Práctica 5: Daemons en Linux. Implementación de un Servidor de Ficheros con colas de mensajes POSIX

Demonios

Un demonio en UNIX (servicio o programa residente en otros sistemas operativos) es un proceso no interactivo que se ejecuta en *background* y sin terminal de control que ofrece servicios a otros procesos y/o supervisa el sistema. Generalmente los demonios inician su ejecución en tiempo de arranque del sistema y su `uid` es `root` u otro usuario especial (`postfix`, `apache`, `daemon`, `avahi`,...). Por convenio, el nombre de los demonios suele finalizar en la letra 'd' (`crond`, `sshd`, `atd`, `nptd`, ...).

Los dos requisitos básicos de un demoni son:

- No debe estar conectado a un terminal.
- Debe ejecutarse como hijo del proceso `init` (`systemd`).

Los demonios de Linux se han implementado tradicionalmente usando el esquema propuesto en Unix System V (*System V daemons*). Actualmente se suelen utilizar esquemas más simples (*new-style daemons*) bajo el control de `systemd`.

Para crear un demonio según el modelo tradicional de System V, el proceso que arranca el demonio debe seguir una serie de pasos como parte de su inicialización:

1. **Cerrar todos los descriptores de fichero abiertos** excepto `stdin`, `stdout` y `stderr`. De este modo se evita que el demonio herede ficheros abiertos , como pueden ser ficheros de configuración del propio demonio. En linux se pueden conocer los ficheros abiertos por un proceso consultando el directorio `/proc/self/fd`. Cada entrada de este directorio es un enlace simbólico, cuyo nombre es el número de un descriptor de fichero abierto, que apunta al recurso que dicho descriptor representa. Por ejemplo, si el proceso tiene abierto el fichero `/etc/myserverconf` y su descriptor es el número 3, el directorio `/proc/self/fd` tendrá una entrada:

```
lrwx------ 1 usuario usuario 64 Apr 1 21:55 3 -> /etc/myserverconf
```

2. **Restablecer todos los manejadores de señales al manejador por defecto (`SIG_DFL`)**. Se evita que el demonio herede cualquier manejador personalizado que podría interferir con las operaciones del demonio. La constante `_NSIG` definida en `<signal.h>` representa el número de señales definidas en el sistema y puede utilizarse para recorrer todas las señales posibles.

3. **Restablecer la máscara de señales con `sigprocmask()`**. Se desbloquean cualquier señal que pudiera estar bloqueada, asegurando que el demonio pueda manejar todas las señales apropiadamente.

4. Eliminar **variables de entorno** innecesarias para la operación del demonio.

5. **Double fork() trick**. Se crea un nuevo proceso hijo con `fork()`, que crea una nueva sesión independiente llamando a `setsid()`, desvinculandose de la terminal de control, convirtiéndose en líder de sesión y líder de grupo. Este proceso hijo vuelve a crear un nuevo proceso con un segundo `fork()` que será el que se convierta finalmente en demonio. La idea con este doble fork es que el demonio, al no ser líder de sesión, si abre un terminal no se convierte en terminal de control.

6. Una vez creado el proceso que se convertirá en demonio (nieto del proceso inicial), el proceso hijo finaliza con `exit()`. De este modo, el padre del demonio pasa a ser el proceso `init` (actualmente el proceso `systemd` correspondiente).
7. En el proceso demonio se deberá además:

