Manejo de archivos a bajo nivel

JLM

14 de junio de 2024

Capítulo 1

Acceso a ficheros de bajo nivel

1.1. Introducción

Posix proporciona un conjunto de llamadas al sistema para la manipulación de ficheros. Todas las aplicaciones o utilidades que trabajan con ficheros están fundamentadas en estos servicios básicos. La biblioteca estándar de C dispone de un conjunto de funciones para utilizar directamente estas llamadas al sistema, proporcionando al programador la misma visión que sobre los recursos tiene el sistema operativo.

Estas funciones se suelen denominar *"de bajo nivel"*. Este apartado explica algunas de las llamadas al sistema del UNIX que trabajan con ficheros, que les permitirán:

* Abrir y cerrar un fichero
* Crear y borrar un fichero
* Leer en un fichero
* Escribir en un fichero
* Desplazarse por un fichero

El manejo de ficheros en Linux sigue el modelo de la sesión. Para trabajar con un fichero hay primero que abrirlo con una invocación a la función *open*. Ésta devuelve un descriptor de fichero (file descriptor en inglés), un número entero que servirá de identificador de fichero en futuras operaciones. Finalmente hay que cerrar el fichero, con la función *close*, para liberar los recursos que tengamos asignados.

Existen al menos tres descriptores ya establecidos en la ejecución de un programa (ya los han abierto por nosotros). El **descriptor 0 es la entrada estándar (normalmente el teclado)**, el **descriptor 1 es la salida estándar (normalmente la pantalla)** y el **descriptor 2 el fichero estándar de visualización de errores (también la pantalla, normalmente)**. Los pueden considerar como simples ficheros que ya han sido abiertos, y pueden trabajar con ellos con cierta normalidad. Incluso los pueden cerrar.

Los ficheros en UNIX (Posix, Linux) permiten tanto el acceso directo como el secuencial. Cada fichero abierto dispone de un puntero que se mueve con cada lectura o escritura. Hay una función especial llamada *lseek* para posicionar ese puntero donde se quiera dentro del fichero.

Los archivos existen para almacenar información y permitir que se recupere posteriormente. Distintos sistemas proveen diferentes operaciones para permitir el almacenamiento y la recuperación. Las llamadas al sistema más comunes relacionadas con los archivos son:

1. **Creat.** El archivo se crea sin datos. El propósito de la llamada es anunciar la llegada del archivo y establecer algunos de sus atributos.
2. **Delete.** Cuando el archivo ya no se necesita, se tiene que eliminar para liberar espacio en el disco. Siempre hay una llamada al sistema para este propósito.
3. **Open.** Antes de usar un archivo, un proceso debe abrirlo. El propósito de la llamada a open es permitir que el sistema lleve los atributos y la lista de direcciones de disco a memoria principal para tener un acceso rápido a estos datos en llamadas posteriores.
4. **Close.** Cuando terminan todos los accesos, los atributos y las direcciones de disco ya no son necesarias, por lo que el archivo se debe cerrar para liberar espacio en la tabla interna. Muchos sistemas fomentan esto al imponer un número máximo de archivos abiertos en los proceso. Un disco se escribe en bloques y al cerrar un archivo se obliga a escribir el último bloque del archivo, incluso aunque ese bloque no esté lleno todavía.

5. **Read.** Los datos se leen del archivo. Por lo general, los bytes provienen de la posición actual. El llamador debe especificar cuántos datos se necesitan y también debe proporcionar un búfer para colocarlos.

6. **Write.** Los datos se escriben en el archivo otra vez, por lo general en la posición actual. Si la posición actual es al final del archivo, aumenta su tamaño. Si la posición actual está en medio del archivo, los datos existentes se sobrescriben y se pierden para siempre.

7. **Append.** Esta llamada es una forma restringida de write. Sólo puede agregar datos al final del archivo. Los sistemas que proveen un conjunto mínimo de llamadas al sistema por lo general no tienen append; otros muchos sistemas proveen varias formas de realizar la misma acción y algunas veces ésos tienen append.

