



Em geral, o que pode causar **lentidão** em um programa?



Na maioria dos casos tem relação com o
I/O ou Input and Output

Enquanto a memória é acessada na ordem de
nanosegundos os dados em disco ou na rede
levam de milisegundos a até mesmo segundos

L1 cache latency:	3 cycles
L2 cache latency:	14 cycles
RAM latency:	250 cycles
Disk latency:	41 000 000 cycles
Network:	240 000 000 cycles

Durante um ciclo, que é basicamente um **pulso elétrico**, uma tarefa específica é executada como obter dados da memória, fazer um cálculo ou armazenar dados em um registrador e cada instrução leva uma determinada quantidade de ciclos para ser executada



Como um programa é executado?

O **sistema operacional** é responsável pelo gerenciamento de processos e memória além de interagir com dispositivos de I/O como o disco, a rede e outros periféricos

User Programs

User Interface

System Calls

Program
Control

I/O

File
System

Comms

Error
Manage
ment

Resource

Auditing

Security

Hardware

O processador é o responsável, de fato, por **executar instruções na máquina**. Imagine uma API de baixo nível onde temos operações como alocação de valores em memória, soma, comparação, entre outras.

Todo código-fonte que você escreve em alto nível
é convertido para essas instruções em algum
momento, seja na interpretação ou na compilação
do programa

High-level
language
program in C

```
void swap(int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Compiler

swap:

muli	\$2, \$5, 4
add	\$2, \$4, \$2
lw	\$15, 0(\$2)
lw	\$16, 4(\$2)
sw	\$16, 0(\$2)
sw	\$15, 4(\$2)
jr	\$31

Assembler

```
000000001010001000000000011000
000000001000111000110000100001
10001100011000100000000000000000
10001100111100100000000000000000
10101100111100100000000000000000
10101100011000100000000000000000
00000011111000000000000000000000
```

Binary
machine
language
program
(for MIPS)

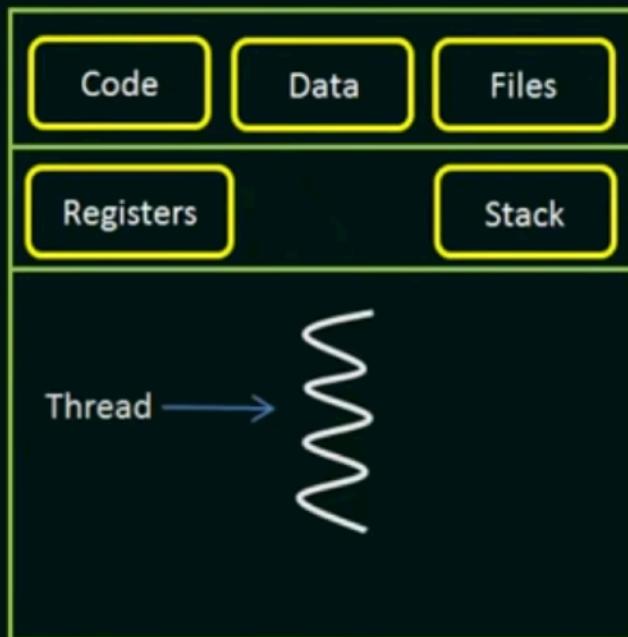
Assembly
language
Program
(for MIPS)

Um processo é uma instância de um programa em execução no sistema operacional que utiliza recursos como espaço de memória, variáveis de ambiente, processos filhos, descritores de arquivo, alarmes, tratadores de sinalização

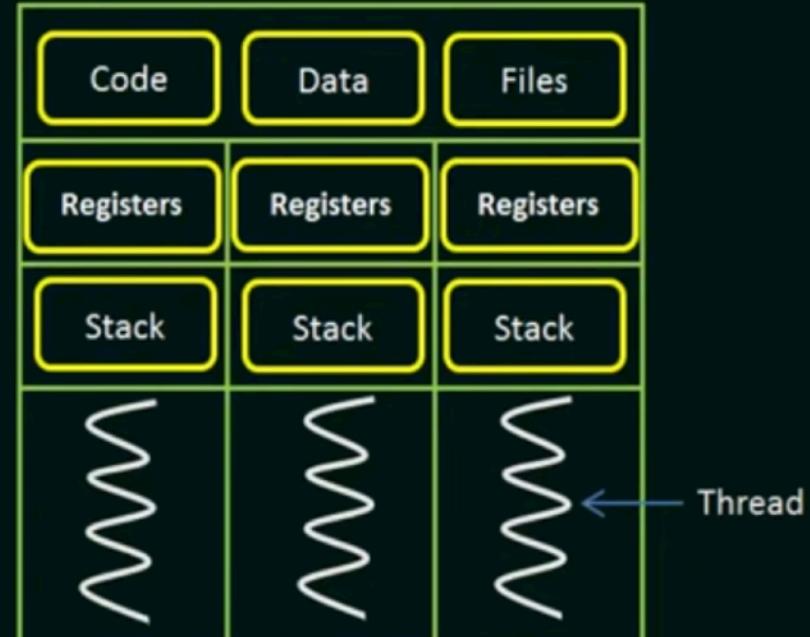
Todo processo tem pelo menos uma thread, ela é a unidade básica de execução, é o que de fato será executado, ela tem um identificador, contador, conjunto de registradores e pilha

Toda thread **compartilha recursos com outras threads** do mesmo processo como a memória, descritores de arquivo e tratadores de sinalização

Single-Thread



Multi-Thread



A comunicação entre processos geralmente é mais lenta quando feita por IPC, ou Inter-Process Communication enquanto as threads compartilham a memória

Apesar de existirem formas de compartilhar memória entre diferentes processos

Um processo só consegue **utilizar de mais de um núcleo de CPU** se tiver mais de uma thread



O que um programa precisa fazer para executar operações que envolvem **I/O**?

POSIX System Call

`ssize_t read(int fd, void *buf, size_t count);`

1. file descriptor of the file
2. buffer where the read data is to be stored
3. number of bytes to be read from the file

Windows System Call

```
ReadFile(HANDLE hFile, LPVOID lpBuffer,  
DWORD nNumberOfBytesToRead, LPDWORD  
lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);
```

1. handle of file to read
2. pointer to buffer that receives data
3. number of bytes to read
4. pointer to number of bytes read
5. pointer to structure for data

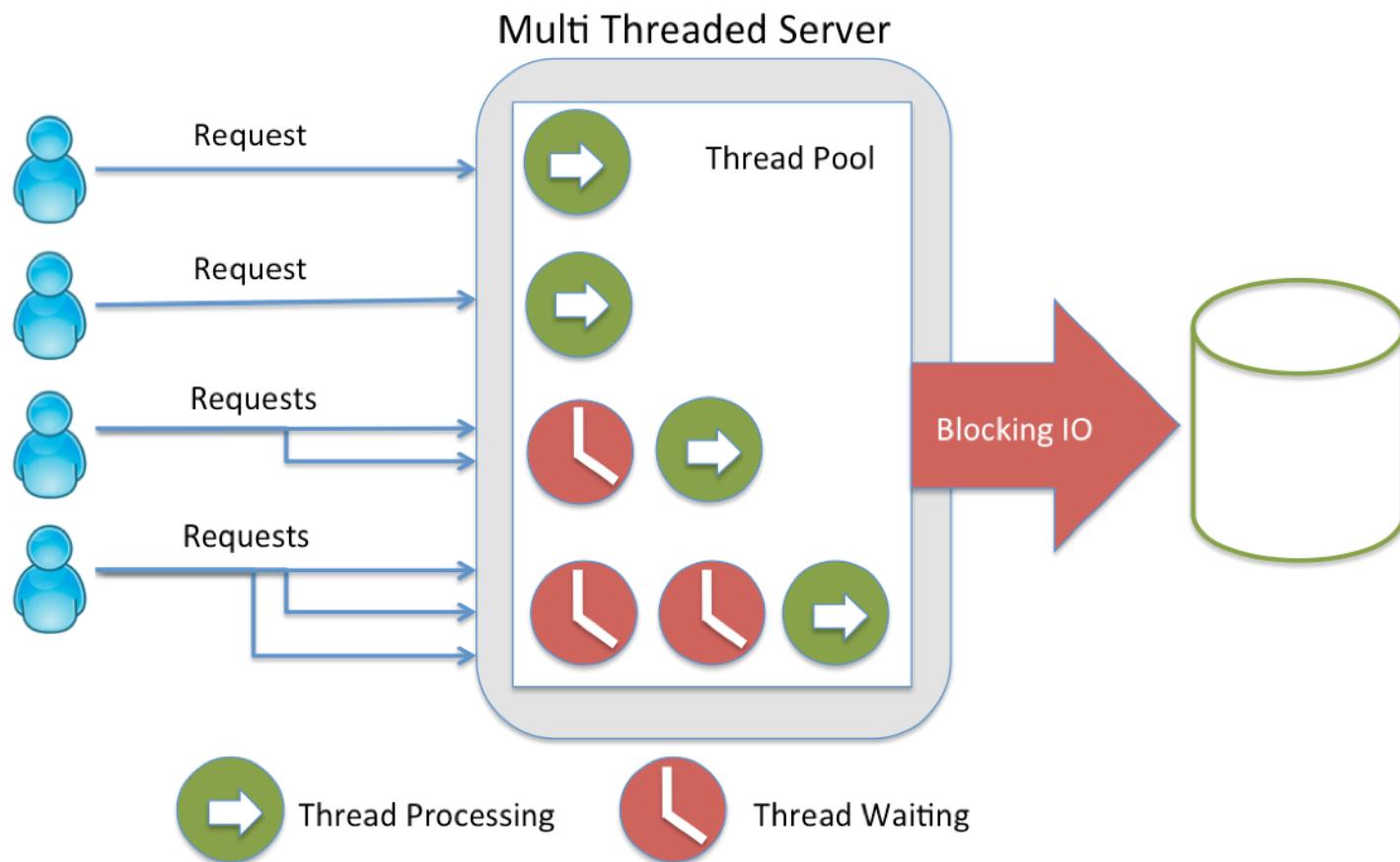
Exemplo utilizando a linguagem Java

