ROSTAQI Yossef
PENG Zhigang

Rapport projet SE : Création d'un démon

Ce projet avait pour but de coder la gestion d'une file d'attente (spool) et son démon qui analyse périodiquement cette file d'attente (représentée par un répertoire) en effectuant une tâche bien précise (traiter le job pour vider le spool). En général le cas du démon est illustré par l'exemple d'une imprimante qui imprime des fichiers mais dans le cadre de ce projet il nous a été demandé d'effectuer simplement d'archiver les fichiers déposés dans la file d'attente.

Pour bien installer le projet sur un poste, nous avons ajouté le bit « set user id on exec » aux exécutables pour que tout utilisateur n'ayant pas accès au spool puisse quand même utiliser le programme, comme demandé dans l'énoncé.

Le chemin vers le spool est soit indiqué via un « #define » soit la variable d'environnement « PROJETSE » existe et dans ce cas il est prioritaire sur le chemin indiqué en « #define » (les tests se font via la fonction « getenv », quelque soit le chemin indiqué les programmes supposent que les répertoires existent déjà, dans le cas échéant une erreur sur la sortie standard s'affichera.

I. Le programme « déposer »

La signature de la fonction « deposer » du fichier « deposer.c » est :

```
void deposer(char ** fichiers, int nb)
```

Elle prend en paramètres un tableau de chaînes de caractères contenant le ou les noms de fichiers à déposer dans le spool et un entier « nb » qui correspond au nombre de fichiers qui vont être déposés (dans le main la fonction est appelée avec « argv » et « argc » afin de connaître le nombre de fichiers qui vont être déposés par l'utilisateur). La fonction ne renvoie rien, elle affiche seulement l'id du fichier déposé, l'id par ailleurs est généré grâce à la fonction mkstemp qui génère un nom aléatoire à partir d'un nom de fichier finissant par 6 « X ».

Le programme crée aussi un fichier « dXXXXXX » correspondant aux métadonnées du fichier, soit le nom du fichier original déposé, la date de dépôt, la taille du fichier et le login de la personne qui l'a déposé (obtenu grâce à la fonction « getlogin() ») et aussi l'uid de l'utilisateur réel qui le dépose (ça servira au démon de traiter des fichiers qu'il ne peut pas lire).

Pour déposer un fichier dans le spool nous avons décidé d'effectuer de le faire en deux temps afin de respecter les consignes données concernant la sécurité des fichiers. En effet un utilisateur qui n'a pas accès au répertoire spool devrait quand même pouvoir y déposer ses fichiers et ce, même si les droits de son fichier restreignent la lecture à son propriétaire

exclusivement. Pour ce faire nous avons sauvegardé l'user id effectif du processus (c'est-à-dire l'user id du propriétaire de l'exécutable) dans une variable et – le temps d'effectuer une copie du fichier à déposer via un fork – nous changeons l'user id effectif du processus (grâce à la primitive système « setuid() ») en l'user id réel (c'est-à-dire la personne derrière le terminal qui exécute le programme). Une fois la copie terminée, nous restaurons l'user id sauvegardé auparavant afin d'effectuer le déplacement du fichier à déposer vers le spool.

Et comme le propriétaire de l'exécutable a lui accès au spool, le processus aura les mêmes droits que lui et donc pourra le déposer sans soucis de droits d'écriture.

Le programme respecte les consignes de verrouillage donnés, si le fichier « verrou » se trouve être verrouillé (par un autre programme accédant au spool), le programme fait un « sleep(1) » tant que le fichier verrou est verrouillé et dès que le fichier est déverrouillé, le programme « déposer » verrouille (grâce à la primitive système lockf) le fichier derrière lui et commence à effectuer la procédure permettant d'ajouter un fichier dans le spool.