8. **Seek.** Para los archivos de acceso aleatorio, se necesita un método para especificar de dónde se van a tomar los datos. Una aproximación común es una llamada al sistema de nombre *seek*, la cual reposiciona el apuntador del archivo en una posición específica del archivo. Una vez que se completa esta llamada, se pueden leer o escribir datos en esa posición.

9. **Get attributes.** A menudo, los procesos necesitan leer los atributos de un archivo para realizar su trabajo. Por ejemplo, el programa *make* de UNIX se utiliza con frecuencia para administrar proyectos de desarrollo de software que consisten en muchos archivos fuente. Cuando se llama a *make*, este programa examina los tiempos de modificación de todos los archivos fuente y objeto, con los que calcula el mínimo número de compilaciones requeridas para tener todo actualizado. Para hacer su trabajo, debe analizar los atributos, a saber, los tiempos de modificación.

10. **Set attributes.** Algunos de los atributos puede establecerlos el usuario y se pueden modificar después de haber creado el archivo. Esta llamada al sistema hace eso posible. La información del modo de protección es un ejemplo obvio. La mayoría de las banderas también caen en esta categoría.

11. **Rename.** Con frecuencia ocurre que un usuario necesita cambiar el nombre de un archivo existente. Esta llamada al sistema lo hace posible. No siempre es estrictamente necesaria, debido a que el archivo por lo general se puede copiar en un nuevo archivo con el nuevo nombre, eliminando después el archivo anterior.

12. **dup y dup2** Las llamadas al sistema dup proporcionan un modo para duplicar un descriptor de archivos, presentando dos o más descriptores diferentes que acceden al mismo archivo. Se pueden usar para leer y escribir en diferentes partes del archivo. La llamada al sistema dup duplica un descriptor de archivo, fildes, enviando un nuevo descriptor. La llamada al sistema dup2 copia eficazmente un descriptor de archivo a otro especificando qué descriptor usar para la copia.

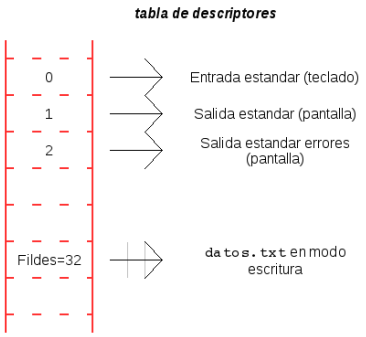
    La llamada dup2 permite generar un duplicado de un descriptor existente, tiene el siguiente prototipo:

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

    Esta función devuelve -1 en caso de error. En caso de éxito, el descriptor fIldes2 pasa a ser un duplicado del descriptor fildes.

    Supongamos la siguiente tabla de descriptores de archivos:

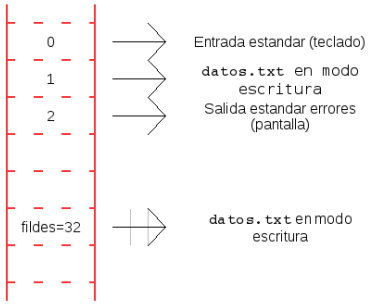


    Fíjese que ya existe un descriptor fildes para manejar el fichero datos.txt, que se obtuvo mediante la llamada open, por tanto, el valor del descriptor es seleccionado por el sistema operativo (en nuestro caso, hemos seleccionado el valor 32 arbitrariamente).

    A partir de dicha situación, tras realizar la siguiente llamada:

```
dup2(fd, STDOUT_FILENO);
```

El resultado sobre la tabla de descriptores es la siguiente:



Por tanto, tras la llamada dup2 se genera el duplicado, de manera que el descriptor 1 (STDOUT_FILENO) ya no apunta a la pantalla sino al fichero datos.txt.

De esta manera, cuando invoquemos a printf el mensaje que se pase como parámetro no se imprimirá en pantalla, sino que se almacenará en el fichero datos.txt.

