Développement et mise en place de demon Unix

Dans le monde Unix, un démon est un processus qui tourne en arrière-plan, généralement pour s'acquitter d'une tâche ne nécessitant pas d'intervention de la part de l'utilisateur. Les applications " serveurs " font notamment partie de ce type de programmes. Bien que leur développement ne soit pas extrêmement complexe, il reste nécessaire, comme pour tout, de respecter un ensemble de bonnes pratiques si l'on désire faire les choses correctement. Cet article propose justement de passer en revue certaines de ces règles.

1. Préambule

Le mot " démon " vient de l'anglais DAEMON (pour Disk And Execution MONitor) et désigne un type d'application s'exécutant en arrière-plan dans le système. Sous Linux, il s'agit d'un programme conçu pour répéter de manière infinie une tâche bien spécifique.
Si l'on se réfère aux anciens numéros de GLMF, la programmation de démons Unix a déjà été traitée dans un précédent article [1]. Ce dernier contient d'ailleurs bon nombre de renseignements intéressants sur le processus de transformation d'un programme en démon. Cependant, il ne répond pas à certaines questions telles que : est-il garanti que mon démon gère correctement ses allocations en mémoire ? Ou encore : comment puis-je installer proprement mon démon ?
Ainsi, cette nouvelle étude est organisée en trois volets. Le premier est dédié à la programmation d'un démon simple. Le deuxième explique la manière de détecter les fuites de mémoire au sein de notre programme. Enfin, le troisième revient sur l'installation d'un démon dans un système GNU/Linux.

1.1 Outils nécessaires

Dans le cadre de cet article, nous aurons besoin d'un certain nombre d'outils. Parmi ceux-ci, on peut compter :

- **gcc**, disponible dans les dépôts de votre distribution ou ici http://gcc.gnu.org/ ;
- **valgrind**, également disponible dans les dépôts de votre distribution ou ici http://valgrind.org/ ;
- la commande **update-rc.d**, disponible dans le paquet **sysv-rc** des dépôts Debian, pour l'installation du démon.

Une fois que tous ces programmes sont présents sur votre système, il est temps de commencer à parler du démon.

1.2 Fonctionnalités de notre démon

Première question : quelle doit être l'action de notre programme ?
Dans notre cas, nous prendrons un exemple assez simple. Nul besoin en effet de coder un nouvel Apache pour ce que nous voulons observer. Ainsi, la principale activité de notre démon sera de nettoyer un répertoire choisi à l'avance. Par " nettoyage ", on entend suppression des fichiers de sauvegarde d'éditeurs de texte que le dossier contient (Vous savez ? Les petites choses comme **mon_fichier~** que Gedit, ou autre, crée après que vous avez modifié **mon_fichier**.). Ce nettoyage sera répété en permanence toutes les secondes.
Afin de suivre l'évolution de notre démon, nous le ferons stocker ses messages dans le journal système **/var/log/messages** sous la forme " mydeamon: <le_message> ". On trouvera en particulier

un message " mydeamon: le démon vient de naître. Mouahahaha !! " et un message " mydeamon: commande ‘rm -f <dossier_a_nettoyer>/*~’ exécutée " (où " <dossier_a_nettoyer> " correspond au dossier dont on veut supprimer les fichiers de sauvegarde). Ce dernier sera répété toutes les secondes après chaque exécution de la commande.

2. Développement de la bête

Notre démon est des plus simples et ne comporte que deux fonctions. La première est, bien entendu, la fonction **main()** qui contient le code de l’activité principale du démon (celle qui sera exécutée en boucle infinie). La seconde fonction sert à initialiser le démon. Nous l’appellerons **demoniser()** et commencerons par étudier celle-là.

2.1 Transformation du processus en démon

Un démon n’est pas un programme comme les autres. Il nécessite ainsi une phase d’initialisation. Cette section a déjà été traitée en profondeur dans l’article sur la transformation d’un processus en démon [1]. Nous nous contenterons donc d’une fonction relativement sommaire, initialisant correctement notre démon sans pour autant être trop sophistiquée.
Le code suivant décrit cette fonction de transformation :

```
/**
* Initialise le démon.
*/
static void demoniser()
{
    pid_t pid; /* PID après le fork() */
```

