Imagen que contiene Icono

El contenido generado por IA puede ser incorrecto.Dibujo en blanco y negro

El contenido generado por IA puede ser incorrecto.

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ingeniería en Sistemas Computacionales

Aplicaciones para comunicaciones en red

Academia “Sistemas Distribuidos”

Plan 2020

Práctica 24:

“**Semáforos**”

2015090269

González González Armando Omar

Profesor: Ojeda Santillan Rodrigo

Contenido

[Objetivo 3](#_Toc201735601)

[Introducción 3](#_Toc201735602)

[Desarrollo 4](#_Toc201735603)

[3.1 Configuración Inicial 4](#_Toc201735604)

[3.2 Manejo de Conexiones de Clientes 4](#_Toc201735605)

[3.3 Función Principal y Ciclo del Servidor 5](#_Toc201735606)

[3.4 Implementación del Cliente 6](#_Toc201735607)

[Conclusión 9](#_Toc201735608)

[Pregunta 9](#_Toc201735609)

[¿Cómo podrían generar un negocio a través de lo visto en la práctica? 9](#_Toc201735610)

[Bibliografía 9](#_Toc201735611)

## Objetivo

Aplicar y demostrar el funcionamiento de los semáforos como mecanismo de sincronización en programación concurrente. Se simulará un balanceador de carga básico en lenguaje C que utiliza semáforos para gestionar y distribuir un número limitado de "servidores" (hilos) que procesan solicitudes de múltiples clientes de forma simultánea.

## Introducción

Los semáforos, concepto introducido por Edsger Dijkstra, son un mecanismo de sincronización fundamental en la programación concurrente y en los sistemas operativos. Su propósito principal es gestionar el acceso de múltiples procesos o hilos a recursos compartidos de manera controlada para evitar condiciones de carrera.

Un semáforo es, en esencia, una variable que controla el acceso a una sección crítica, que es una porción de código que accede a recursos compartidos y no debe ser ejecutada por más de un hilo a la vez.

Existen dos tipos principales:

* **Semáforo binario:** Actúa como un cerrojo, tomando solo valores de 0 o 1 para permitir o bloquear el acceso a un único recurso.
* **Semáforo contador:** Puede tomar valores enteros mayores a uno, permitiendo que un número limitado de hilos accedan a un conjunto de recursos.

El ciclo de vida de un semáforo se basa en dos operaciones atómicas:

1. **Wait (o P):** Decrementa el valor del semáforo. Si el valor es mayor a cero, el proceso continúa. Si es cero, el proceso se bloquea hasta que el recurso se libere.
2. **Signal (o V):** Incrementa el valor del semáforo, usualmente cuando un proceso ha terminado de usar el recurso. Esto puede despertar a un proceso que estaba bloqueado en la operación wait.

En esta práctica, se utilizará un semáforo contador para simular un balanceador de carga que distribuye solicitudes entrantes entre un número finito de hilos servidores.

## Desarrollo

Se implementó un balanceador de carga en lenguaje C (server.c) que simula la distribución de solicitudes de conexión entre varios servidores. El semáforo es la pieza clave que limita el número de solicitudes procesadas simultáneamente.

### **3.1 Configuración Inicial**

Se definieron constantes para el puerto (PORT), el número máximo de clientes en cola (MAX\_CLIENTS) y el número de servidores disponibles (NUM\_SERVERS). Se declaró un semáforo global sem\_t semaphore que controlará el acceso a los hilos servidores.

#define PORT 5555

#define MAX\_CLIENTS 10

#define NUM\_SERVERS 3

### **3.2 Manejo de Conexiones de Clientes**

La función handle\_server es ejecutada por cada hilo y se encarga de gestionar la solicitud de un cliente. Antes de procesar la solicitud, el hilo debe adquirir el semáforo con sem\_wait(&semaphore). Esto asegura que no se exceda el número máximo de servidores (NUM\_SERVERS) activos. Una vez procesada la solicitud, se libera el semáforo con sem\_post(&semaphore), permitiendo que otro hilo pueda atender una nueva conexión.

void \*handle\_server(void \*arg)

{

int client\_socket = \*(int \*)arg;

free(arg);

sem\_wait(&semaphore); // Controlar cuántos servidores procesan solicitudes a la vez

char buffer[1024];

int read\_size = recv(client\_socket, buffer, 1024, 0);

if (read\_size > 0)

{

buffer[read\_size] = '\0';

printf("Servidor procesando solicitud: %s\n", buffer);

char \*response = "Solicitud procesada exitosamente";

send(client\_socket, response, strlen(response), 0);

}

close(client\_socket);

sem\_post(&semaphore); // Liberar un servidor

pthread\_exit(NULL);

}

### **3.3 Función Principal y Ciclo del Servidor**

En la función main, se inicializa el semáforo con sem\_init(&semaphore, 0, NUM\_SERVERS), configurándolo como un semáforo contador que permite hasta NUM\_SERVERS accesos concurrentes.

