Jesús Torres Follow
Docente e Investigador de la Universidad de La Laguna.
May 17, 2013 · 11 min read

Unreal Engine 4 Elemental Demo

Servicios y demonios en Linux

Este artículo hace referencia al viejo estilo de demonio para SysV init y no al nuevo estilo de demonio requerido por systemd, que está sustituyendo al `init` *tradicional en los sistemas Linux.*

. . .

Un demonio o servicio es un programa que se ejecuta en segundo plano, fuera del control interactivo de los usuarios del sistema ya que carecen de interfaz con estos. El término demonio se usa fundamentalmente en sistemas UNIX y basados en UNIX, como GNU/Linux o Mac OS X. En Windows y otros sistemas operativos se

denominan servicios porque fundamentalmente son programas que ofrecen servicios al resto del sistema.

El sistema generalmente inicia los demonios durante el arranque, siendo las funciones más comunes de estos las de ofrecer servicios a otros programas, ya sea respondiendo a las peticiones que llegan a través de la red o atendiendo a procesos que se ejecutan en el mismo sistema, así como responder ante cierta actividad del hardware—por ejemplo `acpid` maneja el apagado del sistema cuando el usuario pulsa el botón de encendido del equipo—. Algunos demonios sirven para configurar hardware —como es el caso de `udevd` en algunos sistemas GNU/Linux— ejecutar tareas planificadas —como hace `cron` — o realizar otras funciones similares.

Tradicionalmente en sistemas UNIX y derivados los nombres de los demonios terminan con la letra *d*. Por ejemplo `syslogd` es el demonio que implementa el registro de eventos del sistema, mientras que `sshd` es el que sirve a las conexiones SSH entrantes.

Crear demonios del sistema

En Linux, cuando ejecutamos un comando cualquiera, como `ls` :

```
# ls
```

el nuevo proceso `ls` es hijo del proceso de la *shell* desde la que lo hemos ejecutado. Dicha *shell* conoce su PID y puede bloquearse esperando a que el comando termine, antes de volver a mostrar el *prompt —este es* carácter o conjunto de caracteres que la *shell* muestra en la línea de comandos para indicar que está a la espera de nuevos comandos —. Incluso si la *shell* decide no esperar a que el comando termine —por ejemplo cuando en `bash` el usuario lo indica utilizando el caracter `&` — permitiendo la ejecución en segundo plano, esta será notificada cuando el proceso hijo termine mediante una señal, pudiendo así controlar el estado de los distintos comandos en ejecución. Además la *shell* tiene acceso al motivo por el que el proceso hijo terminó, así como al código de estado devuelto por este a través de la llamada al sistema `exit()`.

Por lo tanto los procesos que quieren convertirse en un demonio deben asegurarse que siempre se ejecutan en segundo plano, desvinculándose

del proceso de la *shell* que los invocó y de la terminal del sistema desde la que esta se ejecutaba.

Crear un proceso hijo

El primer paso para que un proceso se convierta en un demonio es crear un proceso hijo y a continuación terminar el proceso padre:

```
pid_t pid;

// Nos clonamos a nosotros mismos creando un proceso hijo.
pid = fork();

// Si pid es < 0, fork() falló.
if (pid < 0) {
    // Mostrar la descripción del error y terminar.
    std::cerr << std::strerror(errno) << '\n';
    exit(10);
}

// Si pid es > 0, estamos en el proceso padre.
```

Al finalizar el proceso padre el proceso hijo es adoptado por `init`. El resultado es que la *shell* piensa que el comando terminó con éxito, permitiendo que el proceso hijo se ejecute de manera independiente en segundo plano.

`init` es el demonio antepasado de todos los procesos del sistema. Es el primer proceso que se inicia durante el arranque del sistema y su función principal es lanzar el resto de demonios necesarios para el funcionamiento del sistema. Generalmente también lanza los procesos encargados de solicitar el inicio de sesión a los usuarios.

