Projet Système

ET3 - 2010/11

Table des matières

Historique

Version du **18 juin 2011**.

18 mai 2011	version initiale du document
26 mai 2011	ajout d'un listing des fonctions `C` autorisées
31 mai 2011	description d'une première *milestone*
	nouvelles fonctions `C` autorisées (manipulation du temps)
	précision sur la sortie de `-ls`
01 juin 2011	nouvelles fonctions `C` autorisées (`exec*`)
18 juin 2011	modalités de rendu

1 Modalités de rendu

Pour le rendu, créez une archive *tar* contenant au minimum :

1. Les logins UN*X des membres de votre groupe dans le fichier texte `group.txt`. Chaque login devra apparaitre seul sur sa ligne. Ne rajoutez aucune autre information dans ce fichier.
2. Les sources de votre projet dans le répertoire `src`. A moins que vous utilisiez un système d'automatisation de la compilation (t.q. *make*), votre projet devra compiler par exécution du compilateur C sur l'ensemble de vos fichiers sources.
3. Tout document pouvant aider à l'évaluation de votre projet dans le répertoire `documents`.

Envoyez ensuite l'archive à l'adresse pierre-yves@strub.nu (pour les groupes 1 **et** 2).

La date limite de rendu est le **lundi 27 juin à 17h**.

2 Description générale

On se propose d'implémenter un clone *partiel* de la commande `find`. `find` est une commande UN*X permettant de rechercher les fichiers d'un ou plusieurs répertoires (filtrés par des critères définis par l'utilisateur), et d'exécuter des opérations sur ces derniers t.q. imprimer leurs données relatives ou exécuter une commande externe.

L'utilisateur appellera `find` comme suit :

```
find [chemin1 ...] [options...] <expression....>
```

Votre programme cherchera l'ensemble des fichiers des arbres de répertoires enracinés en un des $\texttt{chemin}_i$, et évaluera, pour chacun d'entre eux, l'expression donnée (de droite à gauche, en respectant la priorité des opérateurs), jusqu'à ce que le résultat de l'expression soit connu de manière sûre (e.g. l'opérande gauche de l'opérateur *et* s'est évaluée à `false`). L'expression pourra exécuter certaines actions t.q. afficher le nom du fichier ou lancer une commande externe sur le fichier.

E.g., la commande suivante affiche le nom de tous les fichiers réguliers appartenant au répertoire courant ou à un de ses sous-répertoires, et dont le nom contient un `a`.

```
find . -name "*a*" -a -type f -a -print
```

3 Description détaillée

3.1 Prédicat

Un prédicat est une fonction d'un nom de fichier vers les booléens (`true`, `false`). Si un prédicat renvoie `true` pour un nom de fichier, on dit qu'il le *valide*. Certains prédicats peuvent prendre des paramètres supplémentaires t.q. un motif de recherche, un masque de permissions ou une commande à exécuter. Parmi les prédicats, on distingue les prédicats purs (qui n'ont qu'un rôle de filtrage), des actions (qui exécutent une action sur le fichier en cours d'examen t.q. afficher son nom ou exécuter une commande).

Les prédicats que vous devez implémenter sont donnés en section 3.3.

Dans la suite, on écrit `pred` $a_1 \cdots a_n$ pour indiquer un prédicat paramétré par les valeurs $a_1 \cdots a_n$.

3.2 Expression

Les expressions sont construites à partir des opérateurs suivants :

expr ::=	`(` expr `)`	
		`pred` $a_1 \cdots a_n$ prédicat
		`-not` expr négation
		expr1 `-a` expr2 *et* logique
		expr1 expr2 *et* logique (*alias*)
		expr1 `-o` expr2 *ou* logique
		expr1 `,` expr2 chaînage

Ils sont donnés par ordre décroissant de priorité (i.e. que e_1 `-o` e_2 `-a` e_3 se comprend e_1 `-o (` e_2 `-a` e_3 `)` et non `(` e_1 `-o` e_2 `) -a` e_3, alors que e_1 `-a` e_2 `-o` e_3 se comprend `(` e_1 `-a` e_2 `) -o` e_3 et non e_1 `-a (` e_2 `-o` e_3 `)`). Tous les opérateurs ont une associativité à gauche (i.e. que e_1 `-o` e_2 `-o` e_3 se comprend `(` e_1 `-o` e_2 `) -o` e_3 et non e_1 `-o (` e_2 `-o` e_3 `)`).

