DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2: `pthreads`

Escape en Sistemas

13 de noviembre de 2014

Sistemas Operativos

Integrante	LU	Correo electrónico
Rodriguez, Pedro	197/12	pedro3110.jim@gmail.com
Benegas, Gonzalo Segundo	958/12	gsbenegas@gmail.com
Barrios, Leandro Ezequiel	404/11	ezequiel.barrios@gmail.com

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) 4576-3359

http://www.fcen.uba.ar

Contents

Parte I Introducción

En el presente trabajo práctico empleamos técnicas de programación paralela. En particular, nos centramos en el diseño y la implementación de un servidor que a través del uso de múltiples hilos de ejecución (`multithreading`) se encarga de atender concurrentemente las peticiones a distintos clientes. Esto lo logramos siguiendo el modelo de *workers* propuesto por la cátedra en el enunciado del TP.

En el TP se nos propone resolver el problema de coordinar adecuadamente la evacuación de un edificio. En el modelo provisto por la cátedra, contamos con un sistema que procesa los pedidos de los clientes de forma secuencial. Y nuestro objetivo es usar técnicas de multiprogramación para procesar estos pedidos de forma paralela. En nuestro modelo, los clientes son personas que están en un espacio rectangular dividido en metros cuadrados. Cada persona está en algún metro cuadrado y cada baldosa tiene un límite predefinido en cuanto a la cantidad de personas que pueden estar en ese espacio al mismo tiempo.

Parte II Desarrollo

Para implementar el `servermulti` nos basamos en el código de `servermono` provisto por la cátedra. Las tareas que tuvimos que realizar para lograr la administración paralela de los pedidos por parte de los clientes fueron:

1. Para cada cliente que pida entrar a nuestro sistema, largamos un *thread* que se encargue de la comunicación entre `servermulti` y dicho cliente. Todos estos *threads* comparten el acceso a un mismo bloque de memoria. Es esta la razón por la cuál tuvimos que emplear varios `mutex` y `variables de condición` para administrar correctamente el acceso a estas posiciones de memoria.
   Observación: se utilizan variables de condición en vez de semáforos porque son los mecanismos con los que cuenta la especificación de `pthreads`.

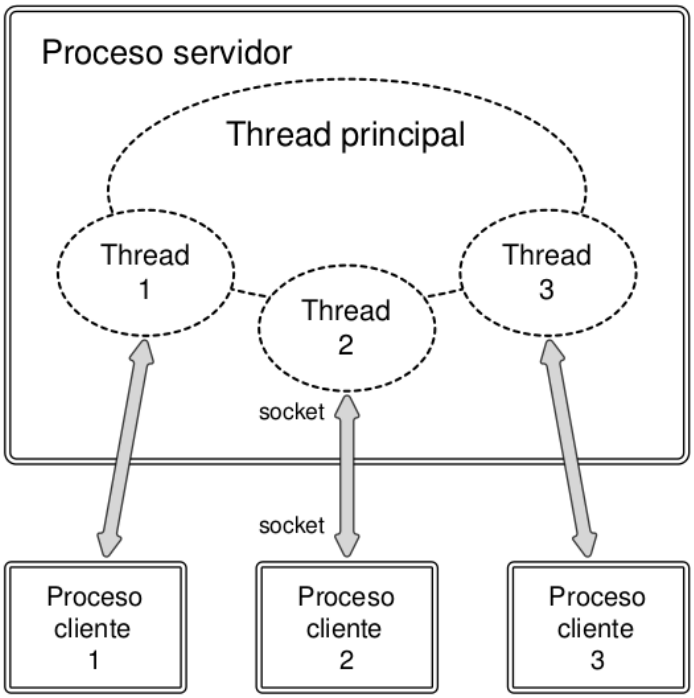

Figure 2.0.1: Croquis del nuevo diseño propuesto

2. Cada vez que un cliente pide desplazarse de una casilla del aula a otra, manejar correctamente el acceso a dichas celdas y actualizar correctamente las variables globales que determinan el estado actual del sistema en cada instante.

3. Manejar correctamente la salida definitiva del edificio de las personas que ya están afuera del mismo. Para esto tuvimos que tener cuidado con, para cada persona que logra salir de la habitación, determinar si hay algún rescatista libre en ese momento para ponerle la máscara y así poder posteriormente (cuando hayan llegado otras cuatro personas junto a ella) salir definitivamente del edificio y salvarse. Si no lo hay, la persona tiene que esperar a que llegue un rescatista y le ponga una máscara antes de poder salvarse definitivamente (podrían haber más de cinco personas fuera del aula pero esperando a que lleguen rescatistas para que le pongan la máscara y recién ahí, cuando hayan cinco personas con máscara, puedan salir).

Un problema importante que nos surgió fue el de evitar *deadlock* cuando dos a y b que están en las casillas A y B piden acceso a B y A respectivamente. Para arreglar este potencial problema,

lo que hicimos fue considerar un orden especial en el tomado de los `mutex`. Consideramos un **orden global**, de acuerdo al índice de la posición en la matriz, primero por fila y después por columna. En general, de esta forma prevenimos cualquier deadlock ocasionado por una situación de espera circular. Como no se cumple una de las condiciones de Coffman, podemos garantizar que el sistema está libre de deadlock.

Para manejar correctamente la espera de los rescatistas utilizamos dos variables: una para contabilizar la cantidad de personas fuera del edificio **con máscara** y por otro lado la cantidad de personas fuera del edificio pero **esperando máscara**.

