Daemon et script init

Gestion des comptes utilisateur

TD 4 – Jérémy Briffaut

1 Daemon et script initilisation

1.1 Ecriture d'un Daemon en C

A daemon (or service) is a background process that is designed to run autonomously,with little or not user intervention. The Apache web server http daemon (httpd) is one such example of a daemon. It waits in the background listening on specific ports, and serves up pages or processes scripts, based on the type of request.

1.2 Basic Daemon Structure

When a daemon starts up, it has to do some low-level housework to get itself ready for its real job. This involves a few steps:

- Fork off the parent process
- Change file mode mask (umask)
- Open any logs for writing
- Create a unique Session ID (SID)
- Change the current working directory to a safe place
- Close standard file descriptors
- Enter actual daemon code

1.3 Forking The Parent Process

A daemon is started either by the system itself or a user in a terminal or script. When it does start, the process is just like any other executable on the system. To make it truly autonomous, a *child process* must be created where the actual code is executed. This is known as forking, and it uses the *fork()* function:

```
pid_t pid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}
```

Notice the error check right after the call to *fork()*. When writing a daemon, you will have to code as defensively as possible. In fact, a good percentage of the total code in a daemon consists of nothing but error checking.

The *fork()* function returns either the process id (PID) of the child process (not equal to zero), or -1

on failure. If the process cannot fork a child, then the daemon should terminate right here.

If the PID returned from *fork()* did succeed, the parent process must exit gracefully. This may seem strange to anyone who hasn't seen it, but by forking, the child process continues the execution from here on out in the code.

1.4 Changing The File Mode Mask (Umask)

In order to write to any files (including logs) created by the daemon, the file mode mask (umask) must be changed to ensure that they can be written to or read from properly. This is similar to running umask from the command line, but we do it programmatically here. We can use the *umask()* function to accomplish this:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        /* Log failure (use syslog if possible) */
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);
```

By setting the umask to 0, we will have full access to the files generated by the daemon. Even if you aren't planning on using any files, it is a good idea to set the umask here anyway, just in case you will be accessing files on the filesystem.

1.5 Opening Logs For Writing

This part is optional, but it is recommended that you open a log file somewhere in the system for writing. This may be the only place you can look for debug information about your daemon.

1.6 Creating a Unique Session ID (SID)

From here, the child process must get a unique SID from the kernel in order to operate. Otherwise, the child process becomes an orphan in the system. The pid_t type, declared in the previous section, is also used to create a new SID for the child process:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
```

```
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
        /* Log any failure */
        exit(EXIT_FAILURE);
}
```

Again, the *setsid()* function has the same return type as *fork()*. We can apply the same error-checking routine here to see if the function created the SID for the child process.

1.7 Changing The Working Directory

The current working directory should be changed to some place that is guaranteed to always be there. Since many Linux distributions do not completely follow the Linux Filesystem Hierarchy standard, the only directory that is guaranteed to be there is the root (/). We can do this using the *chdir()* function:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}

/* Change the current working directory */
if ((chdir("/")) < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}
```

Once again, you can see the defensive coding taking place. The *chdir()* function returns -1 on failure, so be sure to check for that after changing to the root directory within the daemon.

1.8 Closing Standard File Descriptors

One of the last steps in setting up a daemon is closing out the standard file descriptors (STDIN, STDOUT, STDERR). Since a daemon cannot use the terminal, these file descriptors are redundant

and a potential security hazard.

The *close()* function can handle this for us:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
      exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
      exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
      /* Log any failure here */
      exit(EXIT_FAILURE);
}

/* Change the current working directory */
if ((chdir("/")) < 0) {
      /* Log any failure here */
      exit(EXIT_FAILURE);
}


/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```

It's a good idea to stick with the constants defined for the file descriptors, for the greatest portability between system versions.

1.9 Writing the Daemon Code

1.9.1 Initialization

At this point, you have basically told Linux that you're a daemon, so now it's time to write the actual daemon code. Initialization is the first step here. Since there can be a multitude of different functions that can be called here to set up your daemon's task, I won't go too deep into here.

The big point here is that, when initializing anything in a daemon, the same defensive coding guidelines apply here. Be as verbose as possible when writing either to the syslog or your own logs. Debugging a daemon can be quite difficult when there isn't enough information available as to the status of the daemon.

1.9.2 The Big Loop

A daemon's main code is typically inside of an infinite loop. Technically, it isn't an infinite loop, but

it is structured as one:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
      exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
      exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
      /* Log any failures here */
      exit(EXIT_FAILURE);
}


/* Change the current working directory */
if ((chdir("/")) < 0) {
      /* Log any failures here */
      exit(EXIT_FAILURE);
}

/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

/* Daemon-specific initialization goes here */

/*The Big Loop */
while (1) {
   /* Do some task here ... */
   sleep(30); /* wait 30 seconds */
}
```

