Trabajo práctico de S.O I
Sistema de archivos distribuido

7 de agosto de 2014

1. Implementación en C con POSIX Threads

1.1. Comunicación entre workers

Creamos 5 hilos en el arranque del sistema, encargados de realizar los pedidos del cliente, *workers*. Asignamos a cada *worker* una cola de mensajes propia (utilizamos Posix Messages Queues para comunicar a los *workers* entre si, y un a *worker* con el proceso socket correspondiente).

Cuando un cliente se conecta, lanzamos un hilo, el cual atiende los pedidos del cliente, *proceso socket*. Este hilo también posee una cola de mensajes propia. Como muestra la Figura 1, cada *worker* se comunica con los *workers* adyacentes y a lo sumo un proceso socket.

Ante una llamada al sistema, el proceso socket, enviará la consulta a la cola de mensajes del *worker* X, elegido pseudoaleatoriamente entre los *workers* que estén desocupados. Luego, el *worker* X 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.

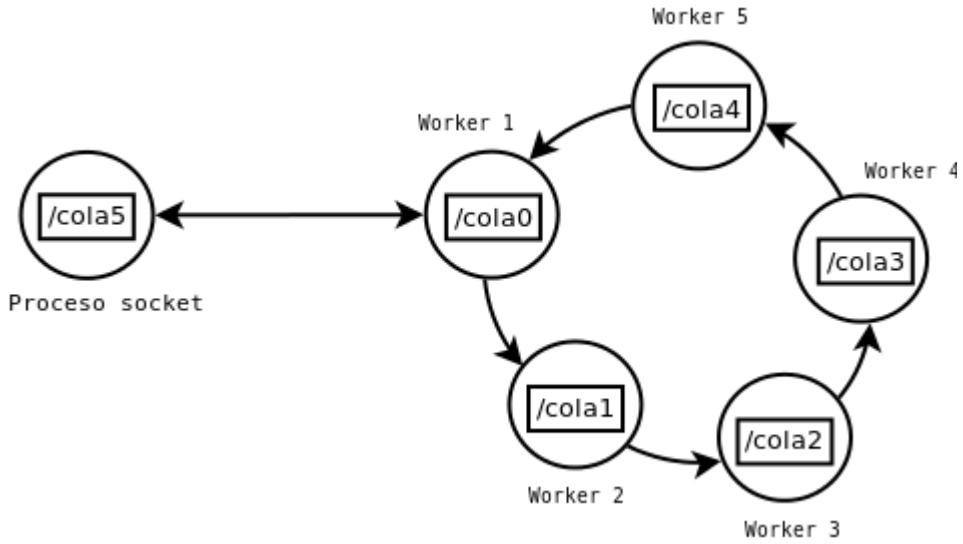

Figura 1: Comunicación en anillo entre workers

1.2. Estructuras de datos

1. Representamos a los mensajes enviados por procesos sockets o *workers* mediante la siguiente estructura:

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

Interpretamos a sus miembros de la siguiente manera:

- `tipo`: indica si se trata de un mensaje entre los *workers*, o entre el proceso socket y un *worker*.
- `contador`: varia entre los caracteres `'1'` y `'5'`. Lo usamos para identificar al *worker* que tiene como destino.
- `dato`: por lo general, indica el resultado de un pedido, si fue exitoso o no.
- `otrodato`: contiene el tamaño en bytes del miembro `texto`. Es útil para los operadores `REA` y `WRT`.
- `nombre`: el nombre de uno o varios archivos.
- `texto`: formará parte del contenido de un archivo.

2. Representamos a los archivos mediante la siguiente estructura:

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

Interpretamos a sus miembros de la siguiente manera:

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

1.3. Problemas y soluciones

1. La elección del descriptor de archivos.

   Creamos una función con el nombre `ins_descriptor`, que retornar el valor de una variable global, antes de ser incrementada. Cuando es llamada por el proceso socket, ante el pedido de apertura de un archivo, este hilo, utiliza el valor que obtiene de `ins_descriptor` como el descriptor del archivo.

2. Cerrar todos los archivos abiertos por un cliente.

   Creamos una lista enlazada global: `descriptores`, mediante una estructura llamada `ListaDescriptores`. Contiene los nombres de todos los archivos abiertos, junto a los *workers* a los pertenecen, entre otros datos. Creamos operadores que actúan sobre `ListaDescriptores`. En su implementación tuvimos en cuenta el uso de *memory barriers*. Con la función `buscar_descriptor`, podemos filtrar los nombres de los archivos abiertos por un cliente determinado que desee desconectarse, para luego cerrarlos.

3. La 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 para 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 el pedido de creación de un archivo, debe consultar a los demás. Entonces, cada *worker* revisa su lista de archivos, para determinar si el archivo ya existe. Hasta este punto, nuestra comunicación en anillo, permitiría la creación de archivos duplicados. Añadimos lo siguiente: cuando un *worker* es consultado, además de revisar su lista de archivos, examina si tiene pendiente un pedido de creación de archivo, y si el nombre de éste coincide con el nombre del archivo buscado.

2. Implementación en Erlang

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

Al igual que la implementación en `C`, lanzamos 5 hilos. Un *worker* envía su consulta a cada uno de los restantes *workers*, y luego espera las respuestas de todos ellos para continuar.

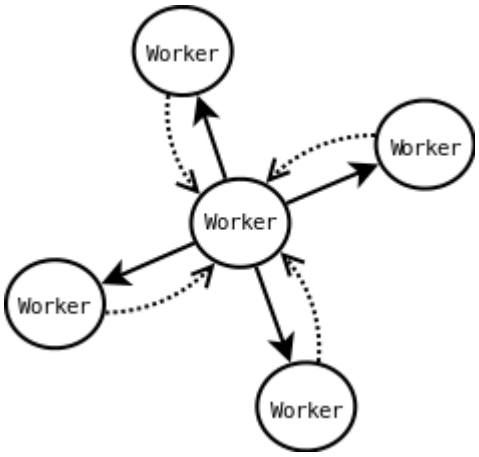

Figura 2: Comunicación de un worker

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 no cambia 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, con el índice de lectura asociado a un Pid (el de un proceso socket).
6. Tamaño en bytes del buffer del punto 4.

2.3. Problemas y soluciones

1. 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 de apertura del archivo con nombre N.
   - P: Pid del *worker* que tiene el archivo N.

   Pasamos la lista por argumento a la función `proc_socket`, para 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.

2. 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} (como vimos en la 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:

- Cada *worker*, realiza los pedidos de a lo sumo un solo proceso socket, en un momento dado. Por lo tanto, solo puede haber hasta 5 clientes a la vez (un cliente para cada *worker*).
- 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 que genera el cliente para confirmar el envio del mensaje.

En la implementación en C:

- El buffer de cada archivo tiene un límite de aproximadamente 500 bytes de almacenamiento.

4. Integrante

- Catacora, Joel.