Rapport valgrind :

```
yossef@DELL-XPS-15-YOSSEF:/mnt/c/Users/yosse/OneDrive/Documents/Cours/L2/S4-A/SE/Projet/finalfinalfinal$ valgrind ./deposer deposer.c
==146== Memcheck, a memory error detector
==146== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==146== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==146== Command: ./deposer deposer.c
==146==
c8FvUX
==146==
==146== HEAP SUMMARY:
==146==     in use at exit: 0 bytes in 0 blocks
==146==   total heap usage: 15 allocs, 15 frees, 3,546 bytes allocated
==146==
==146== All heap blocks were freed -- no leaks are possible
==146==
==146== For counts of detected and suppressed errors, rerun with: -v
==146== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

II. Le programme « retirer »

La signature de la fonction « retirer » du fichier « retirer.c » est :

```
void retirer(char **id, int nb)
```

Elle prend un tableau de caractères en paramètre, contenant un ou plusieurs ids de jobs à retirer du spool et un entier nb correspondant au nombre de jobs à retirer.

Pour l'implémentation : on ouvre le dossier spool (grâce au bit « set user id on exec » présent sur l'exécutable) et on supprime les fichiers correspondant aux ids stockés dans le tableau de chaînes de caractères « id » en paramètre. Sous réserve que soit on possède le spool donc on peut retirer le job de n'importe qui, soit on est un utilisateur qui veut retirer le spool qu'il a lui-même déposé. On retrouve le login de la personne qui a déposé le fichier avec un fscanf sur le fichier « dXXXXXX » et on le compare avec le « getlogin() » lancé par le programme, s'ils sont égaux les fichiers sont supprimés, sinon un message d'erreur sera affiché sur la sortie d'erreur standard.

Remarque : la fonction « getlogin() » ne fonctionne pas sur tous les postes sous linux, nous n'avons pas compris pourquoi mais dans la majorité des machines (réelles ou virtuelles) que nous avons testé, la fonction renvoyait un pointeur « null », modélisé via un fprintf par « (null) ».

Comme pour la fonction « deposer », la fonction « retirer » respecte aussi la notion de verrouillage du fichier « verrou » pour ne pas empiéter sur un autre programme manipulant lui aussi le spool.

Rapport valgrind :

```
yossef@DELL-XPS-15-YOSSEF:/mnt/c/Users/yosse/OneDrive/Documents/Cours/L2/S4-A/SE/Projet/finalfinalfinal$ valgrind ./retirer ZZXjsK
==159== Memcheck, a memory error detector
==159== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==159== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==159== Command: ./retirer ZZXjsK
==159==
==159==
==159== HEAP SUMMARY:
==159==     in use at exit: 0 bytes in 0 blocks
==159==   total heap usage: 10 allocs, 10 frees, 34,175 bytes allocated
==159==
==159== All heap blocks were freed -- no leaks are possible
==159==
==159== For counts of detected and suppressed errors, rerun with: -v
==159== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

III. Le programme « lister »

La signature de la fonction « lister » du fichier « lister.c » est :

```
void lister(int lflag, int uflag, char * user)
```

La fonction prend un entier « lflag » qui montre si oui ou non l'option « -l » a été utilisée par l'utilisateur réel, de même pour « uflag ». L'utilisation des options ont été permis grâce à la fonction « getopt ». La chaîne de caractères « user » correspond à l'argument donné par l'utilisateur s'il a choisi d'activer l'option « -u ».

Les options « -l » et « -u » ne doivent être qu'accessible au propriétaire du spool, cette vérification se fait via l'utilisation de la primitive système « stat » sur le spool et on compare le propriétaire du spool à l'utilisateur réel et on affiche un message sur la sortie d'erreur standard si les deux ne correspondent pas.

Le programme lit le spool jusqu'à la fin (le programme utilise « lockf » aussi donc le spool n'est accédé que par ce programme) et à chaque fichier « dXXXXXX » rencontré on affiche les attributs de ce job en respectant le format donné dans l'énoncé. La récupération de ces données sont simplifiés par le stockage de ces attributs dans le fichier « d » (sauf l'id que l'on retrouve grâce aux « d_name » pendant la boucle de recherche).

Le programme ne renvoie rien et quand le spool est vide le programme n'affiche rien, et si l'usage de la fonction n'est pas correct (si une option non-existante est rentrée le programme affiche comment l'utiliser et se termine).