This typical loop is usually a *while* loop that has an infinite terminating condition, with a call to *sleep* in there to make it run at specified intervals.

Think of it like a heartbeat: when your heart beats, it performs a few tasks, then waits until the next beat takes place. Many daemons follow this same methodology.

1.10 Complete Sample

Listed below is a complete sample daemon that shows all of the steps necessary for setup and execution. To run this, simply compile using gcc, and start execution from the command line. To terminate, use the *kill* command after finding its PID.

I've also put in the correct include statements for interfacing with the syslog, which is recommended at the very least for sending start/stop/pause/die log statements, in addition to using your own logs with the *fopen()*/*fwrite()*/*fclose()* function calls.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>

int main(void) {

        /* Our process ID and Session ID */
        pid_t pid, sid;

        /* Fork off the parent process */
        pid = fork();
        if (pid < 0) {
                exit(EXIT_FAILURE);
        }
        /* If we got a good PID, then
           we can exit the parent process. */
        if (pid > 0) {
                exit(EXIT_SUCCESS);
        }

        /* Change the file mode mask */
        umask(0);

        /* Open any logs here */

        /* Create a new SID for the child process */
        sid = setsid();
        if (sid < 0) {
                /* Log the failure */
                exit(EXIT_FAILURE);
        }



        /* Change the current working directory */
        if ((chdir("/")) < 0) {
                /* Log the failure */
                exit(EXIT_FAILURE);
        }

        /* Close out the standard file descriptors */
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);