Cambiar umask

Todos los procesos tiene una máscara que indica que permisos no deben establecerse al crear nuevos archivos. Así cuando se utilizan llamadas al sistema como `open()` los permisos especificados se comparan con esta máscara, desactivando de manera efectiva los que en ella se indiquen.

La máscara —denominada `umask()`— es heredada de padres a hijos por los procesos, por lo que su valor por defecto será el mismo que el que tenía configurada la *shell* que lanzó el demonio. Esto significa que el demonio no sabe que permisos acabarán tenido los archivos que

intente crear. Para evitarlo simplemente podemos autorizar todos los permisos:

```
umask(0);
```

o establecer específicamente para `umask()` otro valor más seguro según nuestra conveniencia. Incluso algunos desarrolladores permiten que los administradores del sistema especifiquen la máscara deseada a través del archivo de configuración del demonio.

Abrir conexión con el registro de eventos

Puesto que un demonio se ejecuta en segundo plano no debe estar conectado a ninguna terminal. Sin embargo esto plantea la cuestión de cómo indicar condiciones de error, advertencias u otro tipo de sucesos del programa.

Algunos demonios almacenan estos mensajes en archivos específicos o en su propia base de datos de sucesos. Sin embargo en muchos sistemas existe un servicio especifico para registrar estos eventos. En lo sistemas basados en UNIX este servicio lo ofrece el demonio Syslog, al que otros procesos pueden enviar mensajes a través de la función `syslog()`.

```
// Abrir una conexión al demonio syslog
openlog(argv[0], LOG_NOWAIT | LOG_PID, LOG_USER);

...

// Enviar un mensaje al demonio syslog
syslog(LOG_NOTICE, "Demonio iniciado con éxito\n");

...

// Cuando el demonio termine, cerrar la conexión con el
servicio syslog
closelog();
```

Al crear la conexión se especifica un código de recurso —en el ejemplo anterior `LOG_USER` indica mensajes genéricos de nivel de usuario— que indica el tipo de software que genera los mensajes. Además cada mensaje se envía acompañado de un nivel de severidad — `LOG_NOTICE` especifica que nuestro mensaje no es más que una aviso—. El

administrador del sistema puede configurar Syslog para que haga cosas diferentes con los mensajes según el código de recurso y el nivel de severidad. Por ejemplo pueden ser escritos a un archivo concreto en disco, mostrados por una consola, enviados por red a registro de eventos centralizados o filtrados.

Registro de eventos en aplicaciones Qt

Cuando se desarrolla con Qt puede ser interesante echar un vistazo a las funcionalidades de la librería LibQxt. Este proyecto proporciona un conjunto de clases que añaden funcionalidades multiplataforma que no están disponibles en el *framework* Qt.

Entre dichas funcionalidades adicionales está la clase `QxtLogger`, que facilita que el demonio registre los eventos en sus propios archivos de sucesos `QxtLogger` —por ejemplo `/var/log/midemonio/info.log`, `/var/log/midemonio/debug.log`, etc.— de forma más apropiada para una aplicación en C++.

Por ejemplo `QxtLogger` se puede configurar así al iniciar la aplicación:

```
#include <QxtBasicFileLoggerEngine>
#include <QxtLogger>

// Código de inicialiación de QxtLogger

// Configurar QxtLogger para usar dos archivos de registro.
// de severidad de INFO en adelante y otro para los niveles
// Primero se crea un engine por archivo de registro
auto debug = new QxtBasicFileLoggerEngine("debug.log");
auto info = new QxtBasicFileLoggerEngine("info.log");

// Cada engine se registra en el objeto QtLogger asignándol
qxtLog->addLoggerEngine("dbg", debug);
qxtLog->addLoggerEngine("app", info);

// Finalmente se indica que niveles de severidad se envían
// tanto  a cada archivo de registro
```

Para después registrar los eventos así:

```
// Y ya podemos registrar los distintos sucesos.
qxtLog->info("Demonio iniciado con éxito");
```

o así, si queremos registrar el instante de tiempo en el que ocurrió.

```
qxtLog->info(QTime::currentTime(), "Algo ha ocurrido",
3.14);
```

En teoría el mecanismo usado por `QxtLogger` es lo bastante flexible como para que sea posible extenderlo con el objeto de usar el registro del sistema —a través de la función `syslog()`— en lugar de nuestros propios archivos, si estuviéramos interesados en ello.

