

PRÁCTICA 2

Sistemas Operativos

Roberto Loor.
A8590323

Ejercicio 1.
“COMUNICACIONES EN RED”

Planteamiento.

Implementar una aplicación cliente-servidor mediante la utilización de sockets cuya funcionalidad será la ejecución remota de comandos simples. El ejercicio constará de dos partes (procesos): el cliente que enviará los comandos a ejecutar al servidor y el servidor que recibirá las peticiones desde el cliente, ejecutará los comandos y devolverá los resultados al cliente.

El cliente se ejecutará en la máquina local y su función es establecer la comunicación con el servidor, enviar al servidor los comandos y mostrar los resultados. El sistema tiene que permitir la ejecución de múltiples comandos en una misma sesión, para ello, el proceso cliente mostrará un prompt en el que el usuario podrá ir invocando órdenes que se ejecutarán remotamente en el servidor, devolviendo los resultados de éstas al cliente, que mostrará por pantalla las respuestas obtenidas. El programa terminará cuando el usuario ejecute la orden "FIN". El cliente se lanzará con la orden:

ClienteRemoto IP_Servidor

El servidor se lanzará con la orden ServidorRemoto y debe estar en todo momento escuchando por el puerto 9999, preparado para aceptar peticiones.

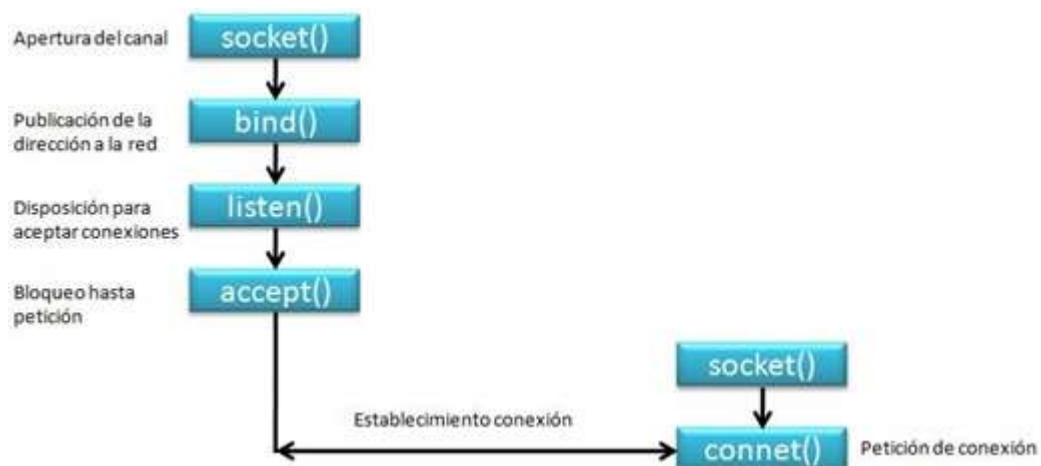
La ejecución de cada uno de los servicios por parte del servidor se realizará por procesos hijos independientes.

El servidor guardará una historia con todos los comandos ejecutados durante la sesión, historia que se podrá consultar con la orden "historia".

La puntuación del ejercicio será la siguiente:

- Implementación correcta del cliente y su estructura. (1 punto)
- Implementación correcta del servidor y su estructura. (1 punto)
- Ejecución remota de comandos. (3 puntos)
- Histórico de ejecución de órdenes. (1 puntos)

Estructura de aplicaciones Cliente-Servidor:



Implementación de Cliente.

Librerías usadas:

#include <stdio.h>: Librería estándar de C. En el programa se utiliza para imprimir mensajes en la consola.

#include <stdlib.h>: Librería estándar de C. En el programa se utiliza para definir constantes y funciones de utilidad.

#include <string.h>: Librería estándar de C. En el programa se utiliza para manipular cadenas de texto.

#include <unistd.h>: Librería estándar de C. En el programa se utiliza para acceder a funciones de sistema.

#include <arpa/inet.h>: Librería para manejar direcciones IP.

Constantes definidas:

#define PORT 9999: Puerto en el que el servidor escuchará las conexiones.

#define BUFFER_SIZE 1024: Tamaño del buffer para leer y escribir mensajes.

Función principal:

Es la función principal de un programa cliente de red que se conecta a un servidor utilizando sockets TCP.