3.2.1 Évaluation

L'évaluation d'une expression, pour un nom de fichier donné, se fera comme suit :

`(` e `)`	Évalue e et retourne son résultat.
`-not` e	Évalue e et retourne la négation de son résultat.
e_1 `-a` e_2	Évalue e_1. Si e_1 s'est évaluée en `false`, retourne `false`. Sinon, évale e_2 et retourne son résultat.
e_1 `-o` e_2	Évalue e_1. Si e_1 s'est évaluée en `true`, retourne `true`. Sinon, évalue e_2 et retourne son résultat.
e_1 `,` e_2	Évalue e_1, puis e_2, et retourne le résultat de e_2.
`pred` $a_1 \cdots a_n$	Évalue le prédicat et retourne son résultat.

3.3 Définition des prédicats

Pour chaque prédicat, on donne son nom, ses paramètres, sa valeur de retour, ainsi que l'action effectuée si ce dernier n'est pas pur.

-true	arguments	*aucun*
	action	*aucune*
	résultat	`true`
-false	arguments	*aucun*
	action	*aucune*
	résultat	`false`
-name	arguments	un motif (*glob*) (cf. section A)
	action	*aucune*

	résultat	`true` si le nom de fichier (sans les répertoires en tête de chemin) correspond au motif donné en argument.
	exemples	`# Trouve tous les fichiers se terminant par .c` `find . -name '*.c'`
-type	arguments	spécification UN*X d'un type de fichier (cf. section B.4)
	action	*aucune*
	résultat	`true` si le fichier est de même type que celui donné en argument, `false` sinon.

	exemples	`# Trouve tous les sous-répertoires du répertoire courant`
		`find . -type d`
		`# Trouve tous les fichiers réguliers se terminant par .c`
		`find . -type f -name '*.c'`

-uid	arguments	un `UID` sous forme numérique
	action	*aucune*
	résultat	`true` si le fichier appartient à l'UID donné. `false` sinon.
-gid	arguments	un `GID` sous forme numérique
	action	*aucune*
	résultat	`true` si le fichier appartient au GID donné. `false` sinon.
-user	arguments	un nom d'utilisateur valide
	action	*aucune*
	résultat	`true` si le fichier appartient à l'utilisateur donné. `false` sinon.
-group	arguments	un nom de groupe valide
	action	*aucune*
	résultat	`true` si le fichier appartient au groupe donné. `false` sinon.
-atime	arguments	un entier n, possiblement préfixé par `+` (*mode plus-de*) ou par `-` (*mode moins-de*)
	action	*aucune*
	résultat	`true` si la date de dernier accès au fichier est exactement n jours dans le passé (resp. strictement plus de n jours dans le passé, strictement moins de n jours dans le passé). Pour la comparaison, lors du calcul du nombre de jours écoulés entre la date d'accès et la date courante, la partie fractionnaire est ignorée (tronquée). `false` sinon.
-ctime	arguments	un entier n, possiblement préfixé par `+` (*mode plus-de*) ou par `-` (*mode moins-de*)
	action	*aucune*
	résultat	`true` si la date de dernière modification des meta-données du fichier est exactement n jours dans le passé (resp. strictement plus de n jours dans le passé, strictement moins de n jours dans le passé). Pour la comparaison, lors du calcul du nombre de jours écoulés entre la date de dernière modification des meta-données et la date courante, la partie fractionnaire est ignorée (tronquée). `false` sinon.
-mtime	arguments	un entier n, possiblement préfixé par `+` (*mode plus-de*) ou par `-` (*mode moins-de*)
	action	*aucune*

