Lab 15.1 - Set Child Process Environment Variable

The labs-1 folder contains an `index.js`, a `child.js` file and a `test.js` file.

The `child.js` file contains the following:

```
'use strict'
const assert = require('assert')
const clean = (env) => Object.fromEntries(
  Object.entries(env).filter(([k]) => !/^(_.*|pwd|shlvl)/i.test(k))
)
const env = clean(process.env)

assert.strictEqual(env.MY_ENV_VAR, 'is set')
assert.strictEqual(
  Object.keys(env).length,
  1,
  'child process should have only one env var'
)
console.log('passed!')
```

The code in `child.js` is expecting that there will be only one environment variable named **`MY_ENV_VAR`** to have the value `'is set'`. If this is not the case the **`assert.strictEqual`** method will throw an assertion error. In certain scenarios some extra environment variables are added to child processes, these are stripped so that there should only ever be one environment variable set in `child.js`, which is the **`MY_ENV_VAR`** environment variable.

The `index.js` file has the following contents:


```
'use strict'
const assert = require('assert')

function exercise (myEnvVar) {
  // TODO return a child process with
  // a single environment variable set
  // named MY_ENV_VAR. The MY_ENV_VAR
  // environment variable's value should
  // be the value of the myEnvVar parameter
  // passed to this exercise function
}
```

Using any **child_process** method except **execFile** and **execFileSync**, complete the exercise function so that it returns a child process that executes the **child.js** file with **node**.

To check the exercise implementation, run **node test.js**, if successful the process will output: **passed!** If unsuccessful, various assertion error messages will be output to help provide hints.

One very useful hint up front is: use **process.argv[0]** to reference the **node** executable instead of just passing **'node'** as string to the **child_process** method.

The contents of the **test.js** file is esoteric, and the need to understand the code is minimal, however the contents of **test.js** are shown here for completeness:

```
'use strict'
const assert = require('assert')
const { equal } = assert.strict
const exercise = require('.')

let sp = null
try {
  sp = exercise('is set')
  assert(sp, 'exercise function should return a child process
instance')
  if (Buffer.isBuffer(sp)) {
    equal(sp.toString().trim(), 'passed!', 'child process
misconfigured')
    process.stdout.write(sp)
    return
  }
```



```
} catch (err) {
  const { status} = err
  if (status == null) throw err
  equal(status, 0, 'exit code should be 0')
  return
}

if (!sp.on) {
  const { stdout, stderr } = sp
  if (stderr.length > 0) process.stderr.write(stderr)
  if (stdout.length > 0) process.stdout.write(stdout)
  equal(sp.status, 0, 'exit code should be 0')
  equal(stdout.toString().trim(), 'passed!', 'child process
misconfigured')
  return
}

let out = ''
if (sp.stderr) sp.stderr.pipe(process.stderr)
if (sp.stdout) {
  sp.stdout.once('data', (data) => { out = data })
  sp.stdout.pipe(process.stdout)
} else {
  // stdio may be misconfigured, or fork method may be used,
  // allow benefit of the doubt since in either case
  // exit code check will still fail:
  out = 'passed!'
}
const timeout = setTimeout(() => {
  equal(out.toString().trim(), 'passed!', 'child process
misconfigured')
}, 1000)

sp.once('exit', (status) => {
  equal(status, 0, 'exit code should be 0')
  equal(out.toString().trim(), 'passed!', 'child process
misconfigured')
  clearTimeout(timeout)
})
```


The **`test.js`** file allows for alternative approaches, once the **`exercise`** function has been completed with one **`child_process`** method, re-attempt the exercise with a different **`child_process`** method.