El programa principal crea el socket del servidor, lo enlaza (bind) a la dirección y puerto especificados, y lo pone en modo de escucha (listen).

Finalmente, entra en un bucle infinito while que acepta conexiones de clientes con accept(). Por cada conexión aceptada, se crea un nuevo hilo (pthread\_create) que ejecutará la función handle\_server para procesar la solicitud de manera concurrente. Al terminar, se destruye el semáforo con sem\_destroy() para liberar los recursos.

int main()

{

int server\_socket, client\_socket,

\*new\_sock;

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t client\_addr\_size = sizeof(struct sockaddr\_in);

sem\_init(&semaphore, 0, NUM\_SERVERS);

server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket == -1)

{

printf("No se pudo crear el socket\n");

return 1;

}

printf("Socket creado\n");

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(PORT);

if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

{

perror("Error en bind");

return 1;

}

printf("Enlace completado\n");

listen(server\_socket, MAX\_CLIENTS);

printf("Esperando conexiones...\n");

while ((client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_addr\_size)))

{

printf("Conexión aceptada, distribuyendo al servidor disponible...\n");

pthread\_t server\_thread;

new\_sock = malloc(1);

\*new\_sock = client\_socket;

if (pthread\_create(&server\_thread, NULL, handle\_server, (void \*)new\_sock) < 0)

{

perror("Error al crear el hilo");

return 1;

}

pthread\_detach(server\_thread);

}

if (client\_socket < 0)

{

perror("Error en accept");

return 1;

}

sem\_destroy(&semaphore);

return 0;

}

### **3.4 Implementación del Cliente**

El programa cliente está diseñado para conectarse al balanceador de carga, enviar una solicitud y recibir una respuesta. Su lógica es la siguiente:

1. **Creación del Socket:** Se crea un socket TCP/IP usando la función socket().
2. **Configuración de la Dirección:** Se configura la estructura sockaddr\_in con la dirección IP del servidor (en este caso, 127.0.0.1 o localhost) y el puerto de escucha.
3. **Conexión:** Se establece la conexión con el servidor mediante la llamada a connect().
4. **Envío y Recepción:** Una vez conectado, el cliente envía un mensaje simple ("Hola") usando send() y luego espera una respuesta del servidor, que lee con la función read().
5. **Cierre:** Finalmente, imprime la respuesta del servidor en la consola y cierra el socket.

Este cliente puede ser ejecutado en múltiples instancias para simular una carga de varios usuarios simultáneos hacia el balanceador.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define PORT 5555

#define BUFFER\_SIZE 1024

int main(int argc, char const \*argv[]) {

int sock = 0;

struct sockaddr\_in serv\_addr;

char buffer[BUFFER\_SIZE] = {0};

char \*message = "Hola";

// Create socket file descriptor

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

printf("\n Socket creation error \n");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary form

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0) {

printf("\nInvalid address/ Address not supported \n");

return -1;

}

// Connect to the server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0) {

printf("\nConnection Failed \n");

return -1;

}

// Send message to server

printf("Conectado al balanceador de carga\n");

send(sock, message, strlen(message), 0);

// Read response from server

int valread = read(sock, buffer, BUFFER\_SIZE);

printf("Server response: %s\n", buffer);

close(sock);

return 0;

}

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 1. Ejecución del server.

A screenshot of a computer program

AI-generated content may be incorrect.  
Figura 2. Ejecución de 5 clientes para probar el funcionamineto del semáforo.

## Conclusión

La práctica demostró exitosamente el uso de semáforos contadores como una herramienta eficaz para gestionar el acceso a un conjunto limitado de recursos, en este caso, hilos servidores. Se observó cómo el semáforo impidió que más de NUM\_SERVERS clientes fueran atendidos simultáneamente, obligando a las nuevas solicitudes a esperar hasta que un "servidor" se liberara.

Se concluye que los semáforos son esenciales para resolver problemas de concurrencia y sincronización, permitiendo construir aplicaciones robustas y eficientes como los balanceadores de carga, que son críticos en arquitecturas de sistemas distribuidos para garantizar la estabilidad y el rendimiento.

## Pregunta

### **¿Cómo podrían generar un negocio a través de lo visto en la práctica?**

El concepto de balanceo de carga es un servicio fundamental en la nube. Un negocio podría ofrecer un servicio de "Balanceo de Carga como Servicio" (LBaaS). Los clientes (desarrolladores, empresas) pagarían una suscripción para dirigir el tráfico de sus aplicaciones a través de este servicio. La empresa se encargaría de gestionar la infraestructura, garantizando alta disponibilidad y distribuyendo la carga de manera óptima para evitar caídas del servicio del cliente, especialmente durante picos de tráfico.

## Bibliografía

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating System Concepts, 10th Edition. Hoboken, NJ: Wiley, 2018.

[2] W. Stallings, Operating Systems: Internals and Design Principles, 9th Edition. Harlow, United Kingdom: Pearson Education, 2017.

[3] M. J. Rochkind, Advanced UNIX Programming, 2nd Edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2004.