Ejemplo. Este programa imprime "Hola Mundo" en un archivo llamado salida.txt mediante dup2:

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

int main()
{
   int fd;

   fd = open("salida.txt", O_WRONLY | O_CREAT | O_TRUNC, 0600);
   if(fd == -1)
   {
      perror("fallo en open");
      exit(EXIT_FAILURE);
   }
   if(dup2(fd, STDOUT_FILENO) == -1){
      perror("fallo en dup2");
      exit(EXIT_FAILURE);
   }
   if(close(fd) == -1){
      perror("fallo en close");
      exit(EXIT_FAILURE);
   }

   printf("Hola Mundo\n");
   return 0;
}
```

1.2. Permisos de los archivos

Las llamadas al sistema *creat* y *open* admiten un parámetro entero en el que se especifican los permisos con los que se crea un archivo. Una de las maneras más cómodas de declararlos es mediante la representación octal.

Los permisos se forman como un número de 9 bits, en el que cada bit representa un permiso, tal y como se muestra en el cuadro (es el mismo orden con el que aparecen cuando hacemos un ls).

RWX	**RWX**	**RWX**
usuario	grupo	otros

Se toman los nueve permisos como tres números consecutivos de 3 bits cada uno. Un bit vale 1 si su permiso correspondiente está activado y 0 en caso contrario. Cada número se expresa en decimal, del 0 al 7, y los permisos quedan definidos como un número octal de tres dígitos. Para poner un número en octal en el lenguaje C, se escribe con un cero a la izquierda. Por ejemplo, los permisos **rw-r–r-x** son el número octal **0645**.

```
/* Crea un fichero con permisos RW-R--R-- */

int fd = creat ( "mi_fichero", 0644);
```

1.3. Apertura, creación y cierre de ficheros

Función open

La función open abre un fichero ya existente, retornando un descriptor de fichero. La función tiene este prototipo:

```
int open ( char* nombre, int modo, int permisos );
```

El parámetro **nombre** es la cadena conteniendo el nombre del fichero que se quiere abrir; *modo* establece la forma en que se va a trabajar con el fichero. Algunas constantes que definen los modos básicos son:

O_RDONLY	abre en modo lectura
O_WRONLY	abre en modo escritura
O_RDWR	abre en modo lectura-escritura
O_APPEND	abre en modo agregar (escritura desde el final)
O_CREAT	crea el fichero y lo abre (si existía, borra lo que tenía)
O_EXCL	usado con O_CREAT. Si el fichero existe, se retorna un error.
O_TRUNC	abre el fichero y trunca su longitud a 0

Para usar estas constantes, han de incluir la cabecera **fcntl.h** . Los modos pueden combinarse, simplemente sumando las constantes, o haciendo un ”or” lógico, como en este ejemplo:

```
O_CREAT | O_WRONLY
```

El parámetro acceso sólo se ha de emplear cuando se incluya la opción O_CREAT, y es un entero que define los permisos de acceso al fichero creado.

La función open retorna un descriptor válido si el fichero se ha podido abrir, y el valor -1 en caso de error.

Función creat

Si desean expresarmente crear un fichero, disponen de la llamada creat. Su prototipo es

```
int creat ( char* nombre, int acceso );
```

Equivale (más o menos) a llamar a open (nombre, O_RDWR—O_CREAT, acceso). Es decir, devuelve un descriptor si el fichero se ha creado y abierto, y -1 en caso contrario.

Función close

Para cerrar un fichero ya abierto está la función close:

```
int close ( int fichero );
```

donde fichero es el descriptor de un fichero ya abierto. Retorna un 0 si todo ha ido bien y -1 si hubo problemas.

Borrado de ficheros

La función

```
int unlink ( char* nombre );
```

borra el fichero de ruta nombre (absoluta o relativa). Devuelve -1 en caso de error.

Lectura y escritura

Para leer y escribir información en ficheros, han de abrirlos primero con *open* o *creat*. Las funciones *read* y *write* se encargan de leer y de escribir, respectivamente:

