DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

24 de junio de 2014 — Sistemas Operativos

Integrante	LU	Correo electrónico
Silvio Vileriño	106/12	svilerino@gmail.com
Ezequiel Gambaccini	715/13	ezequiel.gambaccini@gmail.com
Martin Arjovsky	683/12	martinarjovsky@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

Resumen

En este documento se explicaran y documentaran las decisiones tomadas con respecto al disenõ del servidor multi-thread desarrollado, se detallara el disenõ del mismo, se explicaran los obstaculos encontrados al realizar multi-threaded el servidor, por ejemplo las `race conditions`, como fueron solucionados y con que herramientas del estandar de POSIX fueron implementadas las soluciones. Se explica el nuevo manejo de los hilos del servidor multi al atender peticiones, y algunas modificaciones a tener en cuenta que surgieron a partir de la multiprogramacion generada en el servidor, entre ellas nuevas validaciones necesarias en el protocolo, que solo agregan funcionalidad, sin modificar el protocolo existente, y pueden ser facilmente adaptadas para que el protocolo original quede sin cambios.

1. Diseño: Modelo propuesto para servidor multihilo

Se utilizo como base el servidor mono, para realizar el nuevo diseno del servidor multi, que atiende varios clientes de forma concurrente. A continuacion se explica brevemente el funcionamiento del servidor mono y luego se explica como fue implementado el servidor que atiende pedidos concurrentes.

1.1. Funcionamiento del servidor mono

El servidor mono atiende un cliente a la vez en base al orden de llegada, y hasta que no termina con el cliente que está atendiendo, no pasa a atender al próximo.

El protocolo cliente-servidor que se utiliza es el siguiente:

1. El cliente indica su nombre y posición inicial. Para mantener la simplicidad esta posición no se chequea, por lo que será el cliente el encargado de enviar una posición válida.

2. El cliente indica el próximo movimiento, enviando ARRIBA, ABAJO, IZQUIERDA, DERECHA. El servidor recibe el pedido y responde OK u OCUPADO según corresponda.

3. Las partes repiten este ciclo hasta que el cliente haya salido del lugar. Una vez que esto ocurre, el servidor espera por un rescatista para el alumno y cuando lo obtiene, le coloca la máscara y envía al cliente LIBRE!

Si la conexión se interrumpe en medio de la huida, el servidor da de baja al cliente automáticamente. Ejemplo:

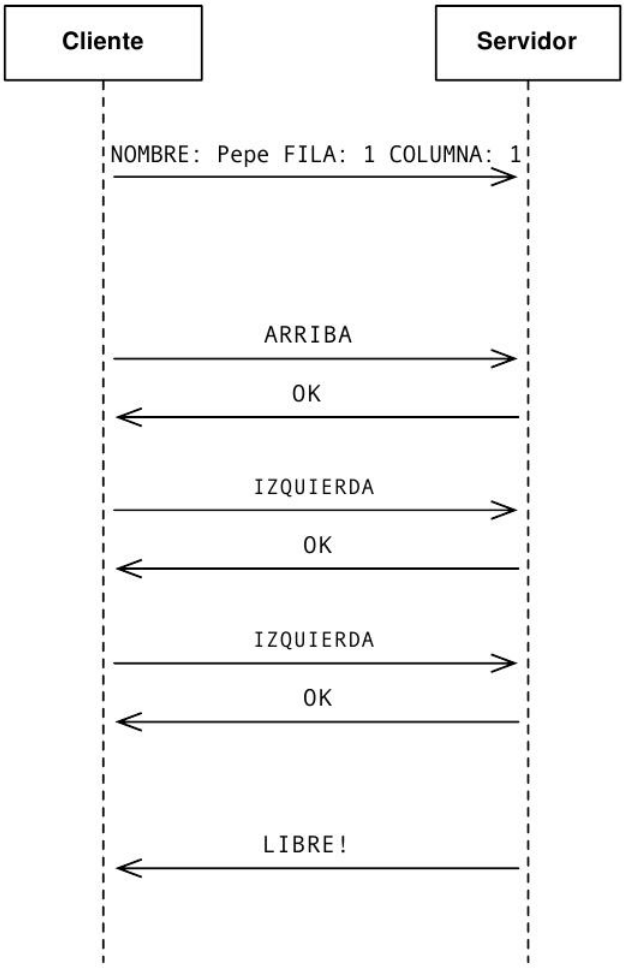

Figura 1: Ejemplo de escape exitoso (con puerta en posicion 0, -1).

1.2. Implementación del servidor multi

Nuestro servidor multi consiste de un thread principal y un thread worker por cada cliente conectado, separando en hilos diferentes las operaciones que realizaba el servidor mono, como se explica a continuación:

- El thread principal se encarga de detectar la conexión de nuevos clientes, tras lo cual crea un worker para que los atienda, si la funcion `pthread_create` devuelve error, se esperan unos segundos y se reintenta la creacion del mismo una cantidad fija de veces antes de ignorar el pedido.
- El worker se encarga de atender a los clientes usando el protocolo cliente-servidor especificado anteriormente, recibiendo como parámetros el socket donde se conectó el cliente, y la estructura compartida tipo `t_aula` aula.