Dans la pratique, lorsqu’un processus se retrouve orphelin, c’est-à-dire quand son processus père est tué, l’enfant se retrouve immédiatement rattaché au processus d’**init** (de PID égal à 1), qui devient alors son nouveau parent. Ainsi, une manière classique et élégante de lancer un démon est d’effectuer un appel à la primitive **fork()** dans son code. Cet appel crée un second processus, fils de celui ayant exécuté le **fork()**. Il ne reste plus qu’à suicider le père pour que le fils se retrouve orphelin et soit rattaché à **init**.
On commence par vérifier si le processus s’exécutant est déjà un démon. On considère ce prédicat comme vrai si son processus père est déjà init. Si c’est le cas, on quitte la fonction.

```
    if(getppid() == 1) return;
```

Ce n’est pas le cas, on duplique donc le processus appelant grâce à la primitive **fork()**. Le PID que nous retourne cette fonction permet de détecter le père et de le tuer afin que le fils soit adopté par **init**.

```
     pid = fork();
    if(pid < 0)
    {
        syslog(LOG_ERR,”le ‘fork()’ a echoue, errno = %d
(%s).”,errno,strerror(errno));
        exit(EXIT_FAILURE);
    }
```

```
    /* On quitte le processus père. */
    if(pid > 0) exit(EXIT_SUCCESS);
```

À ce stade, le père a été tué et l'on exécute le processus fils. On donne tous les droits en lecture/écriture au démon sur les fichiers qu'il crée.

```
     umask(0222);
```

On crée une nouvelle session pour que le processus fils ne reçoive pas les signaux destinés au père.

```
            if(setsid() < 0)
    {
        syslog(LOG_ERR,"le 'setsid()' a echoue, errno = %d
(%s).",errno,strerror(errno));
        exit(EXIT_FAILURE);
    }
```

On change le répertoire par la racine afin de pouvoir démonter la partition à partir de laquelle a été démarré le démon.

```
            if((chdir("/")) < 0)
    {
        syslog(LOG_ERR,"le 'chdir()' a echoue, errno = %d
(%s).",errno,strerror(errno));
        exit(EXIT_FAILURE);
    }
```

On redirige les E/S standards vers **/dev/null** afin d'éviter, entre autres, toute sortie intempestive vers le terminal de l'utilisateur.

```
    freopen("/dev/null", "r", stdin);
    freopen("/dev/null", "w", stdout);
    freopen("/dev/null", "w", stderr);
```

Le démon vient de naître !

```
            syslog(LOG_INFO,"le demon vient de naitre. Mouahahaha !!");
} /* Fin de demoniser() */
```

2.2 Action principale du démon

La fonction **main()** est, quant à elle, encore plus simple. L'intérêt du démon est d'exécuter un même bout de code en tâche de fond et en permanence. C'est la raison pour laquelle nous trouvons une boucle infinie au sein de cette procédure. L'activité principale du démon est à placer entre les deux accolades du **for(;;)**.

```
/* Nombre de secondes de sommeil du démon */
#define NB_SEC_SOMMEIL    1
```

```
...
int main(void)
{
    char * chaineCommande;
    demoniser(); /* On initialise le démon. */

    /* Boucle infinie dans laquelle
    on trouve l'action du démon */
    for(;;)
    {
        ...
        ...
        sleep(NB_SEC_SOMMEIL);
    }
    return 0;
}
```

Pour l'instant, nous décidons de laisser cette partie vide et de passer à la correction des fuites de mémoire.

3. Les fuites de mémoires

3.1 Le cas particulier des démons

Un problème récurrent lorsque l'on programme des démons en C est le memory leak.
Les systèmes d'exploitation permettent aux programmes en cours d'exécution de réserver des zones mémoires pour leur fonctionnement. Quand un programme effectue cette opération (en C : avec la fonction **malloc()/calloc()**), la zone mémoire qu'il vient de réserver devient inaccessible aux autres programmes s'exécutant sur le système. En général, avant de terminer son exécution, le programme libère la zone mémoire (en C : avec la fonction **free()**) qu'il a réservée auparavant ; il se ferme, il n'y a pas alors de fuite de mémoire. Cependant, il arrive parfois qu'un développeur peu consciencieux ne fasse pas libérer la mémoire par son programme. Ainsi, lorsque celui-ci se termine, la mémoire lui est toujours allouée : il y a alors un memory leak. Si plusieurs programmes font la même chose ou si le programme précédent est exécuté plusieurs fois, la mémoire du système finit par être entièrement réservée : on obtient alors soit un plantage de tous les programmes voulant réserver de la mémoire à leur tour, soit un plantage du système.
Les systèmes d'exploitation récents sont relativement bien conçus et font en sorte que dès qu'un programme se termine, l'espace mémoire qu'il a pu réserver pendant son fonctionnement soit libéré. Le problème est qu'un démon n'est, en général, pas censé se terminer, sachant que dans la plupart des cas il s'agit d'une boucle infinie. Autrement dit, lorsque l'on trouve un memory leak dans l'un d'entre eux, il est nécessaire de le corriger immédiatement, car sinon il finira par saturer la totalité de la mémoire disponible et par planter le système.