- **Conectar `stdin`, `stdout` y `stderr` al dispositivo `/dev/null`**. Para comunicarse con los usuarios se pueden utilizar ficheros de *log* en directorios especiales como `/var/log`. Para escribir en los ficheros de log se puede utilizar los servicios del demonio syslog.
- **Restablecer la máscara umask a 0** para que los permisos establecidos con `open()`, `mkdir()` u otras llamadas similares controlen directamente dichos permisos.
- **Cambiar el directorio de trabajo** actual al directorio raíz (/). Con esta operación se puede evitar que el demonio bloquee involuntariamente que se se puedan desmontar puntos de montaje (si la ruta de dicho directorio de trabajo se encuentra dentro de dicho volumen).
- **Escribir el PID del demonio en el fichero PID** asociado a este demonio. Este fichero se utiliza para garantizar que no se inician múltiples instancias simultáneamente del mismo demonio. Para evitar condiciones de carrera se debe proteger el acceso a dicho fichero con locks de ficheros (con `flock()` o `fnctl()`).
- **Renunciar a privilegios**, si es posible y aplicable. Es habitual que el proceso original que arranca el demonio se inicie con privilegios de administrador (root)y si el demonio tiene algún tipo de vulnerabilidad, puede comprometer seriamente el sistema. Por este motivo, una práctica de seguridad habitual para limitar el daño potencial en caso de que el demonio sea explotado es reducir los privilegios del demonio a los de un usuario menos privilegiado que típicamente es específico para este servicio.
- **Notificar al proceso original** que la inicialización está completa. Esto puede implementarse mediante una tubería sin nombre u otro canal de comunicación que se crea antes del primer `fork()` y por lo tanto está disponible tanto en el proceso original como en el proceso del demonio.

8. El proceso original, una vez confirmando que la inicialización del demonio ha sido correcta puede finalizar.

Colas de Mensajes

Las colas de mensajes son un mecanismo de comunicación entre procesos similar a las tuberías, pero orientado a mensajes, permitiendo con ello una mayor versatilidad. Al igual que las tuberías, una cola de mensajes es una estructura de datos en memoria gestionada por el kernel donde se pueden insertar o extraer datos siguiendo un esquema FIFO. Pero a diferencia de las tuberías los datos tienen formato de mensaje y son tratados como un todo indivisible.

Colas de Mensajes SV

En La implementación original de las colas de mensajes de Unix System V los mensajes tienen una etiqueta que sirve para clasificar los mensajes. Esta información se puede utilizar en las lecturas para extraer solo los mensaje de un tipo (etiqueta) determinado o usar la cola como una cola de prioridad.

Las llamadas que se emplean para manipular las colas de mensajes del System V son

- `msget`: para crear una cola o habilitar el acceso a una ya existente.
- `msgctl`: para acceder y modificar la información administrativa y de control que el núcleo le asocia a cada cola de mensajes.
- `msgsnd`: para escribir un mensaje en la cola.
- `msgrcv`: para extraer un mensaje de la cola.

El código de apoyo de esta práctica es un servidor de ficheros implementado con colas de mensajes System V que Michael Kerrisk usa como referencia para ilustrar su uso. Esta aplicación hace uso de múltiples colas de mensajes. El servidor mantiene una cola (con una clave conocida) para recibir las peticiones de los clientes y cada cliente crea su propia cola privada, que se utiliza para comunicar las respuestas del servidor al cliente:

- Los clientes envían a la cola del servidor peticiones en las que se indica el fichero que se quiere obtener y el identificador de su cola privada.
- El proceso servidor usa un bucle de espera para recibir peticiones de los clientes. Cada petición recibida es procesada por un proceso hijo, que se encarga de acceder al fichero solicitado. Si no se puede abrir dicho fichero, se envía una respuesta de fallo al cliente. En caso contrario, el contenido del fichero se envía en una serie de mensajes.

Otros aspectos relevantes de la implementación son:

- La captura de la señal `SIGCHLD` en el servidor para gestionar de forma asíncrona la terminación de los procesos hijos.
- La utilización de la macro de la librería estándar de C `offsetof(type, member)` para conocer el tamaño de los mensajes.
- La utilización de la función `atexit(void (*function)(void))` de la libraría estándar de C para gestionar el borrado de la cola privada en los procesos clientes.

Colas de Mensajes POSIX

Las colas de Mensajes POSIX son un mecanismo de comunicación bidireccional y asíncrona con mensajes indivisibles. A diferencia de los pipes no es posible leer parte de un mensaje y dejar el resto en la cola sin leer o leer/escribir múltiples mensajes con una única lectura/escritura (recepción/envío).
Los mensajes POSIX tienen una prioridad. Los mensajes más prioritarios se encolan antes que los menos prioritarios. En las colas SystemV no existe el concepto de prioridad y los mensajes se encolan en orden de llegada, aunque tienen un tipo que permite su filtrado. Las prioridades de las colas POSIX se pueden utilizar para emular (con limitaciones) los tipos de System V.