El cliente espera recibir una dirección IP del servidor como argumento de línea de comandos, crea un socket, se conecta al servidor en el puerto especificado por la constante PORT, y luego entra en un bucle donde permite al usuario ingresar comandos. Los comandos se envían al servidor, y las respuestas del servidor se reciben y muestran al usuario. El bucle continúa hasta que el usuario ingresa el comando "FIN", momento en el cual el cliente cierra la conexión y sale del programa.

Si el servidor envía la cadena "unuseerver", el cliente informa al usuario de que el servidor se ha cerrado y que solo se puede usar el comando "FIN".

Capturas de pantalla del programa:

```
4 Clientes > @main(char*){
1  #include <stdio.h> // Librería estándar de C. En el programa se utiliza para imprimir mensajes en la consola.
2  #include <stdlib.h> // Librería estándar de C. En el programa se utiliza para definir constantes y funciones de utilidad.
3  #include <string.h> // Librería estándar de C. En el programa se utiliza para manipular cadenas de texto.
4  #include <unistd.h> // Librería estándar de C. En el programa se utiliza para acceder a funciones de sistema.
5  #include <arpa/inet.h> // Librería para manejar direcciones IP.
6
7  #define PORT 9999 // Puerto en el que el servidor escuchará las conexiones.
8  #define BUFFER_SIZE 1024 // Tamaño del buffer para leer y escribir mensajes.
9
10 int main(int argc, char *argv[]) { // Función principal del programa. Se ejecuta al iniciar el programa.
11     int sock; // Descriptor de socket
12     struct sockaddr_in server_addr; // Dirección del servidor
13     char buffer[BUFFER_SIZE]; // Buffer para leer y escribir mensajes
14     char *ip; // Dirección IP del servidor
15
16     if(argc == 2){ // Verificar si se proporcionó una dirección IP como argumento
17         if(argv[1] == "0")
18             ip = "127.0.0.1"; // Dirección IP por defecto
19         else
20             ip = argv[1]; // Dirección IP proporcionada como argumento
21     }
22     else{
23         printf("Error en los argumentos. Debe proporcionar la dirección IP del servidor.\n"); // Imprimir un mensaje de error en caso de que no se haya proporcionado una dirección IP como argumento
24         exit(1); // Salir del programa en caso de error
25     }
26
27     // Crear socket
28     sock = socket(AF_INET, SOCK_STREAM, 0); // Crear un socket de tipo TCP (SOCK_STREAM)
29     if (sock == -1) { // Verificar si se creó el socket
30         perror("Error al crear el socket"); // Imprimir un mensaje de error en caso de que no se haya creado el socket
31         exit(1); // Salir del programa en caso de error
32     }
33
34     // Configurar dirección del servidor
35     memset(&server_addr, 0, sizeof(server_addr)); // Limpiar la dirección del servidor
36     server_addr.sin_family = AF_INET; // Configurar la familia de direcciones (IPv4)
37     server_addr.sin_addr.s_addr = inet_addr(ip); // Configurar la dirección IP del servidor
38     server_addr.sin_port = htons(PORT); // Configurar el puerto del servidor
39
40     // Conectar al servidor
41     if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) { // Conectar al servidor
42         perror("Error en la conexión"); // Imprimir un mensaje de error en caso de que no se haya podido conectar al servidor
43         exit(1); // Salir del programa en caso de error
44     }
45
46     printf("Conectado al servidor.\n"); // Imprimir un mensaje para indicar que se ha conectado al servidor
47
48     while (1) { // Bucle para enviar comandos al servidor. Solo se sale del ciclo si se envía el comando "FIN".
49         printf("Ingrese comando: \n(FIN para finalizar la conexión) \n(HISTORIA para ver el historial de comandos introducidos desde el inicio de la conexión): "); // Imprimir un mensaje para solicitar un comando al usuario. Además, se indica que se puede
50         // utilizar el comando "FIN" para finalizar la conexión y el comando "HISTORIA" para ver el historial de comandos.
51         fgets(buffer, BUFFER_SIZE, stdin); // Leer el comando introducido por el usuario
52     }
```

```

C Servidor.c  C Cliente.c x
C Cliente.c @main.c:que x
10 int main(int argc, char *argv[]) { // Función principal del programa. Se ejecuta al iniciar el programa.
32 server_addr.sin_family = AF_INET; // Configurar la familia de direcciones (IPv4)
33 server_addr.sin_addr.s_addr = inet_addr(ip); // Configurar la dirección IP del servidor
34 server_addr.sin_port = htons(PORT); // Configurar el puerto del servidor
35
36 // Conectar al servidor
37 if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) { // Conectar al servidor
38     perror("Error en la conexión"); // Imprimir un mensaje de error en caso de que no se haya podido conectar al servidor
39     exit(1); // Salir del programa en caso de error
40 }
41
42 printf("Conectado al servidor.\n"); // Imprimir un mensaje para indicar que se ha conectado al servidor
43
44 while (1) { // Ciclo para enviar comandos al servidor. Solo se sale del ciclo si se envía el comando "FIN".
45     printf("Ingrese comando: \n(FIN para finalizar la conexión) \n(HISTORIA para ver el historial de comandos introducidos desde el inicio de la conexión): "); // Imprimir un mensaje para solicitar un
46     comando al usuario. Además, se indica que se puede utilizar el comando "FIN" para finalizar la conexión y el comando "HISTORIA" para ver el historial de comandos.
47     fgets(buffer, BUFFER_SIZE, stdin); // Leer el comando introducido por el usuario
48     buffer[strlen(buffer), '\n'] = 0; // Eliminar el salto de línea al final del comando. Esto se hace para evitar problemas al enviar el comando al servidor ya que se espera un comando sin saltos de línea.
49
50     // Enviar comando al servidor
51     write(sock, buffer, strlen(buffer)); // Enviar el comando al servidor
52
53     // Salir si el comando es "FIN"
54     if (strcmp(buffer, "FIN") == 0) { // Verificar si el comando es "FIN"
55         printf("Finalizando conexión.\n"); // Imprimir un mensaje para indicar que se está finalizando la conexión
56         break; // Salir del ciclo para enviar comandos
57     }
58
59     // Recibir respuesta del servidor
60     int str_len = read(sock, buffer, BUFFER_SIZE - 1); // Leer la respuesta del servidor
61     if (str_len == -1) { // Verificar si se leyó la respuesta
62         perror("Error en la lectura de datos"); // Imprimir un mensaje de error en caso de que no se haya leído la respuesta
63         break; // Salir del ciclo para enviar comandos
64     }
65
66     if (strcmp(buffer, "unusever") == 0) {
67         memset(buffer, '\0', BUFFER_SIZE); // Limpiar el buffer con guiones
68         printf("\nEl servidor se ha cerrado. No se puede ejecutar ningún comando nuevo, a excepción de 'FIN'. \n\n");
69     }
70
71     buffer[str_len] = '\0'; // Agregar un carácter nulo al final de la respuesta ya que se espera una cadena de texto sin saltos de línea
72     printf("Respuesta del servidor:\n%s\n", buffer); // Imprimir la respuesta del servidor
73 }
74
75 close(sock); // Cerrar el socket
76 return 0; // Retornar un valor entero para indicar que el programa finalizó correctamente

