Sistemas Operativos I

Informe sobre Trabajo Práctico Final

Licenciatura en Ciencias de la Computación

Ayrala, Isabel
Lojo, Dolores

20 de febrero de 2025

Introducción

Se pedía realizar un Mini Memcached Distribuido donde el servidor pueda manejar pedidos de clientes. Este servidor debía soportar multi-threading, tener un límite en la cantidad de memoria que iba a utilizar e implementar desalojos si es que esta memoria no es suficiente. Los pedidos de los clientes, por otro lado, se realizaban una vez establecida la conexión al servidor, y podían consistir en 4 tipos de pedidos diferentes: PUT, DEL, GET o STATS. Al hacer un pedido de PUT Clave Valor, lo que se está pidiendo es: ingresar en la base de datos el par (Clave, Valor). Por otro lado, si un cliente realiza un GET Clave, está pidiendo que se busque en la base de datos la clave en cuestión y devuelva su valor asociado. DEL Clave se refiere a la eliminación de esa clave y su valor correspondiente de la base de datos. Y por último, STATS pide al servidor las estadísticas del sistema: la cantidad de comandos de tipo DEL, GET, PUT que se realizaron y la cantidad de claves que hay insertadas en la tabla.

Breve descripción de nuestro código.

La estructura de nuestra tabla hash consiste en: un arreglo `CasillaHash*`, que sería una lista de las casillas de la tabla, una función comparadora (para comparar las claves de nuestra tabla), una función destructora para destruir nuestros ingresos en la tabla y una función hash para calcular el índice de un elemento.
El tipo `CasillaHash` consiste en otra lista. Por lo tanto, como lo indica su nombre, cada casilla de la tabla hash consiste en una lista. A su vez, `CasillaHash` es de tipo `HList*`, la llamamos `CasillaHash` para mayor claridad.
Los pares `(Clave, Valor)` que quiera insertar, eliminar u obtener un cliente, van a ser traducidos al tipo `Comando`, que consiste en una estructura con dos campos: `clave` de tipo `char*` y `valor` de tipo `char*`. Por otro lado, para representar la lista de política LRU elegimos la estructura `ListaLru` que consiste en: una `head` de tipo `HList*`, y una `tail` también de `HList*`. `head` y `tail` son de este tipo, pues cada nodo de la LRU se trata de un nodo de las listas de las casillas hash. Elegimos plantear así la estructura y tipo de la lista LRU, ya que pensamos que sería más eficiente que crear otra estructura paralela a la Tabla Hash, con sus propios nodos almacenados también en memoria. Pues estaríamos ocupando el doble de memoria y no es lo ideal.

Funciones relevantes del código

Una de las funciones más relevante de nuestro código es la función `safe_malloc()`, la cual se usa para cuando queremos realizar un `malloc()`. Como nuestro servidor tiene un límite en la memoria, tenemos que tener en cuenta el caso en el que lo estemos alcanzando, pues podemos no contar con la memoria suficiente para realizar un `malloc()`, por lo tanto tenemos que manejar ésta situación. Dentro de la función se realiza un `malloc()`, si este no retorna NULL, `safe_malloc()` es equivalente a `malloc()`, en caso contrario, como dijimos antes, tenemos que manejar el problema. La forma de manejarla es con un bucle `while`, que chequee si nuestro elemento (el que queremos alojar) es NULL, ya que en ese caso significa que no hemos podido reservar memoria. Dentro de ese bucle lo que se hace es llamar a la función desalojo y posteriormente realizar un `malloc()` para nuestro elemento. Como último punto, cabe aclarar que `safe_malloc()` no es utilizada en la creación de las estructuras principales, pues son esenciales para el funcionamiento de nuestro código.

Ahora podemos hablar de la función `desalojo`. En `desalojo()` utilizamos la política de "Last recently used". Se busca eliminar el ultimo elemento de la lista LRU, que sería el menos recientemente utilizado. Para eliminar el nodo se llama a `eliminar_nodo_tabla()`, en caso exitoso, retorna habiendo liberado memoria. Pero si esta retorna -1, intentaremos eliminar el penúltimo elemento menos re-

cientemente utilizado. Se continúa con este razonamiento hasta que se libere memoria o hasta que se recorra toda la lista LRU sin lograrlo, si esto sucede, vuelve a caer en el bucle de `safe_malloc()`. Éste seria un caso borde que podría pasar en situaciones muy específicas, pero como es algo poco probable, decidimos no manejarlo y centrarnos en los casos más comunes.

Veamos el papel que cumple `eliminar_nodo_tabla()` y su funcionamiento: se encarga de eliminar un par (Clave, Valor) de la Tabla Hash y la lista LRU. Esta función recibe una bandera `funcion` para indicar si venimos de un pedido DEL o del desalojo. Si bien la función se utiliza con el mismo propósito (eliminar un par) en ambos casos, la diferencia se encuentra en como debemos tomar el lock. Cuando se realiza el llamado a la función por un pedido DEL de un cliente, realizamos un `pthread_mutex_lock()` para bloquear la sección de la Tabla Hash en la cual se eliminara el par. En cambio, cuando venimos de desalojo, al estar bloqueando la lista LRU, puede haber secciones de la Tabla Hash que se encuentran bloqueadas y no pueden soltar el lock, ya que para ello (seguramente) necesitan, primero, tomar el lock de la lista LRU (la cual ya se encuentra bloqueada). Si utilizamos un `pthread_mutex_lock()` se podría generar un deadlock, pues vamos a estar esperando que la sección de la tabla haga `pthread_mutex_unlock()` y como dijimos recién, es probable que no pueda hacerlo. Por lo que utilizamos un `trylock()` para únicamente bloquear la sección de la Tabla Hash si es posible. Si no se pudo tomar el lock (trylock), retornamos -1 y `desalojo()` se encarga de ese caso.

