</p>
## [3] <p id="meu-p-favorito" class="azul">Sou um par?grafo azul e ne
```

```
xml_find_all(html, "//*[@class='azul']")
```

```
## {xml_nodeset (2)}
## [1] <p class="azul">Sou um par?grafo azul.</p>
## [2] <p id="meu-p-favorito" class="azul">Sou um par?grafo azul e ne
```

Seletores CSS vs. XPath

```
rvest::html_nodes(html, ".azul")
```

```
## {xml_nodeset (2)}
## [1] <p class="azul">Sou um par?grafo azul.</p>
## [2] <p id="meu-p-favorito" class="azul">Sou um par?grafo azul e ne
```

- Note que `//p` indica que estamos fazendo uma busca na tag `p`, enquanto `//*` indica que estamos fazendo uma busca em qualquer tag.

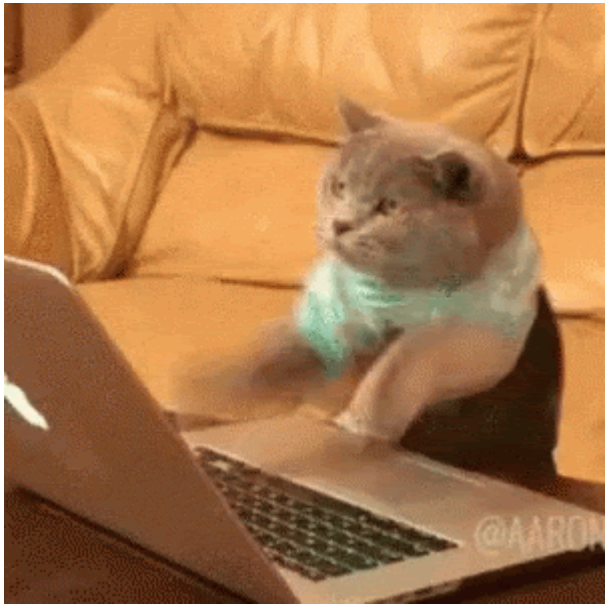
Exemplos

Exemplo 06: HTML simples

Exemplo 07: Chance de gol

Exemplo 08: Jurisprudência Anatel

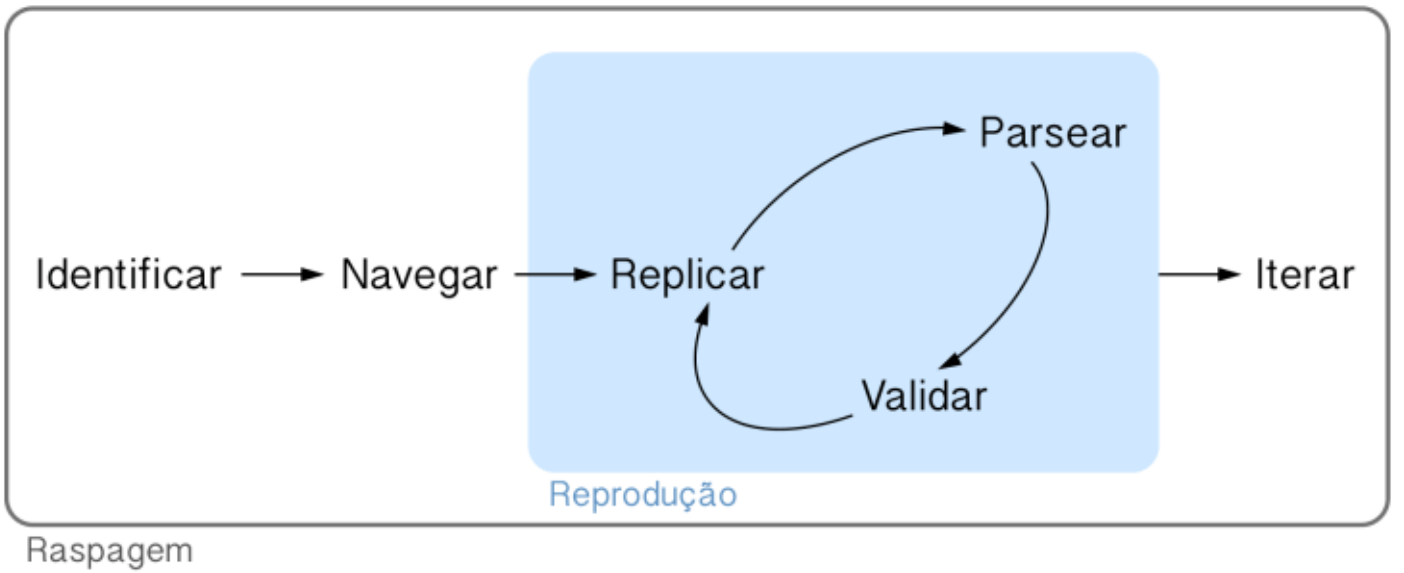
Vamos ao R!



Iteração

O fluxo do web scraping

- Sempre que fazemos um web scraper é bom seguir um fluxo definido
- Por enquanto já foram apresentados elementos da maior parte do passo-a-passo, mas nada foi dito sobre a iteração



Por que iterar?

- Dificilmente queremos fazer uma tarefa de web scraping uma vez só (senão bastaria baixar a página uma vez e raspá-la)
- Podemos querer baixar muitas páginas de uma vez ou uma página a cada certo tempo
- Iteração, tratamento de erros e automatização passam a ser relevantes
    - O pacote `purrr` nos ajudará a iterar
    - O pacote `purrr` retornará para tratar qualquer erro que possa aparecer
    - Falaremos de Github Actions na última aula
- Se você estiver interesse em aprender mais, veja nosso curso de Deploy!

Elementos comuns

- **Rodar em paralelo**. Quanto mais rápido, melhor!
- **Rodar com tratamento de erros**. Coisas dão errado no web scraping.
- **Utilizar barras de progresso**. Remédio para ansiedade.

Introdução

- Iteração é um padrão de programação extremamente comum que pode ser altamente abreviado