```
int read ( int fichero, void* buffer, unsigned bytes );

int write( int fichero, void* buffer, unsigned bytes );
```

Ambas funciones toman un primer parámetro, fichero, que es el descriptor del fichero sobre el que se pretende actuar.

El parámetro *buffer* es un apuntador al área de memoria donde se va a efectuar la transferencia. O sea, de donde se van a leer los datos en la función *read*, o donde se van a depositar en el caso de *write*.

El parámetro bytes especifica el número de bytes que se van a transferir.

Las dos funciones devuelven el número de bytes que realmente se han transferido. Este dato es particularmente útil en la función *read*, pues es una pista para saber si se ha llegado al final del fichero. En caso de error, retornan un -1.

Hay que tener especial cautela con estas funciones, pues el programa no se va a detener en caso de error, ni hay control sobre si el puntero buffer apunta a un área con capacidad suficiente (en el caso de la escritura).

La primera vez que se lee o escribe en un fichero recién abierto, se hace desde el principio del fichero (desde el final si se incluyó la opción O_APPEND). El puntero del fichero se mueve al byte siguiente al último byte leído o escrito en el fichero. Es decir, UNIX trabaja con ficheros secuenciales.

Movimiento del puntero del fichero

C y UNIX manejan ficheros secuenciales. Es decir, conforme se va leyendo o escribiendo, se va avanzando en la posición relativa dentro del fichero. El acceso directo a cualquier posición dentro de un fichero puede lograrse con la función *lseek*.

```
long lseek ( int fichero, long desp, int origen );
```

Como siempre, fichero es el descriptor de un fichero y abierto.

El parámetro *desp* junto con origen sirven para determinar el punto del fichero donde va a acabar el puntero. *desp* es un entero largo que expresa cuántos bytes hay que moverse a partir del punto indicado en origen, parámetro que podrá adoptar estos valores:

0	SEEK_SET	inicio del fichero
1	SEEK_CUR	relativo a la posición actual
2	SEEK_END	relativo al final del fichero

Las constantes simbólicas se encuentran en **¡stdlib.h¿** y **¡unistd.h¿**.

El parámetro *desp* puede adoptar valores negativos, siempre que tengan sentido. Si el resultado final da una posición mayor que el tamaño del fichero, éste crece automáticamente hasta esa posición.

La función *lseek* devuelve un entero largo que es la posición absoluta donde se ha posicionado el puntero; o un -1 si hubo error. Obsérvese que la función *lseek* puede utilizarse también para leer la posición actual del puntero.

1.4. Ejemplos

1. Crear un archivo donde guardar el ”Hola mundo”.

escribeFich.c

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

int main()
{
   const char* cadena = "Hola Mundo";
   int fichero = open("miFichero", O_CREAT|O_WRONLY, 0644);
   if(fichero == -1)
   {
      perror("Falla al abrir el fichero: ");
      exit(1);
   }
   write(fichero, cadena, strlen(cadena));
   close(fichero);
   printf("\n");
   return 0;
}
```

2. Leer desde un fichero existente diez caracteres, a partir de la posición 400.

leeFich.c

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

int main()
{
   char cadena[31];
   int leidos;
   int fichero=open("elFichero", O_RDONLY);

   if(fichero == -1)
   {
      perror("Error al abrir el fichero: ");
      exit(1);
   }
   lseek(fichero, 400, SEEK_SET);  // coloca el fichero en la posicion 400

   leidos = read(fichero, cadena, 30); // lee 10 bytes

   printf("Se leyeron %d bytes.\nLa cadena leida es \"%s\"\n", leidos, cadena);

   return 0;
}
```

3. Diseñe un programa que lea un archivo de la línea de comandos y lo copie a otro archivo.

copyfile.c

```
// Programa para copiar archivos.  Verificacion y reporte de errores
// son minimos.  Ejemplo sacado del libro Sistemas Operativos Modernos
// 3ra edicion Andrew S.Tanenbaum
// **
// Uso: copyfile <archivo_origen> <archivo_destino>

