



스레드 어디서 사용해야 적합한지
노드 장점
아이오장점 싱크로나이즈..
17p 이벤트 시작과 끝만 알면 코드가 알아서.
싱글스레드로 동작..

어렵 : call back :
sequential 이아니고 eventual 임..

앞규리즈 이미 짜여있 · npm

Chapter 08

Node.js

Open Source SW Development
CSE22300

thread: 쪼개서 마치 동시에 실행되는 것처럼
event : 실행의 단위

Thread VS Event

Thread

- **An abstraction of a unit of execution**
 - **Tasks**
- **What is in a task**
 - **Instructions to executes**
 - **Memory**
 - **File descriptors**
 - **Credentials**
 - **Locks**
 - **Network resources**

Thread

- **Threads**
 - Share memory
 - Share file descriptors
 - Share filesystem context
- **Processes**
 - Not share memory
 - Not share most file descriptors
 - Not share filesystem context

Thread Switching

- **To switch from thread T1 to T2:**
 - Thread T1 saves its registers (including pc) on its stack
 - Scheduler remembers T1's stack pointer
 - Scheduler restores T2' stack pointer
 - T2 restores its registers
 - T2 resumes

Thread Scheduler

- **Maintains the stack pointer of each thread**
- **Decides what thread to run next**
 - E.g., based on priority or resource usage
- **Decides when to pre-empt a running thread**
 - E.g., based on a timer
- **Needs to deal with multiple cores**
 - Didn't use to be the case
- **“fork” creates a new thread**

Uses of Threads

어떤 work에 효과적인가
intensive

- **To exploit CPU parallelism** 높이기 위해 스레드 사용.
 - Run two CPUs at once in the same program
- **To exploit I/O parallelism**
 - Run I/O while computing, or do multiple I/O
 - I/O may be “remote procedure call”
- **Effective CPU intensive work**
cpu놀지않게...!

Common Problems

- **Priority Inversion**
 - High priority thread waits for low priority thread
 - Solution: temporarily push priority up (rejected??)
- **Deadlock**
 - X waits for Y, Y waits for X
- **Incorrect Synchronization**
 - Forgetting to release a lock
- **Failed “fork”**
- **Tuning**
 - E.g. timer values in different environment

priority를 높여서 빨리 처리...

Event

- **An object queued for some module**
- **Operations:**
 - `create_event_queue(handler) → EQ`
 - `enqueue_event(EQ, event-object)`
 - **Invokes, eventually, `handler(event-object)`**
- **Handler is not allowed to block**
 - **Blocking could cause entire system to block**
 - **But page faults, garbage collection,**

이벤트가 큐에 들어간다.
->들어간 순서대로 처리.

handler는 절대 block되서는 안된다.

Event Scheduler

- **Decides which event queue to handle next.**
 - Based on priority, CPU usage, etc.
- **Never pre-empts event handlers!**
 - No need for stack / event handler
- **May need to deal with multiple CPUs**

큐안에들어오면 순서를 바꿀 수없다.
대신 여러개의 큐 생성.

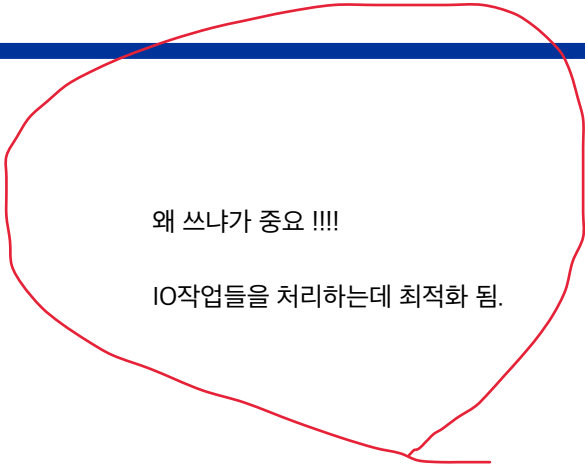
Event Synchronization

- **Handlers cannot block → no synchronization**
- **Handlers should not share memory**
 - **At least not in parallel**
- **All communication through events**

동기화 필요 없당....

