Supongamos que un sistema operativo basado en VFS.

a) ¿Es posible tener diferentes implementaciones de una operación (por ejemplo la asociada a un *read*) para los diferentes elementos del sistema? ¿Cómo permite VFS hacer eso?

A continuación se muestra a nivel de pseudocódigo las llamadas implicadas en una operación de lectura (llamada *read*).

```
sys_read(fd_idx, buff, size) {
  XXX* f=fd[fd_idx]->file;
  n=f->file_read(buff, f->offset, size);
  f->offset+=n;
  return n;
}
```

```
class XXX {
  file_read(buff,off,size){
    YYY* v=get_vnode(this);
    page=v->read_page(off, off+size);
    sys_mem_cpy(buff, page, off, size);
  }
  ...
};
```

```
class YYY {
  read_page(start,end) {
    return ZZZ::get_page(start,end);
  }
  ...
};
```

```
class ZZZ {
  get_page(start, end) {
    page=calculate_mem_page(start, end);
    if(is_not_present(page)) {
      schedule_dev_load(page);
      block_process(page->queue);
    }
    return page;
  }
  ...
};
```

b) En un sistema basado en VFS, ¿qué estructuras de datos se utilizan en el sistema de archivos? Indique: (1) cómo están enlazadas desde el BCP del proceso, (2) cuál de los fragmentos de pseudocódigo anteriores se corresponde a funciones que manipulen dichas estructuras y (3) que otros datos relevantes contienen estas estructuras. Sustituya *XXX*, *YYY* y *ZZZ* por los nombres habituales de las estructuras.

Sobre este sistema se plantea diseñar un nuevo objeto del sistema de ficheros, denominado *stream-log*. Los *stream-log* funcionan de acuerdo con las siguientes condiciones:

i. Almacena datos en bloques (de la misma forma que un fichero).
ii. Las escrituras siempre se realizan al final del contenido del *stream-log* (modo *append*).
iii. Las lecturas se pueden realizar en cualquier parte del contenido.
iv. Sincroniza lectores y escritores de la misma forma que un *pipe* con nombre:
    iv.i. Si se realiza una escritura y no hay lectores la escritura falla.
    iv.ii. Si se realiza una lectura al final del *stream-log* el lector se bloquea.

c) Si se quiere controlar la primera de las condiciones de sincronización (ver condición iv.i), ¿en cuál de las estructuras de datos habría que realizar el control? Modifique dicha estructura si fuese necesario e incluya la comprobación que se haría en pseudocódigo.

d) En el caso de la apertura múltiple del *stream-log* en modo escritura ¿cómo se garantizaría que siempre se escribe al final del mismo (condición ii)? Considere el caso de que haya varios procesos no emparentados escribiendo a la vez. Indique los cambios necesarios en las estructuras VFS antes citadas.

Utilizando las funciones *block_process(…)* y la análoga *release_process(…)* para controlar las colas de espera de eventos en las que se pueden bloquear procesos:

e) Implemente a nivel de pseudocódigo la llamada *read* sobre un *stream-log*, de forma que se cumpla la condición de sincronización iv.ii.

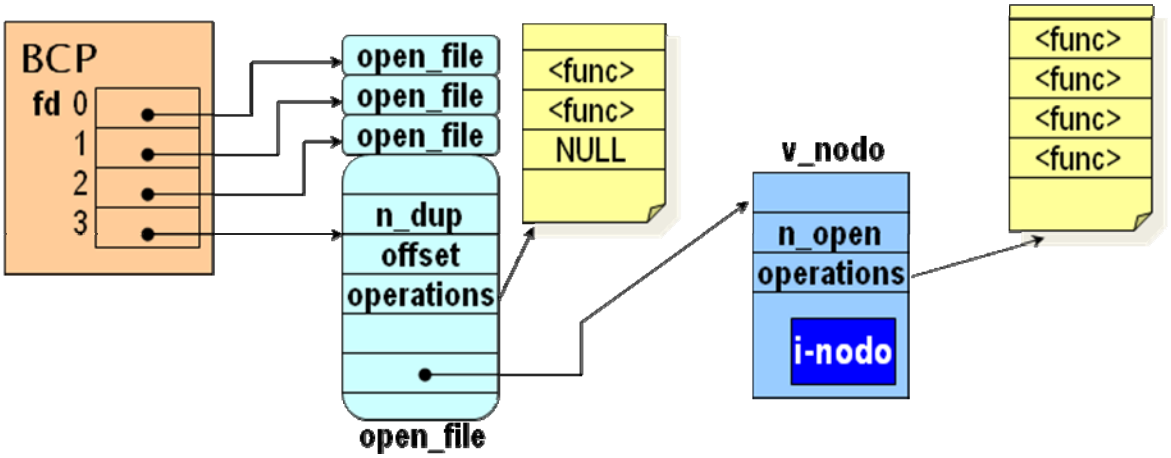
f) Ídem, pero ahora implementa a nivel de pseudocódigo la llamada *write*.