#include <sys/types.h>  // se agregan los archivos de encabezado necesarios
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char *argv[]);  // Prototipo de funcion ANSI

#define TAM_BUF 4096 // 2^12 usa un tamaño de buffer de 4096 bytes
#define MODO_SALIDA 0700 // octal 700 (111 000 000)

int main(int argc, char* argv[])
{
   int ent_da, sal_da, leer_cuenta, escribir_cuenta;
   char bufer[TAM_BUF];

   if(argc != 3) exit(1);  // error de sintaxis si argc no es 3

   // Abre la entrada y crea el archivo de salida
   ent_da = open(argv[1], O_RDONLY); // abre archivo origen
   if(ent_da < 0) exit(2);  // Si no se puede abrir termina
   sal_da = creat(argv[2], MODO_SALIDA);  // crea el archivo de salida
   if(sal_da < 0) exit(3);  //Si no se puede crear, finaliza

   // copia del archivo
   while(1)
   {
      leer_cuenta = read(ent_da, bufer, TAM_BUF);  // Lee un bloque de datos
      if(leer_cuenta <= 0) break;  // Si llega al fin del archivo o
                                   // hay un error, sale del ciclo
      escribir_cuenta = write(sal_da, bufer, leer_cuenta);  // escribe datos
      if(escribir_cuenta <= 0) exit(4); // error
   }

   // Cierra los archivos
   close(ent_da);
   close(sal_da);
   if(leer_cuenta == 0)
      exit(0); // no hubo error en la ultima lectura
   else
      exit(5); // hubo error en la ultima lectura
}
```

1.5. Dispositivos TTY. (Extracto del apunte Control de Periféricos del Dr. Gonzalo Perez Paina)

El nombre TTY, utilizado aún por razones históricas, significa TeleTYpewriter (máquina de escribir remota). Como se mencionó, en los sistemas de tiempo compartidos anteriores a Unix se utilizaban terminales para interactuar con la computadora. Las terminales generaban los mensajes enviados desde un teletipo y la información recibida se imprimía localmente. En estos sistemas las terminales se conectaban de forma remota

con la computadora mediante el cable adecuado y una UART. En los sistemas operativos (SO) modernos como Unix o GNU/Linux se utilizan las terminales para interactuar con el mismo mediante dispositivos TTY. El funcionamiento interno de las terminales lo maneja directamente el kernel o núcleo del SO. Se puede acceder a las terminales de GNU/Linux presionando las teclas Ctrl+Alt+Fx, con x: 1,. . . ,6. También existen programas emuladores de terminales tales como gnome-terminal, konsole, xterm, terminator, entre otras. Un comando útil para ver y modificar la configuración de la terminal es stty. La figura 1.1 muestra la salida del comando stty en la terminal actual y la figura 1.2 de la terminal asociada al dispositivo /dev/tty1. La opción -a es para imprimir toda la información de configuración de la terminal, mientras que la opción -F sirve para indicarle el archivo de dispositivo. Se pueden apreciar los valores de algunos parámetros tales como: longitud de palabra (cs8), baudrate, control de flujo, paridad, etc.

```
joseluis@DESKTOP-7SEFDIA:~$ stty
speed 38400 baud; line = 0;
-brkint -imaxbel iutf8
```

Figura 1.1: Configuración de la terminal actual(/dev/pts/4)

```
joseluis@DESKTOP-7SEFDIA:~$ sudo stty -a -F /dev/tty1
[sudo] password for joseluis:
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z;
rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon
-ixoff -iuclc -ixany -imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0
vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop
-echoprt echoctl echoke -flusho -extproc
```

Figura 1.2: Configuración de (/dev/tty1)

Utilidad `socat`

socat permite generar dispositivos TTY que se comunican entre sí mediante sockets. Es como disponer de dos puertos seriales virtuales conectados entre sí. Por ejemplo, el comando

