Diseño Concurrentes – Proyecto DBMS

Objetivo

Escribir dos programas tales que uno se comportará como un **gestor de una base de datos** compuesta por una tabla de **personas**:

Nombre	char(61)
Dirección	char(120)
Teléfono	char(13)

El otro programa es un **cliente de la base de datos**. Este cliente puede **leer** el contenido de la base de datos y también **puede agregar nuevos registros**.

Se deberá **utilizar cola de mensajes** para comunicar el programa cliente con el gestor de la base.

Como **condiciones adicionales** a las planteadas por el ejercicio, se deberán cumplir las siguientes:

1. La base de datos se deberá persistir en almacenamiento permanente (archivo) cuando el gestor se cierra.
2. Se deberán poder ejecutar simultáneamente más de dos clientes que trabajen contra el gestor.

Diseño General

Procesos

El programa cliente de la base de datos contendrá un único proceso encargado de hacer las peticiones, ya sea de lectura o escritura, y su posterior puesta de información en pantalla.

El programa gestor de base de datos contendrá dos procesos. Estos son:

Proceso DBOrganizer, encargado de:

1. Obtener las peticiones entrantes de los clientes y mandarlas a procesar
2. Devolver las respuestas de las peticiones a los respectivos clientes

Proceso DBWorker, encargado de procesar la petición de cliente. La misma puede ser o bien de lectura o bien de escritura. En cada caso el proceso realizará lo siguiente:

- ***Lectura***: Buscará en la base de datos y en el archivo temporal los registros que cumplan con la condición de búsqueda de la petición; y los devolverá al organizador para su posterior devolución a los clientes.
- ***Escritura***: Escribe en el archivo temporal el nuevo registro indicado.

Comunicación entre procesos

Proceso DBOrganizer y Proceso DBWorker

La comunicación entre ambos se dará por una cola de mensajes. La lógica de comunicación es la siguiente:

Cuando el proceso ***DBOrganizer*** encuentra la petición del cliente que se debe procesar, el mismo lo escribe en la cola de mensajes y se quedará esperando a la respuesta del proceso ***DBWorker*** con mensajes de tipo *ANSWER*. El ***DBWorker*** lee la nueva petición de la cola de mensajes y la procesa.

Cuando el **DBWorker** finaliza dicho procesamiento, y posee todos los registros a devolver (Ya sea uno sólo que indica que la escritura se ha realizado con éxito, o varios que han cumplido con la condición de lectura) comienza a incluirlos en la cola de mensajes para enviárselos al ***DBOrganizer***. Cuando finalice con el envío de todos los registros, incluirá un mensaje extra indicando dicha situación en la estructura de mensaje (*EOMGS*)

El ***DBOrganizer*** irá leyendo los registros de tipo *ANSWER* hasta encontrar el mensaje indicando *EOMGS*. Esto hace que concluya la comunicación entre los procesos para dicha petición del cliente.

Procesamiento de la Base de Datos

Para la búsqueda en la base de datos, el procesamiento se hará primero buscando en el temporal, y luego en lo persistido. De haberse algo de lo persistido en lo temporal, se omitiría el registro persistido por su nueva actualización.