```

Funcionamiento individual:

```

robertolloor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$ make Cliente
make: 'Cliente' está actualizado.
robertolloor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$ ./Cliente
Error en la conexión: Connection refused
robertolloor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$

```

Implementación de Servidor.

Librerías usadas:

#include <stdio.h>: Librería estándar de C. En el programa se utiliza para imprimir mensajes en la consola.

#include <stdlib.h>: Librería estándar de C. En el programa se utiliza para definir constantes y funciones de utilidad.

#include <string.h>: Librería estándar de C. En el programa se utiliza para manipular cadenas de texto.

#include <unistd.h>: Librería estándar de C. En el programa se utiliza para acceder a funciones de sistema.

#include <arpa/inet.h>: Librería para manejar direcciones IP. En el programa se utiliza para configurar la dirección del servidor.

#include <sys/socket.h>: Librería para manejar sockets. En el programa se utiliza para crear sockets y establecer conexiones.

#include <sys/wait.h>: Librería para manejar procesos. En el programa se utiliza para esperar a que los procesos hijos terminen.

#include <sys/ipc.h>: Librería para manejar claves de memoria compartida. En el programa se utiliza para crear claves únicas para la memoria compartida.

#include <sys/shm.h>: Librería para manejar memoria compartida. En el programa se utiliza para compartir el historial de comandos entre procesos.

#include <sys/sem.h>: Librería para manejar semáforos. En el programa se utiliza para controlar el acceso concurrente al historial de comandos.

Constantes definidas:

#define PORT 9999: Puerto en el que el servidor escuchará las conexiones.

#define BUFFER_SIZE 1024: Tamaño del buffer para leer y escribir mensajes.

#define HISTORY_SIZE (BUFFER_SIZE * 10): Tamaño del historial de comandos.