```
public class FileReaderServlet extends HttpServlet {  
  
    @Override  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    File f = new File("file.mp4");  
    FileInputStream is = new FileInputStream(f);  
    byte[] buffer = new byte[(int) f.length()];  
    is.read(buffer, 0, (int) f.length());  
    response.setHeader("Content-Disposition", "attachment;filename=file.mp4");  
    response.setContentLength((int) f.length());  
    OutputStream os = response.getOutputStream();  
    os.write(buffer, 0, f.length());  
    is.close();  
    os.close();  
}  
}
```



A leitura de arquivo é **bloqueante**?

No Java e em outras plataformas, **cada requisição**
é tratada em uma thread diferente e por conta
disso ainda que a leitura do arquivo seja
bloqueante, até um certo limite, conseguem
ser atendidas em paralelo

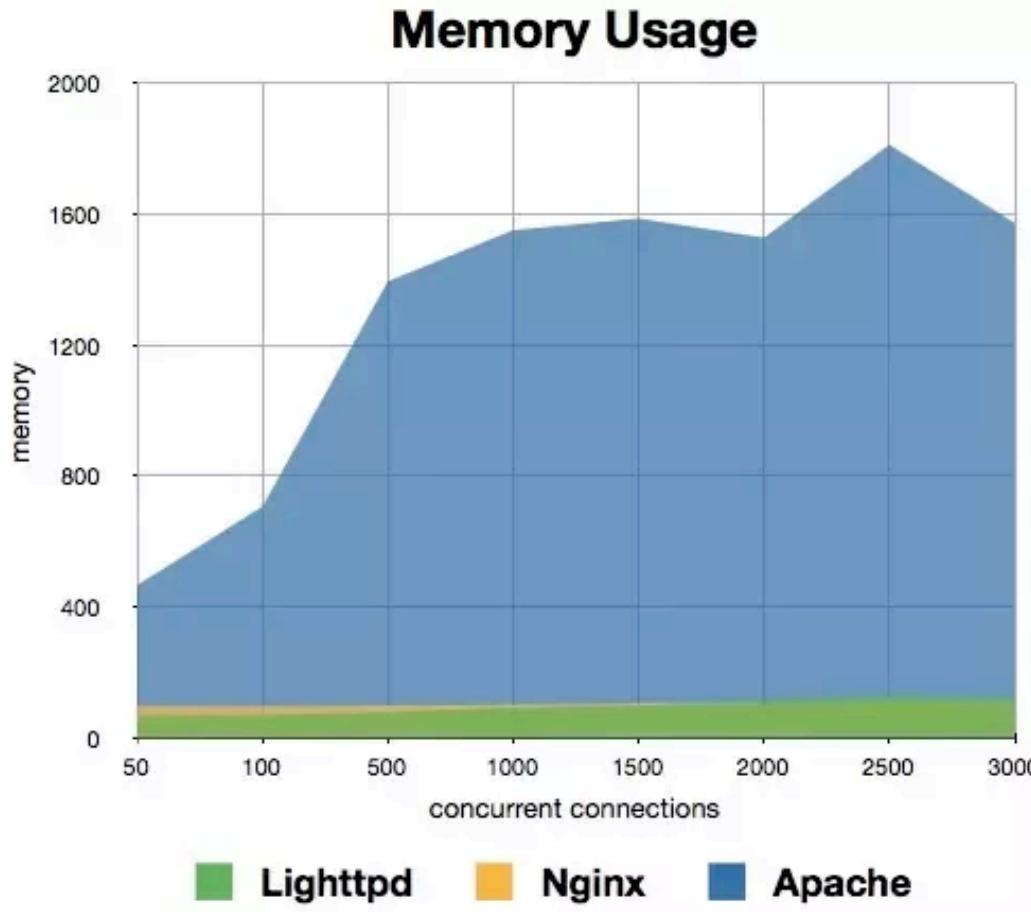




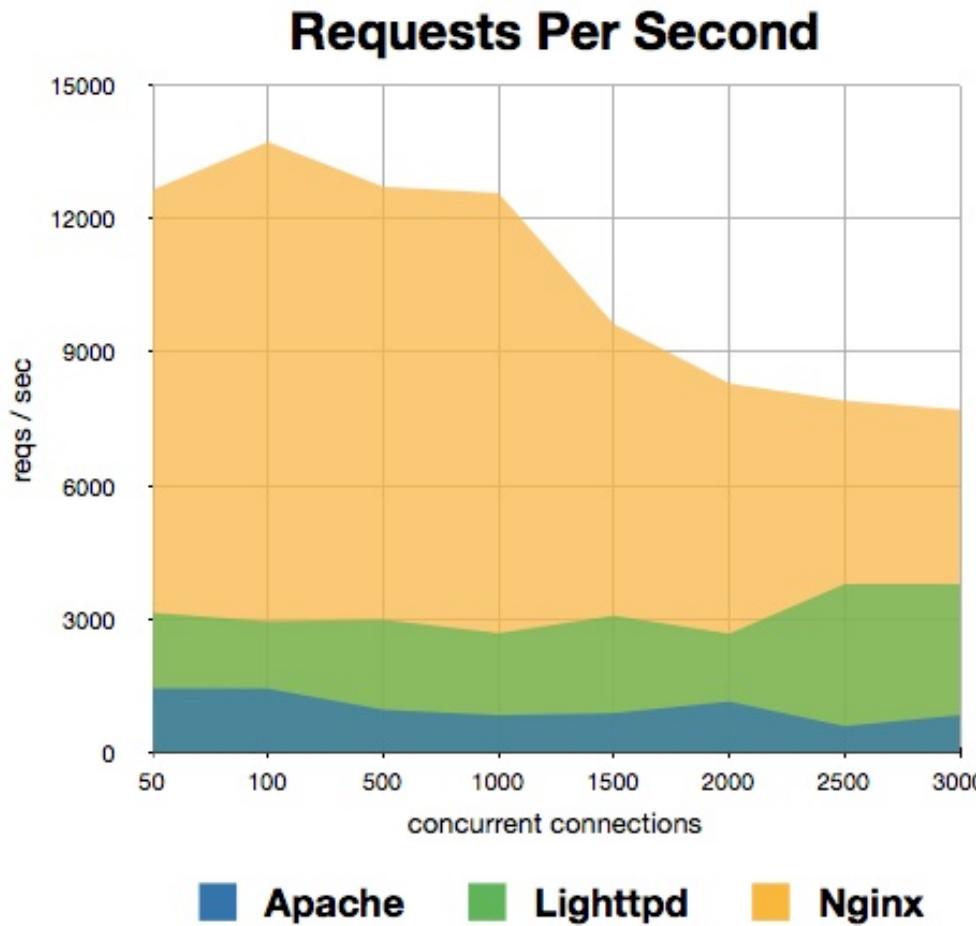
As threads passam a maior parte do tempo aguardando, sem fazer nada



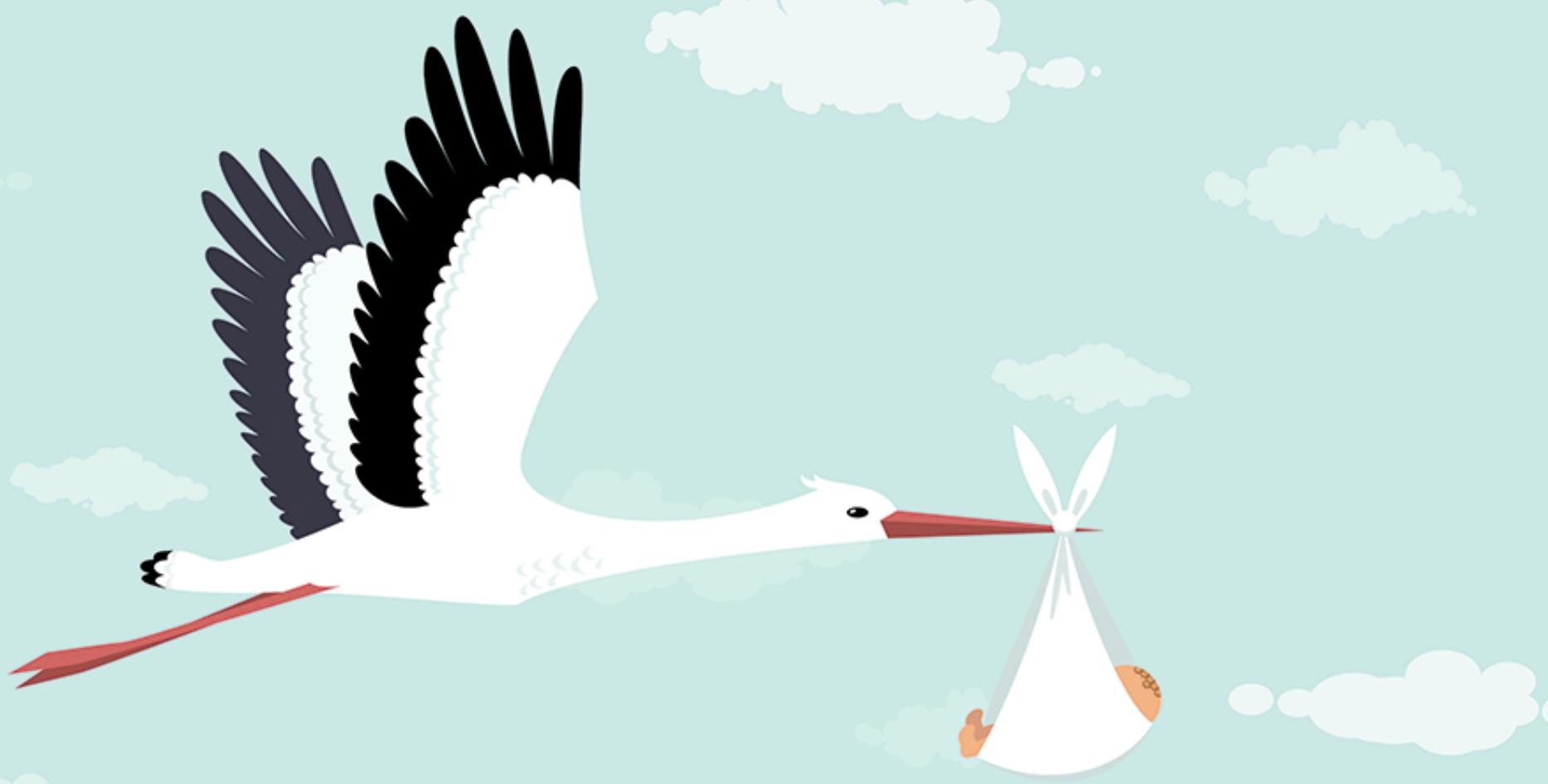
Existe desperdício de recursos com o
chaveamento de contexto



Conforme as requisições aumentam, o consumo de memória sobe muito no Apache, que utiliza uma thread por requisição ao contrário do Nginx



Com isso, a capacidade de atender as requisições
reduz frente ao aumento da concorrência



O Node.js **não** nasceu com a intenção
de levar o JavaScript para o servidor

O Node.js não foi a primeira iniciativa a levar JavaScript para o servidor, a Netscape já havia lançado o **LiveWire** mais de uma década antes do lançamento do Node.js

Na época a linguagem **ainda não era tão madura** e a Netscape passou por uma sucessão de aquisições, o projeto não foi pra frente e acabou sendo descontinuado

```
<HTML>
<HEAD>
  <TITLE>My awesome web app</TITLE>
</HEAD>
<BODY>
  <H1>
    <SERVER>
      /* This tag and its content will be processed on the server side,
      and replaced by whatever is passed to `write()` before being sent to the client. */
      if(client.firstname != null) {
        write("Hello " + client.firstname + " !")
      }
      else {
        write("What is your name?")
      }
    </SERVER>
  </H1>

  <FORM METHOD="post" ACTION="app.html">
    <P>
      <LABEL FOR="firstname">Your name</LABEL>
      <INPUT TYPE="text" NAME="firstname"/>
    </P>

    <P>
      <INPUT TYPE="submit" VALUE="Send"/>
    </P>
  </FORM>
</BODY>
</HTML>
```



Tudo começou com Ryan Dahl

flickr



"I/O needs to be done differently"



Como fazer I/O de forma **mais eficiente**?

É possível consumir recursos como files, sockets, pipes de forma não bloqueante utilizando a system call fcntl() com a flag O_NONBLOCK, com isso o retorno é imediato e tem o valor -1

Enquanto a leitura não for finalizada, qualquer operação de read() falha retornando EAGAIN



Como saber o momento em que certo
para iniciar a **leitura**?

```
resources = [fileA, socketB, pipeC, socketD, ...];
while(!resources.isEmpty()) {
    for(i = 0; i < resources.length; i++) {
        resource = resources[i];
        //try to read
        var data = resource.read();
        if(data === NO_DATA_AVAILABLE)
            //there is no data to read at the moment
            continue;
        if(data === RESOURCE_CLOSED)
            //the resource was closed, remove it from the list
            resources.remove(i);
        else
            //some data was received, process it
            consumeData(data);
    }
}
```

Esse tipo de **algoritmo de polling** normalmente é **ineficiente** já que consomem CPU e na maior parte das vezes os dados ainda não estão disponíveis



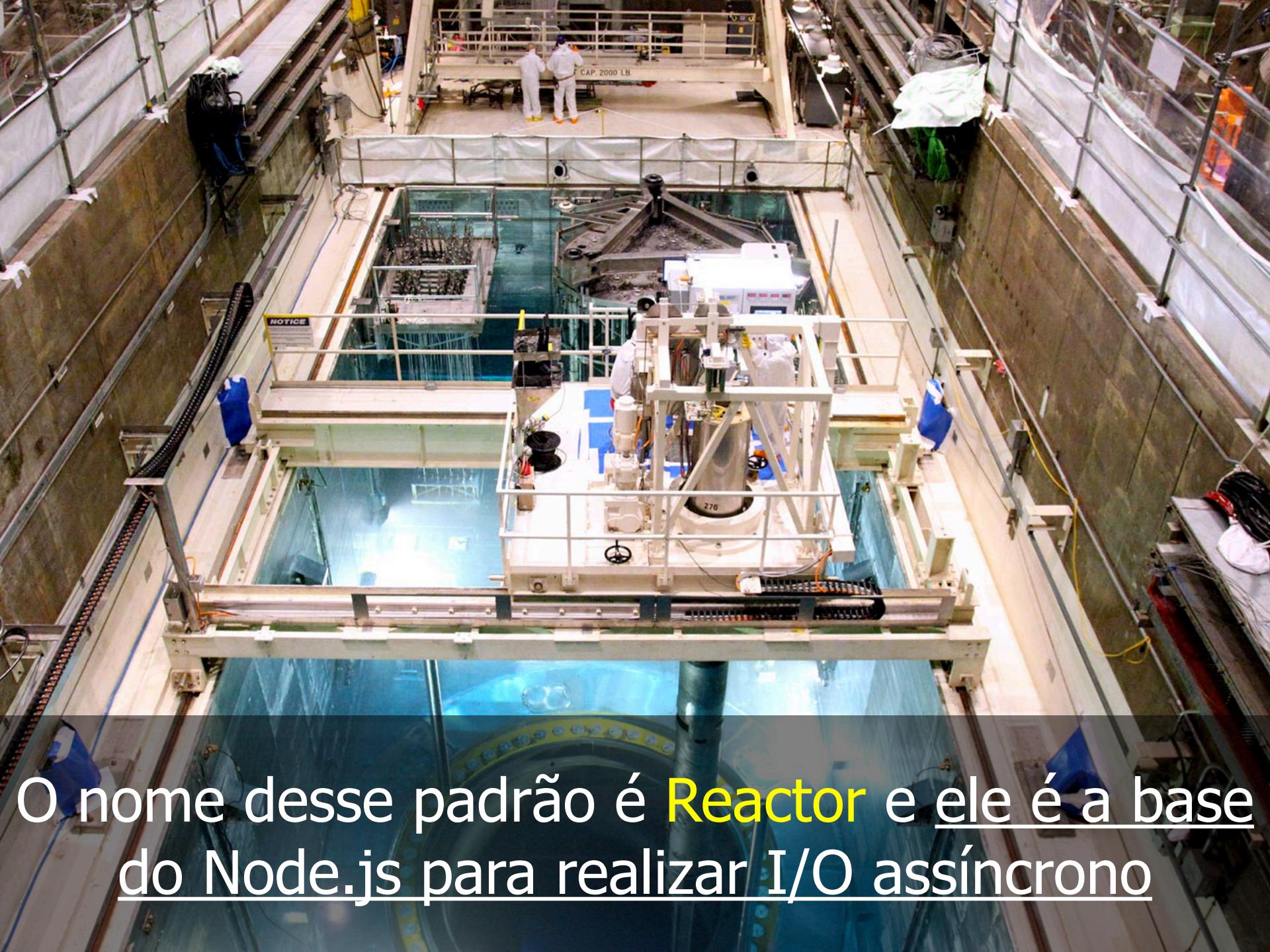
Tem alguma forma de ser **notificado**?

Esse problema é resolvido por meio do **Event Demultiplexer**, uma espécie de implementação de publish/subscribe que notifica quando o recurso desejado estiver disponível

Cada sistema operacional tem a sua própria implementação de **Event Demultiplexer**, epoll no Linux, kqueue no OSX, IOCP no Windows, event ports no Solaris

Uma proposta de implementação envolve a utilização de um event loop que bloqueia enquanto aguarda por recursos e invoca um handler quando o recurso estiver disponível

