Système Unix

42sh

Contact b-psu-050@epitech.eu

Abstract:

Table des matières

.1 Détails administratifs

- Les sources doivent être rendues sur le dépôt PSU_année_42sh
  ex : PSU_2013_42sh pour l'année 2013-2014
- **[UPDATE]** Nom du binaire : 42sh
- Votre binaire devra compiler avec un Makefile.
- Les questions sont à poser sur le forum dans le thread "42sh" du module B2 Systeme Unix.
- Seules Les réponses du responsable du module seront considérées comme officielles.
- Il vous est conseillé de travailler à plusieurs groupes et confronter vos résultats. Cependant, chaque groupe doit avoir une implémentation qui lui est propre.
- Il doit y avoir un fichier auteur avec un (et un seul) login par ligne.
- **[UPDATE]** Chaque membre du groupe doit pouvoir expliquer le fonctionnement général et la structure d'ensemble du projet, les structures utilisées, ainsi que l'utilité de chaque partie. Il doit aussi être capable d'expliquer ce qu'il a lui-même fait et pouvoir le modifier ou le refaire lors du pitch.
- Vous pouvez exclure des personnes du groupe jusqu'à 2 mois du rendu. Aucun crédit ne sera apporté à vos requètes passé ce délai. Il faut pour cela que la majorité du groupe soit en accord avec cette décision.
- Vous n'êtes pas obligé d'avoir un chef de groupe mais nous vous le conseillons fortement.
- Tout ce dont vous avez besoin doit se trouver dans le répertoire de rendu et ses sous-répertoires (rien ailleurs, même avec les droits et des paths absolus).

.2 Sujet

- Il s'agit ici d'écrire un SHELL Unix.
- Le projet est composé de 2 parties décrites ci-dessous :
  - Une partie **obligatoire**, à réaliser impérativement, notée sur 8 points
  - Une partie **optionelle**, qui ne sera prise en considération que si la partie obligatoire fonctionne dans son intégralité.
- la stabilité et l'utisabilité de l'ensemble sera largement prise en compte. Il serait souhaitable d'être conforme aux usages et habitudes.

.3 Partie obligatoire

- Cette partie doit **IMPERATIVEMENT ETRE INTEGRALEMENT FONCTIONNELLE**. Dans le cas contraire, votre note sera de 0.
- Une acquisition de ligne minimale :
  - affichage d'un prompt (plus ou moins élaboré)
  - récupération de la ligne tapée (un get_next_line(0) devrait suffir)
- Exécution des commandes avec leurs paramètres

```
ex: $>ls –l /
```

  - gestion correcte des espaces et tabulations
  - gestion du PATH (pas forcément de système de cache)
  - gestion des erreurs et de la valeur de retour

```
ex: $> ./str_maxlenoxc "ddd" "dd" "who"
    segmentation fault (core dumped)
    $>
```

- Les redirections :

```
ex: $> </etc/hosts od -c | grep xx | wc >> /tmp/z –l
```

  - « < > »
- les pipes
- Les builtins :
  - cd (avec cd seul et cd -)
  - echo
  - exit
- Les séparateurs :
  - ;
  - &&
  - ||

.4 Partie optionelle

C'est sur cette partie que vous gagnerez en théorie la majorité des points. Elle est globalement libre. Vous pouvez faire ce que vous voulez. Cependant, la cohérence de l'ensemble sera prise en compte.
Encore une fois la stabilité sera beaucoup plus importante que la quantité. N'incluez pas une option qui pose un problème au reste du programme (surtout à la partie obligatoire). Pensez avant toute chose à l'utilisabilité et la stabilité.
[UPDATE] Pour les différentes commandes et la compatibilité (syntaxe), le shell de référence utilisé sera le tcsh.

