

TP n° 4

Programmation avec nodejs

Le but de ce TP est de se familiariser avec nodejs et sa bibliothèque standard. On n'utilisera pas pour ce TP de système de package (comme npm) ni de système de build (rollup, ...). Mais uniquement les outils suivants :

- un éditeur de texte (VSCode);
- un terminal avec le toplevel node pour les tests :

```
$ node  
> const answer = 42;  
$
```

Cette console permet de tester des expressions et autorise la complétion des expressions et des objets avec tab

- la documentation de la bibliothèque standard

<https://nodejs.org/docs/latest-v20.x/api/>

- l'interpréteur node pour exécuter des programmes :

```
$ node monprogramme.js
```

Attention, au PUIO, il faut avoir défini correctement le chemin vers la bonne version de NodeJS. Une façon simple est de s'assurer que le fichier ~/.bashrc contient la ligne :

```
source /public/kn/setup-kn.sh
```

1 Commande ls

Note : pour tout ce TP, on utilisera les versions **synchrones** (i.e. bloquantes) des fonctions de l'API. La plupart du temps, cela signifie qu'il faut utiliser la version xyzSync() d'une fonction si elle existe.

Le but de cet exercice est d'implémenter une version de plus en plus complexe de la commande ls (dont on change légèrement les spécifications). La commande doit fonctionner de la manière suivante :

```
$ node ls.js [options] [chemin]
```

La commande attend une liste (potentiellement vide) d'options et au plus **un** chemin. Si le chemin est absent ou qu'il ne correspond pas à un répertoire existant, alors le chemin "." est utilisé à sa place. Les options que l'on souhaite supporter sont : Sur invocation de la commande, les entrées du répertoire sont listées. L'affichage est influencé par les options.

- 1 provoque un affichage sur une seule colonne plutôt qu'en lignes
- t ordonne les fichiers par date de dernière modification décroissante
- l provoque un affichage détaillé des fichiers. Implique un affichage en colonne.

Le code principal du programme sera dans un fichier ls.js.

1. Écrire une fonction parseCmdLine(argv) qui prend en argument un tableau de chaînes représentant la ligne de commande et renvoie un objet Javascript de la forme :

```
{ path : "chemin",  
  long : true,  
  column : true,  
  sorted : false  
}
```

où les propriétés `path` dénote le chemin vers le répertoire donné sur la ligne de commande (ou `."` si le chemin donné est invalide ou ne correspond pas à un répertoire) et les propriétés `long`, `column` et `sorted` valent `true` ou `false` selon que les options `-l`, `-1` et `-t` sont présentes ou absentes. Pour tester qu'un chemin existe et est un répertoire on pourra utiliser les fonctions `realpathSync` et `lstat` du module `fs` (chapitre «*File system*» de la documentation).

Écrire une fonction `main()` qui appelle `parseCmdLine` sur le tableau `process.argv` et qui affiche le résultat de cette appel dans la console au moyen de `console.log`. Ajouter en fin de fichier un appel à `main()` et tester votre programme.

2. Modifier maintenant le code de la fonction `main()` pour effectuer les actions suivantes :
 - Charger, au moyen de `fs.readdirSync` le tableau des entrées du répertoire passé sur la ligne de commande.
 - Si ce tableau est vide, quitter la fonction
 - Sinon, transformer le tableau d'entrées du répertoire en tableau de tableaux de trois cases. Pour chaque entrée `e`, calculer le tableau `[s, p, e]` où :
 - `p` est le chemin complet de l'entrée (obtenu en concaténant le répertoire listé et le nom de fichier `e`, au moyen de `path.join` documentée dans le chapitre «*Paths*»)
 - `s` est le résultat de `fs.lstatSync(p)` (un objet de type `Stat`, décrit dans la documentation du module `fs`)
 - `e` est l'entrée initiale
 - Une fois ce tableau créé, itérer dessus. Si l'option `-l` a été passée en paramètre du programme, afficher toutes les entrées séparées par un `"\n"` sinon afficher toutes les entrées séparées par un `" "`.
3. Si l'option `-t` a été passée en argument du programme, trier le tableau de triplets calculés à la question précédente selon les temps de modification décroissants (propriété `.mtime` de la structure `Stat`).