        /* Daemon-specific initialization goes here */

        /* The Big Loop */
        while (1) {
           /* Do some task here ... */

           sleep(30); /* wait 30 seconds */
        }
   exit(EXIT_SUCCESS);
}
```

1.11 Complete Sample with Logging

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <syslog.h>
#include <errno.h>
#include <pwd.h>
#include <signal.h>

/* Change this to whatever your daemon is called */
#define DAEMON_NAME "mydaemon"

/* Change this to the user under which to run */
#define RUN_AS_USER "daemon"

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1


static void child_handler(int signum)
{
    switch(signum) {
    case SIGALRM: exit(EXIT_FAILURE); break;
    case SIGUSR1: exit(EXIT_SUCCESS); break;
    case SIGCHLD: exit(EXIT_FAILURE); break;
    }
}

static void daemonize( const char *lockfile )
{
    pid_t pid, sid, parent;
    int lfp = -1;

    /* already a daemon */
    if ( getppid() == 1 ) return;

    /* Create the lock file as the current user */
    if ( lockfile && lockfile[0] ) {
        lfp = open(lockfile,O_RDWR|O_CREAT,0640);
        if ( lfp < 0 ) {
            syslog( LOG_ERR, "unable to create lock file %s, code=%d (%s)",
                    lockfile, errno, strerror(errno) );
            exit(EXIT_FAILURE);
        }
    }

    /* Drop user if there is one, and we were run as root */
    if ( getuid() == 0 || geteuid() == 0 ) {
        struct passwd *pw = getpwnam(RUN_AS_USER);
        if ( pw ) {
            syslog( LOG_NOTICE, "setting user to " RUN_AS_USER );
            setuid( pw->pw_uid );
        }
    }

    /* Trap signals that we expect to recieve */
    signal(SIGCHLD,child_handler);
    signal(SIGUSR1,child_handler);
    signal(SIGALRM,child_handler);

    /* Fork off the parent process */
    pid = fork();
    if (pid < 0) {
        syslog( LOG_ERR, "unable to fork daemon, code=%d (%s)",
                errno, strerror(errno) );
        exit(EXIT_FAILURE);
    }
    /* If we got a good PID, then we can exit the parent process. */
    if (pid > 0) {

        /* Wait for confirmation from the child via SIGTERM or SIGCHLD, or
```

```
          for two seconds to elapse (SIGALRM).  pause() should not return. */
        alarm(2);
        pause();

        exit(EXIT_FAILURE);
    }
    /* At this point we are executing as the child process */
    parent = getppid();
    /* Cancel certain signals */
    signal(SIGCHLD,SIG_DFL); /* A child process dies */
    signal(SIGTSTP,SIG_IGN); /* Various TTY signals */
    signal(SIGTTOU,SIG_IGN);
    signal(SIGTTIN,SIG_IGN);
    signal(SIGHUP, SIG_IGN); /* Ignore hangup signal */
    signal(SIGTERM,SIG_DFL); /* Die on SIGTERM */

    /* Change the file mode mask */
    umask(0);

    /* Create a new SID for the child process */
    sid = setsid();
    if (sid < 0) {
        syslog( LOG_ERR, "unable to create a new session, code %d (%s)",
                errno, strerror(errno) );
        exit(EXIT_FAILURE);
    }

    /* Change the current working directory.  This prevents the current
       directory from being locked; hence not being able to remove it. */
    if ((chdir("/")) < 0) {
        syslog( LOG_ERR, "unable to change directory to %s, code %d (%s)",
                "/", errno, strerror(errno) );
        exit(EXIT_FAILURE);
    }

    /* Redirect standard files to /dev/null */
    freopen( "/dev/null", "r", stdin);
    freopen( "/dev/null", "w", stdout);
    freopen( "/dev/null", "w", stderr);

    /* Tell the parent process that we are A-okay */
    kill( parent, SIGUSR1 );
}

int main( int argc, char *argv[] ) {
    /* Initialize the logging interface */
    openlog( DAEMON_NAME, LOG_PID, LOG_LOCAL5 );
    syslog( LOG_INFO, "starting" );

    /* One may wish to process command line arguments here */

    /* Daemonize */
    daemonize( "/var/lock/subsys/" DAEMON_NAME );

    /* Now we are a daemon -- do the work for which we were paid */
    while(1){
        syslog( LOG_NOTICE, "alive" );
        sleep(10);
    }
    /* Finish up */
    syslog( LOG_NOTICE, "terminated" );
    closelog();
    return 0;
}
```

1.12 Exercices

Le code de ces deux exemples est disponible sur le serveur enseignement (**2ASTIADMINSYS**) dans le fichier td/td4.zip

Question 1 : tester le daemon daemon2.c

Que fait ce daemon?

Comment le compiler?

Comment l'exécuter, vérifier qu'il fonctionne?, le tuer ?

Comment le mettre en service système au démarrage? (pour cela aider vous du fichier ***script-init-exemple***)

Question 2 : Seveur tcp d'impression de fichier

En vous aidant du fichier daemon2.c, écrire un serveur tcp qui écoute sur le port 7000, reçoit un chemin correspondant à un fichier et renvoi les 1024 premier octect de ce fichier.

Votre serveur devra pouvoir prendre en compte plusieurs connections.

Il devra géré correctement les erreurs (fichier non existant, port 7000 déjà utilisé, ...) et afficher ces erreurs dans syslog.

Il devra, de plus, envoyer à syslog deux informations :

- nom du fichier à lire
- nombre d'octet lu

Vous tester votre daemon à l'aide de telnet, exemple de session :

```
briffaut TD4 # telnet localhost 7000
Trying 127.0.0.1…
Connected to localhost.
Escape character is '^]'.
/proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 CPU         6400  @ 2.13GHz
stepping        : 2
cpu MHz         : 2128.009
cache size      : 2048 KB
[…..]
```

Question 3 : Ecrire un script d'init pour ce daemon, et ajouter ce daemon au démarage du système.

Modifier ce daemon pour qu'il créé un fichier /var/lib/printserver/printserver.pid contenant le pid du daemon.

Modifier la fonction **stop** du script d'init pour tuer ce daemon en envoyant le signal sigkill au daemon.

Modifier le daemon pour terminer correctement le service (fermer la socket, ...) lorsqu'il reçoit un SIGTERM.

Modifier la fonction **stop** du script d'init pour tuer ce daemon en envoyant le signal SIGTERM au daemon.

