Trabajo práctico de S.O I
Sistema de archivos distribuido

3 de marzo de 2014

1. Implementación en C con POSIX Threads

1.1. Comunicación entre workers

Utilizamos Posix Messages Queues para comunicar los distintos workers, encargados de realizar los pedidos del cliente. Tenemos 5 workers, y como muestra la Figura 1, asignamos una cola de mensajes a cada unos de ellos, con un número entre 1 y 5, con el fin de identificarlos. También, el hilo encargado de atender los pedidos del cliente (proceso socket) tiene una cola de mensajes.

Ante una llamada al sistema, el proceso socket, enviará la consulta a la cola de mensajes de algún worker (elegido pseudoaleatoriamente). Luego, este worker responderá al pedido, enviando el resultado a la cola de mensajes del proceso socket, previo a emitir un mensaje por el anillo, para consultar a los demás workers, si es necesario.

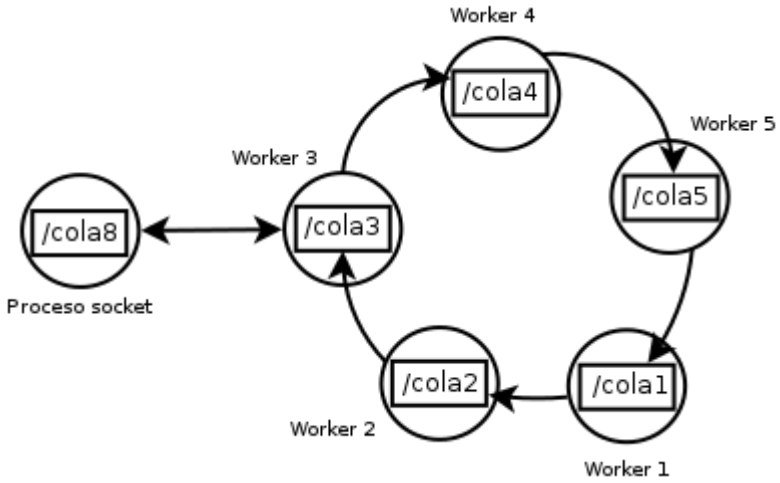

Figura 1: Procesos en anillo

1.2. Estructuras de datos

1. Representamos a los mensajes generados por procesos sockets y workers mediante la siguiente estructura:

```
typedef struct msj_{
  char tipo;
  char contador;
  char dato;
  int otrodato;
  char *nombre;
  char *texto;
} Msj;
```

Veamos como interpretamos cada uno de sus miembros:

- `tipo`: indica si es un mensaje entre workers, o entre el proceso socket y un worker.
- `contador`: varia entre los caracteres `'0'` y `'5'`. Lo usamos para determinar si el mensaje dio una vuelta por el anillo, o para identificar el worker que tiene como destino.
- `dato`: por lo general, indica cual fue el resultado de un pedido, exitoso o no.
- `otrodato`: contiene el tamaño en bytes del miembro `texto`. Es útil en los operadores `REA` y `WRT`.
- `nombre`: nombre de uno o varios archivos.
- `texto`: formará parte del contenido de un buffer de archivo.

2. Los archivos son representados por la siguiente estructura:

```
typedef struct archivo_ {
    char *nombre;
    char *texto;
    int estado;
    int indice;
    int tam;
    struct archivo_ *proximo;
} Archivo;
```

Interpretamos sus miembros de la siguiente manera:

- `nombre`: nombre del archivo.
- `texto`: buffer del archivo.
- `estado`: puede ser 0 o 1, indica si el archivo está cerrado o abierto, respectivamente.
- `indice`: posición del cursor de lectura.
- `tam`: tamaño en bytes del archivo.

1.3. Problemas y soluciones

1. La elección del descriptor de archivos.

   Creamos la función `ins_descriptor`, que devuelve el valor de una variable global, antes de ser incrementada. En su implementación tenemos en cuenta el uso de *memory barriers*. La llama el proceso socket, ante el pedido de apertura de un archivo, y utiliza el valor retornado como descriptor.

2. Cerrar todos los archivos abiertos por un cliente.

   Creamos una lista enlazada, mediante la estructura `ListaDes`, para contener los nombres de todos los archivos abiertos, junto a los clientes a los pertenecen, además de otros datos. Así, podemos filtrar de la lista los nombres de los archivos abiertos por un cliente determinado que desee desconectarse, para luego cerrarlos.

3. Comunicación entre un proceso socket y un FS.

   En un principio, el hilo que atendía los pedidos tenia que tomar el mensaje enviado por un worker directamente de la cola de mensajes de este worker; no habíamos creado una cola de mensajes propia a cada proceso socket. Entonces, consulta y respuesta se enviaban a la misma cola de mensajes. Lo cual generaba ciertos inconvenientes. Por ejemplo, un worker podría leer el mismo mensaje que había enviado, o un proceso socket tendría la chance de leer un mensaje emitido solo para circular por el anillo. Por lo tanto, decidimos asignarle una cola de mensajes a cada proceso socket, a medida que éstos son lanzados.

4. Pedidos concurrentes de creación de archivos con el mismo nombre.

   Cuando un worker recibe un pedido de creación de archivo, debe consultar a los demás. Cada worker revisa su lista de archivos, para determinar si el archivo ya existe. Hasta este punto, nuestro sistema de anillos, permitiría la creación de archivos duplicados. Entonces, un worker consultado, además de revisar su lista de archivos, examina si tiene pendiente un pedido de creación de archivo, y si éste coincide con el archivo buscado.