Uses of Events

- **CPU parallelism**
 - Different handlers on different CPUs
- **I/O concurrency**
 - Completion of I/O signaled by event
 - Other activities can happen in parallel



왜 쓰냐가 중요 !!!!

IO작업들을 처리하는데 최적화 됨.

Threads vs. Events

- **Events-based systems use fewer resources**
 - Better performance (particularly scalability)
- **Event-based systems harder to program**
 - Have to avoid blocking at all cost
 - Block-structured programming doesn't work
 - How to do exception handling?
- **In both cases, tuning is difficult**

이벤트에 맞춰
수행을 하기만 하면 됨.

Node.js

NodeJS

- **Uses Googles V8 JavaScript engine as an interpreter**
 - Very fast
- **Not a JavaScript framework**
 - A Runtime Environment
- **Event-driven architecture**
 - All APIs of Node.js library are asynchronous
- **Non-blocking I/O**
- **Single thread**



NodeJS

- **Contains many modules**
 - Similar to the built-in java object
- **Loading modules simply uses ‘require(..)’**
- **All built in (and modules loaded from npm) do not need pathing**
- **Modules created in application need path**

Advantages : Asynchronous

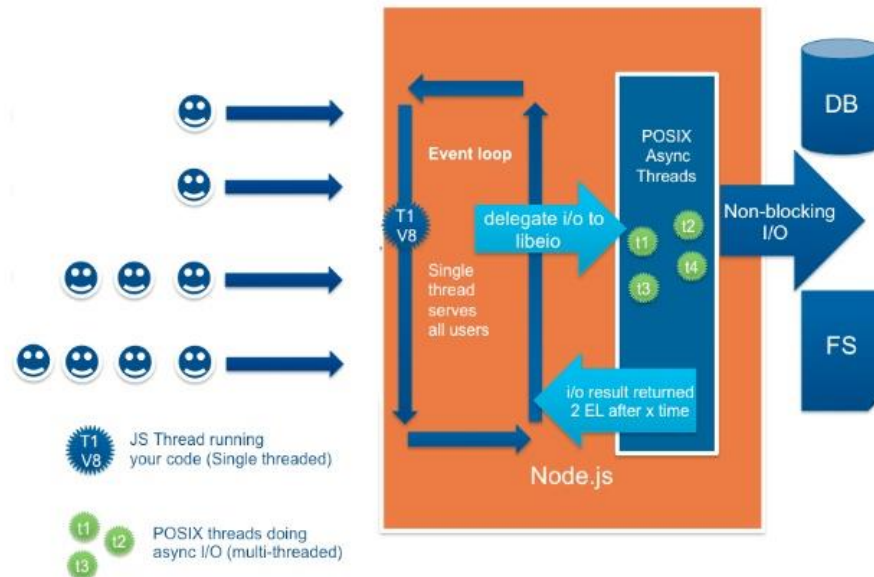
- No waiting time for CPU
 - No blocking
 - Asynchronous I/O

얼굴 : 요청

하나씩 실행.
io작업이 있으면
내부적으로 스레드가
잠시 멈추고....요청
하구 다하면 다시
이벤트루프

IO작업이 끝날때까지
기다리고 스케줄러

Node.js - Single Thread, Event-ed



*Credit: blog.cloudfoundry.com

Saturday, October 6, 12

Advantages: Productivity

- **Tearing down barrier for Web backend**
 - Web backend needs lots of background knowledge for developer
 - JavaScript is used for Web front-end
 - With Node.js, Web front-end developer can code backend-apps
 - Easy access for Web backend
- **No Threads**
 - No Synchronization, Critical section
 - Simple programming
- **Modules**
 - NPM

Disadvantages: Single Thread

- **Intensive CPU Job**

- A certain job can be bottleneck to Node.js
- I/O intensive job is sweet spot for Node.js

cpu 많이 쓰는 작업은 노드로 개발 X

io intensive한건 좋음 노드에!
->웹서버, io관련 코딩시 많이 쓰임

- **Multiple Core**



- Node.js can be run on **single thread and Core**
- Other cores??
- Cluster Module
- Session Sharing??