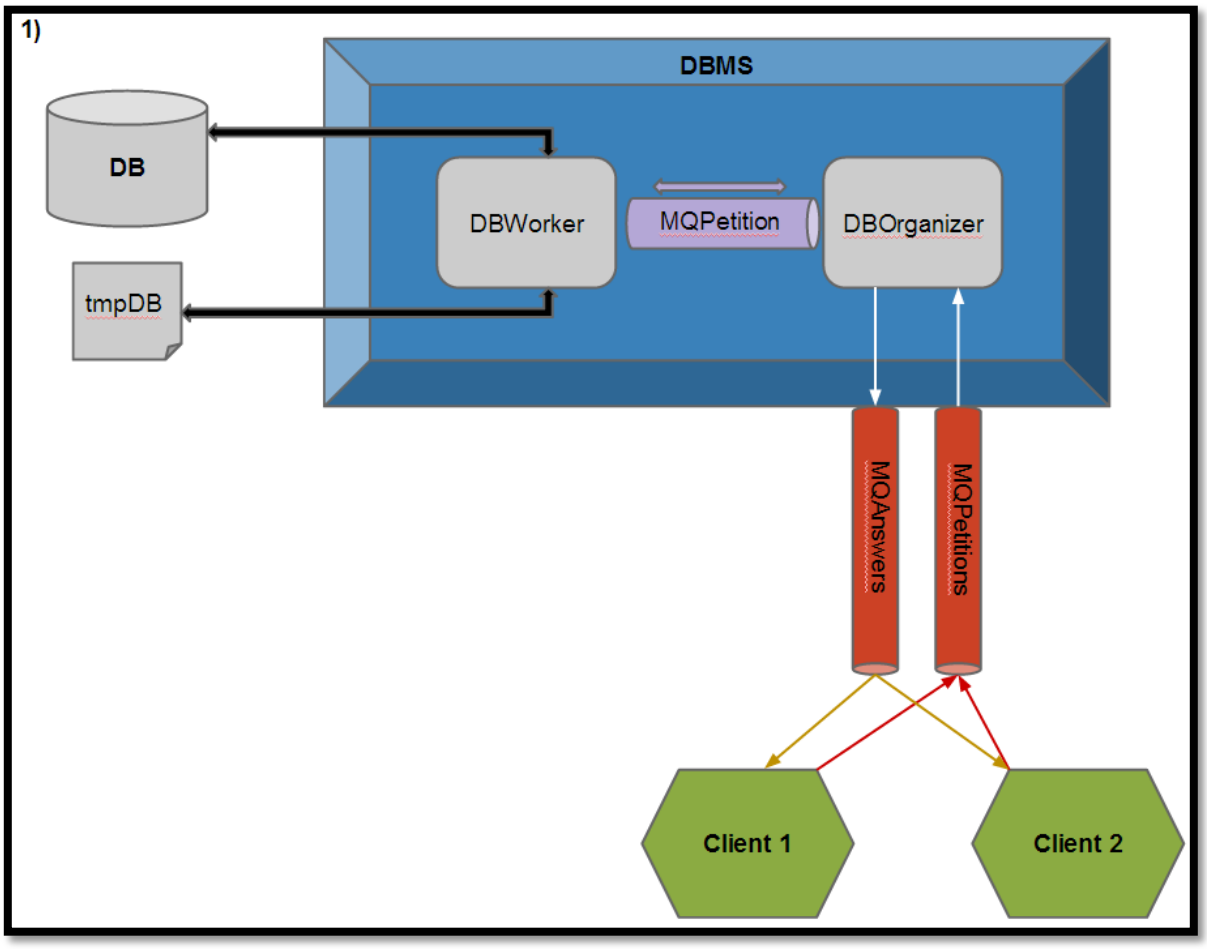
Para la persistencia en la base de datos, se procesará lo guardado en el temporal, obteniendo un temporal limpio (Por si se actualizo un mismo registro varias veces por ejemplo), y luego se procederá a actualizar/agregar los registros en la base de datos.

Cosas a tener en cuenta:

- Tiene que haber si o si 2 colas de mensajes para la comunicación entre el servidor y los clientes. De haber solo una entre servidor y clientes, no se podría utilizar la prioridad para determinar si el mensaje es de tipo *PETITION* o *ANSWER* **y a la vez** saber el tipo de cliente que lo realizó; de manera tal que un cliente este buscando por ejemplo su respuesta de mensaje de petición no podría buscar filtrando por *ANSWER* & *idCliente*
- El ***DBOrganizer*** podría dividirse en 2, uno que busca peticiones y otro que escribe respuestas.

Para el diseño de la comunicación entre clientes y el dbms se me ocurrieron varios diseños. De todos me termine quedando con solo 2 que describo abajo.