3.2 Mise en œuvre dans notre exemple

Reprenons maintenant le code de notre démon et modifions la fonction **main()** de la manière suivante :

```
/* Dossier à nettoyer de ses *~ */
#define DOSSIER_A_NETTOYER    "/home/user/test"
```

```
...
int main(void)
{
    char * chaineCommande;
    demoniser(); /* On initialise le démon. */

    /* Boucle infinie dans laquelle
    on trouve l'action du démon */
    for(;;)
    {
        chaineCommande = calloc(1,256);
        strcat(chaineCommande,"rm -f ");
        strcat(chaineCommande,DOSSIER_A_NETTOYER);
        strcat(chaineCommande,"/*~");
        system(chaineCommande);
        syslog(LOG_INFO,"commande '%s' executee.",chaineCommande);
        sleep(NB_SEC_SOMMEIL);
    }
    return 0;
}
```

Comme vous pouvez le constater, en début de boucle, on réserve 256 octets de mémoire afin de stocker la chaîne de caractères de la commande à exécuter. Bien entendu, il n'est pas forcément nécessaire de jouer avec l'allocation mémoire à chaque fois que l'on veut traiter une suite de caractères (Il eut d'ailleurs été bien plus pratique de déclarer **chaineCommande** de manière statique.). Cependant, nous utilisons volontairement cette méthode dans le cadre de notre exemple afin de montrer les risques d'une mauvaise gestion de la mémoire. L'espace réservé pour le stockage de la chaîne devrait être libéré à l'aide d'un appel à **free**() à la fin de la boucle. Or, là, il ne l'est pas. Si l'on compile à nouveau ce programme et qu'on l'exécute, il fonctionne tout à fait correctement. La commande est bien exécutée à chaque tour de boucle. Le seul bémol se trouve dans l'occupation mémoire du programme. En effet, avec le temps, celle-ci a tendance à grossir. Cela peut se voir facilement en utilisant ce simple script :

```
#!/bin/sh

# Utilisation : ./checkMem.sh <PID>
# <PID> : PID du processus dont on
# veut mesurer l'utilisation mémoire.

while true;
do
    # Dort pendant 5 secondes.
    sleep 5;
    mem=`awk '{print $11 }' < /proc/$1/stat`;
    echo "$mem bytes used";
done;
```

Ces quelques lignes permettent de visualiser l'espace mémoire (en octets) occupé par le processus dont on passe le PID en paramètre. Cette visualisation se fait, ici, toutes les 5 secondes.
Copiez ce code dans un fichier, appelez ce dernier **checkMem.sh**, rendez-le exécutable à l'aide de la commande **chmod +x checkMem.sh**. Enfin, pour vérifier l'occupation mémoire de votre démon, démarrez celui-ci et exécutez **checkMem.sh** en lui passant en paramètre le PID du processus de votre démon. Laissez le script tourner quelque temps. Si le nombre affiché augmente globalement au cours du temps, il est probable que votre programme comporte une fuite de mémoire.

3.3 Correction

Pour résoudre ce problème, nous allons utiliser un outil du nom de **valgrind**. Ce petit logiciel devient fort utile lorsqu'il s'agit de déboguer une application sous Linux. Il possède notamment un module du nom de **memcheck** qui permet, entre autres, de détecter les fuites de mémoire.
Son utilisation est relativement aisée et nous allons voir tout de suite comment nous en servir. Néanmoins, avant, il convient d'apporter juste une petite modification au code. En effet, pour détecter les fuites de mémoire, **valgrind** exécute le binaire. Comme il ne veut quand même pas travailler indéfiniment là-dessus, il ne traite pas les boucles infinies. Donc, il suffit juste de changer le **for(;;)** en **for(i=0; i<10; i++)** par exemple. On compile à nouveau le programme et l'on peut enfin utiliser **valgrind**.
Pour ce faire, tapez la commande **valgrind --leak-check=full ./mydeamon** (où **./mydeamon** correspond au binaire du démon). On obtient le résultat suivant :

