NodeJS

E/S não-bloqueante - Dirigido a Eventos

Átila Camurça

10 de junho de 2013

Sumary

Introdução

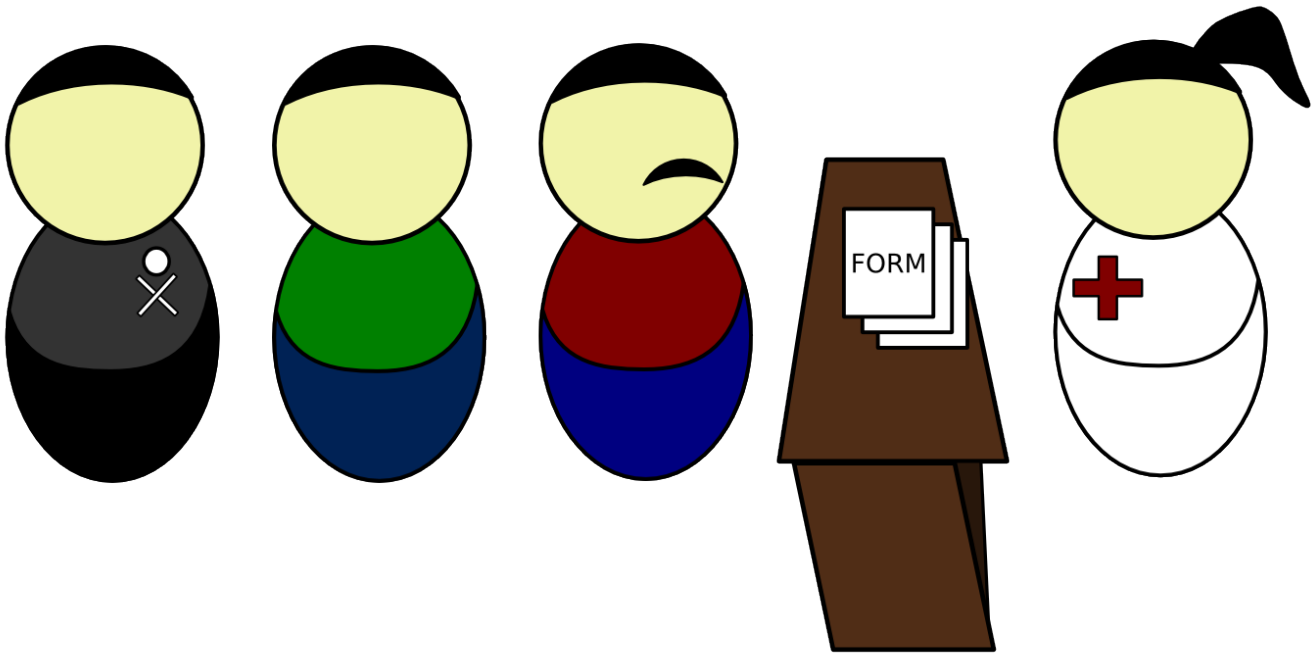
Programas que usam E/S não-bloqueantes tendem a seguir a regra que toda função deve retornar imediatamente.

Isso parece uma thread não?

Existem diferenças. Vejamos um exemplo da vida real.

Recepcionista do médico

Doutor é quem tem doutorado



Imaginem uma fila na recepção de um consultório médico.