Estructuras definidas:

```
struct shared_data { // Estructura de control para el historial compartido
    char history[HISTORY_SIZE]; // Historial de comandos.
    int history_index; // Índice de la próxima posición libre en el historial. Esto se utiliza para
    agregar nuevos comandos.
};
```

Esta estructura se utiliza para compartir el historial de comandos entre procesos dado que la memoria compartida no puede almacenar directamente cadenas de texto.

Función encargada del historial:

Esta función se encarga de agregar un comando al historial. Recibe como argumentos la estructura de datos compartidos, el semáforo, el comando y un indicador de si el comando fue reconocido. Luego, se bloquea el semáforo, se agrega el comando al historial y se libera el semáforo. El semáforo se utiliza para controlar el acceso a la memoria compartida.

Función encargada de la ejecución de los comandos enviados desde el cliente:

Esta función se encarga de ejecutar un comando en el sistema. Recibe como argumentos el comando a ejecutar y un buffer para almacenar el resultado. Primero, se verifica si el comando existe utilizando el comando "command -v". Si el comando existe, se ejecuta

utilizando la función "popen" y se lee la salida del comando. Luego, se almacena la salida en el buffer y se retorna un indicador de si el comando fue reconocido.

Command -v es un comando que se utiliza para verificar si un comando existe en el sistema. Si el comando existe, el comando "command -v" devuelve 0 como código de salida. De lo contrario, devuelve un código de salida diferente de 0.

Función principal:

La función principal de un programa servidor que utiliza sockets TCP para comunicarse con clientes.

El servidor crea un socket, se enlaza a una dirección IP y puerto específicos, y luego escucha conexiones entrantes. Cuando un cliente se conecta, el servidor acepta la conexión y entra en un bucle para recibir comandos del cliente.

Los comandos se ejecutan en procesos hijo separados, y los resultados se envían de vuelta al cliente.

El servidor también maneja un historial de comandos utilizando memoria compartida y semáforos, permitiendo a los clientes solicitar el historial de comandos introducidos. Si el cliente envía el comando "FIN", la conexión se cierra.

Si se envía el comando "killServer" y el servidor confirma, el servidor se cierra y se liberan los recursos de memoria compartida y semáforos.