```
socketA, pipeB;  
watchedList.add(socketA, FOR_READ);  
watchedList.add(pipeB, FOR_READ);  
// block until event is ready to read  
while(events = demultiplexer.watch(watchedList)) {  
    //event loop  
    foreach(event in events) {  
        //This read will never block and will always return data  
        data = event.resource.read();  
        if(data === RESOURCE_CLOSED)  
            //the resource was closed, remove it from the watched list  
            demultiplexer.unwatch(event.resource);  
        else  
            //some actual data was received, process it  
            consumeData(data);  
    }  
}
```



O nome desse padrão é **Reactor** e ele é a base
do Node.js para realizar I/O assíncrono



Como é a arquitetura interna do Node.js?

O Node.js é composto pela junção do
interpretador V8, do Google, com a biblioteca
libuv e por um conjunto de módulos

Node.js Code Library (JavaScript)

http

fs

stream

buffer

net

crypto

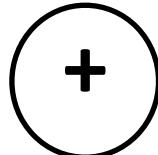
zlib

cluster

dns

...

Node.js Bindings (C++)



libuv

Nas primeiras versões do Node.js, existia suporte somente para Linux e OSX e a implementação do Event Loop e o suporte a I/O era feito na libev, criada por Marc Lehmann



Search or jump to...

/

Pull requests Issues Marketplace Explore

This repository has been archived by

joyent / libuv

Public archive

Code

Issues 52

Pull requests 12

Actions

Projects

Wiki

unix: remove libev #485

Closed

bnoordhuis opened this issue on Jul 2, 2012 · 26 comments



bnoordhuis commented on Jul 2, 2012

This issue tracks the removal of libev in v0.9 / master.



ghost assigned bnoordhuis on Jul 2, 2012