	résultat	`true` si la date de dernière modification du fichier est exactement n jours dans le passé. (resp. strictement plus de n jours dans le passé, strictement moins de n jours dans le passé) Pour la comparaison, lors du calcul du nombre de jours écoulés entre la date de dernière modification et la date courante, la partie fractionnaire est ignorée (tronquée). `false` sinon.
-perm	arguments	permissions UN*X `perms` au format octal (cf. section B), optionnellement préfixées du caractère `/` (mode *masque*) ou `-` (mode *au-moins*).
	action	*aucune*
	résultat	En mode par défaut, retourne `true` si les permissions associées au fichier sont exactement `perms`. En mode *masque*, retourne `true` si le fichier possède au moins une des permissions présentes dans `perms`. (Dans le cas où `perms` ne contient aucune permission, i.e. `perms` = 0000, le test réussi par convention) En mode *au-moins*, retourne `true` si le fichier possède au moins l'ensemble des permissions présentes dans `perms`. Retourne `false` dans tous les autres cas.

	exemples	`# Trouve tous les fichiers réguliers exécutables`
		`# (pour le propriétaire, le groupe propriétaire ou le monde)`
		`find . -perm /111`
		`# Trouve tous les fichiers en lecture/écriture pour le propriétaire`
		`find . -perm -600`

-exec	arguments	*variables :* $[\mathrm{tk}_1 \mid \{\}] \cdots [\mathrm{tk}_n \mid \{\}]$ `;` le lexème `;` (point-virgule) marque la fin des arguments le lexème `{}` représente le fichier traité
	action	Exécute la commande $\mathrm{tk}_1 \cdots \mathrm{tk}_n$, où chaque occurence de `{}` a été remplacée par le nom du fichier traité. Le point-virgule ne fait pas parti de la commande.
	résultat	`true` si la commande a pu être exécutée et s'est terminée avec un code d'erreur (*exit status*) égal à 0. Renvoie `false` sinon.
	exemples	`# Efface tous les fichiers réguliers de la forme '*~'` `find . -type f -name '*~' -exec rm '{}' ';'`
-print	arguments	*aucun*
	action	affiche, sur la sortie standard, le nom complet du fichier.
	résultat	`true`
-ls	arguments	*aucun*
	action	affiche, sur la sortie standard, les informations listées en figure 1, chaque champ étant séparé par un unique espace.
	résultat	`true`
-prune	arguments	*aucun*
	action	si le fichier est un répertoire, ne pas y descendre pour y continuer la recherche.
	résultat	`true`

3.4 Options

Votre programme devra accepter les options suivantes :

Description	Format
numéro d'inode	base 10, min. 6 caractères, justifié à droite
	padding avec des espaces
nombre de blocs (1024o) alloués	base 10, min. 4 caractères, justifié à droite
	padding avec des espaces
type de fichier + permissions UN*X	représentation textuelle (Cf. sections B.2 et B.4)
nombre de liens matériels (*hard link*)	base 10, min. 3 caractères, justifié à droite
	padding avec des espaces
nom (login) du propriétaire⋆	min. 8 caractères, justifié à gauche
	padding avec des espaces
	base 10 si la valeur numérique (UID) est utilisée
nom du groupe propriétaire⋆	min. 8 caractères, justifié à gauche
	padding avec des espaces
	base 10 si la valeur numérique (GID) est utilisée
taille du fichier (octets)	base 10, min. 8 caractères, justifié à droite
	padding avec des espaces
date de dernière modification (dans le fuseau horaire local)	format `yyyy-mm-dd hh:mm`
nom complet du fichier	le nom complet du fichier est composé de l'ensemble des répertoires traversés depuis le chemin d'origine, ainsi que du nom du fichier. Si le fichier est un lien symbolique, vous indiquerez la cible du lien à la suite du nom complet - les deux étant séparés par "␣->␣".

⋆ - on utilise la valeur numérique de l'UID (ou du GID) lorsqu'il ne peut pas être résolu.

FIGURE 1 – Format de sortie de la commande `-ls`

-mindepth *n* n'évalue pas l'expression (et donc n'applique aucune action) avant d'atteindre un niveau de profondeur de recherche d'au moins *n*. E.g., `-minpdeth 1` empêche l'évaluation de l'expression sur les chemins donnés en argument, mais pas sur les sous-fichiers.

-maxdepth *n* borne la profondeur de recherche à *n* niveaux. E.g., `-maxdepth 0` n'évalue l'expression que sur les chemins donnés en argument.