```
> socat pty,link=/tmp/ttyS0 pty,link=/tmp/ttyS1
```

crea los archivos de dispositivos TTY, /tmp/ttyS0 y /tmp/ttyS1. Se puede acceder a su configuración mediante stty -a -F /tmp/ttyS0. Se puede modificar los parámetros de la comunicación agregando opciones al comando socat. Una vez generados estos archivos y dejando abierta la terminal donde se encuentra ejecutado socat, se puede establecer una comunicación entre las dos terminales creadas. Por ejemplo, al ejecutar

```
> cat /tmp/ttyS1
```

en una terminal, y

```
> echo "Hola TTY" > /tmp/sttyS0
```

en otra, se estará enviando el mensaje ”Hola TTY”desde ttyS0 a ttyS1.

Programa que escribe sobre puerto virtual y lee el dato desde otro puerto virtual

GNU/Linux no diferencia entre ficheros (archivo) y dispositivos físicos, a todos los trata como ficheros. En consecuencia un puerto de la PC se verá como un fichero más, sobre el cual puedo escribir o leer datos.

Aprovecharemos los puertos creados por socat para escribir sobre el puerto /tmp/ttyS0 y leeremos desde el puerto /tmp/ttyS1. Los datos recibidos los guardaremos en un fichero llamado FicheroTty.

escribirtty.c

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

int main()
{
int fichero = open("/tmp/ttyS0", O_WRONLY);
int i;
//char c[1];
if(fichero == -1)
{
    perror("Falla al abrir el fichero: ");
exit(1);
}
for(i=0; i<5; i++)
    write(fichero, "Saludos desde ttyS0\n", 20);

close(fichero);
printf("\n");
return 0;
}
```

leertty.c

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

int main()
{
int pttyS1, fichero;
char cadena[21];

pttyS1 = open("/tmp/ttyS1", O_RDONLY);
if(pttyS1 < 0) exit(1);

fichero=open("FicheroTty", O_CREAT|O_WRONLY, 0777);
if(fichero == -1) exit(2);

read(pttyS1, cadena, 20); // lee 20 bytes

write(fichero, cadena, 20);
return 0;
}
```

datos guardados en FicheroTty

```
joseluis@DESKTOP-7SEFDIA:/tmp$ cat FicheroTty
Saludos desde ttyS0
```

1.5.1. Problemas

1. ¿Que realiza el siguiente programa? Investigue las funciones desconocidas

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

```
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <stdio.h>

#define MAX 1024


void err_quit(char *msge);

int main(int argc, char *argv[])
{
   char buffer[8];
   char dest[MAX], *exten;
   int fd_orig, fd_dest, n;
   char *name_orig= argv[1];
   char name_dest[32];

   if (argc!=2) {
      fprintf(stderr, "Modo uso: %s <archivo.txt>\n", argv[0]);
      return -1;
   }

   exten= rindex(name_orig, '.');
   if (strcmp(exten, ".txt"))
      err_quit("Solo archivos de txt plano");

   if ((fd_orig= open(name_orig, O_RDONLY))== -1)
      err_quit("Error al abrir origen");

   strcpy(name_dest, "cifrado-");
   strcat(name_dest, name_orig);

   if ((fd_dest= creat(name_dest, 0666))== -1)
      err_quit("Error al abrir destino");

   while (n= read(fd_orig, buffer, sizeof buffer))
   write(fd_dest, buffer, n);

   close(fd_dest);
   close(fd_orig);

   return 0;
}

void err_quit(char *msge) {
   perror(msge);
   exit(EXIT_FAILURE);
}
```

2. Escriba un programa en C que lea un archivo de texto (.txt) desde la terminal y lo duplique al descriptor "duplicado_<archivo.txt>" en la salida estándar.

3. Cree un archivo temporal en /temp de nombre puerto_salida. A continuación escriba un programa en c que transmita un archivo con los primeros 20 números primos. Estos números serán generados automáticamente por el programa.

4. Cree un archivo temporal en /temp de nombre puerto_entrada. A continuación escriba un programa en c

que reciba los números primos del problema anterior, a continuación realice la sumatoria y la productoria de estos números con un ciclo for.