Recepcionista do médico

Doutor é quem tem doutorado

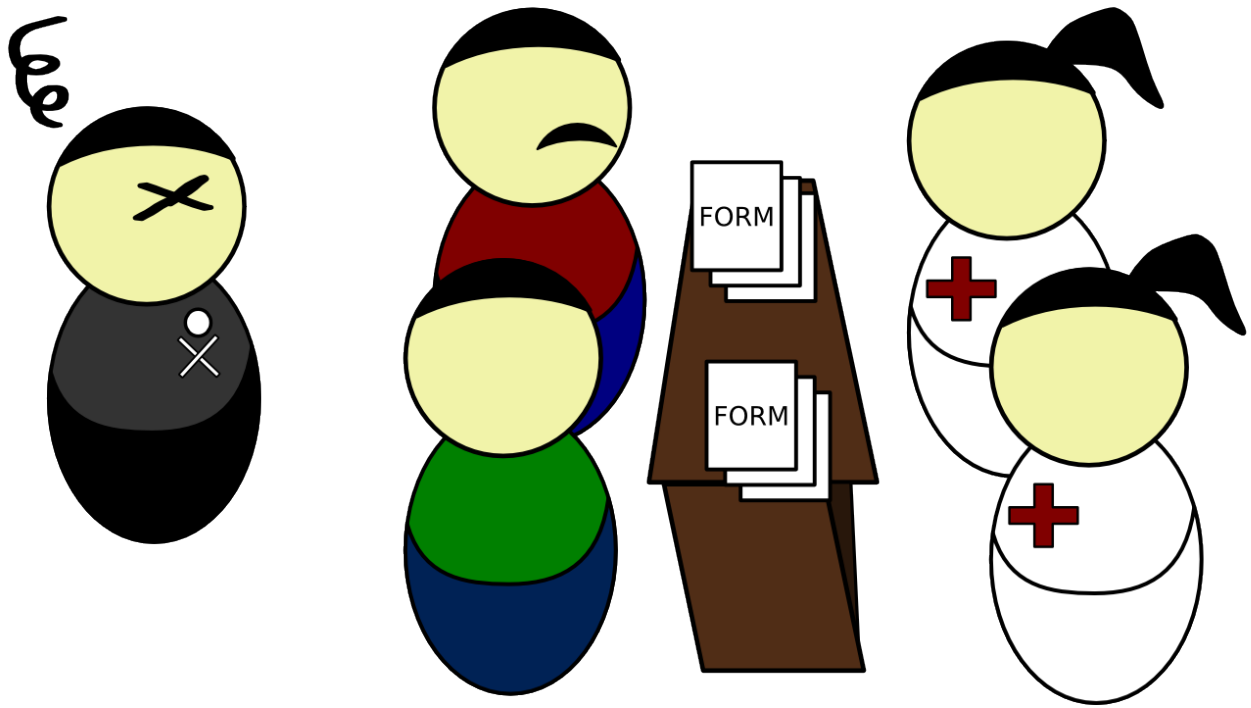
Para ser atendido o paciente precisa preencher **3 formulários**.

Num mundo bloqueante o paciente preencheria os **3 formulários** na própria recepção, fazendo com que a fila espere.

Usando outra *thread* (recepcionista) poderíamos resolver este problema?