-sort parcours les fichiers d'un répertoire par ordre alphabétique sur leur nom, sans prise en compte de la localisation.

3.5 Analyse syntaxique des expressions

`find` a une manière bien précise de prendre l'expression en argument : chaque constituant primitif (i.e. chaque lexème t.q. nom de prédicat, nom d'opérateur, ou ponctuation) de l'expression doit être

passé en tant que paramètre séparé. Ainsi, l'invocation suivante est invalide :

```
find . "-type f"
```

car `-type f` sera passé en tant qu'un seul argument à `find` au lieu de 2 (`-type` et `f`).

Lors de l'invocation suivante :

```
find . -name "*a*" -a -type f -a -print
```

votre programme recevra (en plus de son nom) 8 arguments (ici, séparés par des virgules) : `.`, `-name`, `*a*`, `-a`, `-type`, `f`, `-a` et `-print`. Il est important que votre programme ne fasse aucune analyse lexicale, i.e. aucune séparation d'un de ses arguments en constituants primitifs.

Vous devez prendre en compte la priorité et l'associativité des opérateurs.

3.6 Gestion des liens symboliques

Votre implémentation de `find` ne devra pas suivre les liens symboliques lors du parcours de l'arborescence.

3.7 Ordre des fichiers lors du parcours

Les chemins donnés en argument sont explorés séquentiellement (i.e. que votre programme commencera la recherche pour un chemin passé en argument qu'une fois l'exploration du chemin précédent terminée).

L'exploration d'un chemin se fait en profondeur (i.e. que votre programme devra explorer les fichiers fils d'un répertoire avant de passer au fichier suivant de même niveau). L'évaluation de l'expression de filtrage/actions sur un répertoire se fait avant la recherche des sous-fichiers de ce même répertoire.

Par défaut, votre implémentation de `find` devra parcourir les fichiers d'un répertoire dans l'ordre donné par le système (i.e. tels que `readdir` vous les fournit). Si l'option `-sort` est donnée, l'ordre alphabétique sur le nom, sans prise en compte de la localisation, devra être utilisé. (La non prise en compte de la localisation vous autorise l'utilisation de la fonction `strcmp`. Voir également la fonction `qsort` de la `libc`.)

3.8 Comportement aux limites

3.8.1 Expressions sans action, aucune expression

Si l'expression e passée en argument ne contient aucune action autre que `-prune`, ce dernier devra exécuter l'équivalent de `(` e `) -a -print`.

Si aucune expression n'est passée à votre programme, ce dernier devra exécuter l'équivalent de `-print`.

3.8.2 Aucun chemin

Si aucun chemin n'est donné à votre programme, celui effectuera la recherche à partir du répertoire courant. E.g., `find -ls` est équivalent à `find . -ls`.

3.9 *Exit status* de votre programme

`find` quitte avec un code d'erreur égal à 0 si aucune erreur n'a été levée lors de son exécution. Dans tous les autres cas, `find` quitte avec un code d'erreur différent de 0.

L'évaluation de l'expression en `false`, ou le fait qu'aucun fichier n'ait validé l'expression ne constituent pas une erreur.

3.10 Fonctions autorisées

Si vous pensez qu'une fonction manque, contactez nous.

- Tous les appels systèmes (section 2 des pages *man*)
- Les fonctions suivantes de la `libc` (section 3 des pages *man*) :

  gestion du tas : `malloc`, `realloc`, `free`

  manipulation des blocs mémoires et chaînes de caractères (`string.h`)
  - Famille de fonctions `str*` t.q. `strcmp`, `strcat`, ...
  - Famille de fonctions `mem*` t.q. `memcmp`, `memcpy`, ...

  accès aux codes d'erreurs (`errno.h`, `string.h`) : `errno`, `strerror`

  E/S de haut niveau (`stdio.h`)
  - type `FILE*` et famille de fonctions `f*` t.q. `fopen`, `fprintf`, ...

  accès au contenu des répertoires (`dirent.h`)
  - `opendir`, `readdir`, `closedir`, ...

  accès aux annuaires (`pwd.h`, `grp.h`)
  - `getpwnam`, `getpwuid`, ...
  - `getgrnam`, `getgruid`, ...