Disadvantages: Call Back

- Readability
 - JavaScript vs. Java
- Call Back Hell



```
1 var floppy = require('floppy');
2
3 floppy.load('disk1', function (data1) {
4   floppy.prompt('Please insert disk 2', function () {
5     floppy.load('disk2', function (data2) {
6       floppy.prompt('Please insert disk 3', function () {
7         floppy.load('disk3', function (data3) {
8           floppy.prompt('Please insert disk 4', function () {
9             floppy.load('disk4', function (data4) {
10              floppy.prompt('Please insert disk 5', function () {
11                floppy.load('disk5', function (data5) {
12                  // if node.js would have existed in 1995
13                });
14              });
15            });
16          });
17        });
18      });
19    });
20  });
21 });
22
```

Disadvantages: Learning Curve

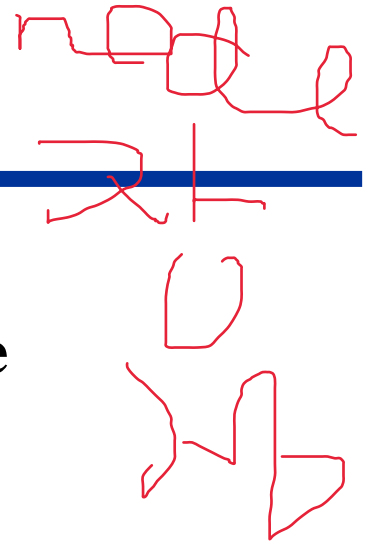
- Sequential Programming → Event Programming

```
FILE *pFile = fopen("file.txt", "r"); // Wait...
char buf[101];
if ( NULL != pFile ) {
    fread(buf, 1, 100, pFile); // Wait...
    buf[(sizeof buf)-1] = 0;
    printf("%s", buf); // Wait...
    fclose(pFile); // Wait...
}
printf("Exit");
```

```
var file = "somefile.txt";
var fs = require("fs");
fs.readFile(file, function(err, data) {
    if ( err ) { throw err; }
    var len = data.toString().split("\n").length - 1; console.log("File Length: " + len);
});
console.log("Exit");
```

NPM

Node Package Manager



- **Online repositories for node.js**
- **Command line utility to install node.js package**
- **NPM comes bundled with node.js installables**

- **Search Modules**

- `npm search <search name>`

온라인에서 간단히 다운받아 설치.
일종의 저장소이자 command line utility.

- **Installing Modules**

- `npm install <module name>`

- **Uninstalling Modules**

- `npm uninstall <module name>`

- **How to Use**

- `var express = require('express');`

Global vs Local

- **Default/Local**
 - NPM installs any dependency in the local mode
 - Installed Packages/dependencies in `node_modules` directory
 - Deployed packages are accessible via `require()` method
- **Global**
 - Installed packaged/dependencies in system directory
 - Directly run installed packages
 - Cannot import using `require()`

Package.json

- **Define the properties of a package**
 - **Package.json** is present in the root directory of any Node application/module
- **Attributes of Packages.json**
 - **name** : name of the package
 - **version** : version of the package
 - **description** : description of the package
 - **homepage** : homepage of the package
 - **author** : author of the package
 - **contributors** : name of the contributors to the package
 - **dependencies** : list of dependencies
 - **repository** : repository type and URL of the package
 - **main** : entry point of the package
 - **keywords** : keywords

Package.json

- Create your own packages.json
 - **npm init**
- Add dependencies in your packages.json
 - **npm install <package> --save**
- Exercise
 - Create a folder
 - **npm init**
 - **npm install express --save**
 - **Check packages.json**

dependency에 express어짜구 생김.