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
malloc/free: in use at exit: 2,560 bytes in 10 blocks.
malloc/free: 85 allocs, 75 frees, 99,483 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 10 not-freed blocks.
checked 59,372 bytes.
2,560 bytes in 10 blocks are definitely lost in loss record 1 of 1
   at 0x402095F: calloc (vg_replace_malloc.c:279)
   by 0x8048875: main (in /home/path/mydeamon)
LEAK SUMMARY:
   definitely lost: 2,560 bytes in 10 blocks.
     possibly lost: 0 bytes in 0 blocks.
   still reachable: 0 bytes in 0 blocks.
        suppressed: 0 bytes in 0 blocks.
```

Les lignes particulièrement intéressantes sont en rouge. On note tout d'abord qu'il y a eu 85 allocations de mémoire pour seulement 75 libérations. Idéalement, ces deux nombres devraient être égaux. À la fin, il est également dit que 2560 octets ont été définitivement perdus, ce qui correspond, dans notre cas, aux 10 tours de boucle allouant chacun 256 octets de mémoire.

Au milieu de tout cela, **valgrind** nous indique les emplacements des **malloc()/calloc()** dont les zones mémoires n'ont pas été libérées. Ici, nous voyons que, pour les 2560 octets, il s'agit des 10 **calloc()** exécutés par la fonction **main()**.

Imaginons un instant que l'on remette la boucle infinie pour un tel programme. Un rapide calcul nous indique qu'un serveur possédant 512 mégaoctets de RAM verrait sa mémoire saturée et nécessiterait un redémarrage en environ 24 jours (2097152 secondes exactement) s'il venait à laisser notre démon tourner. Cela en considérant, bien entendu, que la petite fuite de mémoire présentée ici soit la seule au sein du système.

Afin de résoudre ce problème, il suffit d'ajouter un appel à la primitive free() dans notre programme.

```
int main(void)
{
    ...
    for(;;)
    {
        ...
        syslog(LOG_INFO,"commande '%s' executee.",chaineCommande);
        free(chaineCommande);
```

```
        sleep(NB_SEC_SOMMEIL);
    }
    return 0;
}
```

Si maintenant on lance à nouveau **valgrind**, les fuites de mémoires ont disparu. Notre démon est débogué : il ne reste plus qu'à l'installer.

4. Installation du démon

Maintenant que notre démon fonctionne et que sa gestion de la mémoire a été corrigée, il ne nous reste plus qu'à l'installer au sein du système. Par " installation ", nous entendons trois choses : intégration dans le système, lancement au démarrage de l'ordinateur, et script de contrôle.

Commençons par installer le binaire. Notre démon n'a pas, a priori, besoin d'être exécuté en root (c'est même relativement déconseillé). Ainsi, nous contenterons-nous de le copier dans **/usr/bin**.

Afin de faciliter la vie aux utilisateurs de notre application, nous allons créer un script de contrôle que nous placerons dans **/etc/init.d**. Ce script doit permettre de démarrer, d'arrêter, de redémarrer le démon. On veut également connaître son statut (s'il est en cours d'exécution ou pas).

La rédaction d'un tel script requiert, elle aussi, le respect de certaines bonnes pratiques. Il est possible d'en trouver un exemple dans le fichier **/etc/init.d/skeleton** sous Debian/Ubuntu/... Le nôtre sera cependant moins compliqué comme vous pouvez en juger :

```
#!/bin/bash
PATH=/bin:/usr/bin:/sbin:/usr/sbin
# Optionnel : définit le nom qui s'affichera
# lors du démarrage ou de l'arrêt du du démon
NAME=mydaemon
# Emplacement du binaire du démon
DAEMON=/usr/bin/mydaemon
# On vérifie que le programme du démon
# est exécutable, sinon fin du script.
test -x $DAEMON || exit 0
case "$1" in
    start)
        # On vérifie si le démon
        # n'est pas déjà lancé.
        if [ -z "$(ps -A | grep $NAME)" ]
        then
            echo "$NAME : lancement du démon."
            $NAME & >& /dev/null
        fi
        ;;
    stop)
        echo "$NAME : arrêt du démon."
        killall -9 $NAME >& /dev/null
        ;;
    restart)
        $0 stop
        sleep 1
        $0 start
        ;;
    status)
        echo -n "$NAME : "
        if [ -z "$(ps -A | grep $NAME)" ]
```