  manipulation du temps (`sys/time.h`, `time.h`) :
  - `localtime`, `gmtime`, `strftime`, `mktime`, ...

  sucre autour de la fonction `execve` :
  - `execvp`, `execlp`, ...

  fonction de tri : `qsort`

  fonction de globbing : `fnmatch`

  les inclassables : `exit`

3.11 Première *milestone*

Voici la description d'une première *milestone*, et ce qui constitue certainement le minimal vital à nous rendre[1] :

- `find [chemin] [predicats purs...] [action]`

  i.e. un seul chemin, pas d'option, une expression composée uniquement d'une séquence de prédicats purs (i.e. chainés avec l'opérateur *et* implicite) suivie d'une action,
- implémentation des prédicats `-true`, `-false`, `-type`, `-ctime`, `-uid` et `-gid`,
- implémentation des actions `-print`, `-ls` et `-exec`.

1. Les premières batteries de tests resteront dans le domaine de fonctionnement décrit dans cette section.

A Globbing (filtrage par motif)

Le *globbing* est une instance du filtrage par motif, pour les chaînes de caractères et notamment les noms de fichiers, qui a fait sa première apparition dans les Un*x v6. Dans sa version initiale, un motif est n'importe quelle chaîne de caractères pouvant contenir les patrons suivants :

*	représente une séquence (possiblement vide) de caractères
?	représente un unique caractère
$[c_1 \cdots c_n]$	représente un unique caractère parmi l'ensemble $\{c_1, \ldots, c_n\}$
$\backslash c$	représente le caractère c (échappement, e.g. *)

Une chaîne de caractères correspond à un motif s'il est possible de passer du motif à cette chaîne en remplaçant chaque patron par un représentant valide (cf. la table précédente).

Ainsi, n'importe quel nom de fichier ayant l'extension `.c` correspond au motif `*.c`, alors qu'au motif `*a*` correspondent tous les chaînes contenant le caractère `a`.

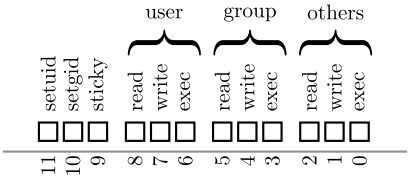
Certains de vos prédicats prendront un motif en argument. On ne vous demande pas d'implémenter la fonction de correspondance entre une chaîne de caractères et un motif. Vous pouvez utiliser la fonction `fnmatch` de la `libc`. Voir les pages `man` suivantes : fnmatch (3), glob (3) et glob (7).

B Permissions Un*x

Les droits sur un fichier Un*x s'attribuent sur trois *actions* différentes possibles : la lecture (`r`), l'écriture (`w`) et l'exécution (`x` - pour un répertoire, représente le droit de se positionner dedans). Pour chaque fichier, ces droits sont donnés pour i) le propriétaire, ii) le groupe propriétaire et iii) les autres utilisateurs. À cela, s'ajoutent trois permissions spéciales : SUID (*set user ID*, modification de l'UID à l'exécution), SGID (*set group ID*, modification du GID à l'exécution) et le sticky bit (obsolète).

B.1 Représentation octale

Il est naturel de représenter chaque permission par un bit dans un mot de 12 bits. [2]



On voit clairement que les permissions peuvent être regroupées en 4 groupes de 3 bits (permissions spéciales, du propriétaire, du groupe propriétaire et des autres). Chaque groupe de permissions code $2^3 = 8$ possibilités différentes et peut donc être représenté par un chiffre entre 0 et 7. Ainsi, l'ensemble des permissions peut être codé par 4 chiffres entre 0 et 7, i.e. par un nombre en base 8 (octal) entre 0o0000 et 0o7777 inclus. [3] Pour chaque permission isolée, on obtient les codes suivants :

2. $|\{\text{read, write, exec}\}| \times |\{\text{user, group, other}\}| + |\{\text{suid, sgid, sticky}\}| = 12$
3. La notation 0oXXXX, héritée de la syntaxe du C, indique une écriture en base 8