Para sincronizar los accesos a los diferentes recursos compartidos del aula, y prevenir race conditions, se utilizaron diferentes estructuras de sincronización del estándar pthread, `pthread_mutex_t` y `pthread_cond_t`, en las funciones `t_aula_ingresar`, `t_aula_liberar`, `intentar_moverse`, y `atendedor_de_alumno`. La variable de condición solo fue utilizada para la función `atendedor_de_alumno`.

Una race condition es cuando varía la ejecución dependiendo del orden de acceso a un recurso compartido, si este orden no es el esperado. Para evitar esto, se fuerza un orden determinado mediante diferentes estructuras de sincronización, como mutex y semáforos.

Se utilizan en total 3 mutex y 1 variable de condicion:

- Se utiliza un mutex para sincronizar el acceso a la matriz de posiciones del aula.
- Se utiliza otro mutex para sincronizar el acceso a la cantidad de personas del aula.
- Por último, se utilizan en conjunto un mutex y una variable de condición para sincronizar el acceso a los rescatistas del aula.

Las diferentes funciones especificadas anteriormente acceden a recursos compartidos con todos los clientes, por lo que se usan las estructuras antes especificadas para que el acceso al recurso sea hecho de manera atómica y ordenada.

Una consideración especial que tomamos para nuestro servidor multi fue validar que la posición inicial este dentro del rango de la matriz, y que la celda correspondiente a esta posición tenga lugar disponible para el nuevo cliente. Si no se cumpliera alguna de las condiciones anteriores, se le envía un mensaje de error al cliente, se cierra la conexión y finaliza la ejecución de ese worker particular. Para realizar esto también se tuvo que tener en cuenta posibles race conditions, por lo que se sincronizo el acceso con los mutex especificados anteriormente.

1.3. Funcionamiento de los threads

Las operaciones que realiza el hilo principal son las siguientes:

1. Crear el socket servidor. bindearlo y empezar a escuchar.
2. Inicializar las estructuras de sincronización.
3. Quedarse en un loop infinito aceptando conexiones entrantes, creando workers en modo DETACHED para que atiendan las nuevas conexiones realizadas, pasando como parámetro el socket de la conexión y un puntero al aula. Se los crea en modo DETACHED ya que no importa cuando terminen ni su valor de retorno, son independientes al thread principal.

Las operaciones que realiza un hilo worker son las siguientes:

1. Recibir el nombre y posición del cliente.

2. Tratar de hacer entrar al alumno (nuevo cliente), al aula. Si lo logra, se prosigue con la operación normal, en caso contrario, como se especificó anteriormente, se envía un mensaje de error al cliente, se cierra la conexión, y se finaliza la ejecución de ese worker particular.

3. Luego, el cliente manda una dirección y el servidor le dice si puede avanzar o no. Se repite este ciclo hasta que el cliente sale. Las funciones de acceso al aula son de acceso atómico gracias a los mutex.

4. Mediante la combinación mutex/variable de condición, el worker se queda esperando hasta que la cantidad de rescatistas disponibles sea mayor a 0, para luego marcar que el alumno tiene la mascara puesta.

5. Finalmente, el worker emite un broadcast a la variable de condición para indicar que terminó, incrementa en 1 la cantidad de rescatistas, avisa al cliente que es libre, cierra la conexión, y finaliza su ejecución.

Dado que en la implementacion monothread, podia haber un único cliente moviendose por la matriz, salvo que la cantidad de personas por posicion fuera menor que uno, no había problemas de saturación al momento de entrar. Ahora, dado que puede haber muchos clientes moviendose por la matriz, se debe tener en cuenta la cantidad de personas actualmente en una posicion (x, y) ante un pedido de movimiento o de ingreso al aula. En esta situación, se debe validar que la posición destino a donde va a terminar el cliente que solicito el pedido, este en rango y ademas que haya lugar para esa persona. Esto provocó tener que crear un nuevo tipo de mensaje de respuesta del servidor (sin modificar el protocolo ya existente) que ante el pedido de ingreso al aula de una posicion (x, y) saturada, se le niegue el ingreso devolviendo dicho nuevo estado `POSICION_LLENA_O_FUERA_DE_RANGO`. Esto no modifica de ninguna forma el protocolo existente, dado que el cliente que vino en el bundle no valida errores. Si el servidor original devolvía errores (el mono devuelve error y el cliente no valida) el cliente no tiene comportamiento definido ante esto. Una forma de solucionar el problema de cambio de protocolo, es definir que `POSICION_LLENA_O_FUERA_DE_RANGO` sea equivalente a `ERROR`, cosa que ya estaba contemplada en el protocolo original presentado.