```
nums <- 1:10
resp <- c()
for (i in seq_along(nums)) {
  resp <- c(resp, nums[i] + 1)
}
resp
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
library(purrr)
map_dbl(nums, ~.x + 1)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

A função map

- A função `map()` recebe um vetor ou uma lista de entrada e aplica uma função em cada elemento do mesmo
- Podemos especificar o formato da saída com a família de funções `map_***()`
- A função pode ser declarada externamente, internamente ou através de um *lambda*

```
soma_um <- function(x) {
  x + 1
}
map(nums, soma_um)
map(nums, function(x) x + 1)
map(nums, ~.x + 1)
```

Utilidade do map

- Se tivermos uma lista de URLs, podemos iterar facilmente em todos sem abrir mão da sintaxe maravilhosa do Tidyverse

```
urls <- c(
  "https://en.wikipedia.org/wiki/R_language",
  "https://en.wikipedia.org/wiki/Python_(programming_language)
)
urls |>
  map(read_html) |>
  map(xml_find_first, "//h1") |>
  map_chr(xml_text)
```

```
## [1] "R (programming language)"      "Python (programming language)
```

Tratando problemas

- Ao repetir uma tarefa múltiplas vezes, não podemos garantir que toda execução funcione
- O R já possui o `try()` e o `tryCatch()`, mas o `purrr` facilita ainda mais o trabalho

```
read_html("https://errado.que")
```

```
## Error in open.connection(x, "rb"): Could not resolve host: errado.
```

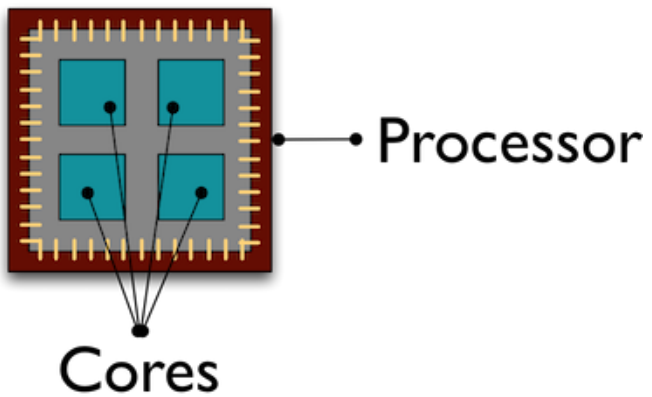
```
maybe_read_html <- possibly(read_html, NULL)
maybe_read_html("https://errado.que")
```

```
## NULL
```

Paralelismo

O que isso significa?