Permissions		Binaire	Octal
spéciaux	SUID	0b100000000000	`0o4000`
	SGID	0b010000000000	`0o2000`
	sticky	0b001000000000	`0o1000`
propriétaire	lire	0b000100000000	`0o0400`
	écrire	0b000010000000	`0o0200`
	exécuter	0b000001000000	`0o0100`
groupe proprio.	lire	0b000000100000	`0o0040`
	écrire	0b000000010000	`0o0020`
	exécuter	0b000000001000	`0o0010`
autres	lire	0b000000000100	`0o0004`
	écrire	0b000000000010	`0o0002`
	exécuter	0b000000000001	`0o0001`

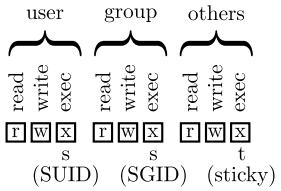
Par exemple, le nombre 0o0644 (= 0b000110100100) code les permissions de lecture pour tout le monde (propriétaire, groupe propriétaire et autres), ainsi que la permission d'écriture pour le propriétaire.

B.2 Représentation textuelle

Chaque groupe de permissions *lecture, écriture, exécution* peut être représenté par trois caractères. Le premier (resp. second, troisième) sera égal à `r` (resp. `w`, `x`) si le bit de lecture (resp. d'écriture, d'exécution) est à 1, et sera égal à `-` sinon. Ainsi, `r--` représente la permission en lecture seule, alors que `rw-` représente les permissions de lecture/écriture. Comme il y a 3 instances de ce groupe de permissions (*propriétaire, groupe propriétaire, autres*), il suffit de répéter 3 fois le principe et de concaténer le résultat, la norme étant de placer en première (resp. seconde, troisième) position les droits pour le propriétaire (resp. le groupe propriétaire, les autres). Ainsi, `rw-r--r--` code la permission de lecture pour tout le monde (propriétaire, groupe propriétaire et autres), ainsi que la permission d'écriture pour le propriétaire seul.

Reste à coder les permissions spéciales. Le SUID est codé au niveau du caractère de la permission d'exécution pour le propriétaire. Si le bit de SUID est à 1, alors le caractère d'exécution pour le propriétaire vaut `s` ou resp. `S` suivant que le bit d'exécution pour le propriétaire soit à 1 ou non. Le GID est codé de la même manière, mais en lieu et place du caractère (et de la permission) d'exécution pour le groupe. Pour le *sticky bit*, on se sert du dernier caractère d'exécution restant. Ce dernier vaut, lorsque le *sticky bit* est à 1, `t` ou resp. `T` suivant que le bit d'exécution pour les autres soit à 1 ou non.

On peut résumer le tout ainsi :



À titre d'exemple, l'ensemble de permissions qui vaut 0o755 en octal est représenté par `rwxr-xr-x`, alors que l'ensemble de permissions qui vaut 0x4755 (resp. 0x1750) en octal est représenté par `rwsr-xr-x` (resp. `rwxr-w--T`).

B.3 Permissions étendues

La plupart des systèmes de fichiers actuels propose une gestion étendue des permissions, notamment via l'utilisation d'ACL (*Access Control List*). On ne vous demande pas de gérer ce type de permissions.

B.4 Type de fichier

Les types UN*X pour les fichiers sont représentés par les caractères suivants, où le premier caractère est celui utilisé pour l'affichage (e.g. pour le prédicat `-ls`), alors que le deuxième est celui utilisé pour l'argument du prédicat `-type` :

`-`	`f`	fichier régulier
`d`	`d`	répertoire
`l`	`l`	lien symbolique
`s`	`s`	socket UN*X
`b`	`b`	device en mode blocs
`c`	`c`	device en mode caractères
`f`	`p`	FIFO

Il est courant de préfixer le type de fichier à la représentation textuelle de ses permissions UN*X. Par exemple, pour indiquer qu'un fichier est un répertoire accessible en lecture/écriture pour seulement le propriétaire, on indiquera `drwx------`, alors que `-rw-r--r--` indique un fichier régulier, non exécutable, lisible par tout le monde mais seulement modifiable par son propriétaire.

C Binaire de référence

Un binaire de référence, pour LINUX, vous est fourni. Il est fortement conseillé de comparer votre sortie à celle du binaire fourni, par exemple à l'aide de la commande `diff`.