Las llamadas que se emplean para manipular las colas de mensajes POSIX son (consultar el manual de Linux como referencia):

- `mq_open`: para crear una cola o abrir el acceso a una cola ya existente.
- `mq_close`: para cerrar una cola que se ha abierto previamente.
- `mq_unlink`: para eliminar una cola (las colas de mensajes tienen persistencia kernel). Con esta operación se elimina inmediatamente el nombre pero la cola no se destruye hasta que no se hayan cerrado todas las referencias.
- `mq_getattr`: obtener atributos de una cola de mensajes.
- `mq_setattr`: fijar atributos de una cola de mensajes. Se puede establecer un modo no bloqueante con el flag O_NONBLOCK.
- `mq_send`: enviar datos a una cola de mensajes, con la prioridad indicada.
- `mq_timedsend`: envío con timeout.
- `mq_recv`: recibir el mensaje con la mayor prioridad de la cola más antíguo. La función devueve el número de bytes leidos y en el argumento `msg_prio` se obtiene por referencia la prioridad del mensaje recibido.
- `mq_timedreceive`: recepción con timeout.
- `mq_notify`: Registrar/desregistrar un método de notificación para conocer eventualmente que hay mensajes disponibles en la mqueue.

Objetivo de la práctica

Implementar una nueva versión del servidor de fichero presentado por Michael Kerrisk en la que:

- El proceso servidor debe ser un demonio System V.
- La comunicación entre clientes y servidor se realiza con colas de mensaje POSIX.
- En el proceso servidor, en lugar de utilizar procesos hijos para transferir los ficheros solicitados a los clientes se utilizarán threads. Para implementar esta funcionalidad se utilizará `mq_notify`, especificando la opción SIGEV_THREAD como opción de notificación.Dos consideraciones imporantes sobre el uso de mq_notify() en nuestra aplicación:
  - La notificación de mensajes ocurre solo cuando llega un nuevo mensaje y la cola estaba previamente vacía. Si la cola no estaba vacía cuando se llamó a `mq_notify()`, entonces una notificación ocurrirá solo después de que la cola se vacíe y llegue un nuevo mensaje.
  - La notificación ocurre solo una vez: después de que se entrega una notificación, el registro de notificación se elimina y otro proceso puede registrarse para la notificación de mensajes. Si el proceso notificado desea recibir la próxima notificación, puede usar `mq_notify()` de nuevo para solicitar una notificación adicional. Esto debe hacerse antes de vaciar todos los mensajes no leídos de la cola. (Poner la cola en modo no bloqueante es útil para vaciar la cola de mensajes sin bloquear una vez que está vacía).

El siguiente ejemplo, extraído de la página de manual de `mq_notify()` ilustra su uso:

```c
#include <mqueue.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

/* Thread start function */
static void  tfunc(union sigval sv) {
    struct mq_attr attr;
    ssize_t nr;
    void *buf;
    mqd_t mqdes = *((mqd_t *) sv.sival_ptr);
    /* Determine max. msg size; allocate buffer to receive msg */
    if (mq_getattr(mqdes, &attr) == -1)
        handle_error("mq_getattr");
    buf = malloc(attr.mq_msgsize);
    if (buf == NULL)
        handle_error("malloc")
    nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);
    if (nr == -1)
        handle_error("mq_receive")
    printf("Read %zd bytes from MQ\n", nr);
    free(buf);
    exit(EXIT_SUCCESS);          /* Terminate the process */
```

```c
int main(int argc, char *argv[]){
    mqd_t mqdes;
    struct sigevent sev
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <mq-name>\n", argv[0]);
        exit(EXIT_FAILURE);

    mqdes = mq_open(argv[1], O_RDONLY);
    if (mqdes == (mqd_t) -1)
        handle_error("mq_open")
    sev.sigev_notify = SIGEV_THREAD;
    sev.sigev_notify_function = tfunc;
    sev.sigev_notify_attributes = NULL;
    sev.sigev_value.sival_ptr = &mqdes;    /* Arg. to thread func. */
    if (mq_notify(mqdes, &sev) == -1)
        handle_error("mq_notify")
    pause();    /* Process will be terminated by thread function */
}
```