Capturas de pantalla del programa:

```
1 // Servidor
2 #include <stdio.h> // Librería estándar de C. En el programa se utiliza para imprimir mensajes en la consola.
3 #include <stdlib.h> // Librería estándar de C. En el programa se utiliza para definir constantes y funciones de utilidad.
4 #include <string.h> // Librería estándar de C. En el programa se utiliza para manipular cadenas de texto.
5 #include <unistd.h> // Librería estándar de C. En el programa se utiliza para acceder a funciones de sistema.
6 #include <sys/types.h> // Librería para manejar direcciones IP. En el programa se utiliza para configurar la dirección del servidor.
7 #include <sys/socket.h> // Librería para manejar sockets. En el programa se utiliza para crear y conectar sockets.
8 #include <sys/wait.h> // Librería para manejar procesos. En el programa se utiliza para esperar a que los procesos hijos terminen.
9 #include <sys/ipc.h> // Librería para manejar claves IPC. En el programa se utiliza para generar claves únicas.
10 #include <sys/shm.h> // Librería para manejar memoria compartida. En el programa se utiliza para compartir datos entre procesos.
11 #include <sys/sem.h> // Librería para manejar semáforos. En el programa se utiliza para controlar el acceso a la memoria compartida.
12
13 #define PORT 9999 // Puerto en el que el servidor escuchará las conexiones.
14 #define BUFFER_SIZE 1024 // Tamaño del buffer para leer y escribir mensajes.
15 #define HISTORY_SIZE (BUFFER_SIZE * 10) // Tamaño del historial de comandos.
16
17 struct shared_data { // Estructura para almacenar los datos compartidos.
18     char history[HISTORY_SIZE]; // Historial de comandos.
19     int history_index; // Índice del historial.
20 };
21
22 void add_to_history(struct shared_data *shared_history, int semaphore, const char *command, int recognized) { // Función para agregar un comando al historial.
23     struct sembuf sb = {0, 1, 0}; // Estructura para operaciones de semáforo. Se utiliza para bloquear el semáforo. El 0 indica el índice del semáforo, el 1 indica que se va a restar 1 al valor del semáforo y el 0 indica que se va a esperar a que el valor del semáforo sea 0.
24     semop(&semaphore, &sb, 1); // Bloquear el semáforo.
25     sprintf(shared_history->history + shared_history->history_index, HISTORY_SIZE, "%s\n", command); // Agregar el comando al historial.
26     shared_history->history_index += strlen(command); // Si el comando fue reconocido, se agrega una cadena vacía. De lo contrario, se agrega "(No reconocido)".
27     shared_history->history_index += strlen(command) + recognized ? 1 : strlen(" (No reconocido)\n"); // Actualizar el índice del historial.
28     sb.sem_op = 1; // Configurar la operación del semáforo para liberarlo. El 1 indica que se va a sumar 1 al valor del semáforo.
29     semop(&semaphore, &sb, 1); // Liberar el semáforo.
30 }
31
32 int execute_command(const char *command, char *result) { // Función para ejecutar un comando.
33     int recognized = 1; // Indicador de si el comando fue reconocido. Se inicializa en 1.
34     char test_command[BUFFER_SIZE]; // Buffer para almacenar el comando de prueba.
35     sprintf(test_command, sizeof(test_command), "command -v %s > /dev/null 2>&1", command); // Crear el comando de prueba. Se redirige la salida estándar y de error a /dev/null para evitar que se muestre en la consola.
36     if (system(test_command) != 0) { // Verificar si el comando existe.
37         recognized = 0; // El comando no fue reconocido.
38         sprintf(result, BUFFER_SIZE, "El comando '%s' no existe.\n", command); // Almacenar un mensaje de error en el buffer.
39     } else { // El comando existe.
40         FILE *fp = popen(command, "r"); // Ejecutar el comando y obtener un puntero al archivo de salida.
41         if (fp != NULL) { // Verificar si se pudo ejecutar el comando.
42             printf("Ejecutando comando: %s\n", command); // Imprimir un mensaje para indicar que se está ejecutando el comando.
43             size_t total_read = 0; // Total de bytes leídos.
44             while (fgets(result + total_read, BUFFER_SIZE - total_read, fp) != NULL) { // Leer la salida del comando.
45                 total_read += strlen(result); // Actualizar el total de bytes leídos.
46                 if (total_read >= BUFFER_SIZE - 1) { // Verificar si se ha leído todo el buffer.
47                     break; // Salir del ciclo.
48                 } // Continuar leyendo la salida del comando.
49             }
50             pclose(fp); // Cerrar el archivo de salida.
51             printf("Comando finalizado: %s\n", command); // Imprimir un mensaje para indicar que se ha finalizado el comando.
52             printf("\n"); // Imprimir un salto de línea.
53         } else { // No se pudo ejecutar el comando.
54             recognized = 0; // El comando no fue reconocido.
55             sprintf(result, BUFFER_SIZE, "Error al ejecutar el comando.\n"); // Almacenar un mensaje de error en el buffer.
56         }
57     }
58 }
```

```

53 int execute_command(const char *command, char *result) { // Función para ejecutar un comando.
54 } else { // Si el comando existe.
55     return recognized; // Retornar el indicador de si el comando fue reconocido.
56 }
57
58 int main() {
59     int server_sock, client_sock; // Descriptores de socket para el servidor y el cliente.
60     int shmid = shmget(key, sizeof(struct shared_data), 0666 | IPC_CREAT); // Crear un segmento de memoria compartida.
61     struct sockaddr_in server_addr, client_addr; // Direcciones del servidor y del cliente.
62     socklen_t client_addr_size; // Tamaño de la dirección del cliente.
63     char buffer[BUFFER_SIZE]; // Buffer para leer y escribir mensajes.
64
65     key_t key = ftok("historial", 65); // Generar una clave única para la memoria compartida y los semáforos.
66     struct shared_data *shared_history = (struct shared_data *) shmat(shmid, NULL, 0); // Asociar la memoria compartida a la estructura de datos.
67     shared_history->history_index = 0; // Inicializar el índice del historial.
68
69     int semaphore = semget(key, 1, 0666 | IPC_CREAT); // Crear un semáforo para controlar el acceso a la memoria compartida.
70     semctl(semaphore, 0, SETVAL, 1); // Inicializar el valor del semáforo en 1.
71
72     server_sock = socket(PF_INET, SOCK_STREAM, 0); // Crear un socket de tipo TCP (SOCK_STREAM).
73     if (server_sock == -1) { // Verificar si se creó el socket.
74         perror("Error al crear el socket"); // Imprimir un mensaje de error en caso de que no se haya creado el socket.
75         exit(1); // Salir del programa en caso de error.
76     }
77
78     memset(&server_addr, 0, sizeof(server_addr)); // Limpiar la dirección del servidor.
79     server_addr.sin_family = AF_INET; // Configurar la familia de direcciones (IPv4).
80     server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // Configurar la dirección IP del servidor.
81     server_addr.sin_port = htons(PORT); // Configurar el puerto del servidor.
82
83     if (bind(server_sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) { // Asociar el socket con la dirección del servidor.
84         perror("Error en el bind"); // Imprimir un mensaje de error en caso de que no se haya podido asociar el socket.
85         exit(1); // Salir del programa en caso de error.
86     }
87
88     if (listen(server_sock, 5) == -1) { // Escuchar las conexiones entrantes.
89         perror("Error en el listen"); // Imprimir un mensaje de error en caso de que no se haya podido escuchar las conexiones.
90         exit(1); // Salir del programa en caso de error.
91     }
92
93     printf("Servidor escuchando en el puerto %d...\n", PORT); // Imprimir un mensaje para indicar que el servidor está escuchando en el puerto especificado.
94
95     while (1) { // Ciclo para aceptar conexiones de clientes. Este ciclo se ejecuta de forma indefinida o hasta que se cierre el servidor.
96         client_addr_size = sizeof(client_addr); // Obtener el tamaño de la dirección del cliente.
97         client_sock = accept(server_sock, (struct sockaddr*)&client_addr, &client_addr_size); // Aceptar la conexión de un cliente.
98         if (client_sock == -1) { // Verificar si se aceptó la conexión.
99             perror("Error en el accept"); // Imprimir un mensaje de error en caso de que no se haya aceptado la conexión.
100             continue; // Continuar con el ciclo para aceptar conexiones.
101         }
102         printf("Cliente conectado.\n"); // Imprimir un mensaje para indicar que un cliente se ha conectado.
103
104         while (1) { // Ciclo para recibir comandos del cliente. Este ciclo se ejecuta de forma indefinida o hasta que el cliente envíe el comando "FIN".
105             memset(buffer, 0, BUFFER_SIZE); // Limpiar el buffer.
106             int str_len = read(client_sock, buffer, BUFFER_SIZE - 1); // Leer el mensaje enviado por el cliente.
107             if (str_len == -1) { // Verificar si se leyó el mensaje correctamente.
108                 perror("Error en la lectura del mensaje"); // Imprimir un mensaje de error en caso de que no se haya leído el mensaje.
109                 break; // Salir del ciclo para recibir comandos.
110             }