2 Gestion des utilisateurs Unix

2.1 *Définition des utilisateurs (à traiter comme utilisateur)*

1- Combien d'utilisateurs UNIX sont définis localement sur votre système ? Parmi ceux-ci, combien correspondent à des utilisateurs réels (humains ou humanoïdes) ?
2- Quel est le shell de l'utilisateur root ? Et celui de l'utilisateur halt ? A quoi sert ce dernier ?
3- Que fait la commande /sbin/nologin ? Quels comptes l'utilisent-elle, pourquoi ?

2.2 *Création d'utilisateurs et de groupes (à traiter comme administrateur (root))*

Gestion des utilisateurs
La création d'un utilisateur UNIX requiert au moins les étapes suivantes : 1. ajouter les informations dans les fichiers /etc/passwd et /etc/shadow, ou dans l'annuaire (NIS, LDAP ou autre) utilisé. 2. créer le répertoire de connexion de l'utilisateur, et y placer les fichiers de configuration minimaux ; 3. configurer si nécessaire le système de messagerie électronique (e-mail). La création d'un utilisateur local (défini simplement sur votre système) est facilité par la commande useradd. La création d'un groupe est similaire (mais sans création de répertoire), via la commande groupadd. Le mot de passe d'un utilisateur est changé par la commande passwd. L'administrateur (root) peut changer le mot de passe d'un utilisateur quelconque en indiquant passwd login. Pour ajouter un utilisateur à un groupe, on édite le fichier /etc/group. Le shell d'un utilisateur est changé par la commande chsh. useradd création utilisateur local groupadd création groupe local passwd modification mot de passe chsh modification shell de login

1- Lire la documentation de la commande *useradd*, puis créer quelques utilisateurs, dont un avec votre nom et prénom.
Immédiatement après création, quel est le mot de passe de l'utilisateur ? Pourquoi ?

2- Dans quels groupes sont vos utilisateurs ?

3- Créer un groupe *tpgtr* réunissant deux de vos utilisateurs.

2.3 *Droits (commandes `chown`, `chgrp`, `chmod`)*

1- Changer (en tant qu'etudiant) les droits sur le compte "etudiant" afin que les autres utilisateurs ne puisse pas y accéder.

2- Créer un répertoire dans /*tmp* qui ne soit accessible (rx) que par les membres du groupe *etudiant*, puis y créer (toujours en tant qu'etudiant) un fichier toto qui soit lisible et modifiable par les utilisateurs du groupe `etudiant`, mais pas par les autres.
Tester (ajouter un autre utilisateur au groupe *etudiant*).

Les utilisateurs du groupe *etudiant* peuvent ils supprimer le fichier *toto* ? Pourquoi ?

2.4 Droits d'accès

Exercice à traiter comme utilisateur (etudiant), non root !
1- Essayer (dans un shell etudiant) de supprimer ou de modifier le fichier */var/log/messages*.
Que se passe-t-il ? Expliquer la situation à l'aide de la commande *ls -l*

2- A l'aide de la commande id, vérifier votre identité et le(s) groupe(s) auquel vous appartenez.

3- Créer un petit fichier texte (de contenu quelconque), qui soit lisible par tout le monde, mais pas modifiable (même pas par vous).

4- Créer un répertoire nommé *secret*, dont le contenu soit visible uniquement par vous même. Les fichiers placés dans ce répertoire sont ils lisibles par d'autres membres de votre groupe ?

5- Créer un répertoire nommé *connaisseurs* tel que les autres utilisateurs ne puissent pas lister son contenu mais puissent lire les fichiers qui y sont placés. On obtiendra :

```
$ ls connaisseurs
ls : connaisseurs: Permission denied
$ cat connaisseurs/toto
<...le contenu du fichier toto (s'il existe)...>
```

6- Chercher dans le répertoire */usr/bin* trois exemples de commandes ayant la permission *SUID*. De quelle genre de commande s'agit il ?

2.5 Limites et restrictions

Modifier le fichier */etc/security/limits.conf* afin de limiter les utilisateurs de votre système à *:*

- *3 connexions maximum*
- *20 processus*
- *1 minutes de temps CPU*

Tester ensuite ces paramètres avec un compte utilisateur.

N'autoriser que les connections sur le terminal 1 à l'aide du fichier */etc/securetty.*

N'autoriser que le *bash* dans */etc/shells.*

Etudier le fichier */etc/login.defs.* Que permet de faire ce fichier?