Sea el siguiente programa de nombre "c0p1" para copiar archivos (por simplicidad se ha eliminado todo control de errores):

```
 1    /* c0p1 origen destino
 2     *    Copia "origen" sobre "destino" byte a byte,
 3     *    haciendo uso de los descriptores 0 y 1.
 4     */
 5    int main(int argc char * argv[])
 6    {     unsigned char byte[1];
 7          close(0); open(argv[1], O_RDONLY);
 8          close(1); open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0666);
 9          while(write(1,byte,read(0,byte,1))>0)
10                continue;
11          return 0;
12    }
```

Se pide:

A) Describa claramente la evolución de los descriptores 0 y 1 del proceso durante la ejecución de las líneas 7 y 8. Para ello, dibuje las tablas internas que el Sistema Operativo utiliza que para gestionar los descriptores de fichero (comenzando en el BCP del proceso **`c0p1`** y terminando en los nodos-i involucrados). Explique el objetivo de estas tablas y muestre el contenido de sus campos más relevantes.

A continuación, y razonando sobre las tablas y campos mostrados en el apartado anterior, explique claramente el comportamiento y el resultado final de la ejecución del mandato **`c0p1 origen destino`** en los supuestos siguientes:

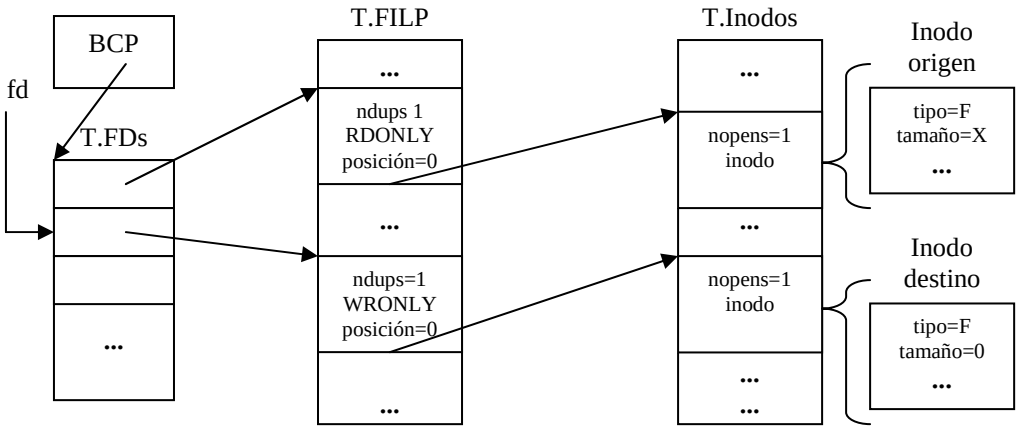
B) **`origen`** y **`destino`** son dos ficheros convencionales distintos de tamaños 1MB y 2MB respectivamente.
C) **`origen`** y **`destino`** son dos enlaces simbólicos distintos que apuntan a un mismo nombre (**`fichero`**), que es un fichero convencional de tamaño 4MB.
D) **`origen`** y **`destino`** son dos enlaces simbólicos distintos que apuntan a un mismo nombre (**`terminal`**), que fue creado como enlace duro (*hard link*) al fichero especial orientado a carácter **`/dev/tty`** que representa el terminal de control asociado al proceso.

Por último, y volviendo a considerar las condiciones del apartado **B**, y que el Sistema de Ficheros está basado en nodos-i con direcciones de 32 bits y bloques de 2KB y que se utiliza una cache de bloques con capacidad para 1Mbyte, política de reemplazo LRU, *delayed-write* para datos y *write-through* para metadatos, conteste:

E) Sin contar los accesos necesarios para la decodificación de los nombres de los ficheros,
   a. ¿Cuántos accesos a disco ocurrirán durante la ejecución completa del mandato?
   b. ¿Cuál será el contenido detallado de la cache en el instante de terminación del proceso?

A)
La llamada **`close`** cierra el descriptor dado como argumento. La llamada **`open`** devuelve el descriptor más bajo disponible asociado al fichero dado como primer argumento abierto en el modo indicado en el segundo argumento.
Las tablas internas que el Sistema Operativo utiliza que para gestionar los descriptores de fichero son:

- Tabla de Descriptores de Fichero: Propia de cada proceso, apuntada desde el BCP del mismo e indexada por el descriptor de fichero. Cada entrada es una referencia a una entrada de la Tabla Intermedia.
- Tabla Intermedia: También llamada Tabla FILP o Tabla Fichero-Posición. Una única en el sistema. Sus entradas son ubicadas cada vez que se abre (o crea) un fichero en el sistema. Cada entrada contiene **`ndups`** ó número de descriptores de fichero que apuntan a esta entrada, el modo de apertura, el puntero de posición sobre el ifichero y una referencia a una entrada de la Tabla de Inodos.
- Tabla de Inodos: Una única en el sistema. Sus entradas son ubicadas cada vez que un inodo debe ser llevado de disco a memoria. Cada entrada contiene **`nopens`** ó número de entradas de la Tabla Intermedia que apuntan a esta entrada, copia del inodo en cuestión y otros campos.
- Estructura Inodo: Entro otros campos, contiene el tipo de objeto (fichero, directorio, enlace simbólico, fichero especial orientado a carácter o a bloque, etc.) y tamaño, válido para los tres primeros tipos indicados, pero interpretado como la pareja **`major-minor`** para los dos siguientes.