```

```

111         // Servidor > @main
112         int main() {
113             while (1) { // Ciclo para aceptar conexiones de clientes. Este ciclo se ejecuta de forma indefinida o hasta que se cierre el servidor.
114                 while (1) { // Ciclo para recibir comandos del cliente. Este ciclo se ejecuta de forma indefinida o hasta que el cliente envíe el comando "FIN".
115                     if (str_len == -1) { // Verificar si se leyó el mensaje correctamente.
116                         break; // Salir del ciclo para recibir comandos.
117                     }
118                     buffer[str_len] = '\0'; // Agregar el carácter nulo al final del mensaje.
119                     printf("Comando recibido: %s\n", buffer); // Imprimir el comando recibido.
120
121                     if (strcmp(buffer, "FIN") == 0) { // Verificar si el comando es "FIN".
122                         printf("Conexión finalizada por el cliente.\n"); // Imprimir un mensaje para indicar que el cliente ha finalizado la conexión.
123                         break; // Salir del ciclo para recibir comandos.
124                     }
125
126                     if (strcmp(buffer, "HISTORIA") == 0) { // Verificar si el comando es "HISTORIA".
127                         write(client_sock, shared_history->history, strlen(shared_history->history)); // Enviar el historial de comandos al cliente.
128                         continue; // Continuar con el ciclo para recibir comandos.
129                     }
130
131                     if (strcmp(buffer, "killServer") == 0) { // Verificar si el comando es "killServer".
132                         printf("¿Desea cerrar el servidor? (S/N): "); // Solicitar confirmación para cerrar el servidor.
133                         char respuesta[1]; // Buffer para almacenar la respuesta.
134                         fgets(respuesta, 2, stdin); // Leer la respuesta del usuario.
135                         if (respuesta[0] == 'S') respuesta[0] = '\0'; // Verificar si la respuesta es "S" o "s".
136                         printf("Cerrando servidor...\n"); // Imprimir un mensaje para indicar que se está cerrando el servidor.
137                         write(client_sock, "muere", 5); // Enviar un mensaje al cliente para indicar que el servidor se cerrará.
138                         close(client_sock); // Cerrar el socket del cliente.
139                         close(server_sock); // Cerrar el socket del servidor.
140                         shmctl(shmid, IPC_RMID, NULL); // Eliminar el segmento de memoria compartida.
141                         semctl(semaphore, 0, IPC_RMID); // Eliminar el semáforo.
142                         exit(0); // Salir del programa.
143                     } else if (la respuesta es "S" o "s".
144                         printf("Continuando con el servidor...\n"); // Imprimir un mensaje para indicar que se continuará con el servidor.
145                         continue; // Continuar con el ciclo para recibir comandos.
146                     }
147                 }
148
149                 pid_t pid = fork(); // Crear un proceso hijo para ejecutar el comando.
150                 if (pid == 0) { // Verificar si se está en el proceso hijo.
151                     close(server_sock); // Cerrar el socket del servidor en el proceso hijo.
152
153                     int recognized; // Indicador de si el comando fue reconocido.
154                     char result[BUFFER_SIZE] = {0}; // Buffer para almacenar la respuesta del comando.
155
156                     recognized = execute_command(buffer, result); // Ejecutar el comando y obtener el indicador de si fue reconocido.
157
158                     add_to_history(shared_history, semaphore, buffer, recognized); // Agregar el comando al historial.
159
160                     write(client_sock, result, strlen(result)); // Enviar la respuesta del comando al cliente.
161
162                     close(client_sock); // Cerrar el socket del cliente en el proceso hijo.
163                     exit(0); // Salir del proceso hijo.
164                 } else if (pid < 0) { // Verificar si hubo un error al crear el proceso hijo.
165                     perror("Error en el fork"); // Imprimir un mensaje de error en caso de que no se haya podido crear el proceso hijo.
166                 }
167             }