상대방에게 전달해주면 맞춰서 설치하면 알아서 exptree...=○

Your Own Module

- **Use exports keyword**
 - Make properties and methods available outside the module file

- myfirstmodule.js

```
exports.myDateTime = function () {  
    return Date();  
};
```

외부에서 오픈 할 것만 exports!!!!
이런식으로 모듈 만들어서 배포 . !

- app.js

```
var dt = require('./myfirstmodule');  
  
console.log(dt.myDateTime());
```

Callback Concept

Blocking Code

- **Callback is an asynchronous function**
 - Callback is called at the completion of a given task
- **Blocking Code Example**
 - Creates input.txt
 - npm install fs –save
 - **Copy codes as block.js** 이건 시퀀스하게 부름! ㅇ....

```
var fs = require("fs");  
var data = fs.readFileSync('input.txt');  
console.log(data.toString());  
console.log("Program Ended");
```

- **node block.js**

Non-Blocking Code

- **Non-Blocking Code Example**

- **Creates input.txt**
- **npm install fs --save**
- **Copy codes as nonblock.js**

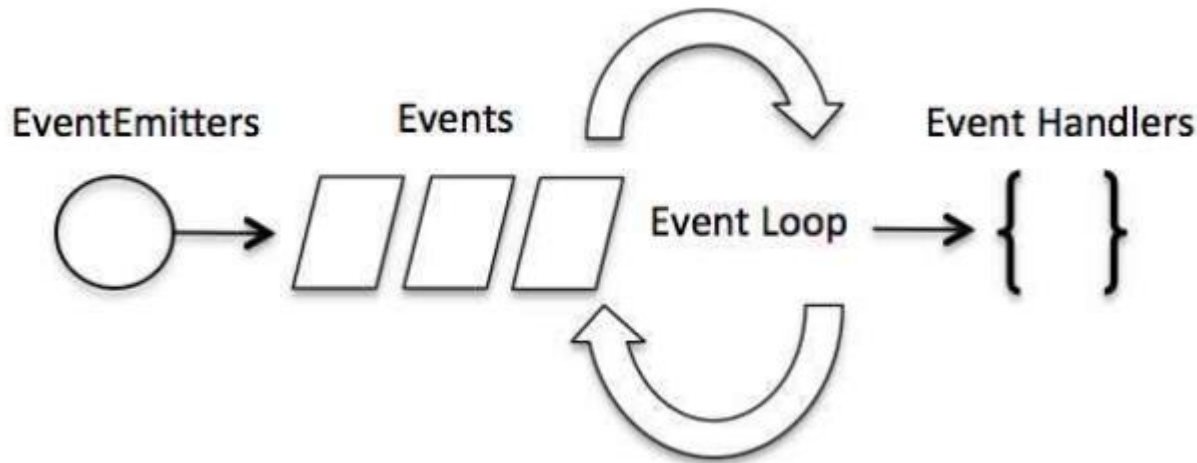
```
var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});
console.log("Program Ended");
```

- **node nonblock.js**

Event Loop

Event-Driven Programming

- **Node.js is a single-threaded application**
 - **Event and Callbacks**
 - **Triggers a callback function when one of those events is detected**



하나씩 하면서 event handler -> 필요한 callback함수 불른다.

Event-Driven Programming

- Import **events module**
 - `var events = require('events');` 생
- Create an `eventEmitter` object
 - `var eventEmitter = new events.EventEmitter();` 만들오
- Bind event and event handler
 - `eventEmitter.on('eventName', eventHandler);` 핸들러...
- Fire an event
 - `eventEmitter.emit('eventName');` 이벤트발

Event-Driven Programming

- **Example**

```
var events = require('events');
var EventEmitter = new events.EventEmitter(); 이벤트발생할수잇는..?
var connectHandler = function connected() { connect handler
  console.log('connection successful. ');
  EventEmitter.emit('data_received');
}

EventEmitter.on('connection', connectHandler);

EventEmitter.on('data_received', function() {
  console.log('data received successfully. ');
});

EventEmitter.emit('connection');
console.log("Program Ended.");
```

Event Emitter

Method	Description
<code>addListener(event, listener)</code> <code>on(event, listener)</code>	Adds a listener at the end of the listeners array for the specified event.
<code>once(event, listener)</code> 딱 한번만 실행...	Adds a one-time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed.
<code>removeListener(event, listener)</code> 제	Removes a listener from the listener array for the specified event.
<code>removeAllListeners([event])</code>	Removes all listeners, or those of the specified event.
<code>setMaxListeners(n)</code>	By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event.