A **libuv** é uma biblioteca multi-plataforma, feita em C e criada pela equipe do Node.js e é responsável pela realização de **I/O assíncrono**, fornecendo a implementação do **event loop**, que é single threaded, e do **thread pool**, juntamente com o suporte a rede, resolução de DNS, descritores de arquivos, processos, sinalização entre outras

libuv

Network I/O

TCP

UDP

TTY

Pipe

...

File
I/O

DNS
Ops

User
code

`uv__io_t`

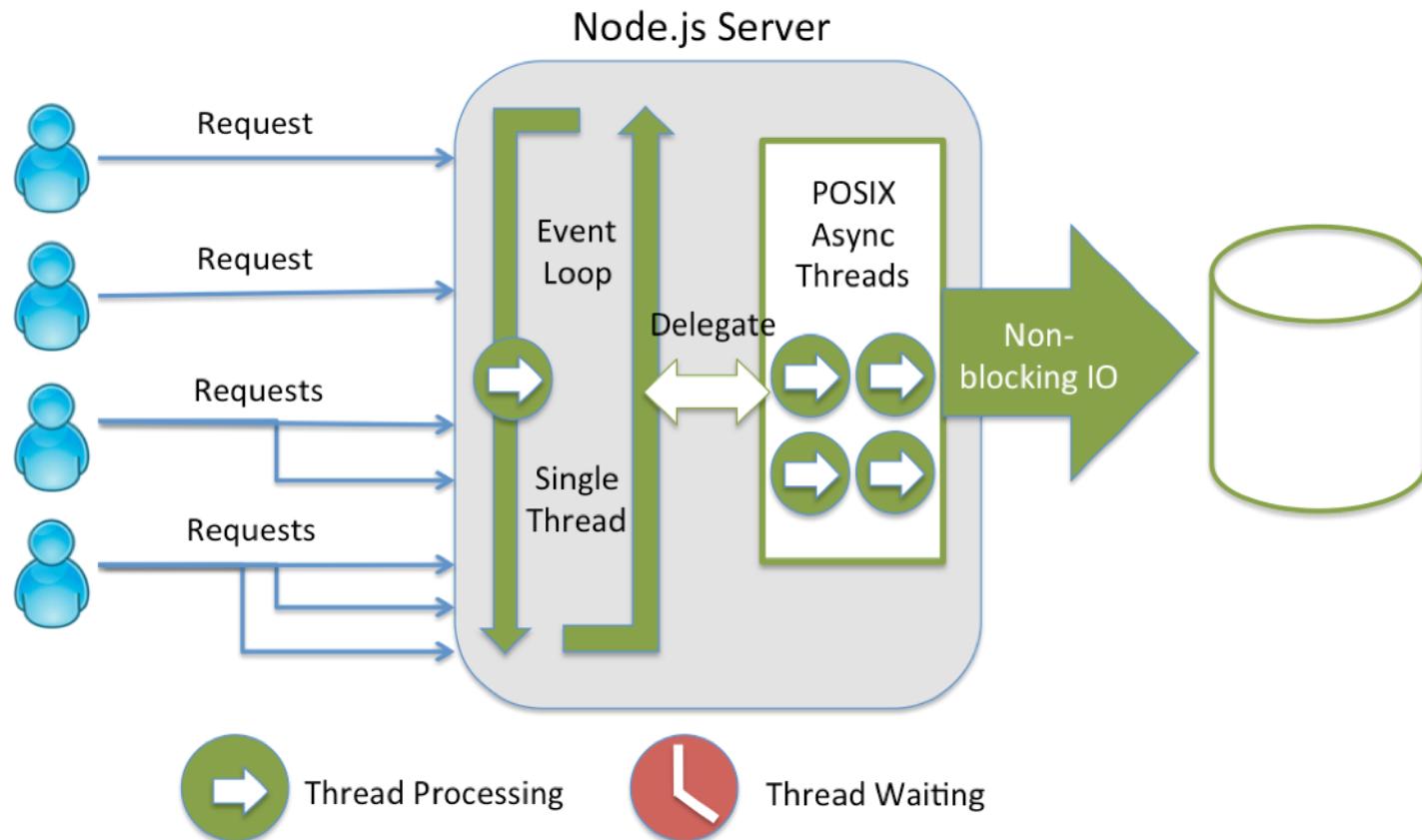
epoll

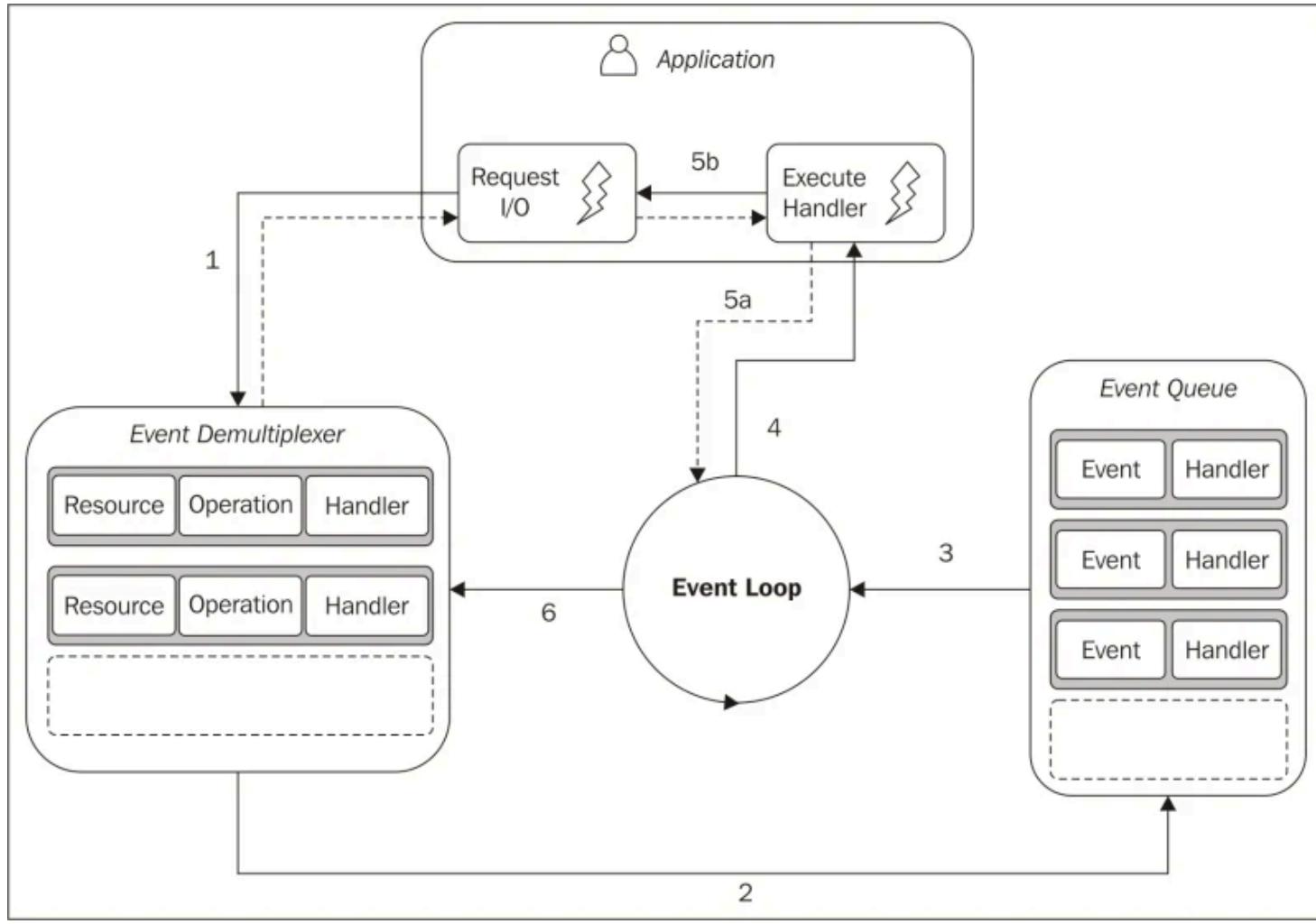
kqueue

event ports

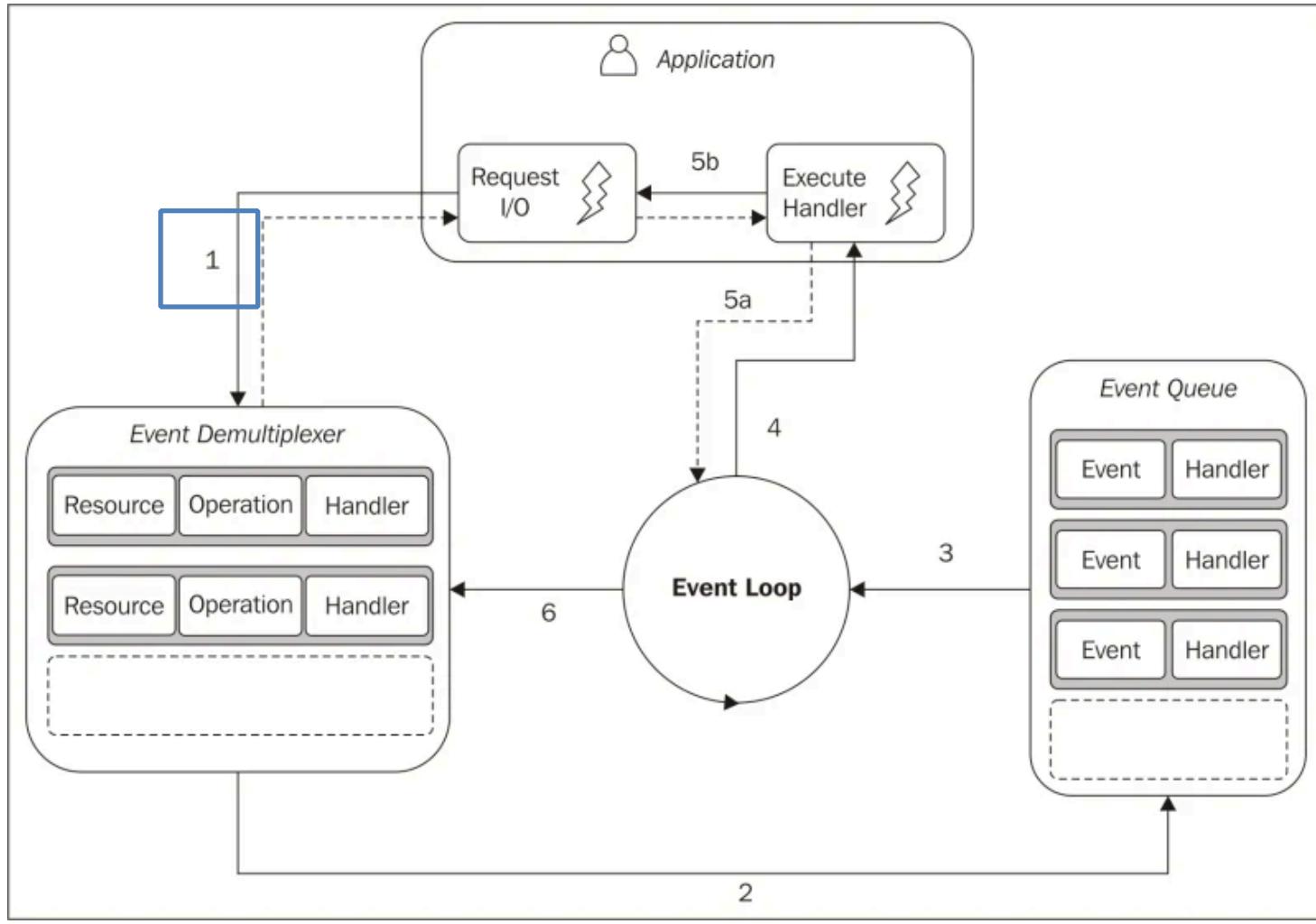
IOCP

Thread Pool

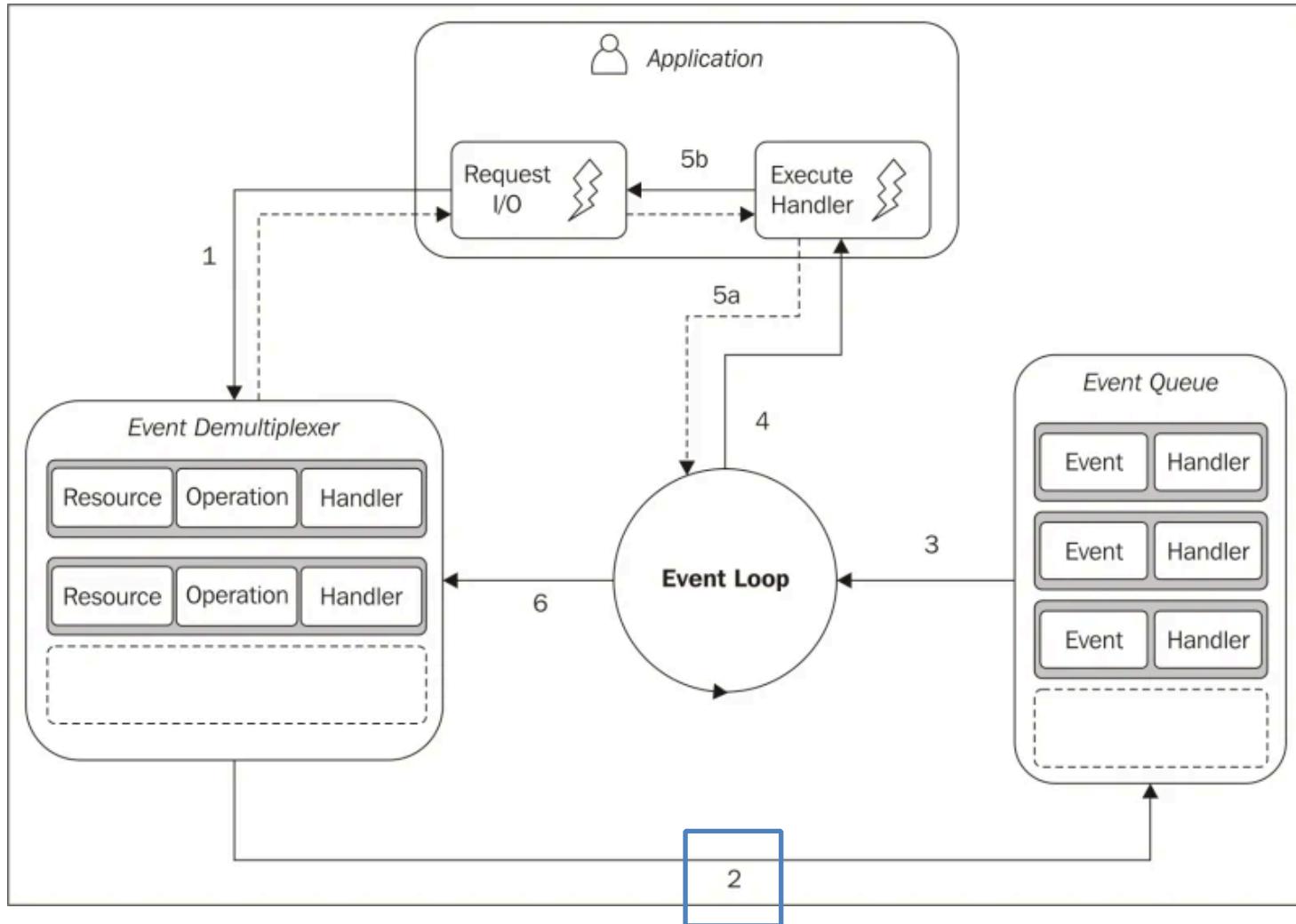




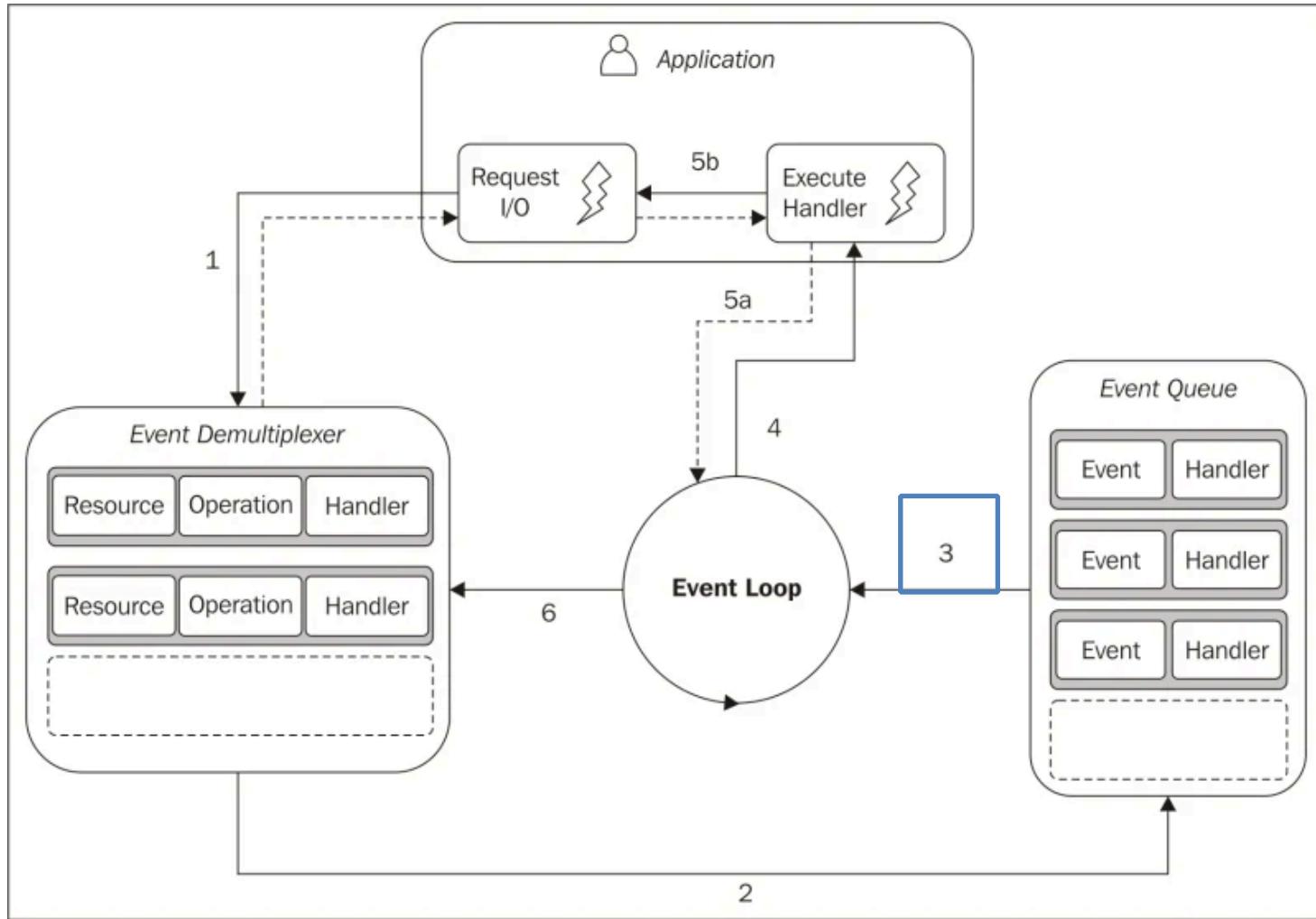
Funcionamento da aplicação no **Event Loop** executando em conjunto com o Event Demultiplexer e o Event Queue



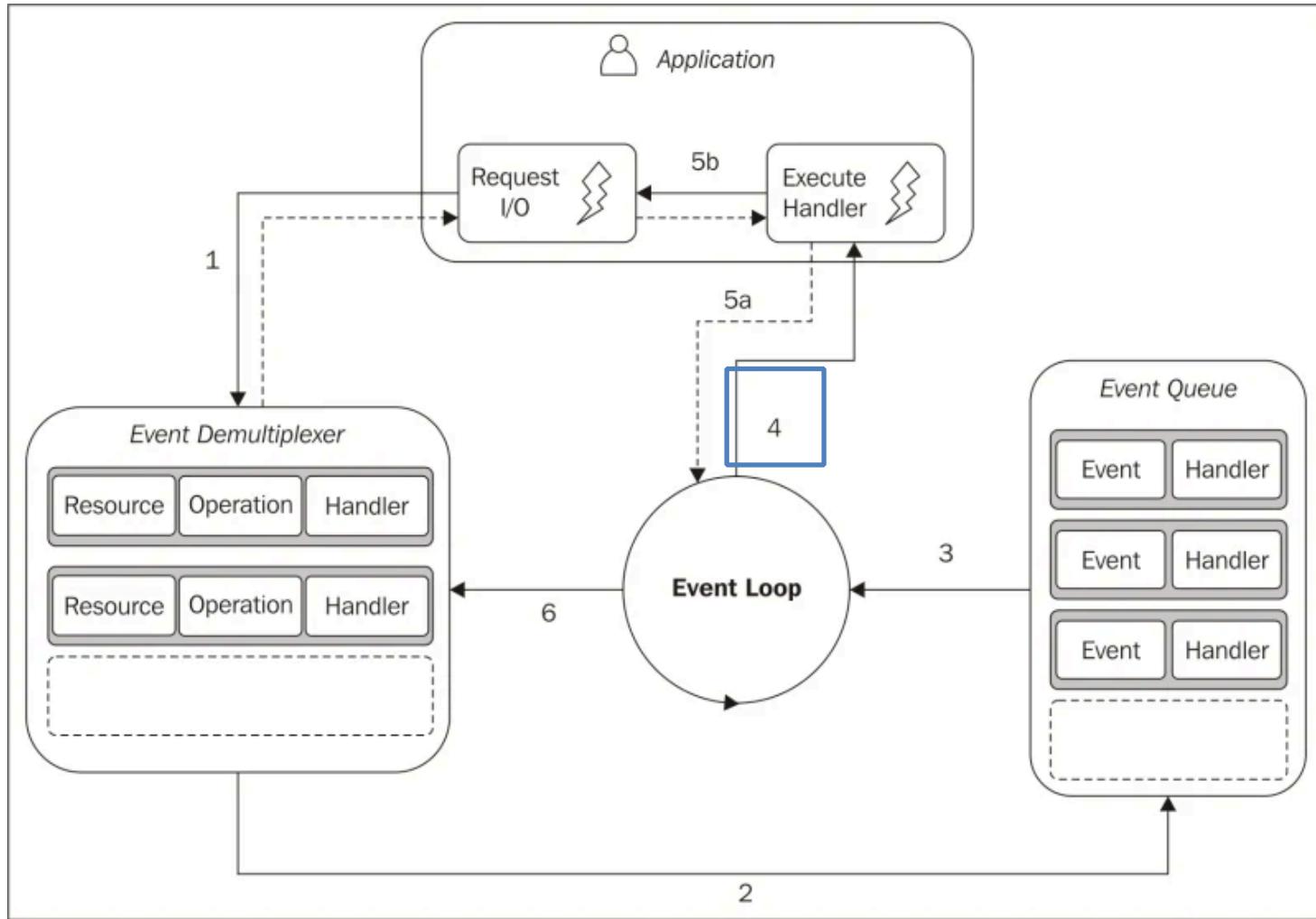
Quando a aplicação precisa realizar um I/O (ler um arquivo, acessar a rede, gravar no banco de dados), a operação é adicionada ao **Event Demultiplexer** e um handler é criado para quando ela finalizar, não bloqueando o event loop



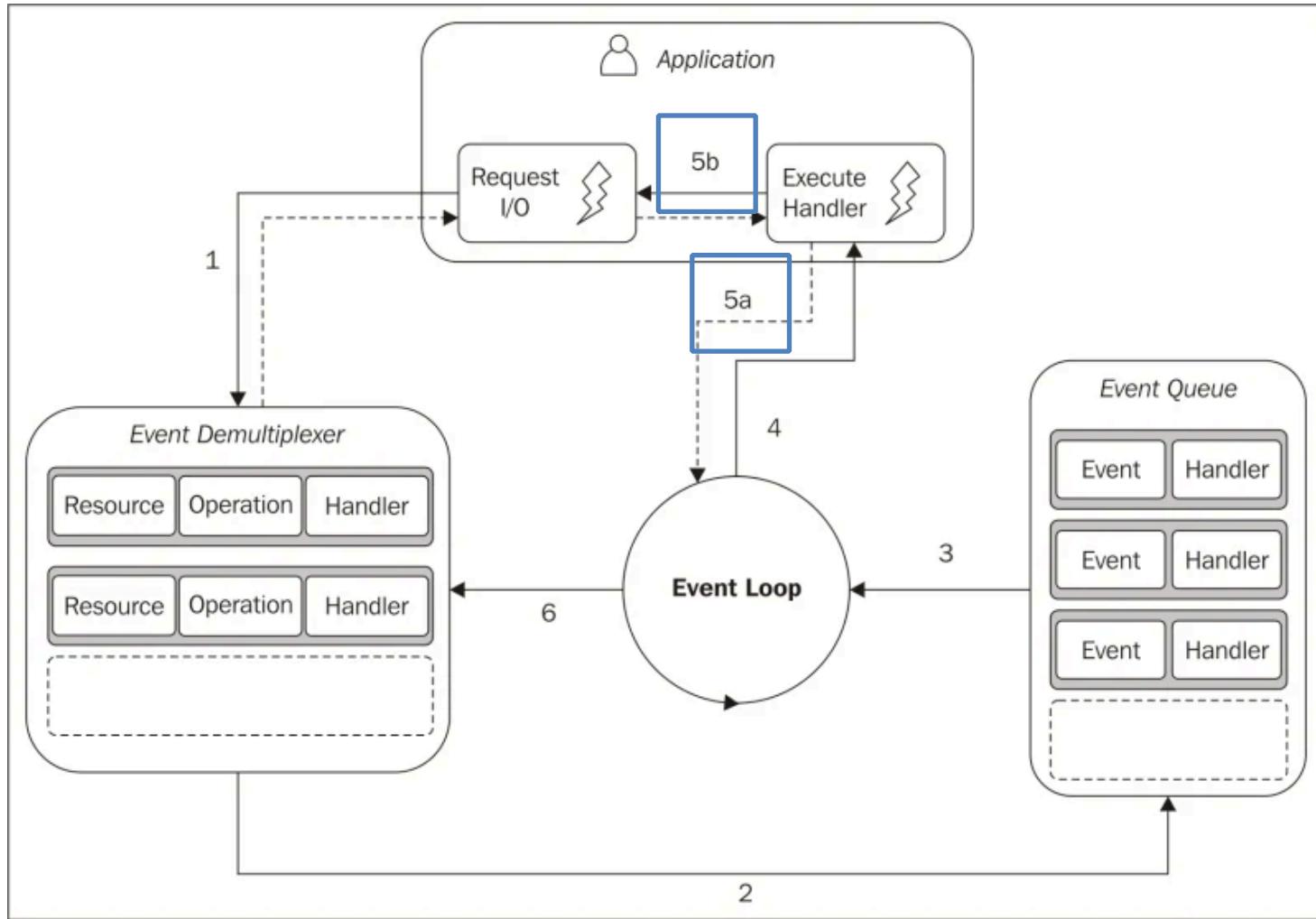
Quando o I/O finalizar a função de callback do handler é adicionada na **Event Queue** e fica aguardando o Event Loop estar disponível para executá-la



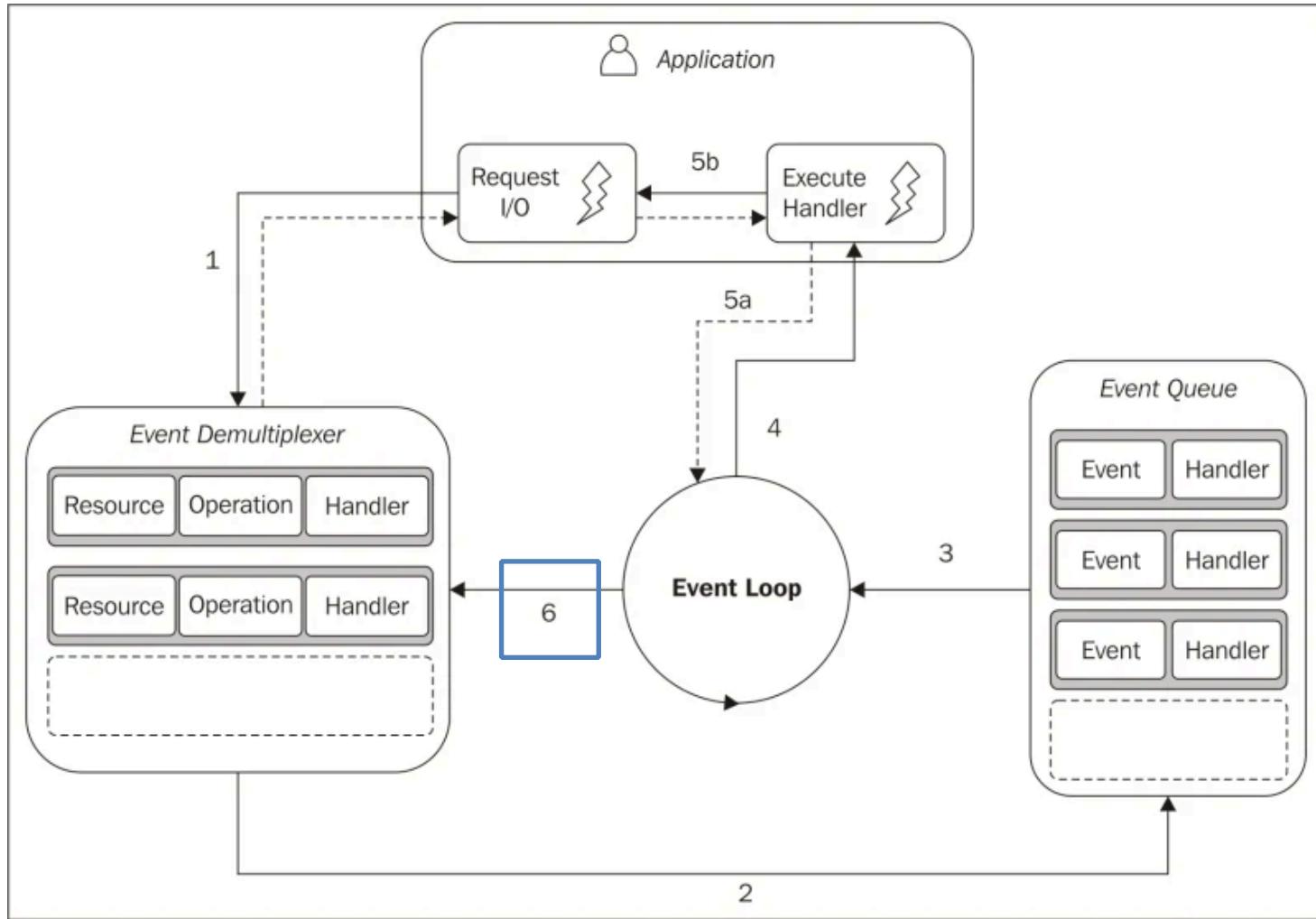
O event loop consome a Event Queue conforme a ordem de término dos eventos sendo uma fila do tipo FIFO



Ao retornar ao Event Loop o callback do handler é invocado



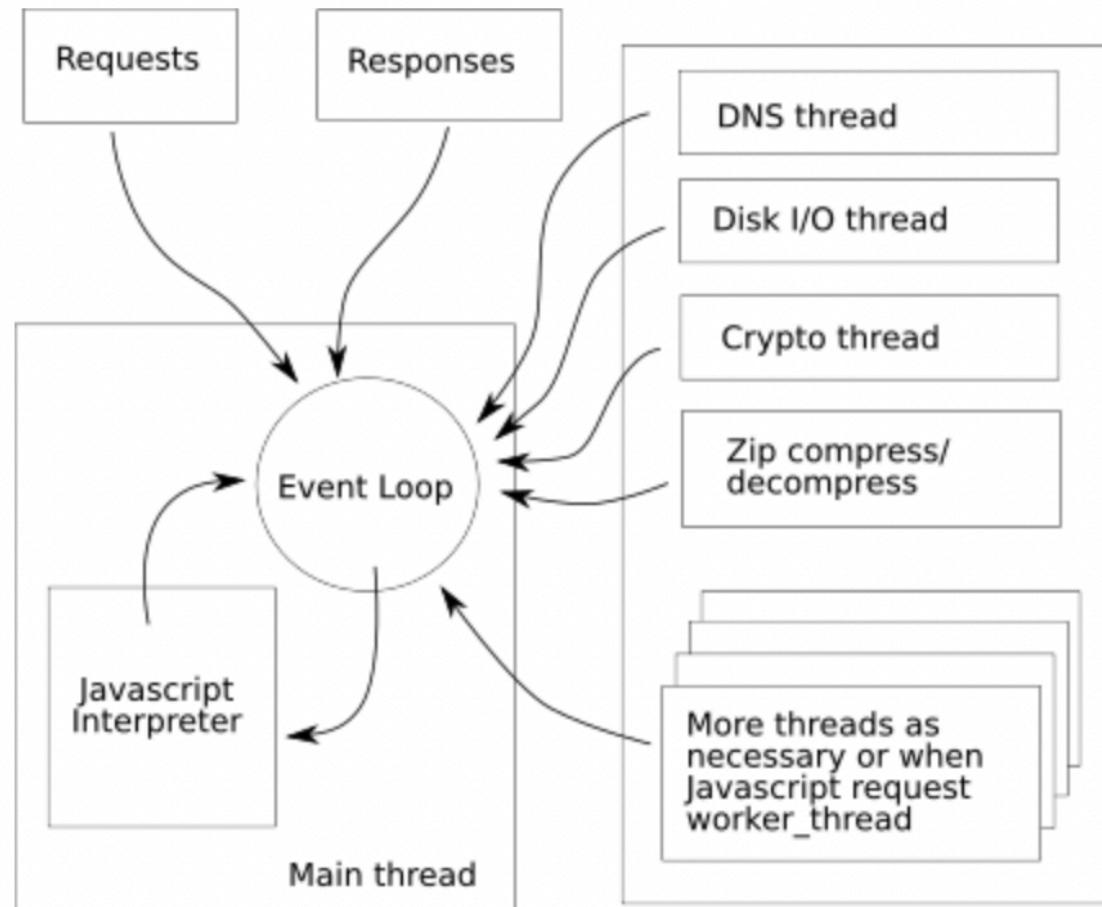
Ao finalizar, o controle é retornado para o Event Loop ou é possível que novas operações de I/O sejam executadas



Caso não exista mais nada para executar, o Event Loop bloqueia e fica aguardando o Event Demultiplexer disparar outros eventos, caso não exista eventos pendentes, a aplicação é encerrada



Porque a libuv tem um **thread pool**?