Liste d'options souhaitables :

- les inhibiteurs " '
  ex: $> ls "who|'" '"'"slt\""
- le globing * ? [ ]
  ex: $> echo {a*[^c],b??.*[a-z]}/b*.{c,h}
- le background
  ex: $> sleep 100 &
- les ‘ (magic quote)
  ex: $> kill -9 ‘ps ax | grep netscape | awk ’{print $1}’‘
- les ( )
  ex: $> (cut -d\ -f2 .note | tr ’\n’ +;echo 0)| bc -l
- les variables (local et d'env).
  ex: $> setenv a val;echo $a;ls $a;$a
- variables spéciales : term, precmd, cwdcmd, cwd, ignoreof ...
- history
  ex: $history
  - avec !
    ex: $> !ls
    ex: $> !12
    ex: $> !-4
  - avec ! et modificateur
    ex: $> !ls:s/.c/.h
  - linker avec l'édition de ligne

- alias
- édition de ligne :
  - multi ligne
  - avec rebinding dynamique
  - complétion dynamique (fichier, commande, contextuel ....)
- job control (très apprecié)
- scripting (très long)

.5 Conseils

- Conseil 1 : Formez un groupe solide
  - Vérifiez que vous pouvez vraiment travailler ensemble (heures, temps, caractères).
  - Travaillez vraiment en groupe (ensemble et en discutant).
  - Passez beaucoup de temps à analyser les choses à tous les niveaux.
  - Confrontez ensemble vos idées avant de vous lancer dans la réalisation.
  - Vérifiez que vous avez bien compris et que les autres membres de votre groupe ont compris la même chose.
  - Parlez en avec d'autres groupes.

- Conseil 2 : Avancez petit à petit
  - Ne codez rien avant que tout soit clair.
  - Ne codez rien avant d'avoir tous vos minishells totalement fonctionnels (pour tous les membres du groupe). Nous vous conseillons même de les refaire complètement en groupe, histoire de voir comment vous codez ensemble.
  - Faites des scénarios complets de fonctionnement de votre shell. Faites vous des jeux de test pour tout ce que vous comptez coder. Cherchez tous les cas de figures (nous les testerons lors de la soutenance).
  - Faites ces jeux de tests avant de commencer à coder.
  - Confrontez vos listes de cas avec les autres groupes.
  - Faites une liste claire des options que vous voulez faire en séparant bien les étapes.
  - Faites un plan général sur papier avant d'écrire la première ligne.
  - Testez tout au fur et à mesure du développement de vos fonctionnalités. N'attendez pas de l'avoir "terminée".
  - N'hésitez pas à effacer des parties qui vous semblent louches, bancales ou mal écrites (même fonctionelles). Cela vous rendra service pour la suite.
  - Ne faites rien que vous ne compreniez complètement.
  - Assemblez très souvent (au moins 1 à 2 fois par semaine) et codez à côté les uns des autres.

Quand votre projet vous semblera fonctionel, faites le tester par d'autres groupes.

.6 Fonctions autorisées

- access
- open
- read
- write
- close
- pipe
- dup
- dup2
- fork
- getpid
- getuid
- geteuid
- getgid
- vfork
- execve
- stat
- lstat
- fstat,
- getsid
- getpgid
- getpgrp
- setpgrp
- setpgid
- setsid
- tcsetpgrp
- tcsetattr
- tcgetpgrp
- tcgetattr
- isatty,
- getpwnam
- getpwent

- getpwuid
- getcwd
- chdir
- opendir
- readdir
- closedir
- glob
- signal
- kill
- wait
- waitpid
- wait3
- wait4
- les fonctions printf et derivees,
- les fonctions de string.h et strings.h
- les fonctions errno, malloc, realloc, calloc, free, bzero, memcpy, memcmp, memset, strerror, va_start, va_arg, va_list, va_end, va_copy
- tout ce qui a trait aux termcaps est autorisé (la gestion de fenêtres de la libcurses ne fait pas partie des termcaps).

.7 [UPDATE] Moulinette

- Une Moulinette corrigera une première partie du projet et vous autorisera à aller en pitch si vous passez la première étape.
- Dans le sujet se trouve un exemple type de moulinette qui sera utilisée à la correction. Nous vous encourageons à la tester sur votre projet !
- Utilisation de la moulinette :
  - Copiez le fichier moul_42sh.sh dans le répertoire où se trouve votre 42sh
  - Exécutez la moulinette : $> sh moul_24sh.sh
- Nous vous encourageons à lire cette moulinette, à étudier son fonctionnement, et surtout, à créer de nouveaux jeux de test pour celle-ci que vous pourrez vous échanger sur le forum !
- La moulinette d'évaluation finale aura, bien entendu, des tests différents de cette moulinette de démonstration