El tester python otorgado, tampoco validaba si el servidor devolvía `ERROR`, y se limitaba a enviar cíclicamente mensajes de forma serial a todos los sockets abiertos. Cuando el servidor devuelve error, cierra el socket y cuando el tester envia datos en el proximo ciclo a un socket cerrado hay un error. Para evitar esto, se validaron las respuestas del servidor, de forma tal que cuando un cliente era liberado/había error, se eliminaba del ciclo de proceso de peticiones.

Para testear paralelismo, se corren varios testers de python en paralelo usando bash. En este archivo se pueden encontrar variables de ajuste de parametros tales como cantidad de clientes por thread y cantidad de threads, lo que da una carga del servidor igual a *cantidad de threads* $\times$ *cantidad de clientes por thread*

2. Testing: Analisis de escalamiento

2.1. Race conditions y Leaks de memoria

El servidor multi fue testeado para ambas cosas con las herramientas `valgrind --leak-check=full` y `valgrind --tool=helgrind` y se intentó realizar una prueba con `thread-sanitizer` de `clang`, pero debido a problemas de instalación de `clang/compilacion` en `x86/x64` no se logró realizar los tests a tiempo.

2.2. Testing: Analisis de escalabilidad

A priori existe una cantidad de clientes $k \in \mathbb{N}$ tal que el servidor comienza a tener una ejecucion poco viable, lease consume mucha memoria ram, debido a la creacion excesiva de nuevos hilos, este valor es dificil de determinar dado que si queremos asegurar buen funcionamiento, debemos correr el servidor con valgrind para chequear leaks de memoria y problemas de pthreads y esto produce mucho overhead en el uso de ram del proceso del servidor sobre valgrind, pero a partir de un numero aproximado de 175 clientes concurrentes comienza a haber problemas de conexion, errores de `colabuf` de la biblioteca proporcionada y de lectura-escritura en los sockets.

2.2.1. Respecto a cambios en el hardware y la arquitectura fisica del sistema

Hasta cierto punto, puede ser viable agregar mas memoria ram y tener un servidor totalmente dedicado a atender clientes. El calculo de la RAM asignada al servidor debe hacerse con una estimacion aproximada del peor caso acerca de la carga total del servidor (cantidad de clientes simultaneos + 1), dado que este sera el numero maximo de hilos que puede manejar, habria que calcular en peor caso, cuanta memoria RAM utiliza cada thread(tienen entre otras cosas, pila y registros unicos por thread) y cuanta memoria ram usa el thread principal y el overhead del sistema operativo y la creacion del proceso que encapsula todos los threads. A partir de que esta solucion no mejore, se puede recurrir a sistemas distribuidos que balanceen la carga de forma mas uniforme y permitan el escalamiento a la cantidad de clientes requerida.

2.2.2. Respecto al software

Al utilizar threads y no procesos(hijos creados con fork, por ejemplo), se gana en performance dado que el fork es bastante mas costoso que un pthread_create. Asimismo los cambios de contexto tiene un overhead mucho menor en threads respecto a procesos, aproximadamente de un orden de magnitud teorico. Fueron utilizados en principio por netscape navigator y tuvieron exito dada la ganancia en performance mencionada anteriormente. Como desventaja, todos los threads comparten los mismos datos y es necesario tomar recaudos para la correcta sincronizacion entre ellos. Otro problema es que cualquier libreria que se use debe ser thread-safe. Como eleccion para maximizar el rendimiento y minimizar el consumo es buena la eleccion de threads sobre procesos como mencionamos anteriormente, pero ademas otra medida podria ser minimizar el uso de variables de stack en los threads, dado que esto aparentemente disminuiria muy poco, pero hay que multiplicarlo por la cantidad de hilos corriendo, puede ser una mejora significativa. Servidores Web como Apache utilizan threads, ver aqui `http://httpd.apache.org/docs/2.2/mod/worker.html`, una idea que podriamos tomar de ahi, es hacer uso de un `pool` de threads libres, los cuales sean asignados cuando se registra un pedido, evitando el overhead de la creacion del thread, y reponiendo en el `pool` otro hilo concurrentemente.

3. Apéndice: Entregable

Se entregara tanto el informe como el codigo completos, en un archivo comprimido.

3.1. Compilación y ejecución

Dado que la única modificación fue la creacion de un servidor paralelo servidor multi, con solo compilar tipeando `make clean all` se obtendrán todos los ejecutables necesarios para llevar a cabo cualquier prueba o ejecución del sistema. Para ejecutar el servidor se debe tipear `./servidor_multi` y para realizar las pruebas `multi_threaded_tester.sh`. Los parámetros modificables que permiten varios tipos de pruebas del servidor se encuentran en `biblioteca.h` y los de testing en `multi_threaded_tester.sh`.