Diseño 1

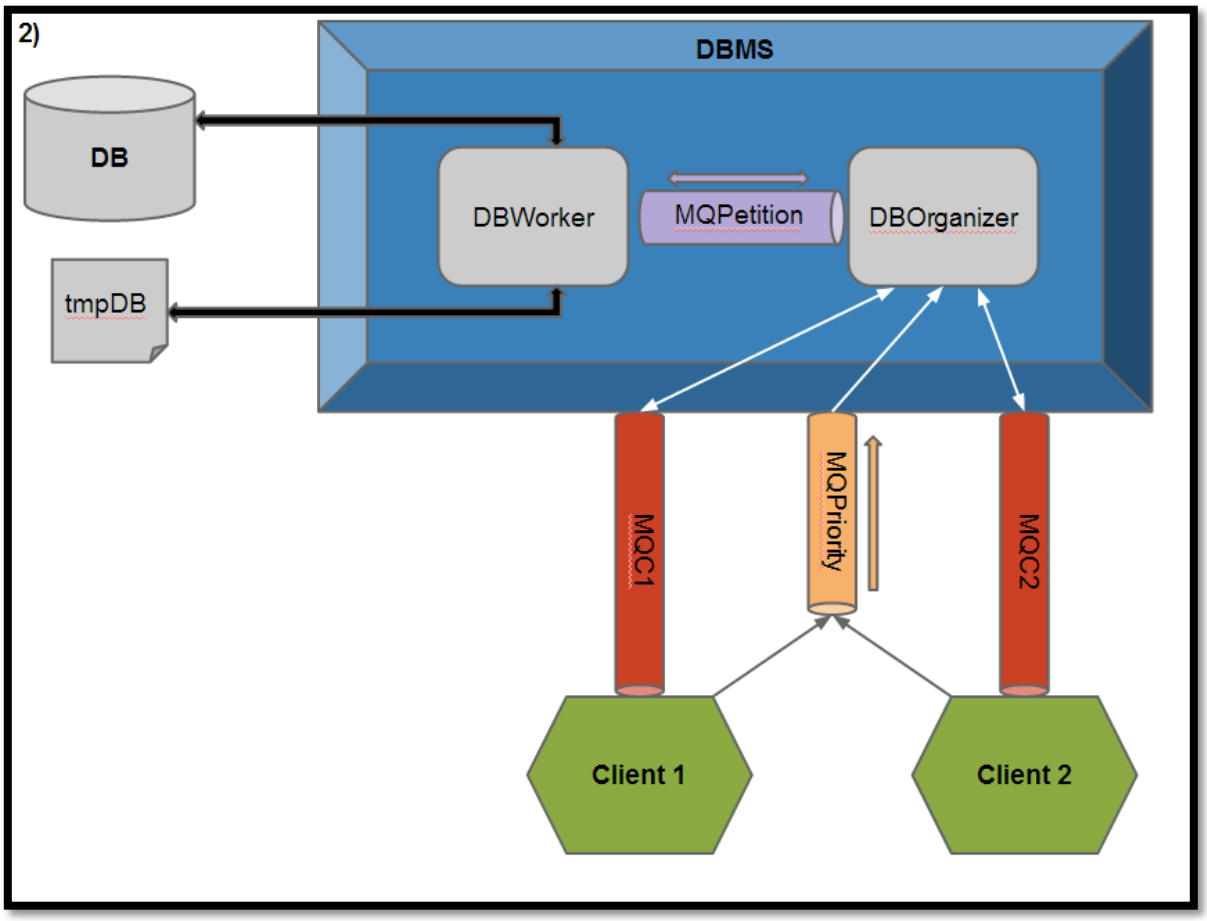


En este diseño, tenemos 2 colas de mensajes. Una de peticiones y otra de respuestas. Los clientes al querer efectuar una petición al DBMS, agregan el mensaje de petición en la cola ***MQPetitions***. La estructura del mensaje contendrá el id del cliente, para poder mantener una referencia para la respuesta final.

El ***DBOrganizer*** estará leyendo constantemente de ***MQPetitions*** para encontrar los nuevos mensajes. En el orden que lleguen las peticiones, las irá procesando (De manera iterativa, una petición por vez, como fue descripto en el diseño general).

La respuesta de las peticiones las colocará en la cola de respuestas ***MQAnswers***, indicando en los mensajes de respuesta el id del cliente que había solicitado la petición. De esta manera, los clientes luego de realizar la petición se quedan escuchando en ***MQAnswers*** por mensajes que contengan su id (Usamos la prioridad acá, para buscar mensajes propios), e irán obteniendo los mensajes de respuesta (Inclusive el mensaje de *EOMGS* que indica que finalizo la respuesta de su petición)

Diseño 2



En este diseño, hay una cola de mensajes ***MQPriority*** y una cola de mensajes por cada cliente. A diferencia del anterior, donde solo habían dos colas de mensajes, en este el cliente al hacer la petición primero escribirá en la cola de mensajes ***MQPriority*** un mensaje con su id, y luego escribirá el mensaje de petición en su cola propia ***MQCi***. El ***DBOrganizer*** irá obteniendo los mensajes de prioridad que le marcarán de donde leer la próxima petición para procesar y responder. (Si llega un mensaje de prioridad con un id que el ***DBOrganizer*** no tiene, indicará que hay un cliente nuevo, y se crea la respectiva conexión)

Feedback de mati

Comunicación entre cliente-servidor

Me cierra más el diseño 1 pues los programas los codeamos nosotros y le damos al usuario la posibilidad de modificar el armado del struct con el **PID** (ósea, estamos seguros que un cliente no lee del **MQAnswers** respuestas de otro proceso porque lo dejamos estáticamente leyendo por el **PID** que es único).

El diseño 2 es como que si, separas los mecanismos de devolución de respuesta pero no parece necesario y gasta más recursos del sistema al tener que crear las **MQ**.

Comunicación entre procesos del servidor

No me queda claro esto: "Cuando el proceso ***DBOrganizer*** encuentra la petición del cliente que se debe procesar, el mismo lo escribe en la cola de mensajes y se quedará esperando a la respuesta del proceso ***DBWorker*** con mensajes de tipo *ANSWER*".

¿Quiere decir que se loquea ahí esperando la respuesta?