Ainda que o Event Demultiplexer seja utilizado para notificar quando o recurso está disponível a leitura ainda é bloqueante em operações como acesso ao sistema de arquivos, DNS lookup, crypto, zlib menos sockets (net/dgram/http/tls/ https/child_process/pipes/stdin/stdout/stderr)

UV_THREADPOOL_SIZE = 4 (default)



Se o event loop é single threaded, como o Node.js consegue **escalar**?

Existem duas formas de escalar threads,
preemptiva e não preemptiva, na primeira o
escalonador que interrompe a execução e faz o
chaveamento enquanto na segunda a execução
vai até o fim

Um escalonador não preemptivo conta com a boa vontade do programa para interromper seu próprio fluxo de execução e liberar recursos para que outros processos executem



Será que o JavaScript é rápido?



O V8 é um **interpretador de JavaScript de alta performance** desenvolvido pelo Google e lançado no Chrome em 2008. Ele tem o código aberto e foi desenvolvido em C++ e é utilizado em vários projetos importantes como o Node.js

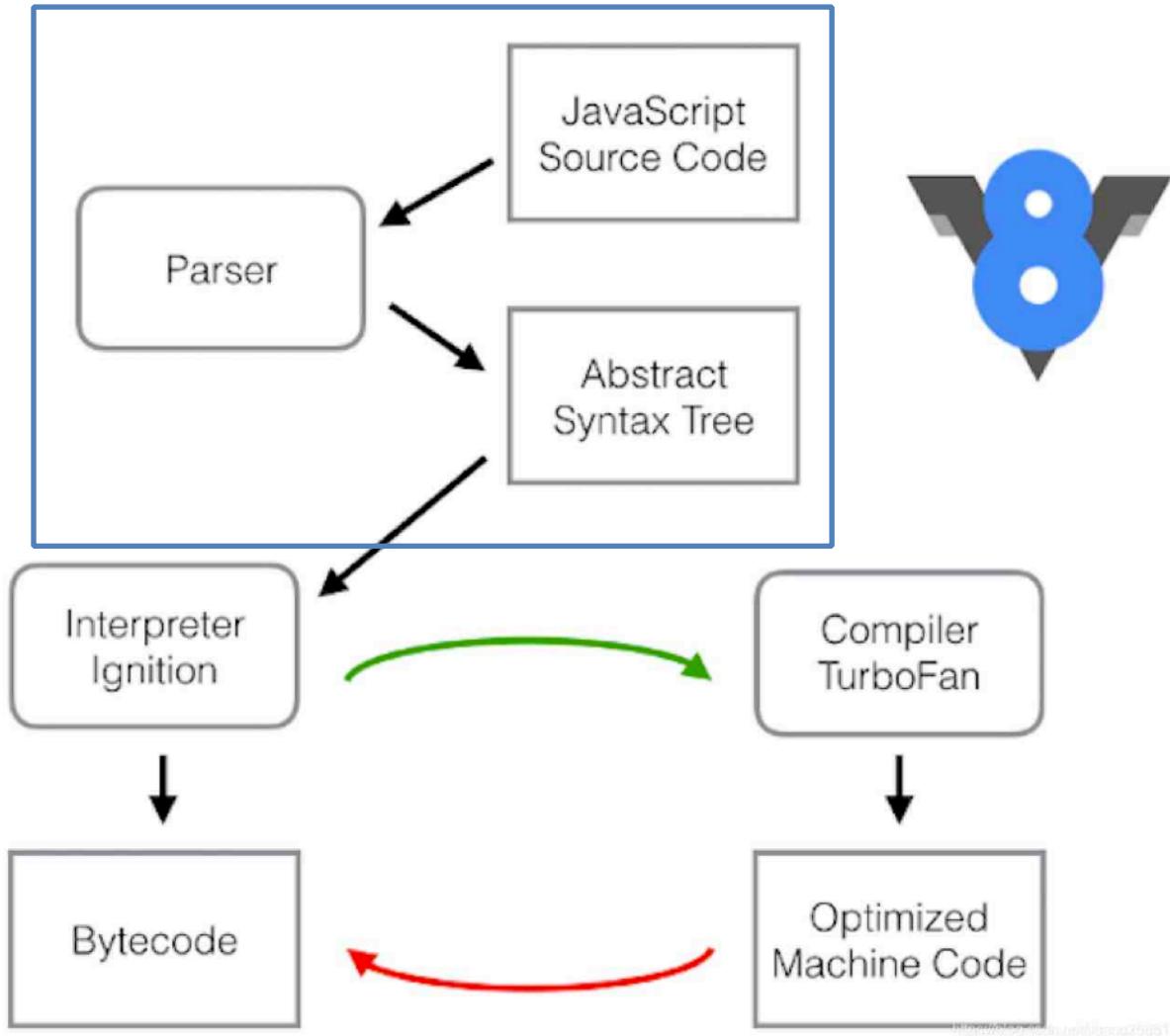
Foi inspirado no Java HotSpot Virtual Machine desenvolvido pela Sun Microsystems e existem muitas similaridades entre eles

Além do Node.js, o **V8** é utilizado no Chrome,
Opera, Edge, Brave, MongoDB, CouchDB,
Electron, Deno e muito mais

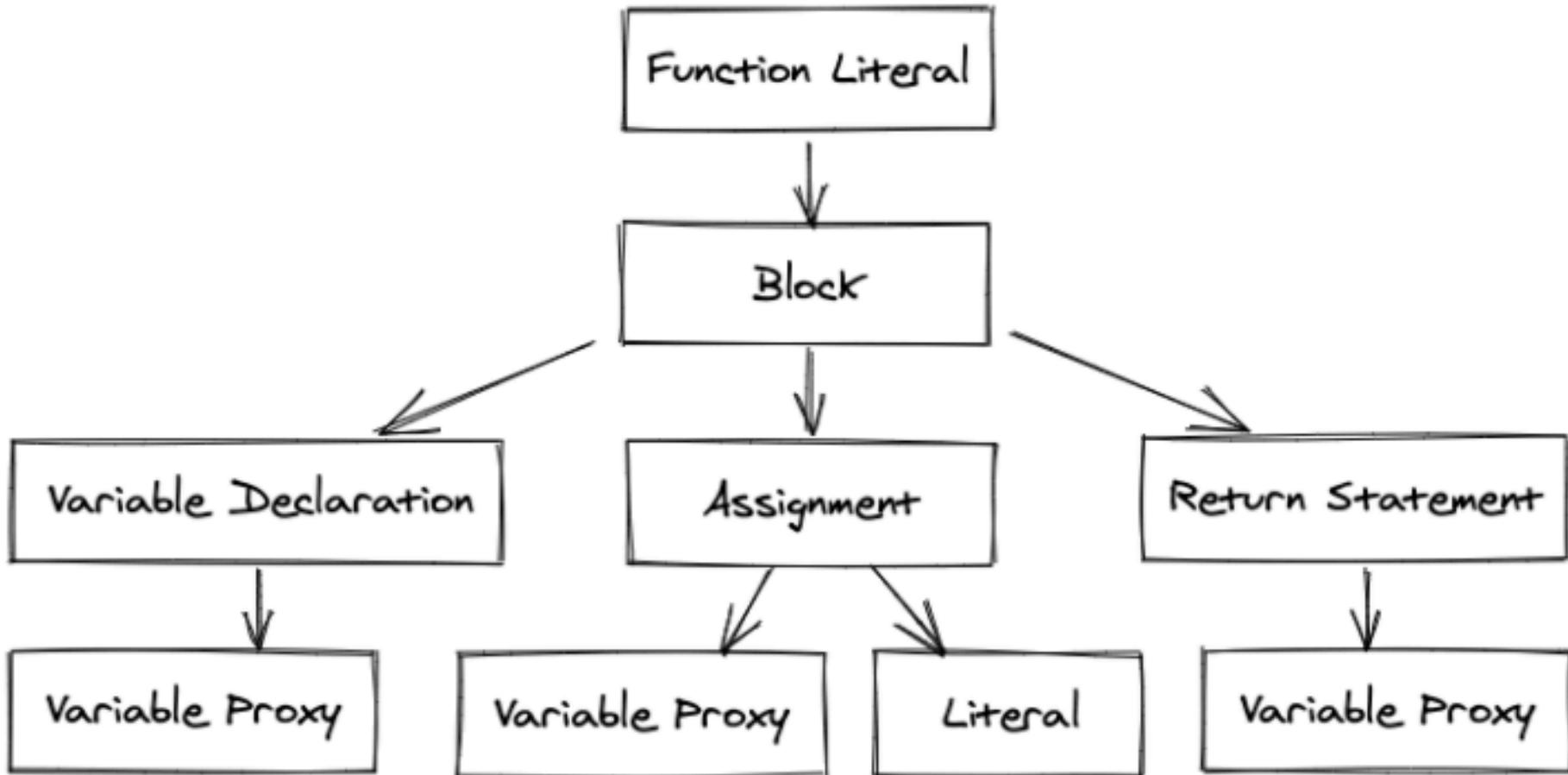
Existem outras **engines** como o SpiderMonkey, utilizado no Firefox, JSC utilizado no Safari e o Chakra utilizado no Edge, que foi substituído recentemente pelo V8



Como o V8 funciona?



O **Parser** converte o código-fonte escrito em JavaScript em uma AST ou Abstract Syntax Tree

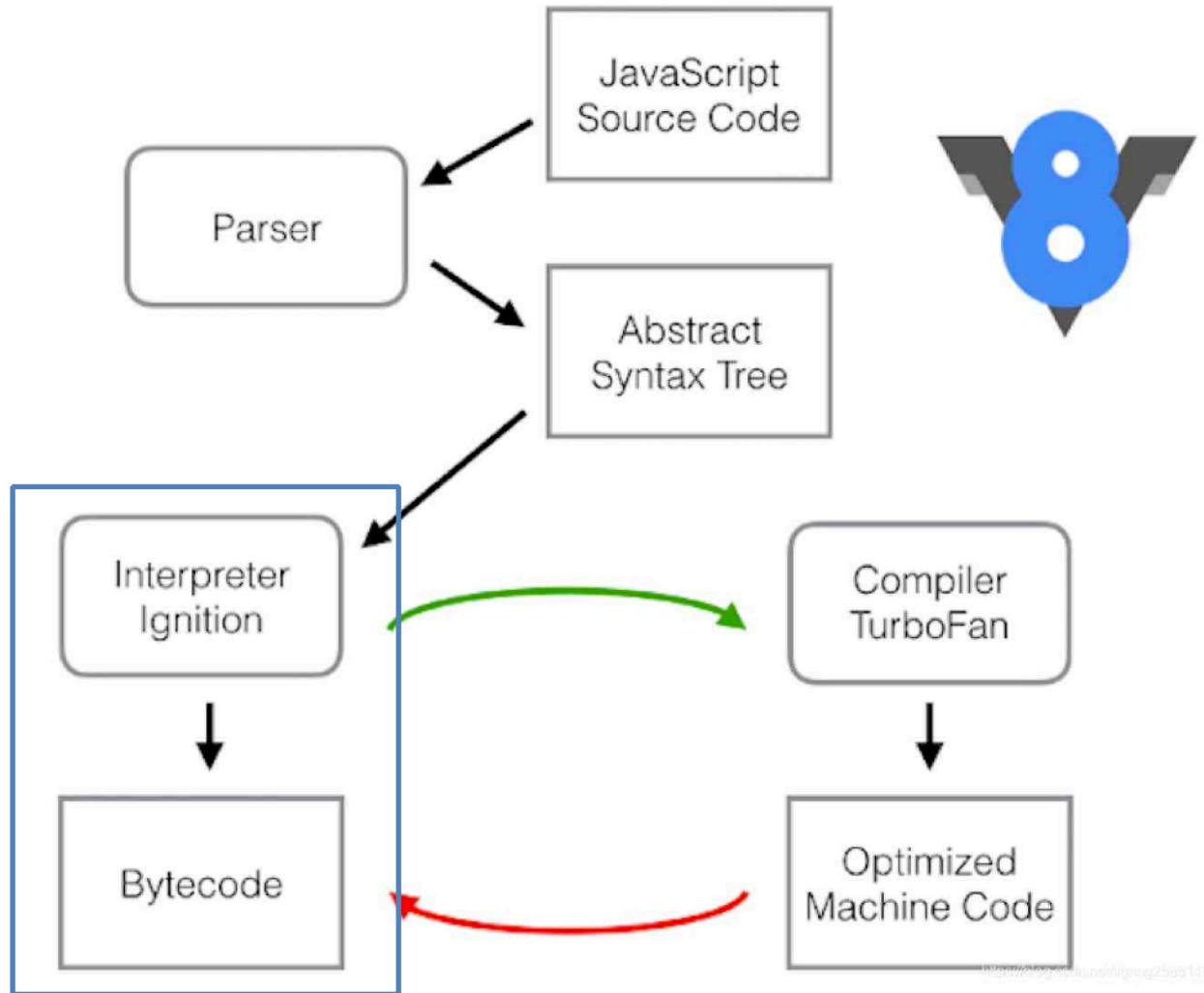


AST explorer astexplorer.net

AST Explorer Snippet JavaScript acorn Transform default ? Parser: acorn-8.3.0 3ms