g) Tal y como se ha hecho la implementación y considerando las posibilidades que permite VFS, ¿sería posible usar un fichero regular ya creado como si fuese un stream-log? ¿podría hacerse con una opción especial al abrir? ¿qué debería hacer la operación de abrir que no hace en un caso normal? No incluya código, sólo coméntelo.

a) *¿Es posible tener diferentes implementaciones de una operación (por ejemplo la asociada a un read) para los diferentes elementos del sistema? ¿Cómo permite VFS hacer eso?*

VFS gestiona una serie de implementaciones de sistemas de ficheros. Estas implementaciones se cargan en sistemas como Linux de forma modular. Esto provee a VFS de varias funciones (por ejemplo de implementaciones de read). La asociación de una implementación a un elemento del sistema se hace por medio de tablas de operaciones (lo más general son punteros a funciones aunque en un modelo de objetos se puede hacer por medio de subclases).

b) *En un sistema basado en VFS, ¿qué estructuras de datos se utilizan en el sistema de archivos? Indique: (1) cómo están enlazadas desde el BCP del proceso, (2) cuál de los fragmentos de pseudocódigo anteriores se corresponde a funciones que manipulen dichas estructuras y (3) que otros datos relevantes contienen estas estructuras. Sustituya XXX, YYY y ZZZ por los nombres habituales de las estructuras.*



Las estructuras son la tabla de archivos abiertos (asociadas a los índices de descriptores de ficheros), la tabla intermedia de v-nodo – posición (tabla de ficheros abiertos) y la tabla de v-nodos.

Además de estas estructuras de datos habría que considerar la Cache de bloques

La traducción de los valores sería:

XXX → *Open_File*: que incluiría datos como en número de referencias (duplicados) que tiene y la posición (offset) del puntero de lectura/escritura.
YYY → *V_Node*: que incluiría el número de referencias (opens) que tiene desde la tabla de ficheros abiertos.
ZZZ → Cache: Caché de bloques. El contenido de la cache son los bloques de datos recuperados de disco.

c) *Si se quiere controlar la primera de las condiciones de sincronización (ver condición iv.i), ¿en cuál de las estructuras de datos habría que realizar el control? Modifique dicha estructura si fuese necesario e incluya la comprobación que se haría en pseudocódigo.*

Lo que diferencia un lector de un escritor es el modo de apertura del fichero (es algo que se fija en las opciones del open correspondiente). Por lo tanto dos elementos que compartan la misma estructura open_file actuarán de la misma forma (lector o escritor). Esto sucede por ejemplo entre dos procesos padre e hijo que desde sus respectivos BCP apunta a la misma

estructura open_file. Por otro lado una estructura open_file no es consciente de aperturas del mismo fichero (son dos estructuras que no tienen una relación directa). El nexo que las une es que ambas apuntan al mismo v_node. Supongamos que se llama size.

Los v_node tienen, como se ve en el gráfico, un campo que indica el número de estructuras open_file que les apuntan, es el campo n_open. Ese campo indica cuantas de las estructuras anteriores le apunta, pero no distingue entre las de lectura y la escritura. Para ello es mejor dividirlo en dos contadores, que llamaremos n_read_open y n_write_open, que son los contadores del número de estructuras open_file de lectura y escritura respectivamente.

El código de control quedaría como:

```
class open_file {
  file_write(buff,off,size){
    v_node* v=get_vnode(this);
    if(v->n_read_open==0)
      return(ERROR_NO_READERS);
  }
  ...
};
```

d) *En el caso de la apertura múltiple del stream-log en modo escritura ¿cómo se garantizaría que siempre se escribe al final del mismo (condición ii)? Considere el caso de que haya varios procesos no emparentados escribiendo a la vez. Indique los cambios necesarios en las estructuras VFS antes citadas.*

El indicador de la posición de escritura es un valor que aparece en la estructura open_file, el campo offset. Eso hace que dicho puntero se comparta entre todos los procesos emparentados que tienen la entrada correspondiente al decriptor, dentro del BCP, apuntando a la misma estructura. Sin embargo, el indicador del tamaño de un fichero reside en el v_node.

Una opción, es hacer que al abrir el fichero se inicialice offset al valor de size del v_node correspondiente. Eso es lo que se hace habitualmente cuando abres un fichero en modo append.