Línea 7: Se redirige la entrada estándar al fichero **`origen`**, esto es, la entrada estándar deja de apuntar a donde fuera para apuntar a una entrada de la tabla FILP, recién ubicada que indica modo de apertura **`O_RDONLY`**, posición 0 (bytes) y referencia al inodo del fichero **`origen`**.
Línea 8: Se redirige la salida estándar al fichero **`destino`**, esto es, la salida estándar deja de apuntar a donde fuera para apuntar a una entrada de la tabla FILP, recién ubicada que indica modo de apertura **`O_WRONLY`**, posición 0 (bytes) y referencia al inodo del fichero **`destino`**. Si tal fichero no existía se creó (**`O_CREAT`**) con permisos **`rw-rw-rw-`** (**`0666`**), pero si existía fue truncado (**`O_TRUNC`**), esto es, todo el espacio que ocupara fue liberado y finalmente su tamaño pasó a ser de 0 bytes.

B)
Por efecto de la llamada **open** de la línea 8, el fichero convencional **destino** será truncado (todo el espacio que ocupase será liberado, y su tamaño pasará a ser de 0 bytes) El bucle principal del programa copiará byte a byte el contenido del fichero convencional **origen** sobre el fichero convencional **destino**. Cada llamada al sistema **read** y **write** afectará a un byte. Los punteros de posición asociados a la entrada y salida estándar irán avanzando de uno en uno. El bucle terminará cuando el puntero asociado a la entrada estándar alcance un valor igual al tamaño del fichero asociado (1MB), momento en el cuál la llamada **read** devolverá 0 bytes leídos, la llamada **write** devolverá 0 bytes escritos y esto hará que el bucle finalice. Finalmente **destino** tendrá un tamaño y contenido idénticos al fichero **origen**.

C)
Los enlaces simbólicos son un tipo especial de inodo que redirigen a otra ruta. Esta redirección la detecta y realiza automáticamente el Sistema Operativo durante la decodificación de los nombres de fichero. De manera que las dos llamadas **open** terminarán por abrir exactamente el mismo fichero convencional de nombre **fichero**. Por esta razón, en este caso las dos entradas de la tabla FILP referirán a una misma entrada de la tabla de Inodos, la correspondiente al inodo del fichero convencional de nombre **fichero**, que tendrá **nopens** a 2.
Por otro lado, y por efecto del parámetro **O_TRUNC** de la segunda llamada **open**, este fichero será truncado y así lo reflejará su inodo, teniendo un tamaño de 0 bytes.
El bucle de copia del programa terminará, como en el caso anterior, cuando el puntero asociado a la entrada estándar alcance un valor igual al tamaño del fichero asociado (0 bytes). Luego, realmente no llegará a copiarse ningún byte, dado que **fichero** está ahora vacío.

D)
Un enlace duro (*hard link*) **no** es un tipo de inodo, sino tan solo una entrada de directorio, esto es, una asociación entre nombre y número de inodo. En este caso, las dos llamadas **open** terminarán por abrir exactamente el mismo fichero especial orientado a carácter de nombre **terminal** Al igual que en el caso anterior, las dos entradas de la tabla FILP referirán a una misma entrada de la tabla de Inodos, la correspondiente al inodo de este fichero especial, con **nopens** a 2.
Los ficheros especiales orientados a carácter representan dispositivos que son fuente y/o sumidero de caracteres, por ello no les es aplicable la idea de "posición sobre el fichero" ni así mismo tiene sentido en ellos el campo tamaño de su inodo.
Por esta razón, y a diferencia del caso anterior, el parámetro **O_TRUNC** de la segunda llamada **open**, no tiene efecto alguno.
En este caso, como en los anteriores, para que el bucle de copia del programa termine, es necesario que la llamada **read** devuelva 0 bytes leídos. Como para estos ficheros especiales no cabe comparar puntero y tamaño, se ha de disponer de otro mecanismo que permita indicar la condición de fin de datos.
En el caso de los terminales, como es el caso en este apartado, la combinación de teclas **Ctrl-D** permite que el usuario indique esta situación.
De manera que en este caso, el programa copiará los caracteres que el usuario introduzca sobre el mismo terminal, hasta que pulse **Ctrl-D**.