- Antigamente, computadores eram capazes de executar apenas uma sequência de comandos por vez
- Avanços tecnológicos permitiram que o processador fosse capaz de fazer "malabarismo" com diversos processos
- Paralelismo (ou multiprocessamento) chegou apenas com os primeiros *dual-core*



Em mais detalhes

- A unidade de processamento central pode ter mais de um **núcleo** (*multicore*)
- Um **processo** é composto por uma sequência de comandos ou tarefas
- Cada núcleo consegue executar apenas um **comando** por vez
- Os comandos de um processo podem ser interrompidos para que sejam executados os de outro (*multitasking*)
- O computador pode executar várias tarefas simultaneamente escalonando os comandos para seus diferentes núcleos (*multithreading*)
- Muitos computadores possuem **núcleos virtuais**, permitindo dois comandos por vez em cada núcleo (*hyperthreading*)

Exemplo mínimo

O pacote `parallel` já vem instalado junto com o R e consegue rodar comandos paralelamente tanto no Windows quanto em outros sistemas. Por padrão, ele quebra a tarefa em 2.

```
library(parallel)
library(tictoc)

tic()
res <- map(1:4, function(x) Sys.sleep(1))
toc()
```

```
## 4.06 sec elapsed
```

```
tic()
res <- mclapply(1:4, function(x) Sys.sleep(1))
toc()
```

```
## 4.06 sec elapsed
```

Futuros

- O pacote `{future}` expande o pacote `{parallel}`, permitindo o descolamento de tarefas da sessão principal
    - Ele pode operar em 2 níveis: *multicore* e *multissession*
- Em cima do `{future}`, foi construído o `{furrr}` com o objetivo de emular a sintaxe do `{purrr}` para processamento paralelo
- Diferentemente do `{parallel}`, o `{future}` é capaz de descobrir sozinho o número de núcleos virtuais do computador

```
library(future)
availableCores()
```

```
## system
##      8
```

Barras de progresso

- Com o pacote {progressr} (recente!), é possível adicionar barras de progresso à suas chamadas, mesmo se a chamada for em paralelo.

```
# coloca o script no contexto
progressr::with_progress({

  # cria a barra de progresso
  p <- progressr::progressor(4)

  purrr::walk(1:4, ~{
    # dá o passo
    p()
    Sys.sleep(1)
  })

})
```

Como faz?

Vamos estabelecer um plano de execução paralela com a função `plan()`. Entender a diferença entre todos os planos disponíveis.

```
plan(multisession)
```

- `sequential`: não executa em paralelo, útil para testes
- `multicore`: mais eficiente, não funciona no Windows nem dentro do RStudio
- `multisession`: abre novas sessões do R, mais pesado para o computador

Como faz?

Agora vamos criar uma função que retorna o primeiro parágrafo de uma página da Wikipédia dado o fim de seu URL (como "/wiki/R_language"). Dicas: textos são denotados pela *tag* <p> em HTML; pule o elemento de classe "mw-empty-elt".

```
download_wiki <- function(url) {
  url |>
    paste0("https://en.wikipedia.org", .) |>
    read_html() |>
    xml_find_first("//p[not(@class='mw-empty-elt')]") |>
    xml_text()
}
```

Como faz?

Executar a função anterior em paralelo para todas as páginas baixadas no exercício de iteração. Dicas: utilize `future_map()` do pacote `furrr`; não se esqueça do `possibly()`"!

```
library(furrr)
prgs <- "https://en.wikipedia.org/wiki/R_language" |>
  read_html() |>
  xml_find_all("//table[@class='infobox vevent']//a") |>
  xml_attr("href") |>
  future_map(possibly(download_wiki, ""))
prgs[[3]]
```

```
## [1] ""
```

Exemplos

Exemplo 08: Jurisprudência Anatel (continuação)

Exemplo 09: Wiki

Exemplo 10: Jobs

Exemplo 11: TJSP (vídeo gravado)

Exemplo 12: DEJT (extra)

Vamos ao R!