Además existe una librería alternativa a `QxtLogger` llamada `QsLog`.

Crear una nueva sesión

Cada proceso es miembro de un grupo y estos a su vez se reúnen en sesiones. En cada una de estas hay un proceso que hace las veces de líder, de tal forma que si muere todos los procesos de la sesión reciben una señal `SIGHUP`. La idea es que el líder muere cuando se quiere dar la sesión por terminada, por lo que mediante `SIGHUP` se notifica al resto de procesos esta circunstancia para que puedan terminar ordenadamente.

Obviamente no estamos interesados en que el demonio termine cuando la sesión desde la que fue creado finalice, por lo que necesitamos crear nuestra propia sesión de la que dicho demonio será el líder:

```
pid_t sid;
```

```
// Intentar crear una nueva sesión
sid = setsid();
if (sid < 0) {
    syslog(LOG_ERR, "No fue posible crear una nueva
sesión\n");
    exit(11);
}
```

Cambiar el directorio de trabajo

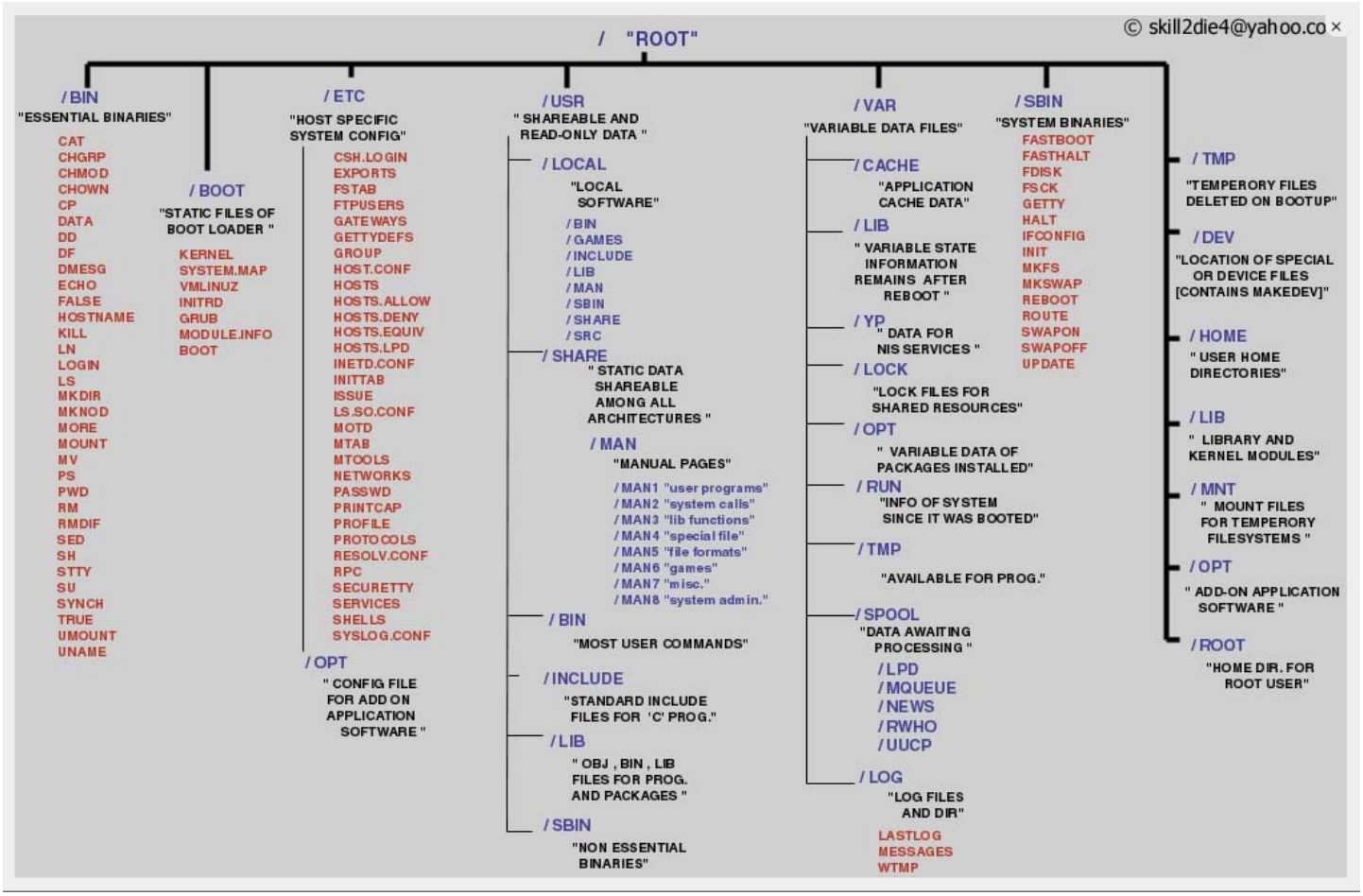
Hasta el momento el directorio de trabajo del proceso es el mismo que el de la *shell* en el momento en el que se ejecutó el comando. Este podría estar dentro de un punto de montaje cualquiera del sistema, por