```



```

182 int main() {
183     while (1) { // Ciclo para aceptar conexiones de clientes. Este ciclo se ejecuta de forma indefinida o hasta que se cierre el servidor.
184         while (1) { // Ciclo para recibir comandos del cliente. Este ciclo se ejecuta de forma indefinida o hasta que el cliente envíe el comando "FIN".
185             if (strcmp(buffer, "HISTORIA") == 0) { // Verificar si el comando es "HISTORIA".
186                 continue; // Continuar con el ciclo para recibir comandos.
187             }
188             if (strcmp(buffer, "killServer") == 0) { // Verificar si el comando es "killServer".
189                 printf("¿Desea cerrar el servidor? (S/N): "); // Solicitar confirmación para cerrar el servidor.
190                 char respuesta[2]; // Buffer para almacenar la respuesta.
191                 fgets(respuesta, 2, stdin); // Leer la respuesta del usuario.
192                 if (respuesta[0] == 'S' || respuesta[0] == 's') { // Verificar si la respuesta es "S" o "s".
193                     printf("Cerrando servidor...\n"); // Imprimir un mensaje para indicar que se está cerrando el servidor.
194                     write(client_sock, "unuse", 5); // Enviar un mensaje al cliente para indicar que el servidor se cerrará.
195                     close(client_sock); // Cerrar el socket del cliente.
196                     close(server_sock); // Cerrar el socket del servidor.
197                     shmctl(const void *) shared_history; // Desasociar la memoria compartida.
198                     shmctl(shmid, IPC_RMID, NULL); // Eliminar el segmento de memoria compartida.
199                     semctl(semaphore, 0, IPC_RMID); // Eliminar el semáforo.
200                     exit(0); // Salir del programa.
201                 } else { // La respuesta no es "S" ni "s".
202                     printf("Continuando con el servidor...\n"); // Imprimir un mensaje para indicar que se continuará con el servidor.
203                     continue; // Continuar con el ciclo para recibir comandos.
204                 }
205             }
206         }
207     }
208     pid_t pid = fork(); // Crear un proceso hijo para ejecutar el comando.
209     if (pid == 0) { // Verificar si se está en el proceso hijo.
210         close(server_sock); // Cerrar el socket del servidor en el proceso hijo.
211         int recognized; // Indicador de si el comando fue reconocido.
212         char result[BUFFER_SIZE] = {0}; // Buffer para almacenar la respuesta del comando.
213         recognized = execute_command(buffer, result); // Ejecutar el comando y obtener el indicador de si fue reconocido.
214         add_to_history(shared_history, semaphore, buffer, recognized); // Agregar el comando al historial.
215         write(client_sock, result, strlen(result)); // Enviar la respuesta del comando al cliente.
216         close(client_sock); // Cerrar el socket del cliente en el proceso hijo.
217         exit(0); // Salir del proceso hijo.
218     } else if (pid < 0) { // Verificar si hubo un error al crear el proceso hijo.
219         perror("Error en el fork"); // Imprimir un mensaje de error en caso de que no se haya podido crear el proceso hijo.
220     }
221     waitpid(pid, NULL, 0); // Esperar a que el proceso hijo termine.
222     close(client_sock); // Cerrar el socket del cliente.
223 }
224 shmctl(const void *) shared_history; // Desasociar la memoria compartida.
225 shmctl(shmid, IPC_RMID, NULL); // Eliminar el segmento de memoria compartida.
226 semctl(semaphore, 0, IPC_RMID); // Eliminar el semáforo.
227 close(server_sock); // Cerrar el socket del servidor.
228 return 0; // Devolver 0 para indicar que el programa finalizó correctamente.
229 }