Event Emitter

Method	Description
<code>listeners(event)</code>	Returns an array of listeners for the specified event.
<code>emit(event, [arg1], [arg2], [...])</code>	Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

Event Emmitter

- **Example**

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
var listner1 = function listner1() {
  console.log('listner1 executed.');
```



```
}
var listner2 = function listner2() {
  console.log('listner2 executed.');
```



```
}

eventEmitter.addListener('connection', listner1);
eventEmitter.on('connection', listner2);
var eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");
```



```
eventEmitter.emit('connection');
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");
```



```
eventEmitter.emit('connection');
eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");
console.log("Program Ended.");
```

File System

Open File

- **Standard POSIX functions**
- **Import FS**
 - `var fs = require("fs");`
- **Open a File**
 - `fs.open(path, flags, [mode], callback);`
 - **path** : file name path
 - **flags** : read/write flags
 - **mode** : file permission only if the file was created
 - **callback** : get two arguments (err, fd)

Open File

- **Example**

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

console.log("Going to open file!");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
});
```


Writing File

- **Writing a File**
 - **fs.writeFile(filename, data,[options], callback)**
 - **path : file name path**
 - **data : String or buffer**
 - **mode : The third parameter is an object which will hold {encoding, mode, flag}**
 - **callback : get single argument (err)**

Writing File

- **Example**

```
var fs = require("fs");
console.log("Going to write into existing file");
fs.writeFile('input.txt', 'Simply Easy Learning!', function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("Data written successfully!");
  console.log("Let's read newly written data");
  fs.readFile('input.txt', function (err, data) {
    if (err) {
      return console.error(err);
    }
    console.log("Asynchronous read: " + data.toString());
  });
})
```

Reading File

- **Reading a File**

- **fs.read(fd, buffer, offset, length, position, callback)**
- **path : file name path**
- **buffer : The data will be written to** data를 담을 변수
- **offset : the buffer to start writing at** 버퍼에서 시작할 위치 어느위치부터데이터 담을지
- **length : The number of bytes to read** 읽을 data 길
- **position : where to begin reading from in the file** 파일의 특정 위치부터 시작
- **callback : get single arguments (err)**

Reading File

- **Example**

```
var fs = require("fs");
var buf = new Buffer(1024);
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
    if (err){
      console.log(err);
    }
    console.log(bytes + " bytes read");
    if(bytes > 0){
      console.log(buf.slice(0, bytes).toString());
    }
  });
});
```

Deleting File

- **Deleting a File**
 - **fs.unlink(path, callback)**
 - **path : file name path**
 - **callback : get single argument (err)**

Deleting File

- **Example**

```
var fs = require("fs");
console.log("Going to delete an existing file");
fs.unlink('input.txt', function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("File deleted successfully!");
});
```

Creating Directory

- **Creating a directory**
 - **fs.mkdir(path[, mode], callback)**
 - **path : file name path**
 - **mode : directory permission to be set**
 - **callback : get single argument (err)**

Creating Directory

- **Example**

```
var fs = require("fs");
console.log("Going to create directory /tmp/test");

fs.mkdir('/tmp/test',function(err){
  if (err) {
    return console.error(err);
  }
  console.log("Directory created successfully!");
});
```


Others

- **Closing a File**

- `fs.close(fd, callback)` 열었으면 닫아

- **Truncate a File**

- `fs.ftruncate(fd, len, callback)` 자

- **Read a Directory**

- `fs.readdir(path, callback)` 디렉토리에 있는 data 읽어냄.
뭐가있는지....?

- **Remove a Directory**

- `fs.rmdir(path, callback)` 디렉토리 지움