lo que no tenemos garantías de que vaya a seguir estando disponible durante la ejecución del proceso.

Por eso es probable que prefiramos cambiar el directorio de trabajo al directorio raíz, ya que podemos estar seguros de que siempre existirá:

```
// Cambiar el directorio de trabajo de proceso
if ((chdir("/")) < 0) {
    syslog(LOG_ERR, "No fue posible cambiar el directorio de
"
                    "trabajo a /\n");
    exit(12);
}
```

Otra opción es cambiar el directorio de trabajo a donde se almacenan los datos de nuestra aplicación o a algún otro directorio similar, siempre según el Filesystem Hierarchy Standard.



Reabrir los descriptores estándar

Como ya hemos comentado, un demonio no interactúa directamente con los usuarios, por lo que no necesita usar la E/S estándar, así que podemos cerrar sus descriptores:

```
// Cerrar los descriptores de la E/S estándar
close(STDIN_FILENO);            // fd 0
close(STDOUT_FILENO);           // fd 1
close(STDERR_FILENO);           // fd 2
```

Sin embargo esto puede no ser lo más conveniente porque los siguientes archivos que se abran —archivos de registro, *sockets*, etc— reusarán esos mismos descriptores, lo que puede ser un problema si se utiliza alguna librería que pueda intentar hacer uso de la E/S estándar. En su lugar puede ser preferible mantenerlos abiertos pero conectados al archivo `/dev/null`:

```
// Cerrar los descriptores de la E/S estándar
close(STDIN_FILENO);                    // fd 0
close(STDOUT_FILENO);                   // fd 1
close(STDERR_FILENO);                   // fd 2

int fd0 = open("/dev/null", O_RDONLY);  // fd0 == 0
int fd1 = open("/dev/null", O_WRONLY);  // fd0 == 1
int fd2 = open("/dev/null", O_WRONLY);  // fd0 == 2
```

Usar daemon()

Algunos de los pasos comentados anteriormente se pueden hacer de una sola vez mediante la función `daemon()`:

- Hace el `fork()` y termina el proceso padre.
- Opcionalmente puede cambiar el directorio de trabajo pero sólo al directorio raíz.
- No ajusta `umask()`.
- Opcionalmente reabre los descriptores de archivo estándar a `/dev/null`.

En todo caso el principal inconveniente es que no es una función del estándar POSIX, por lo que en algunas plataformas puede no existir o tener un comportamiento diferente.