2. Implementación en Erlang

Realizamos los siguientes puntos adicionales.

- Permitimos la apertura de sólo lectura y sólo escritura, dejando múltiples aperturas en modo lectura, y a lo sumo una apertura en modo lectura y escritura.

  Para ello, modificamos la forma del operador `OPN` : `OPN MODE ARG1 ARG2`. Ahora, este operador abre el archivo `ARG2`, en el modo `ARG1`, donde `ARG1` puede ser `RDONLY` o `RDWR`.

- Implementamos el operador `RM`, de la forma `RM ARG1`. El archivo `ARG1` será borrado una vez que ya no se mantenga abierto por ningún cliente.

- Mensajería tolerante a fallas. Si un worker no responde al proceso socket, o el proceso socket no atiende al cliente, luego de aprox. 100 ms, el cliente recibirá el siguiente mensaje: `ERROR 62 ETIME`.

2.1. Comunicación entre workers

Reutilizamos parte del código del ejercicio 8, de la práctica de Erlang, en el cual debíamos implementar un Broadcaster. Esto nos permite comunicar cada worker con todos los demás, mediante un único proceso (lo llamamos broadcaster).

Cuando uno de los workers quiere consultar a lo otros, envía el pedido al broadcaster. Este se encarga de retransmitir el pedido a los 4 workers restantes, y juntar las respuestas en un solo mensaje, que luego transfiere al worker que inicio la consulta.

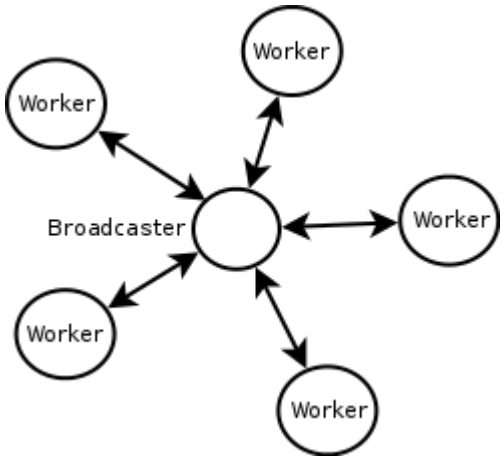

Figura 2: Sistema de broadcast

2.2. Estructura de datos

Implementamos un archivo con una tupla de la forma:

{{ 1 , 2 }, { `nombre` , 3 }, { `data` , 4 , 5 , 6 }}.

1. Atomo, que puede ser `cerrado`, `l` o `lye`. Indica que el archivo no fue abierto por ningún cliente, o el modo de apertura del último cliente, con una salvedad: si es abierto en modo lectura y escritura por un cliente `C`, este campo se mantiene invariable hasta que `C` lo cierre.
2. Atomo, que puede ser `true` o `false`. Indica si puede borrarse por pedido del operador `RM`.
3. Atomo, que tiene el nombre del archivo.
4. Buffer del archivo, una lista.
5. Lista de tuplas, contienen el índice de lectura asociado a un Pid de un proceso socket.
6. Tamaño en bytes del buffer dado en el punto 4.

2.3. Problemas y soluciones

1. El caracter `'\r '` que añade telnet en el fin de linea.

   Al principio, no lo tuvimos en cuenta. Por ejemplo, no lograbamos matchear `"BYE\r\n"` con `"BYE\n"`.

2. Cerrar todos lo archivos abiertos por un cliente.

   Para solucionarlo creamos una lista de tuplas de la forma {`N`,`M`,`P`}. Así interpretamos las variables:

   - `N`: Atomo que tiene el nombre de un archivo.
   - `M`: Atomo que denota el modo en el que archivo `N` fue abierto.
   - `P`: Pid del worker que tiene el archivo `N`.

   Pasamos la lista por argumento a la función `proc_socket`, para de mantener los nombres de los archivos abiertos, junto a los FS a los que pertenecen. Luego, cada archivo que se abre o cierra, añade o elimina respectivamente, un elemento de la lista, con el fin de mantenerla actualizada.

3. Soportar múltiples índices de lectura para un archivo.

   Este problema surge a raíz de la implementación de uno de los puntos adicionales. Tenemos que mantener un índice de lectura para cada cliente, incluso si todos leen el mismo archivo. Entonces, creamos en la tupla que representa un archivo, una lista de tuplas {`P`,`I`} (lo vimos en el sección 2.2).

   - `P`: es un Pid.
   - `I`: un número.

   Con esta tupla identificamos el índice `I` del cliente que tiene como proceso socket aquel con pid `P`. Actualizamos la lista, con la apertura, cierre o lectura del archivo por parte de un cliente, añadiendo, quitando o modificando la tupla correspondiente.

3. Observaciones

En ambas implementaciones:

- No se acepta la creación de archivos con nombres que incluyan espacios.
- No son admisibles operaciones con caracteres cuya codificación en UTF-8 requiera más de un byte.
- Solo es posible la comunicación cliente-servidor mediante el protocolo telnet.
- En la operación `WRT FD ARG0 SIZE ARG1 ARG2`, consideramos a los espacios contiguos en el buffer `ARG2` como un solo espacio. Además, en el archivo no incluimos el salto de linea de `ARG2`.

En la implementación en C, el buffer de cada archivo tiene un límite de 507 bytes de almacenamiento, y solo es posible enviar un pedido de escritura (mediante el op. `WRT`), donde el buffer `ARG2` tenga un tamaño de lo sumo 492 caracteres.

4. Integrantes

- Catacora, Joel.
- Ciunne, Melina.
- Oviedo, Juan Manuel.