```
1 function f (a, b, c) {  
2     var d = c - 100;  
3     return a + d * b;  
4 }  
5 f(5, 2, 150);
```

	Tree	JSON
1	{	"type": "Program", "start": 0, "end": 75, "body": [
2		6 {
3		7 "type": "FunctionDeclaration", 8 "start": 0, 9 "end": 61, 10 "id": {
4		11 "type": "Identifier", 12 "start": 9, 13 "end": 10, 14 "name": "f"
5		15 }, 16 "expression": false, 17 "generator": false, 18 "async": false, 19 "params": [
6		20 {
7		21 "type": "Identifier", 22 "start": 12, 23 "end": 13, 24 "name": "a"
8		25 }, 26 {
9		27 "type": "Identifier", 28 "start": 15, 29 "end": 16, 30 "name": "b"
10		31 }, 32 {
11		33 "type": "Identifier", 34 "start": 18, 35 "end": 19, 36 "name": "c"



O Ignition, interpretador do V8, **gera o byte code a partir da AST**

JS f.js X

```
1 function f (a, b, c) {
2     var d = c - 100;
3     return a + d * b;
4 }
5 f(5, 2, 150);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

rodrigobranas node_architecture \$ node --print-bytecode --print-bytecode-filter=f f.js
[generated bytecode for function: f (0x23a829077ce1 <SharedFunctionInfo f>)]
Bytecode length: 15
Parameter count 4
Register count 1
Frame size 8
OSR nesting level: 0
Bytecode Age: 0