Cambiar la personalidad del proceso

Los demonios son generalmente lanzados por el usuario `root`, lo que les da acceso a todos los recursos del sistema. Esto puede ser muy peligroso si un atacante se saltara las medidas de seguridad incorporadas en el programa y fuera capaz de hacer que ejecutara código arbitrario.

Para evitarlo es muy común que los demonios cambien su personalidad a un usuario y grupo convencional del sistema:

```
#include <sys/types.h>
#include <pwd.h>

// ...

// Cambiar el usuario y el grupo efectivo del proceso a 'mi
passwd* user = getpwnam("midemonio");
group* group = getgrnam("midemonio");

if (user == nullptr || group == nullptr) {
    syslog(LOG_ERR, "No fue posible cambiar el usuario y/o
```

Señales

Una vez el demonio está en ejecución las señales ser convierten en un mecanismo muy potente para comunicarnos con él:

- Debe atender la señal `SIGTERM` para terminar en condiciones de seguridad.
- Es muy común que intercepte la señal `SIGHUP` ya que así se le suele indicar que recargue los archivos de configuración sin tener que reiniciarlo.

Si estamos desarrollando nuestra aplicación con Qt, lo más conveniente es echar un vistazo al final del artículo Aplicaciones de consola con Qt ya que allí se explican las particularidades del uso de señales POSIX en este tipo de aplicaciones.

Arranque del servicio

Los demonios `init` de muchos sistemas operativos operan en base a *niveles de ejecución*. Cada nivel de ejecución define un estado de la

máquina después del arranque, estableciendo que demonios deben ser iniciados y/o detenidos para alcanzarlo. Por lo tanto en cada momento sólo puede haber un nivel de ejecución activo.

Por lo general existen 7 niveles —del 0 al 6— alguno de los cuales suelen ser para:

- Entrar en modo monousuario (generalmente el 1 o S).
- Entrar en modo multiusuario sin soporte de red.
- Entrar en modo multiusuario con soporte de red.
- Apagar el sistema (generalmente el 0).
- Reiniciar el sistema (generalmente el 6).

El uso concreto de los niveles depende del sistema operativo en cuestión. Por ejemplo, el estándar de Linux LSB fija que los niveles 2 y 3 se utilicen para entrar en el modo multiusuario, sin soporte y con soporte de red respectivamente. Mientras que el nivel 5 es como el nivel 3 pero añadiendo el arranque del entorno gráfico. LSB o Linux System Standard es una iniciativa de la Linux Foundation para reducir las diferencias entre las distintas distribuciones de Linux, limitando así los costes derivados de portar las aplicaciones de unas distribuciones a otras.

Pero no todas las distribuciones cumplen estándar LSB al 100%. Por ejemplo, distribuciones como Debian y derivadas no hacen de serie distinciones en los niveles del 2 al 5. Todo ellos permiten arrancar el modo multiusuario con soporte de red y entorno gráfico, siempre que este último haya sido instalado.

Script de inicio

Para iniciar o detener los demonios según el nivel de ejecución, estos suelen ir acompañados de un *script* de inicio que por lo general se almacena en el directorio `/etc/init.d`. En aquellas distribuciones de Linux que siguen el estándar LSB5 la estructura de este *script* suele ser muy similar a la siguiente:

```
#!/bin/sh
#
# midemoniod    /etc/init.d initscript para MiDemonio
#

### BEGIN INIT INFO
# Provides:          midemonio
# Required-Start:    $network $local_fs $remote_fs
# Required-Stop:     $network $local_fs $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: inicia y detiene midemonio
# Description:       MiDemonio es el mejor demonio. Y este
#                    es un ejemplo de como crear un script
#                    inicio.
### END INIT INFO

# Salir inmediatamente si un comando falla
# http://www.gnu.org/software/bash/manual/bashref.html#The-
set -e

# Importar funciones LSB:
# start_daemon, killproc, status_of_proc, log_*, etc.
. /lib/lsb/init-functions
NAME=midemoniod
PIDFILE=/var/run/$NAME.pid
DAEMON=/usr/sbin/$NAME
DAEMON_OPTS="--daemon"

# Si el demonio no existe, salir.
test -x $DAEMON || exit 5

start()
{
    log_daemon_msg "Starting the $NAME process"
    start_daemon -p $PIDFILE -- $DAEMON $DAEMON_OPTS
    log_end_msg $?
}

stop()
{
    log_daemon_msg "Stoppping the $NAME process"
    killproc -p $PIDFILE
    log_end_msg $?
```

```
45  }
46
47  case "$1" in
48      start)
49          start
50          ;;
```