- **Si la respuesta es sí**: no tiene sentido que sean dos procesos. Podría hacer el proceso el mismo y listo. Porque de todos modos se queda bloqueado.
- **Si la respuesta es no**: es como que agregamos un proceso más que esta encargado de quitar de una cola de mensajes los request y ponerlo en una cola más (como en un pasa mano para descargar cosas) si usamos el diseño 1 de comunicación entre cliente-servidor. Esto tiene sentido nada mas si usamos el segundo diseño de comunicación entre cliente servidor porque asi el **DBOrganizer** resuelve la lógica de crear y elegir por que **MQ** responde los request de los clientes.

Otra propuesta

Usando el diseño 1 para la comunicación clientes-servidor:

Servidor

Su ciclo de vida es

```
1. PrepareDatabase()
2. CreateMQs() //o lo que sea que tiene que hacer para levantar el
   canal de comunicación con los futuros clientes
3. while( isWorking )
      a. request = ReadFromMQRequest() // en un modelo cliente-
         servidor se usa la palabra request, no? Es lo mismo igual..
         es un nombre...
      b. registers = Process(request )
      c. structsResponce = CreateStructs( request.sender_pid,
         registers )
      d. SendToMQResponce( structsResponce )
```

```
4. PersistData()
5. ReleaseMQs()
```

El pseudo-código se explica solo.

Queda indicar:

1. **Como se llega a isWorking := false**: levantamos un cliente especial (como en modo administrador seteandole algún campo de entrada en **argv**[]) que envía el **request** de "finalizar servicio". Al recibir el **request** el procesar detecta esta orden y pone en false el **isWorking**. Luego se le responde al cliente-administrador que se llevo a cabo la operación y listo, el cliente termina y el servidor sale del **while** y termina.
2. El **CreateStructs()** recibe una lista de Personas y genera una lista de **structs** respuesta con el **pid** del cliente más el **struct** de fin de respuesta.

Ventajas:

Creo que es una solución más simple que nos ahorra la comunicación entre procesos y la coordinación para finalizar el servicio (se debe finalizar porque el enunciado dice: "la base de datos se deberá persistir… …cuando el gestor se cierre").

Esta resuelto el gracefull quit del server.

Clientes

Generamos un cliente que tiene este ciclo de vida

```
1. request = ParseRequest() // parsea los argumentos de entrada del
   programa
2. ConnectWithServer() // crea la conexión con las dos MQs
3. SendRequestToMQRequest( request )
4. partialResponce = ReadFromMQResponce()
5. While( EOMGS <> partialResponce.type )
      a. PrintPerson( partialResponce )
      b. partialResponce = ReadFromMQResponce()
6. DisconnectFromServer() // elimina su coneccion con las MQs
```

Creo que se entiende solo.

Esta solución levanta un proceso solo para hacer la request, printea y muere. Simple. Como la solución que nos dijo la profesora de levantar un proceso cada vez que pasamos una carta.

La lógica esta en el **ParseRequest()** que agarra los campos de **argv**[] para generar los campos filtros de persona, o verifica si el cliente es administrador y genera el "finalizar servicio".

Un ejemplo de la ejecución por consola seria:

```
➢ ./client –name matias –address * -phone *
➢ ./client –n matias –d * -p * // idem a la anterior
➢ ./client –name matias // idem a la anterior
➢ ./client –mode admin // cliente que manda el “finalizar servicio”
➢ ./client –m admin // idem anterior
```

El servidor se lanzaría

```
➢ ./server
```

Se podría mejorar lo del cliente administrador de la siguiente manera:

```
➢ ./server -key <key>
```

```
➢ ./client –m admin –k <key>
```

Acá lo que se agrega es que el cliente debe ingresar una clave que mandaría en el “finalizar servicio” si la clave no matchea con la que se lanza el server entonces no finaliza el servicio y en lugar de enviar al cliente un “listo finalize” le mandamos un “error de clave, no sos administrador”.

Para tener en cuenta en esta solución:

1. Si el proceso **init** tiene le **PID = 1**, podríamos enviar el request “finalizar servicio” con el ID = 1 y estamos seguros que no se va a confundir con otro proceso. Entonces el gestor lee en lugar de con 0, con 1, y si no retorna nada leer con 0. Entonces es como que el único mensaje con prioridad es el de ID = 1, que parece lógico porque es un cliente-administrador. En el caso de que hayan muchos request y el administrador da de baja el servicio, a todos los request se les responde con un “fin de servicio”. Y al proceso administrador, así como el manda con 1 se le responde con 1.
2. Necesitamos la key de las MQs, para esto generamos un archivo en /tmp que se llame “fileRequest.mq” y otro que se llame “fileResponce.mq”. El cliente cuando va a preparar la conexión se fija si están esos archivos. Si no están es porque no hay server corriendo. El server crea esos archivos al comenzar y los elimina la terminar.