2 Affichage détaillé

1. Écrire une fonction `printStats(s, p, e)` où `s` est un objet de type `Stats`, `p` est un chemin vers le fichier correspondant à `s` et `e` est le nom du fichier au sein du répertoire. Cette fonction doit renvoyer une chaîne de caractères au format suivant :

```
drwxr-xr-x 0 0 4096 2019-02-26T16:45:52.122Z bin
lrwxrwxrwx 0 0 33 2019-03-07T08:52:33.585Z initrd.img -> boot/initrd.img-4.18.0-16-generic
-rw-r--r-- 1000 120 333 2019-02-20T15:30:26.629Z foo.txt
```

- le premier caractère vaut `d` si le fichier est un répertoire, `l` si le fichier est un lien symbolique et `-` sinon.
 - les 9 caractères suivants correspondent aux 3 permissions `read`, `write` et `execute` du fichier, pour le propriétaire, le groupe et les autres. Ces valeurs sont respectivement les bits 7-5-6, 5-4-3 et 2-1-0 de l'entier stocké dans le champ `mode` de l'objet `stat`
 - un espace et l'uid du propriétaire
 - un espace et le gid du propriétaire
 - un espace et la taille du fichier
 - un espace et la date de dernière modification (convertie en chaîne avec `.toISOString()`)
 - un espace et le nom du fichier
 - si le fichier est un lien symbolique, un espace, `->`, un espace et le chemin vers la cible (que l'on peut obtenir avec `fs.readlinkSync`)
2. Modifier la fonction `main()` pour utiliser la fonction `printStats` si l'option `-l` était présente sur la ligne de commande.

3 Résolution de l'uid et du gid

Un dernier aspect peu satisfaisant est que l'uid et le gid du propriétaire du fichier sont affichés de manière numérique. En C, la traduction entre identifiants numériques et symboliques peut être faite au moyen de la fonction `getpwuid` ou `getpwgid`. Ces fonctions n'ont pas d'équivalent dans la bibliothèque standard

node. On peut cependant utiliser la commande Unix `getent passwd xxx` pour obtenir la ligne complète concernant l'utilisateur d'uid `xxx` ou `getent group yyy` pour obtenir la ligne complète du groupe `yyy`. Par exemple, sur une machine du PUIO :

```
$ getent passwd 10093
knguye10:*:10093:933:Kim Nguyen:/home/tp-home002/knguye10:/bin/bash
```

```
$ getent group 933
Vérification d'Algorithmes, Langages et Systèmes:*:933:
```

On pourra donc procéder de la manière suivante :

1. créer un module `uid_resolve.js` (dans un autre fichier)
2. y ajouter une fonction `readFromGetent(map, value, field)` qui appelle la commande Unix `getent` sur la `map` passée en argument ("passwd" ou "group") avec la valeur d'id `value` et qui extrait du résultat de cette commande le champ `field` (les champs sont séparés par des « : »). Dans cette fonction, rattraper toute exception levée et renvoyer simplement `value` dans ce cas (i.e. si on demande le nom symbolique d'un uid "999" et que cet uid n'existe pas ou qu'une autre erreur se produit on renvoie simplement "999"). On pourra utiliser la fonction `execSync` du module `child_process` pour exécuter une commande.
3. Écrire deux fonctions `getUserName(uid)` et `getGroupName(gid)` qui appellent `readFromGetent` avec les bonnes valeurs et renvoie le nom de l'utilisateur et du groupe respectivement. Faire en sorte que ces deux fonctions soient exportées par le module.
4. charger le module `uid_resolve` dans `ls.js` et utiliser les deux fonctions exportées pour que les noms symboliques apparaissent dans l'affichage détaillé
5. Cette manière de procéder (appeler une commande avec `execSync`) est relativement couteuse. Imaginer comment ajouter un *cache* aux fonctions `getUserName` et `getGroupName` pour éviter de rappeler la fonction `readFromGetent` sur des valeurs déjà résolues précédemment.