Recepcionista do médico

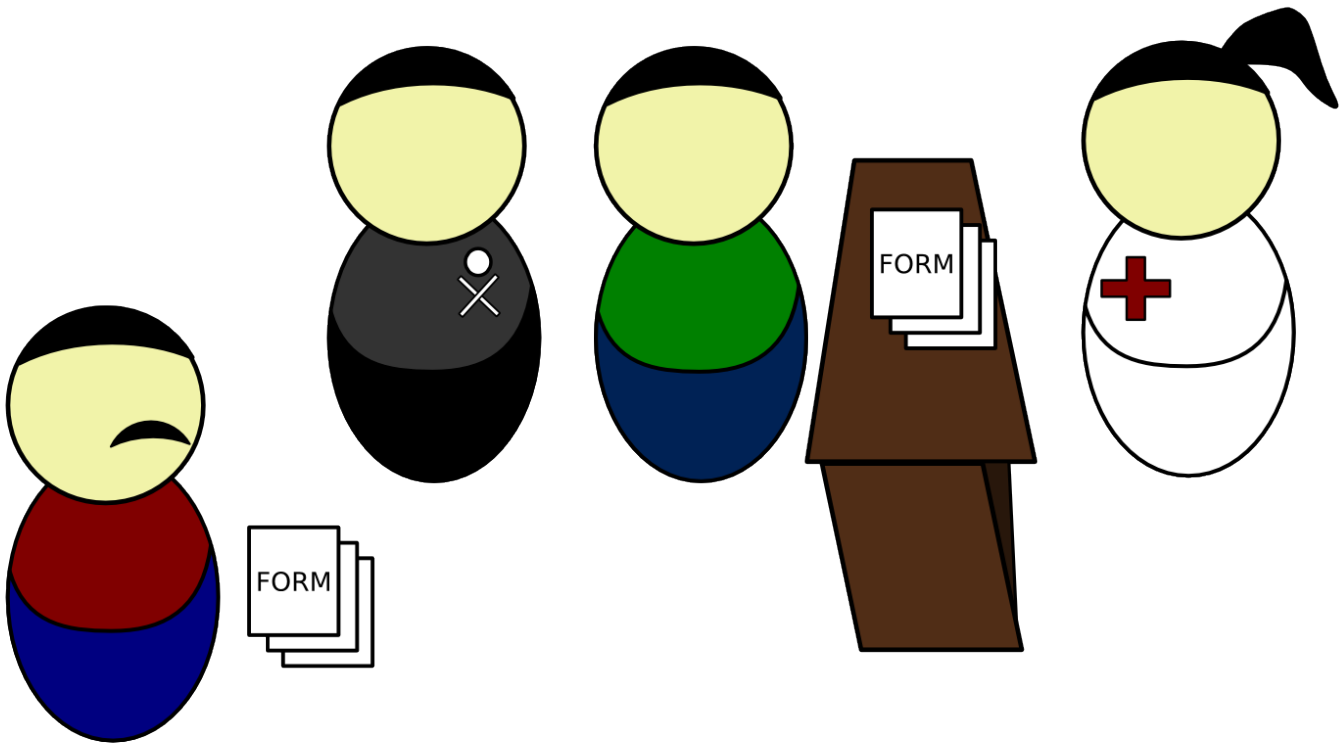
Doutor é quem tem doutorado



Parcialmente, já que mesmo assim a fila deve esperar. Mas como fazer com que a fila não precise esperar?

Recepcionista do médico

Doutor é quem tem doutorado



Enquanto um preenche o formulário, o outro é antendido. E assim por diante. Dessa forma quando um paciente termina de preencher ele volta a fila apenas para entregar o formulário e ser atendido em seguida.

NodeJS

Uma das formas de implementarmos esse comportamento em uma linguagem de programação é usando NodeJS. Rodando em cima do ambiente V8 do Google Chrome, NodeJS consegue trabalhar fora do browser usando JavaScript.

Hello World

Instalação: `$ sudo apt-get install nodejs`

```
console.log("hello world.");
```

Salve como `hello.js`. Execute num terminal:

```
$ node hello.js
```

Emitindo eventos

```
var util = require('util'),
    EventEmitter = require('events').EventEmitter;

var Recepcionista = function() {
    EventEmitter.call(this);
};

var Paciente = function() {
    EventEmitter.call(this);
};

util.inherits(Paciente, EventEmitter);
util.inherits(Recepcionista, EventEmitter);
```