```
        then
            echo “le démon n’est pas lancé."
        else
            echo "le démon est lancé."
        fi
        ;;
    *)
        echo "Usage: $0 start|stop|restart|status" >&2
        exit 1
        ;;
esac
exit 0
## Fin du script !
```

Appelons-le **myapplid** (" d " pour " daemon ") et copions-le dans **/etc/init.d**.

Il est maintenant possible d’exécuter les commandes suivantes :

- **/etc/init.d/myapplid** start pour démarrer le démon ;
- **/etc/init.d/myapplid** stop pour arrêter le démon ;
- **/etc/init.d/myapplid** restart pour le redémarrer ;
- **/etc/init.d/myapplid** status afin de savoir s’il est démarré ou non.

Une fois cela fait, il ne nous reste plus qu’à offrir la possibilité de lancer notre démon au démarrage de l’ordinateur. De même, il est nécessaire de le tuer à son extinction.

Pour définir cela, il faut indiquer au système ce qu’il doit faire du démon pour les différents niveaux d’exécution (ou runlevel) du système. Dans notre cas, nous voulons qu’il l’arrête en cas d’arrêt/redémarrage du système (niveaux 0 et 6) et au niveau mono-utilisateur (niveau 1). Pour tous les autres niveaux, nous voulons qu’il le démarre.

Tout cela se fait tout simplement à l’aide de la commande suivante : **update-rc.d myapplid start 30 2 3 4 5 . stop 10 0 1 6** .

Faites bien attention, les points " . " juste avant le " stop " et à la fin de la commande sont obligatoires.

Nous n’expliquerons pas ici les subtilités de la commande **update-rc.d**. Il faut juste savoir que cela demande au système de démarrer en position 30 (c’est-à-dire après beaucoup d’autres applications) le démon pour les runlevels de démarrage, et de le tuer en position 10 (c’est-à-dire avant beaucoup d’autres applications) pour les runlevels d’arrêt/redémarrage/... Plus de renseignements sont disponibles en [2] et à travers la page de **man** de la commande **update-rc.d**.

L’exécution de ladite commande devrait nous retourner la chose suivante :

```
 Adding system startup for /etc/init.d/myapplid ...
   /etc/rc0.d/K10myapplid -> ../init.d/myapplid
   /etc/rc1.d/K10myapplid -> ../init.d/myapplid
   /etc/rc6.d/K10myapplid -> ../init.d/myapplid
   /etc/rc2.d/S30myapplid -> ../init.d/myapplid
   /etc/rc3.d/S30myapplid -> ../init.d/myapplid
   /etc/rc4.d/S30myapplid -> ../init.d/myapplid
   /etc/rc5.d/S30myapplid -> ../init.d/myapplid
```

Comme nous l’avions dit, il est prévu de tuer en position 10 (K10) notre démon aux niveaux 0, 1, 6, et de le démarrer en position 30 (S30) aux niveaux 2 à 5.

Si l'on redémarre maintenant le système, une des premières lignes que nous obtenons est " mydaemon : arrêt du démon. ". De même, au redémarrage, une des dernières lignes qui s'affiche est " mydaemon : lancement du démon. ". Cela à condition, bien sûr, d'avoir, au préalable, désinstallé toute application de démarrage/extinction graphique du système.

Notre démon est donc correctement débogué et installé. Si l'on va voir dans /home/user/test (ou le dossier que vous avez spécifié avant la compilation de votre code), les fichiers terminés par un tilde (" ~ ") sont régulièrement supprimés. Vous pouvez essayer d'en créer dans le dossier pour tester le résultat.

Conclusion

Il serait illusoire de prétendre couvrir tout ce qui concerne le développement et l'installation des démons Unix en quelques pages. Le lecteur pourra cependant attester que le sujet a été traité de manière globale, afin de montrer les grandes étapes de la mise en place de ces démons.

Ainsi, nous avons commencé par montrer comment donner proprement le statut de service à une application. Nous avons également mis en exergue le danger des fuites de mémoire au sein d'un tel type de programmes, et la manière de colmater ces fuites. Enfin, nous nous sommes attardés sur l'intégration du démon dans un système GNU/Linux Debian.

Le développement de serveurs complets à l'aide de démons est, bien entendu, beaucoup plus complexe. Cet article pose cependant les bases de leur programmation. Approfondira qui voudra.