Como se puede observar, la mayor parte de las opciones del *script* necesitan poner comunicarse con el demonio en ejecución:

- **start**
  Necesita poder determinar si el demonio ya ha sido iniciado para no hacerlo por segunda vez.
- **status**
  También necesita poder determinar si el demonio ya ha sido iniciado o no.
- **stop**
  Necesita poder enviar la señal `SIGTERM` al proceso.
- **reload**
  Necesita poder enviar la señal `SIGHUP` al proceso.

Para que todo esto sea posible el demonio suele crear una archivo con extensión `.pid` en el directorio `/var/run`. En el almacena únicamente su propio identificador de proceso, del tal forma que las funciones LSB `start_daemon`, `killproc`, `pidofproc` y `status_of_proc` asumen que:

- Si el archivo no existe, el demonio no se está ejecutando.
- En otro caso y si contiene uno o más valores numéricos separados por espacios en la primera línea, estos valores son considerados los PID de los procesos del demonio.

Las funciones anteriores considerarán que el demonio está en ejecución si existen procesos con esos PID y enviarán las señales que correspondan.

Automatización del inicio y parada del servicio

Al sistema hay que indicarle en qué *niveles de ejecución* se debe iniciar el servicio y en cuales debe ser detenido. De esta forma automatizamos el inicio y parada del demonio durante el arranque del sistema.

Para hacerlo lo más sencillo es utilizar el comando `update-rc.d` de la siguiente manera:

```
# update-rc.d midemonio defaults
```

Al usar la opción *defaults*, `update-rc.d` configurará que el arranque del servicio sea en los *niveles de ejecución* 2345 y la parada en los niveles 016.

El orden en el que se inician los servicios viene preestablecido por un código de secuencia, de tal forma que primero se inician los servicios con menor valor. Al usar la opción *defaults* este código es 20, por lo que primero se iniciarán los servicios con código menor de 20 y después los que tengan un valor mayor.

A efectos prácticos el comando:

```
# update-rc.d midemonio defaults
```

es equivalente a:

```
# update-rc.d midemonio start 20 2 3 4 5 . stop 20 0 1 6 .
```

donde se puede apreciar como se indica el código de secuencia y los *niveles de ejecución* para el inicio —*start*— y parada —*stop*— del servicio `midemonio`.

Opciones de línea de comandos

Los demonios, igual que cualquier otro programa, pueden recibir opciones de línea de comandos para configurarlos o alterar su comportamiento. Por ejemplo es muy común que ofrezcan opciones para sobrescribir los parámetros indicados en el archivo de configuración. También es habitual que un proceso de un programa diseñado para ejecutarse como un demonio no se convierta en tal si no es especificada en la línea de comandos una opción del estilo de `-d` o `--daemon`. Esto es así, por ejemplo, porque es más sencillo depurar el

programa y comprobar su funcionamiento cuando éste no se ejecuta en segundo plano.

En el código del proyecto rifftree, que ya comentamos en el artículo Resource Interchange File Format, se puede observar como se utiliza la familia de funciones `getopt` para procesar adecuadamente la línea de comandos de un programa cualquiera.

Referencias

- Wikipedia—Demonio.
- Writing a Daemon in C
- What's the difference between calling daemon() and calling fork(), setsid(), fork(), etc.?
- Proyecto LibQxt.
- The GNU C Library—Users and Groups
- Wikipedia—Runlevel
- Linux Standard Base Core Specification 3.2—System Initialization