El problema se plantea cuando dos procesos, que no compartan la misma estructura open_file hacen una escritura. Al abrir el campo offset apunta al final del fichero, pero si uno de ellos escribe el contador del otro no se modifica. Para poder realizar esta operación correctamente ambas estructuras deben tomar el campo size como referencia.

```
sys_write(fd_idx, buff, size) {
  open_file* f=fd[fd_idx]->file;
  v_node* v=get_vnode(f);
  n=f->file_write(buff, v->size, size);
  v->size+=n;
  f->offset=v->size;
  return n;
}
```

Aunque no se incluye en la solución, es evidente que esta operación requiere de algún mecanismo de sincronización para asegurarse que el uso y actualización de size no se convierte en una condición de carrera. Por motivo de simplicidad no se incluye en esta solución.

e) *Implemente a nivel de pseudocódigo la llamada read sobre un stream-log, de forma que se cumpla la condición de sincronización iv.ii.*

Para implementar esta operación y la análoga de escritura podemos optar por su realización (con los aspectos especíicos que tiene a diferentes niveles (de open_file o de v_node). La decisión es hasta cierto punto arbitraria y depende de otras consideraciones de diseño en las

que no hace falta entrar a este nivel. Lo que sí que es importante es identificar la necesidad de una cola de espera para poder realizar el bloque de la operación de lectura. Esta cola se tiene incluir el elemento común a las operaciones de lectura y escritura, es decir que debe ser un campo del v_node. Si ponemos la cola en la estructura open_file sólo se compartiría por parte de los procesos que compartan un mismo open_file sobre el mismo fichero, pero no por parte de otros que hayan hecho un open de forma independiente sobre ese mismo fichero. A esta cola de espera que ponemos en el v_node la hemos llamado is_empty_queue;

```
class open_file {
  file_read(buff,off,size){
    v_node* v=get_vnode(this);

    if(off==v->size) // Condición de bloqueo
       block_process(v->is_empty_queue);

    page=v->read_page(off, off+size);
    sys_mem_cpy(buff, page, off, size);
  }
  ...
};
```

f) Ídem, pero ahora implementa a nivel de pseudocódigo la llamada write.

De forma análoga:

```
class open_file {
  file_write(buff,off,size){
    v_node* v=get_vnode(this);
    if(v->n_read_open==0)
      return(ERROR_NO_READERS);

    //realizamos la escritura
    page=v->read_page(off, off+size);
    // No incluimos el código de aprovisionamiento de una nueva página
    // en el caso de que el especio de esta se hubiese terminado
    sys_mem_cpy(page, buff, 0, size); // Copiamos en sentido inverso
                                      // (del buffer a la pagina)

    if(¡has_waiting(v->is_empty_queue) // SI la cola tiene procesos en espera
       release_process(v->is_empty_queue);
  }
  ...
};
```

g) Tal y como se ha hecho la implementación y considerando las posibilidades que permite VFS, ¿sería posible usar un fichero regular ya creado como si fuese un stream-log? ¿podría hacerse con una opción especial al abrir? ¿qué debería hacer la operación de abrir que no hace en un caso normal? No incluya código, sólo coméntelo.

En teoría sí, si tenemos una forma de abrir un fichero o de indicarle este modo de operación (como podría ser por medio de una IOCTL) se podría hacer. Lo que es importante es entender que debe modificar las operaciones asociadas a ese fichero para su lectura y escritura, es decir cambiar las implementaciones de file_read y file_write de la estructura open_file asociada. Hacer eso, en los modelos de VFS, aunque es poco habitual es posible.

Sin embargo hay un problema, supongamos que un fichero está abierto ya como fichero estándar para su escritura por un proceso y ahora otro proceso independiente lo quiere abrir como un stream-log. El nuevo proceso puede crear su open_file con la implementación que le convenga, pero el proceso que lo ha abierto antes lo usa como un fichero regular con la implementación estándar. Eso que implica, que, por ejemplo la posibilidad de que se quede bloqueado en la lectura, al final del stream-log cuando no hay más datos y se desbloquee al hacer una nueva escritura no existe. El open_file de lectura puede tener la implementación de la operación vista en el apartado e), pero el de escritura no tendrá la implementación de

f). Esto hace que uno, bajo unas determinadas condiciones se bloquee y el otro nunca lo desbloquee (no sabe que tiene que hacerlo).

En realidad es un grave problema tener implementaciones de diferente tipo manipulando las mismas estructuras, la única solución es bajar todas las implementaciones específicas al elemento común, el v_node. Eso implica que los elementos anteriores deben tener implementaciones estándar de todas las operaciones y que funcionen sobre cualquier cosa que haga el v_nodo. Algunas operaciones en el diseño (como la de escribir en modo append todos las operaciones de los escritires es difícil de implementar en el v_node, que desconoce la existencia de los punteros de posición, que están en los open_file.