E.a)
La ejecución completa del mandato empieza en la función main y termina cuando el proceso lo hace.
Consideraremos primero los accesos originados por las redirecciones de las líneas 7 y 8.

- Las llamadas **`close`** implican la actualización de los inodos inicialmente asociados a la entrada y salida estándar (**2 escrituras inmediatas** de metadatos).
- Sin contar los accesos necesarios para la decodificación de los nombres de los ficheros, la primera llamada **`open`** deberá acceder al bloque que contiene el inodo del fichero origen para llevarlo a la tabla de Inodos (**1 lectura** de metadatos).
- La segunda llamada **`open`** hará lo mismo(**1 lectura/acierto**) (posible acierto en cache si ambos inodos están en el mismo bloque), pero además debe truncar el fichero de 2MB asociado. Para ello deberá de recorrer todos los punteros a bloque del inodo, mancándolos en el bitmap como libres. Deberán marcarse 1024 bloques de datos (de 2KB) como libres más los tres bloques de indirección que el inodo usa para referirlos (**3 lecturas**)(simple indirecto, doble indirecto y la primera entrada de este). Para liberar cada uno de estos 1027 bloques se precisará una lectura del bloque de bitmap asociado (**1027 lecturas/aciertos**) (podrían darse aciertos en cache o caer todos en bloques distintos) más una escritura inmediata por cada uno (**1027 escrituras inmediatas** de metadatos). Finalmente debe reescribirse el inodo con fecha de modificación actualizada (**1 escritura inmediata)**.

Origen y destino ocuparán finalmente 512 bloques de datos:

$2^{20}_{(\text{bytes/fichero})}/2^{11}_{(\text{bytes/bloque})} = 2^{9}_{(\text{bloques/fichero})}$.

Consideraremos únicamente el momento en que se copia el primer byte de cada bloque, dado que la copia de los 2047 bytes restantes implica solamente aciertos en lectura y escrituras retardadas y ningún acceso efectivo a disco.

- Al copiar el primer byte de cada uno de los 512 bloques que componen el fichero de origen, sucede un fallo de cache en la lectura (**512 lecturas**) y así mismo, en ese momento hay que ubicar un nuevo bloque de destino (**512 ubicaciones**).

Cada bloque de indirección contiene 512 direcciones:

$2^{11}_{(\text{bytes/bloque})}/2^{2}_{(\text{bytes/dirección})} = 2^{9}_{(\text{direcciones/bloque})}$.

Para direccional 512 bloques (tanto de origen como de destino) es necesario utilizar el bloque de indirección apuntado por el simple indirecto de los inodos, del cuál se utilizarán 512-10 = 502 entradas (suponiendo 10 punteros directos por inodo).

- En el caso de origen sucederán **1 lectura** y **501 aciertos**, al bloque indirecto.
- En el caso de destino sucederán **1 ubicación** y **501 escrituras inmediatas**.
- Para ubicar cada uno de los 512 + 1 (el indirecto) de destino es necesario acceder al bitmap para encontrar uno libre (**513 ó más lecturas/aciertos**) y marcarlo como ocupado (**513 escrituras inmediatas**) así como actualizar en disco el inodo o bloque indirecto que lo apunta (**513 escrituras inmediatas**), ya que son metadatos (de las cuales 501 del indirecto han sido consideradas ya).

Finalmente, al terminar el proceso se cerrarán automáticamente sus descriptores:

- Las llamadas **`close`** implícitas de la entrada y salida estándar (ficheros origen y destino en este caso) precisarán actualizar las fechas en los inodos (**2 escrituras inmediatas** de metadatos).

E.b)
La cache de bloques tiene capacidad para 512 bloques:

$2^{20}_{(bytes/cache)}/2^{11}_{(bytes/bloque)} = 2^{9}_{(bloques/cache)}$.

Para detallar el contenido de la cache de bloques, bastará repasar hacia atrás el apartado anterior para enumerar cuales son los últimos 512 bloques que han sido accedidos, que son en este orden:

- **2** (ó 1 si comparten bloque) bloques de inodos donde residen los inodos de origen y destino respectivamente.
- **1**, el último bloque de datos (número 511) de destino.
- **1** bloque indirecto de destino.
- **1** (o puede que más) bloque de bitmap.
- **1**, el último bloque de datos (número 511) de origen.
- **1** bloque indirecto de origen.
- Restan (512-7=)505 bloques que serían, en orden decreciente y alternado los últimos bloques de destino y origen, (números 510, 509,...al 260 aprox).
- Además, entre ellos encontraríamos muy probablemente otros bloques del bitmap utilizados para ubicar los nuevos bloques de destino.