Rapport valgrind :

```
yossef@DELL-XPS-15-YOSSEF:/mnt/c/Users/yosse/OneDrive/Documents/Cours/L2/S4-A/SE/Projet/finalfinalfinal$ valgrind ./lister
==184== Memcheck, a memory error detector
==184== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==184== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==184== Command: ./lister
==184==
DYcdsj (null)           Tue Nov  7 21:27:29 2017
rNERoj (null)           Tue Nov  7 21:21:39 2017
uNUWkF (null)           Tue Nov  7 21:27:29 2017
YCw1HX (null)           Tue Nov  7 21:27:29 2017
==184==
==184== HEAP SUMMARY:
==184==     in use at exit: 0 bytes in 0 blocks
==184==   total heap usage: 20 allocs, 20 frees, 37,823 bytes allocated
==184==
==184== All heap blocks were freed -- no leaks are possible
==184==
==184== For counts of detected and suppressed errors, rerun with: -v
==184== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

IV. Le programme « démon »

Le fichier « demon.c » contient plusieurs fonctions, l'une servant à mettre le processus en arrière-plan et l'autre servant à analyser le spool périodiquement afin de traiter les jobs dans la file d'attente (ici on décide de les mettre en archive avec la commande gzip).

La signature de la première fonction « init_daemon » :

```
void init_daemon(void)
```

Elle ne prend aucun argument et elle ne renvoie rien. Elle sert à créer le démon en utilisant un processus fils via un fork, dont on tue immédiatement le père pour le détacher du terminal courant et que le fils continue en arrière-plan.

La signature de la deuxième fonction « demon » :

```
void demon(int dflag, int iflag, int delai, char *fichlog)
```

Elle prend en paramètre des flags d'option (sauf celui de « -f » qui est géré dans la fonction main, on n'appelle pas la fonction « init_daemon » dans ce cas, pour faire tourner le programme en avant-plan).

Le « dflag » correspond à l'option qui affiche ou non des informations de débogage permettant de se situer dans le programme. Le « iflag » et son argument « delai » permettent d'effectuer un « sleep » de « delai » secondes après chaque balayage du démon.

Le « fichlog » est le fichier dans lequel le démon écrira les informations dans le format demandé par l'énoncé. Il est créé s'il n'existe pas et le remet à 0 s'il existe. Certaines informations sont reprises du fichier « dXXXXXX » et d'autres se retrouvent dans cette fonction même.

Le programme analyse périodiquement (10 secondes par défaut, « delai » secondes sinon) le spool (via un « while(1) ») pour traiter les fichiers déposés. Si un fichier n'est pas lisible par le propriétaire de l'exécutable il n'y aura pas de problème car avant d'exécuter le « gzip », on effectue un déplacement du job vers le répertoire courant (avec les droits effectifs du processus qui sont égaux à celui du propriétaire du spool), puis on s'octroie les droits de la personne ayant déposé le fichier avec un « seteuid » et en argument l'uid stocké dans le fichier « dXXXXXX ».

Le « gzip » réalisé par le biais d'un « fork » s'effectue donc sans problème de droit, on restaure les droits effectifs et on analyse l'archive créé avec « stat » pour afficher des informations le concernant dans le fichier log. On supprime ensuite le fichier « dXXXXXX » pour vider le spool de toute trace du job et on recommence après un certain délai.

Rapport valgrind :

```
yossef@DELL-XPS-15-YOSSEF:/mnt/c/Users/yosse/OneDrive/Documents/Cours/L2/S4-A/SE/Projet/finalfinalfinal$ valgrind ./demon -f log
==573== Memcheck, a memory error detector
==573== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==573== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==573== Command: ./demon -f log
==573==
^C==573==
==573== Process terminating with default action of signal 2 (SIGINT)
==573==    at 0x4F062F0: __nanosleep_nocancel (syscall-template.S:84)
==573==    by 0x4F06259: sleep (sleep.c:55)
==573==    by 0x4019D1: demon (demon.c:238)
==573==    by 0x401A9A: main (demon.c:275)
==573==
==573== HEAP SUMMARY:
==573==     in use at exit: 0 bytes in 0 blocks
==573==   total heap usage: 39 allocs, 39 frees, 42,092 bytes allocated
==573==
==573== All heap blocks were freed -- no leaks are possible
==573==
==573== For counts of detected and suppressed errors, rerun with: -v
==573== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

V. Résultat du script de test (script1.sh et script 2.sh)

- Script2.sh :

Le script a été mis à jour aujourd'hui, corrigeant les bugs reportés à la fin du rapport.