Cuando testeamos nuestro código, vimos que el tiempo que tardaban los rescatistas en ponerle la máscara a cada una de las personas era muy corto. Entonces, para que la existencia de rescatistas tenga sentido y su existencia sea considerable cuando corremos nuestro `servermulti`, agregamos un tiempo de tardanza que tarda cada rescatista tarde en colocar una máscara a una persona.

2.1 Escalamiento

Cantidad de Puertos

La limitación más evidente es que, con la implementación actual, cada thread mantendría abierta una conexión local ocupando un puerto de origen distinto (todas con destino al puerto `5555`. Dado que el número de puerto es un `unsigned short`, la cantidad máxima de puertos es 65536. Por lo tanto, en estas condiciones estaríamos lejos de poder atender 1000000 de clientes.

Verificamos la afirmación del párrafo anterior mediante la ejecución del comando `netstat -tupaln`. Se adjunta a continuación un output de ejemplo, para 30 clientes.

Listing 1: netstat -tupaln

En un sistema en donde los clientes se conecten desde distintas IPS, este factor no representaría necesariamente una limitación, ya que cada cliente utilizaría el puerto que tuviera disponible (el servidor siempre utiliza el puerto 5555).

Recursos del sistema

Ya que la creación de un thread consume cierta cantidad de recursos, una decisión posible es tener un *pool* de threads creados al inicializar el servidor.

Una primera consideración es que, al crecer la cantidad threads, se les otorga *quantum* de procesador con cada vez menor frecuencia. Dada la urgencia de la evacuación, el *turnaround* puede ser muy alto para ser factible. La persona podría estar esperando mucho tiempo sin recibir respuesta por parte del servidor. La mejora propuesta, en este caso, es de hardware, y consiste en aumentar la cantidad de recursos.

El segundo aspecto a tener en cuenta, es la cantidad de memoria que ocupa cada *Thread*. En un acercamiento experimental lanzamos el servidor con 1000 clientes, y verificamos la cantidad de memoria ocupada del sistema operativo antes de lanzar la prueba (1GB), y luego de que se abrieran

todas las conexiones (4GB). La diferencia, 3GB, dividido 1000, la cantidad de clientes, da una cantidad de memoria de aproximadamente 3MB por thread.

Hay que tener en cuenta que en el análisis anterior se tomó la memoria del servidor, y la que ocupa cada cliente que se conecta, ya que todo el test corrió localmente. Intentamos realizar el test en forma remota, y por alguna razón (a pesar de que los clientes abrían la conexión correctamente) no logramos hacer que ninguno de los clientes se muevan.

Intentamos entonces realizar un monitoreo exhaustivo de la memoria ocupada por el proceso `server_multi`, junto con sus threads, a través de `ps`, `top / htop`, `valgrind`, pero al parecer el sistema operativo no brinda fácilmente esa información, ya que en todos los casos la memoria del proceso particular se mantenía constante luego de haber abierto cientos de conexiones (y por consiguiente, cientos de threads).

Bajo las consideraciones anteriores, asumimos entonces que necesitaríamos un máximo aproximado de 3MB por thread. En un servidor con 1 millón de clientes conectados al mismo tiempo, esto se traduce en 3TB de memoria, una cantidad que, si bien es inmensa, sigue siendo razonable en un entorno de `supercomputadora`.

Asumiendo que no estamos utilizando una supercomputadora, podemos pensar en una solución basada en un `pool de threads`. Esto es, a grandes rasgos, una estructura (ejemplo, un arreglo) de threads de cantidad fija o dinámica. A su vez, se mantiene una cola de tareas a ejecutar, las cuales se van asignando a los threads del `pool` a medida que estos se van liberando. Este modelo se complementa bien con nuestra implementación, ya que se basa en que cada thread del pool es un worker, lo mismo que ya ocurre con nuestra versión del servidor multithread. La versión más básica de un pool de threads mantiene siempre una cantidad fija de threads. Esto puede ser perjudicial, por ejemplo, si hay muy pocos threads efectivamente en uso (ej, pocas conexiones) estamos desperdiciando memoria y recursos para mantener muchos threads que no se están usando. Si por el contrario hay demasiados clientes, vamos a tener muchos threads ocupados constantemente. Suponiendo que la cantidad de clientes pueda variar inesperadamente, o que respete cierto patrón, otra posibilidad es utilizar una cantidad de threads dinámica, de forma tal que el servidor pueda realizar un autoescalamiento en la medida que le lleguen muchos clientes (y se mantengan en el tiempo), y que en el caso de que la cantidad de clientes descienda drásticamente pueda liberar los recursos utilizados por los threads que se encuentran disponibles.

Parte III Conclusión

Primero que nada, tenemos que decir que el trabajo práctico nos pareció muy entretenido. La implementación del modelo propuesto por la cátedra fue para nosotros un desafío ya que implicó el aprendizaje y manejo básico de herramientas importantes para un programador, como los `pthreads` o `variables de condición`.

Durante el desarrollo de la implementación, nos encontramos frente a los problemas comunes que se pueden presentar en la programación de sistemas paralelizados. En estos casos, tuvimos que aplicar las buenas prácticas de programación en este tipo de sistemas, como por ejemplo la protección de variables compartidas y/o memoria compartida.

Además, nos pareció muy estimulante acercarnos por primera vez a lo que es la programación de un servidor, y ver de primera mano cómo un servidor puede y debe ser tuneado para soportar una cantidad grande de pedidos por parte de una gran cantidad de clientes.