Erlang

Pasando a la parte del código en Erlang, vamos a hablar un poco de cómo compilar el programa `cliente.erl`. Explicamos el procedimiento de a pasos, y tomamos como ejemplo el caso en que queramos realizar un put:

1. Abrimos una sesión de Erlang con `erl`
2. Establecemos conexión con el servidor con `c(cliente).`
3. Lanzamos el proceso con `cliente:start([IP1,...,IPN]).`, el cual recibe una lista de IPs (de los servidores conectados). El sistema nos devolverá un identificador de proceso, como `<0.87.0>`, por ejemplo. También nos pedirá ingresar un ID para identificar a cada cliente junto con la información que cada uno ingresa.
4. Llamamos a la función para testear con `cliente:test_put(N, <0.87.0>)`, la cual realizará N puts con clave N y valor N. Otra opción es hacer una llamada a put() ingreando el pid del proceso y la clave y valor a ingresar, `cliente:put(<0.87.0>, çlave", "valor").`
5. Esperamos respuesta del servidor y podemos seguir testeando con otras funciones.9

Una aclaración con respecto al Id que se ingresa por teclado. Su uso es más que nada para poder a acceder a claves y valores ya ingresadas por el mismo cliente en el caso de que vuelva a establecer una conexión con los servidores donde había almacenado su información previamente. Sin embargo, no funciona para realizar un status(). Si un cliente vuelve a conectarse, ingresa su ID y hace un status() no podrá ver esta información, esta función puede utilizarse (y funciona correctamente) unicamente cuando se realizan pedidos al servidor (PUTS).

Makefile

posee dos formas de compilación y ejecución:
- make (compila) y make run (ejecuta) es para cuando no se utiliza un puerto privilegiado.
- make privileged (compila) y make run_privileged (ejecuta) es para el uso de puertos privilegiados.

Puertos privilegiados

Para el uso de puertos privilegiados usamos la libreria `<sys/capability.h>`. Usamos código obtenido en "The Linux Programming Interface", capítulo 39, para la creación y uso de `modifyCap()` y `dropAllCaps()`. Decidimos usar la biblioteca `libcap`, con la cual podemos darle permisos a procesos (no root) sin necesidad de otorgarle los privilegios completos de root. `modifyCap()` modifica el estado de una capacidad dentro del conjuntos de capacidades efectivas de un proceso. Si queremos habilitar `cap_net_bind_service`, debemos compilar usando `sudo setcap cap_net_bind_service = p ./a.out`, esto le brinda la capacidad al ejecutable. `dropAllCaps()` elimina las capacidades del proceso una vez que ya las utilizó para bindearse con el puerto privilegiado.

Uso de Inteligencia Artificial

El uso de inteligencia artificial (Chat GPT) fue necesario para resolver algunas cuestiones.
Por ejemplo: al usar Valgrind para chequear pérdidas o errores en cuanto al manejo de memoria, no se entendían bien los resultados que daba Valgrind y necesitamos ayuda para entenderlos. También nos ayudo a aprender cómo inicializar y crear mutexes recursivos. Incluso, cuando ya pasamos mucho tiempo tratando de resolver algún problema en la ejecución, enviamos nuestro código para recibir alguna sugerencia y poder seguir avanzando. También lo utilizamos para la creación del makefile.

Problemas que surgieron

Algunos problemas surgieron durante el desarrollo de nuestro trabajo. Uno de ellos fue respecto al desalojo, quisimos implementar la idea de desalojar una cierta cantidad de elementos de nuestra lista LRU (más específicamente, el 10 %), pero nos encontramos con algunos inconvenientes. Primero, nos dimos cuenta que esto iba a costarnos tiempo, pues mientras estemos desalojando la lista LRU, íbamos a tomar el lock respectivo por un largo tiempo, y los demás hilos, si estaban necesitando tomar ese lock, no iban a poder continuar con sus tareas. Otra cosa que notamos, fue que si varios hilos se encontraban con que no había memoria, iban a desalojar y al eliminar tantos elementos, vaciaban la lista LRU. Al darnos cuenta de que implementar esta solución no iba a ser fácil, decidimos descartarla, ya que previamente habíamos implementado el desalojo de un solo elemento o de los elementos necesarios para recuperar memoria, y esta estrategia no nos había generado problemas.

Por otro lado, al momento de elegir el formato de la lista LRU, cuando empezamos a diseñar el trabajo, decidimos que los nodos de la lista LRU sean aparte de la Tabla Hash, por lo tanto reservábamos memoria tanto para los nodos de la lista LRU, como para los nodos de la Tabla Hash. Posteriormente, nos percatamos de que no iba a ser la mejor opción, pues estábamos desperdiciando memoria. Por eso decidimos que los nodos de la lista LRU sean los mismos que los de la Tabla Hash.

Otro problema a destacar, fue respecto a la estructura `epoll_event`. Esta estructura se crea en la función main, antes de lanzar los threads, y por lo tanto es global. No tuvimos en cuenta eso, y en la función `wait_for_clients()`, al aceptar una nueva conexión en el socket de escucha `lsock` y obtener el descriptor de socket para el cliente, se generaba un conflicto. Esto ocurría porque el descriptor de socket de escucha `lsock` se almacenaba en ésta variable global que mencionamos, lo que provocaba que se sobrescribiera durante cada intento de conexión. Lo solucionamos definiendo a la estructura `epoll_event` dentro de la función `wait_for_clients()` así los hilos la manejarían como una variable local.

Forma de trabajo

Trabajamos de forma virtual, siempre en videollamada para coordinarnos y resolver problemas juntas en el momento. Nos manejamos con un repositorio en GitHub para tener el código organizado.