Résultat du script :

```
yossef@yossef-VirtualBox:~/Documents/OneDrive-2017-11-08$ ./script2.sh .
Cleaning: ok
Running daemon (as 'yossef'): success
1. dropping one file in spooler
   waiting for job(s) to finish..... ok
1b.checking log... ok
1c.checking 'lister -l' usage... ok
2. dropping two files simultaneously in spooler
   waiting for job(s) to finish..... ok
3. dropping two files simultaneously and removing one
   waiting for job(s) to finish..... ok
4. protecting folders and trying to drop one file
   waiting for job(s) to finish..... ok
5. checking daemon running options: ok
Stopping daemon and cleaning files: ok
```

- Script1.sh (rapport obsolète, le script2 posté d'aujourd'hui corrige ces problèmes) :

Après avoir fini l'implémentation du programme, l'exécution du script de test affichait systématiquement que la taille du fichier gzip affichée dans le fichier de log produit par « demon » était incorrecte alors qu'en réalité c'était le script qui affichait une taille pour gzip de 8 octets de moins.

La taille du fichier était récupérée via un « stat » donc on ne sait pas pourquoi le script afficherait 8 octets en moins. Peut-être que le « wc -c » omet quelque chose mais je ne m'avancerais pas plus.


```
yrostaqi@yrostaqi-VirtualBox:~/Téléchargements/final$ sh script1.sh .
Cleaning: ok
Running daemon (as 'yrostaqi'): success
Usage : lister -l -u <user>
1. dropping one file in spooler
   waiting for job(s) to finish.................................................. ok
1b.checking log... compressed file size (160) not found in log
```

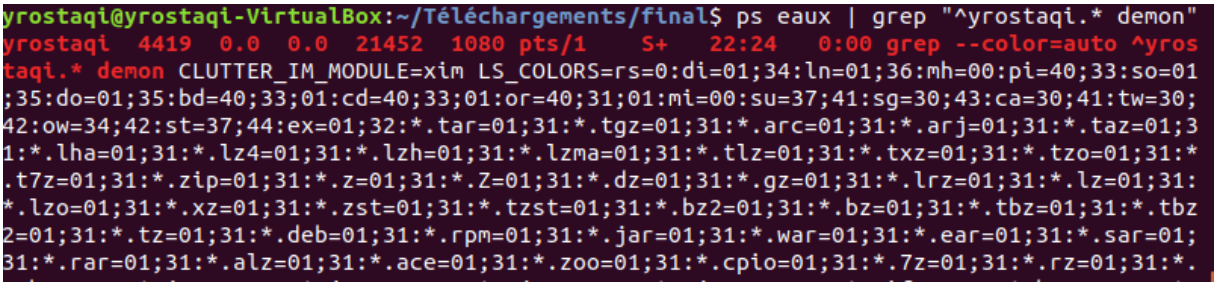
Donc on a modifié notre programme pour qu'il affiche 8 octets en moins pour qu'il passe ce test qui bloquait et nous retombions face à une erreur :

```
yrostaqi@yrostaqi-VirtualBox:~/Téléchargements/final$ sh script1.sh .
Cleaning: ok
Running daemon (as 'yrostaqi'): success
Usage : lister -l -u <user>
1. dropping one file in spooler
   waiting for job(s) to finish.................................................. ok
1b.checking log... ok
1c.checking 'lister -l' usage... ok
2. dropping two files simultaneously in spooler
   waiting for job(s) to finish.................................................. ok
3. dropping two files simultaneously and removing one
   waiting for job(s) to finish.................................................. ok
4. protecting folders and trying to drop one file
   waiting for job(s) to finish.................................................. ok
5. checking daemon running options: Usage : demon -dfi <delai> fichier_sortie
'demon -f' process not killed by 'kill 2223'
yrostaqi@yrostaqi-VirtualBox:~/Téléchargements/final$ ps -ef | grep demon
yrostaqi  2233  1036  0 19:33 pts/1    00:00:00 grep --color=auto demon
```

Là encore on ne comprenait pas parce que le kill fonctionnait très bien lorsque nous le faisions manuellement avec deux terminaux ouverts (un avec le demon qui tourne en avant-plan et un autre qui servait à le tuer).