32 S> 0x23a82907867e @	0 : 0b 05	Ldar a2
34 E> 0x23a829078680 @	2 : 45 64 00	SubSmi [100], [0]
0x23a829078683 @	5 : c3	Star0
42 S> 0x23a829078684 @	6 : 0b 04	Ldar a1
55 E> 0x23a829078686 @	8 : 3a fa 02	Mul r0, [2]
51 E> 0x23a829078689 @	11 : 38 03 01	Add a0, [1]
59 S> 0x23a82907868c @	14 : a8	Return

Constant pool (size = 0)
Handler Table (size = 0)
Source Position Table (size = 14)
0x23a829078691 <ByteArray[14]>
rodrigobranas node_architecture \$ █

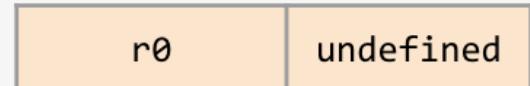
```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```

→ LdaSmi #100
Sub a2
Star r0 →
Ldar a1
Mul r0
Add a0
Return



```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	undefined

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	undefined
accumulator	undefined

```
function f(a, b, c) {  
  var d = c - 100;  
  return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	undefined
accumulator	100

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSm #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	undefined
accumulator	50

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	50
accumulator	50

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	50
accumulator	2

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	50
accumulator	100

```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```



a0	5
a1	2
a2	150
r0	50
accumulator	105

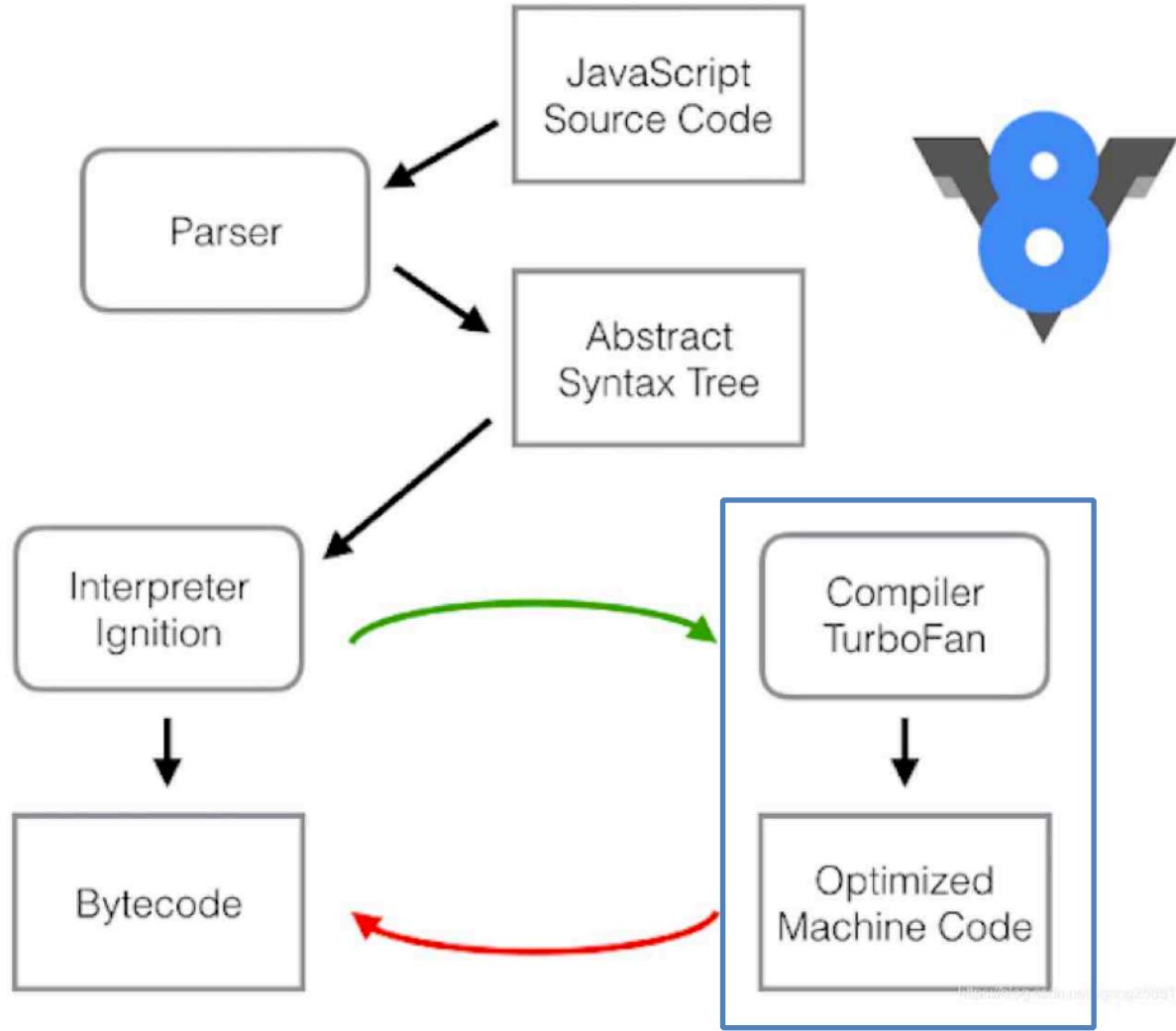
```
function f(a, b, c) {  
    var d = c - 100;  
    return a + d * b;  
}
```



```
LdaSmi #100  
Sub a2  
Star r0  
Ldar a1  
Mul r0  
Add a0  
Return
```