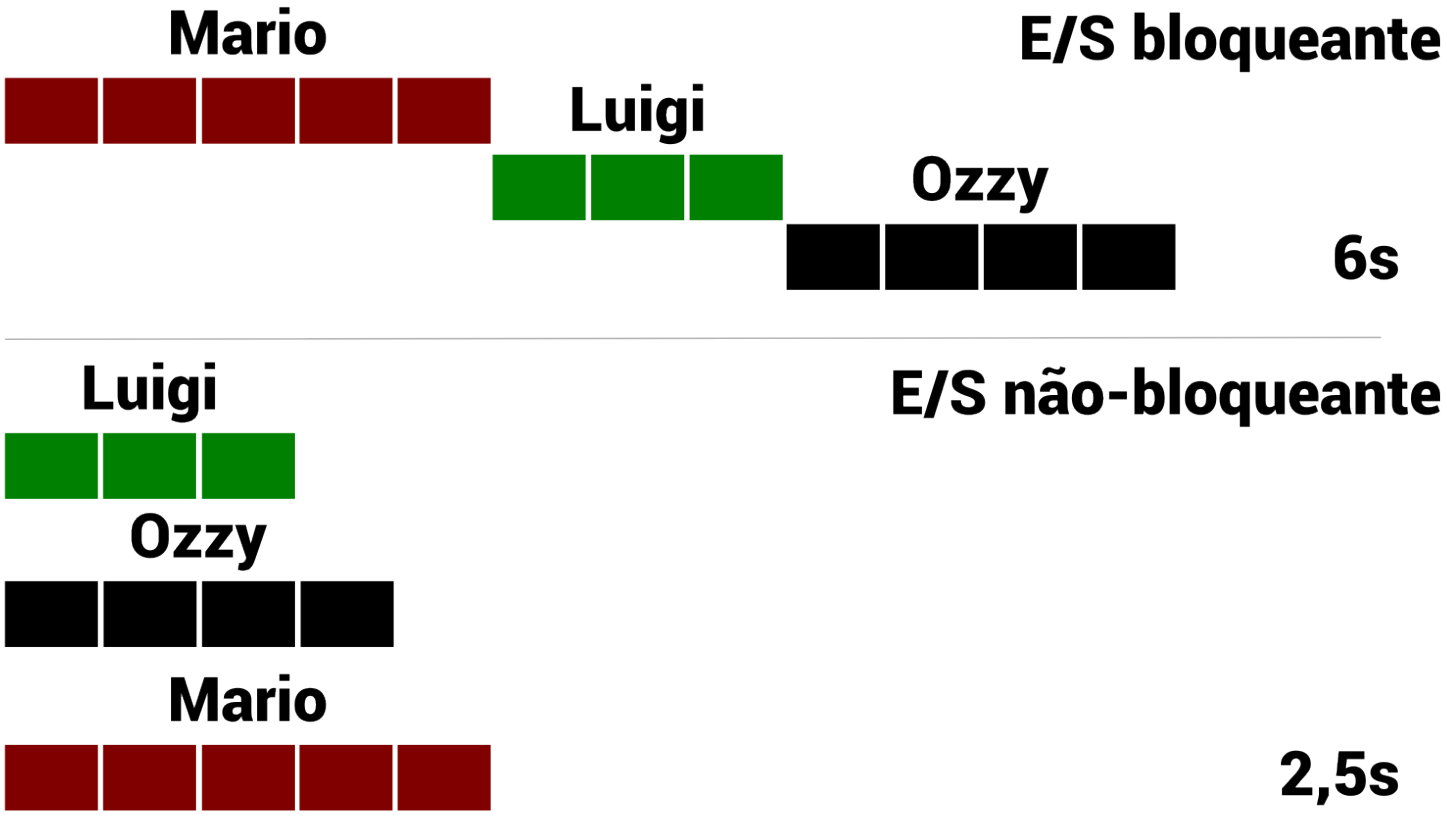
Emitindo eventos

```
Recepcionista.prototype.atender = function() {
    console.log("Preencha o formulário.");
};

Paciente.prototype.preencher = function(recep, nome, t) {
    setTimeout(function() {
        recep.emit('entregar', {nome: nome});
    }, t);
};
```

Emitindo eventos

```
var r = new Recepcionista();
r.on("entregar", function(dados) {
   console.log("Ok Sr(a). " + dados.nome
      + ", você será atendido.");
});
var p = new Paciente();

r.atender();
p.preencher(r, 'Mario', 2500);
r.atender();
p.preencher(r, 'Luigi', 1500);
r.atender();
p.preencher(r, 'Ozzy', 2000);
```

Emitindo eventos



Veja a diferença na vazão.

E/S

Operações com arquivos

```
var fs = require('fs');

fs.readFile('arq1.txt', 'utf-8', function(err, data) {
   if (err) throw err;
   console.log(data);
});

fs.unlink('arq1.txt', function (err) {
  if (err) throw err;
  console.log('successfully deleted arq1.txt');
});
```

Este trecho de código eventualmente irá funcionar como esperado. Isso se fosse uma operação bloqueante, onde o método `fs.unlink` iria esperar o método `fs.readFile`.

Operações com arquivos

Entretanto, estes métodos são assíncronos e podem não funcionar como esperado.

Então como fazer para resolver isso? Utilize chamadas aninhadas.

```
var fs = require('fs');

fs.readFile('arq1.txt', function(e, data) {
    if (!e) {
        console.log(data);
        fs.unlink('arq1.txt');
    } else {
        console.log('ocorreu em erro ao ler.');
    }
});
```

Operações com arquivos

Outra opção é utilizar chamadas síncronas do mesmo método.

```
var fs = require('fs');

var data = fs.readFileSync('arq1.txt',
   {encoding: 'utf-8'});
console.log(data);
fs.unlink('arq1.txt');
```

http://nodejs.org/api/fs.html