Nous avions ouvert le script pour l'examiner et nous avons vu que le kill était bien effectué mais qu'ensuite au test suivant le script voyait toujours un processus avec le mot-clé « demon » tourner donc il affichait cette erreur (alors que pourtant il l'a bien tué, on peut le voir dans la capture d'écran ci-dessus).

Nous pensons que le test est falsifié sur nos postes car lorsque nous faisons un « ps -ef | grep demon » ou un « ps eaux | grep "^yrostaqi.* demon" » comme dans le script, le processus généré par le grep est affiché aussi :



On a donc modifié le script pour affiner le test et envoyer une erreur uniquement quand le processus n'a pas été tué :

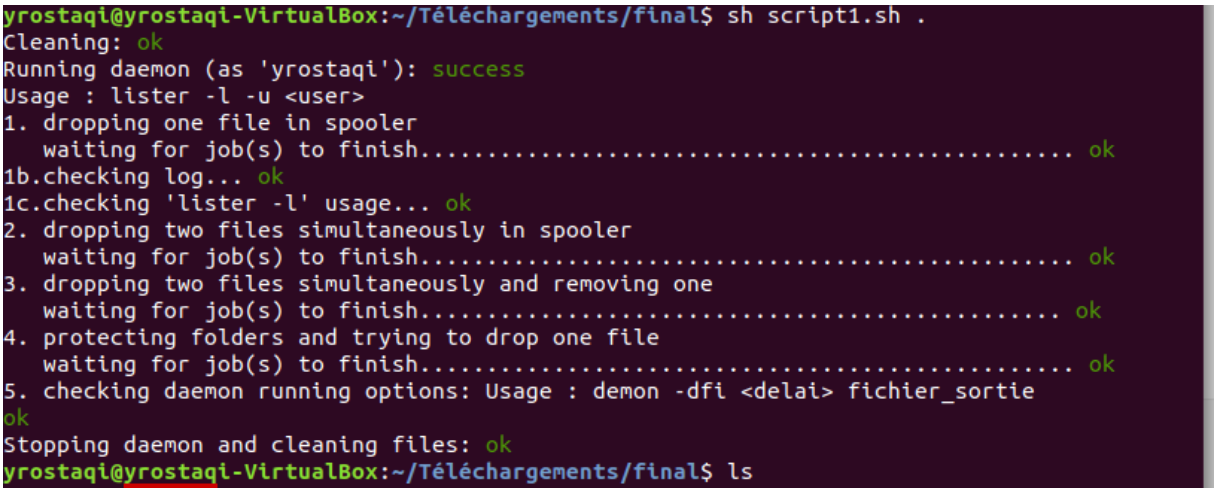
Avant :

```
# vérifie que -f lance le demon en avant-plan
pid=`( demon -f $SPOOLDIR.log >/dev/null & echo $! )`
ps eaux | grep "^$SPOOLUSER[ ]*$pid.* demon" >/dev/null \
    || plouf "No demon process (run with option -f) owned by '$SPOOLUSER'"
sudo -u $SPOOLUSER kill -9 $pid
ps eaux | grep "^$SPOOLUSER.* demon" >/dev/null \
    && plouf "'demon -f' process not killed by 'kill $pid'"
ok "ok"
```

Après :

```
# vérifie que -f lance le demon en avant-plan
pid=`( demon -f $SPOOLDIR.log >/dev/null & echo $! )`
ps eaux | grep "^$SPOOLUSER[ ]*$pid.* demon" >/dev/null \
    || plouf "No demon process (run with option -f) owned by '$SPOOLUSER'"
sudo -u $SPOOLUSER kill -9 $pid
ps eaux | grep "^$SPOOLUSER[ ]*$pid.* demon" >/dev/null \
    && plouf "'demon -f' process not killed by 'kill $pid'"
ok "ok"
```

Résultat du script :



Je doute que l'affichage de la ligne contenant « demon » lors du grep soit universel à tous les systèmes mais en tout cas sur les postes que j'ai pu tester sous Ubuntu j'ai été confronté à ce problème.

Hormis ces deux accrocs le script de test se déroule sans soucis.