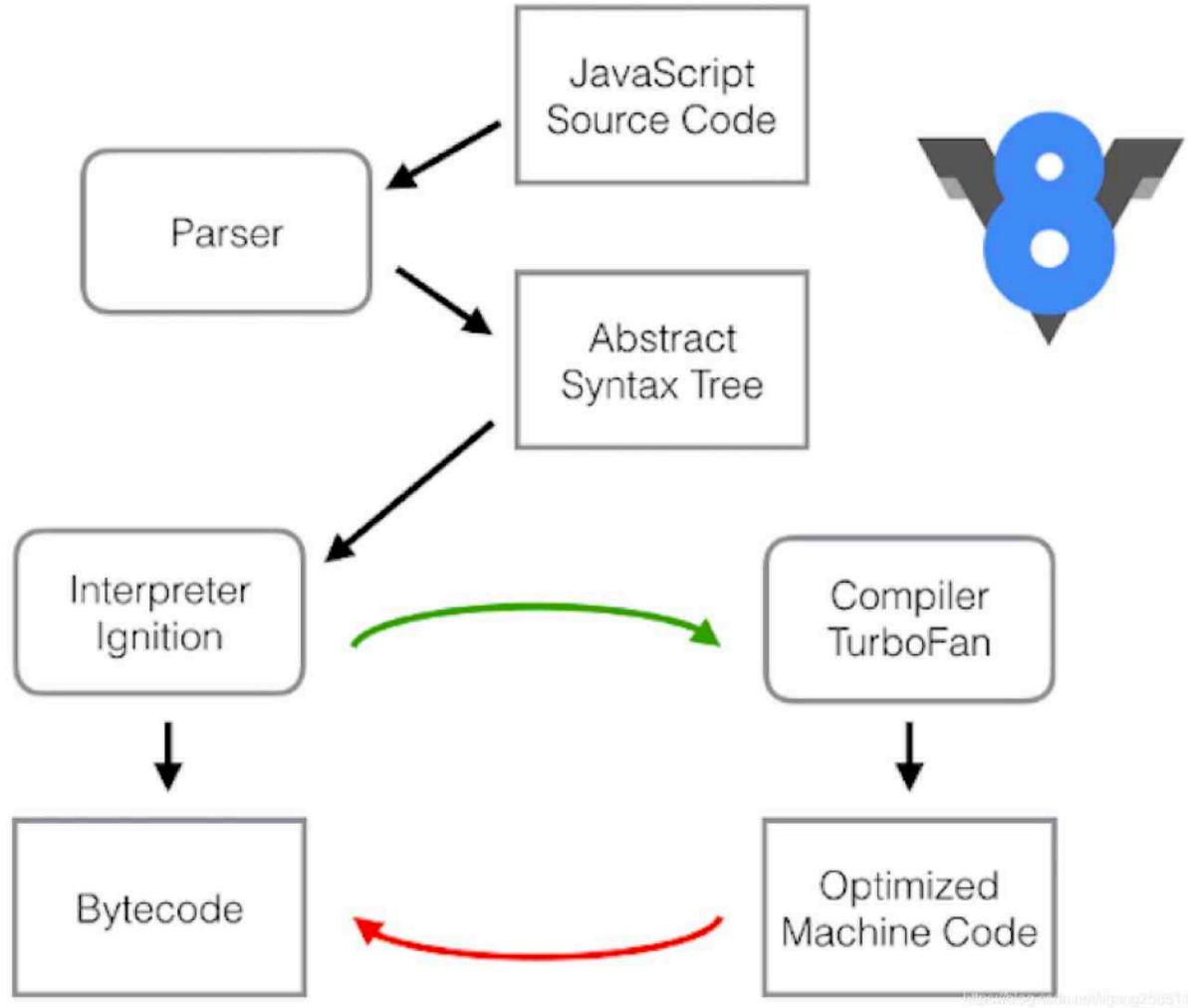


a0	5
a1	2
a2	150
r0	50
accumulator	105

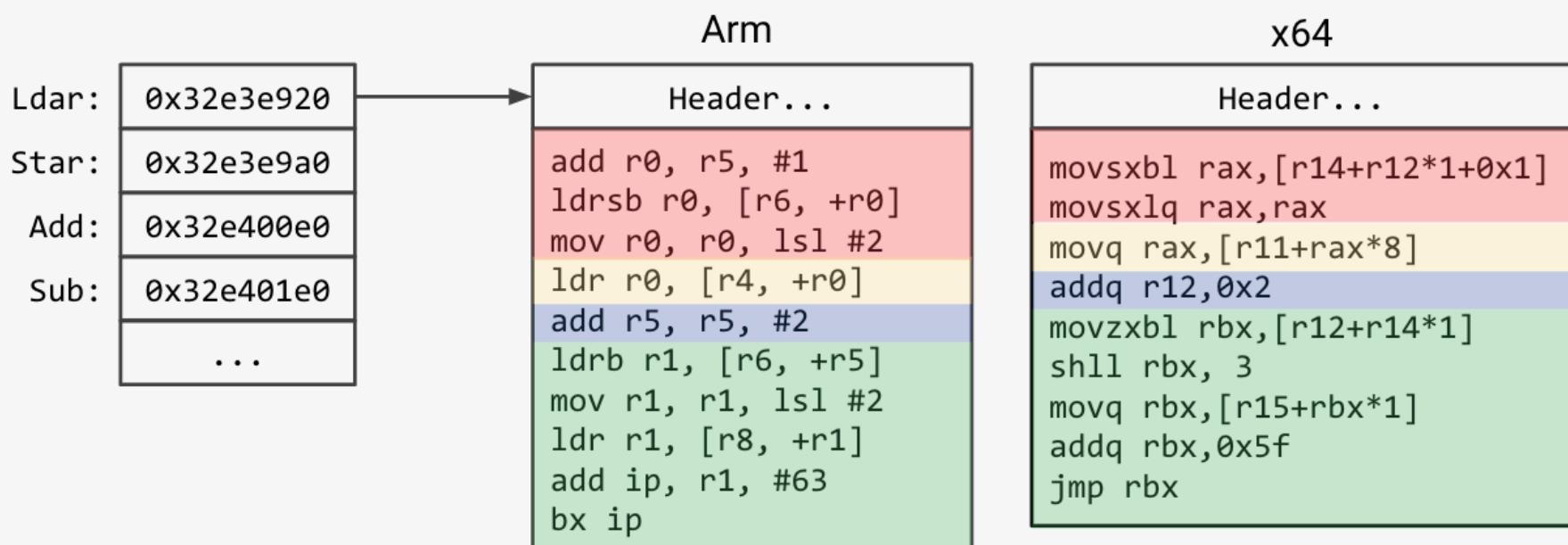


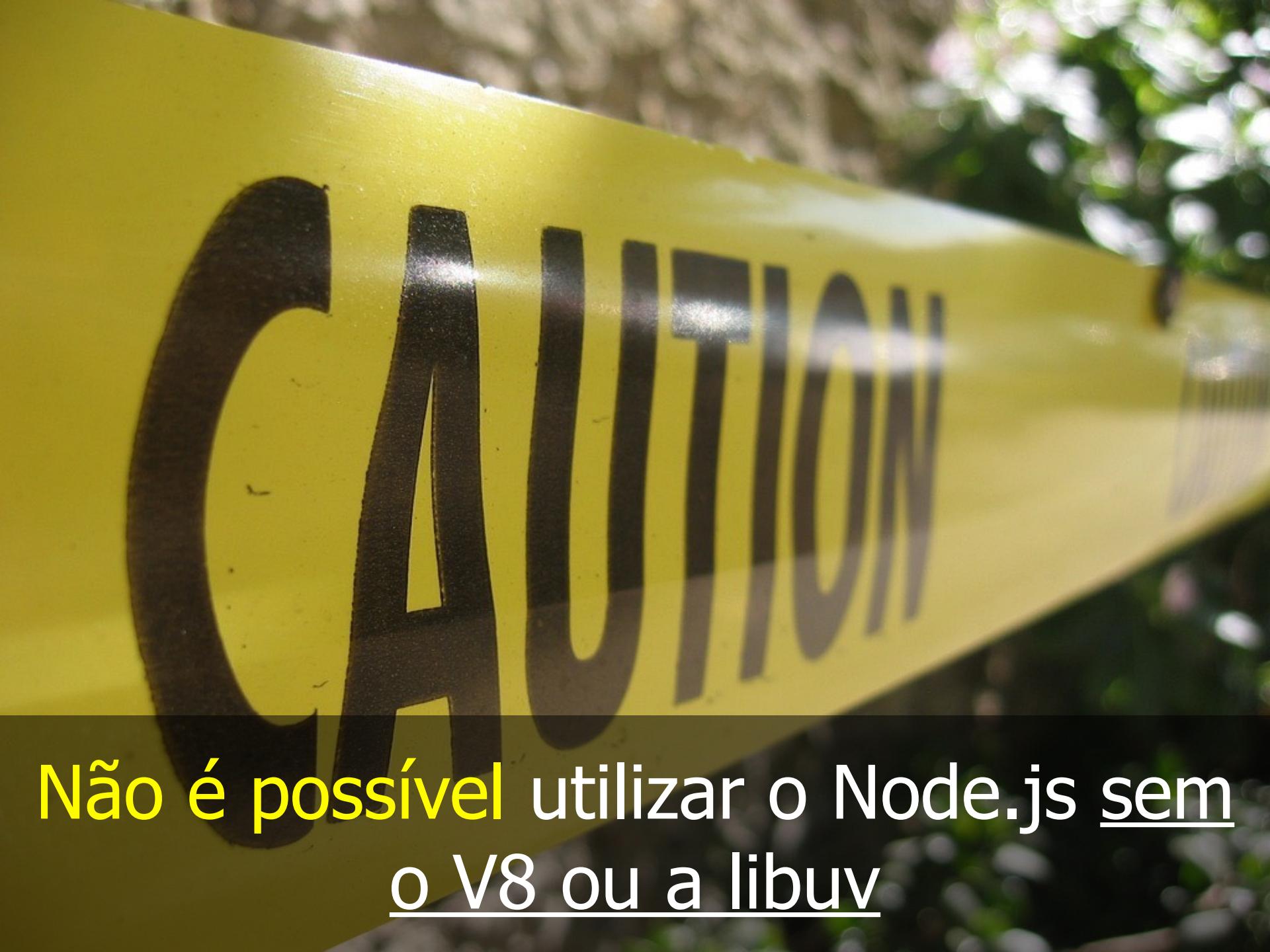
Parte do byte code gerado pelo Ignition, é
compilado e otimizado pelo Turbofan

O V8 tem um JIT Compiler que utiliza tanto um interpretador quanto um compilador no seu pipeline, otimizando partes do código que são executadas com mais frequência



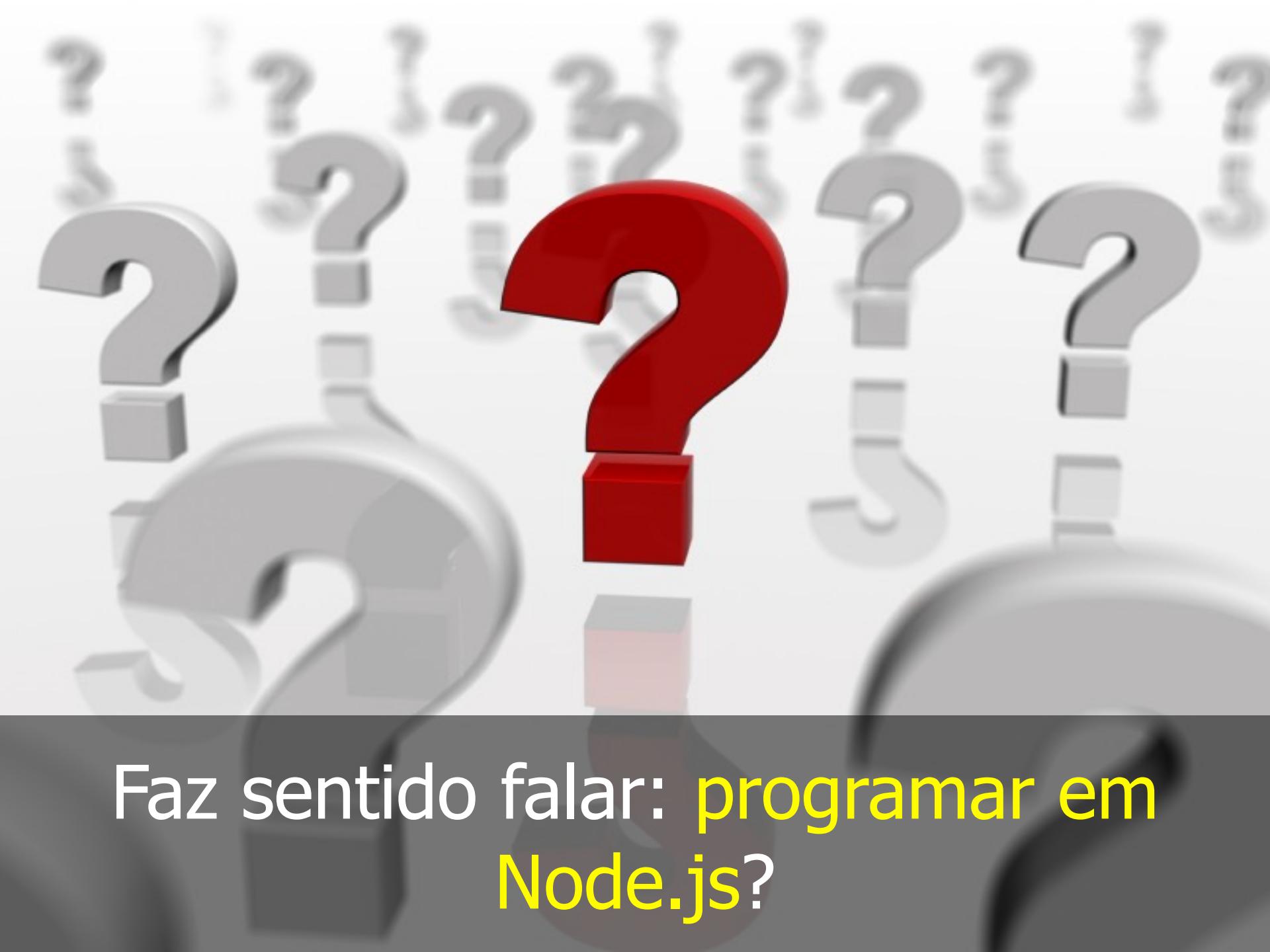
O resultado final é código de máquina otimizado e pronto para ser **executado**





CAUTION

Não é possível utilizar o Node.js sem
o V8 ou a libuv



Faz sentido falar: programar em
Node.js?



E o tal do Deno?

A close-up portrait of Ryan Dahl, the creator of Node.js. He is a young man with dark hair and glasses, wearing a blue shirt. He is looking slightly to the right with a thoughtful expression.

O Ryan Dahl fez a palestra: 10
things I regret about Node.js

Entre os pontos citados por ele estão a **utilização** de uma linguagem não tipada no backend, **no** Deno o JavaScript foi "substituído" pelo TypeScript

Outro fator foi a utilização de callbacks na maior parte das APIs já que na época as promises ainda não eram populares e nem disponíveis na linguagem

No Node.js qualquer dependência acaba tendo acesso irrestrito ao sistema de arquivos e a rede, tanto que existem ataques à dependências com o uso de malwares, no Deno existem flags específicas para liberar o acesso

Na época que o Node.js foi criado, o GYP era o compilador utilizado pelo V8 para compilar C, similar ao CMake. No entanto, com o tempo ele foi substituído e se tornou obsoleto mas ainda é utilizado no Node.js por padrão

Na visão dele as dependências deveriam ser gerenciadas de outra forma, não centralizada, assim como é feito no browser, além do package.json ter acumulado outras responsabilidades com o tempo

O fluxo de resolução do node_modules é hierarquico e relativamente complexo e tem muitas peculiaridades como o index.js

TypeScript front-end

É a interface pública, APIs e funcionalidades que não fazem syscalls ficam nessa camada

É considerado o lado sem privilégios e por padrão não acessa o sistema de arquivos ou a rede, em um sandbox

O acesso é feito por meio do back-end em Rust através do middle-end feito em C++

C++ middle-end

Essa camada intermedia o contato entre o front-end em TS e o backend em Rust

Uma API com métodos de send e receive é disponibilizada para trocar informações

Rust back-end

Todo o acesso privilegiado como sistema de arquivos, rede e ambiente é implementado em Rust

Inicialmente começou a ser feito em Go mas por questões de performance foi feita uma migração para Rust

Tokyo

Ao invés da libuv, que é feita em C/C++, o Deno utiliza o Tokyo feita em Rust

V8

O Deno utiliza o V8, assim como o Node.js, para executar o código-fonte em JavaScript

A close-up photograph of a yellow caution tape. The word "CAUTION" is printed in large, bold, black capital letters. The tape is slightly curved, and the background shows some green foliage.

CAUTION

Projetos escritos utilizando Node.js não são compatíveis com o Deno e vice-versa

Stability | Manual | Deno

deno.land/manual@v1.20.2/runtime/stability

Deno Manual

v1.20.2

1. Introduction

2. Getting Started

- 2.1. Installation
- 2.2. Set up your environment
- 2.3. First steps
- 2.4. Command line interface
- 2.5. Configuration file
- 2.6. Permissions
- 2.7. Debugging your code

3. The Runtime

3.1. Stability

3.2. Program lifecycle

3.3. Permission APIs

3.4. Web Platform APIs

3.5. HTTP Server APIs

3.6. HTTP Server APIs (low level)

3.7. Location API

3.8. Web Storage API

3.9. Workers

3.10. Foreign Function Interface API

4. Linking to external code

4.1. Reloading modules

Search the docs (press / to focus)

Stability

As of Deno 1.0.0, the `Deno` namespace APIs are stable. That means we will strive to make code working under 1.0.0 continue to work in future versions.

However, not all of Deno's features are ready for production yet. Features which are not ready, because they are still in draft phase, are locked behind the `--unstable` command line flag.

```
deno run --unstable mod_which_uses_unstable_stuff.ts
```

Passing this flag does a few things:

- It enables the use of unstable APIs during runtime.
- It adds the `lib.deno.unstable.d.ts` file to the list of TypeScript definitions that are used for type checking. This includes the output of `deno types`.

You should be aware that many unstable APIs have **not undergone a security review**, are likely to have **breaking API changes** in the future, and are **not ready for production**.

Standard modules

Deno's standard modules (<https://deno.land/std/>) are not yet stable. We currently version the standard modules differently from the CLI to reflect this. Note that unlike the `Deno` namespace, the use of the standard modules do not require the `--unstable` flag (unless the standard module itself makes use of an unstable Deno feature).

← The Runtime Program lifecycle →