```

Funcionamiento individual:

```

robertoor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$ make Servidor
cc Servidor.c -o Servidor
robertoor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$ ./Servidor
Servidor escuchando en el puerto 9999...

```

Funcionamiento individual cierre:

```


robertoor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$ make Servidor
cc Servidor.c -o Servidor
robertoor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$ ./Servidor
Servidor escuchando en el puerto 9999...
^Z
[1]+ Detenido ./Servidor
robertoor@MiLinux:~/Escritorio/Sistemas Operativos/Práctica 2$

```

NOTA:

Para ver el funcionamiento de mejor forma, junto con los programas se adjuntan videos sobre el funcionamiento de este programa.

En caso de que los mismos den algún problema, alternativamente se adjuntan los siguiente enlaces:

 [FuncionamientoClienteServidor.mp4](#)

https://drive.google.com/file/d/1o_oCwEfViXK9mPNkuXXtob5bWd4GD-J6/view?usp=sharing

Ejercicio 2.
“CONCURRENCIA”

Planteamiento.

Sincronización de procesos. Semáforos. Diseñar un programa de concurrencia mediante el entorno JBACI.

<https://code.google.com/archive/p/jbaci/downloads>

Implementar mediante semáforos el problema del productor consumidor con buffer limitado estudiado en clase de teoría.

Notas

Para la calificación del ejercicio, además de la corrección del programa se valorarán los siguientes aspectos:

- Uso correcto de los semáforos atendiendo a los dos principales aspectos de exclusión mutua y sincronización.
- Uso del interfaz gráfico para la claridad del problema.

Implementación.

JBACI.

JBACI es un intérprete concurrente basado en Java, desarrollado por Michael Ben-Ari, que utiliza un subconjunto restringido del lenguaje C++, conocido como C--. Este intérprete genera código objeto interpretable (PCODE) y es utilizado principalmente para la enseñanza y experimentación de conceptos de programación concurrente, como la sección crítica, concurrencia y sincronización.

JBACI permite la implementación de problemas de sincronización utilizando semáforos, incluyendo problemas clásicos de concurrencia como el productor-consumidor y el problema de los lectores-escritores con prioridad. Estos problemas son fundamentales para entender cómo gestionar la concurrencia y la sincronización en sistemas multiproceso. Estos problemas mencionados anteriormente son:

- **Problema del Productor-Consumidor:** Este problema se resuelve utilizando semáforos para sincronizar la producción y el consumo de datos en un búfer compartido. JBACI permite implementar este problema de manera sencilla y visualizar su ejecución paso a paso.
- **Problema de los Lectores-Escritores:** Este problema se aborda mediante la utilización de semáforos para gestionar el acceso a un recurso compartido, donde los lectores pueden acceder simultáneamente mientras que los escritores necesitan exclusividad. JBACI facilita la implementación de diferentes variantes de este problema, incluyendo la prioridad de los escritores o los lectores.

En resumen, JBACI es una herramienta educativa valiosa que permite a los estudiantes experimentar y entender conceptos complejos de programación concurrente a través de la implementación de problemas clásicos de sincronización utilizando semáforos.

Constantes definidas:

const int sizeB = 5: Esta constante define el tamaño del buffer. En este caso, el buffer puede contener hasta 5 elementos.

const int figura = 1: Esta constante se utiliza para identificar el tipo de figura que se dibujará en la interfaz gráfica. En este caso, se utiliza para crear objetos con una figura específica.

const int color = 2: Esta constante se utiliza para definir el color inicial de los objetos en la interfaz gráfica. En este caso, se utiliza para crear objetos con un color específico.

int objeto = 0: Esta variable se utiliza como índice para los objetos que se crean y se dibujan en la interfaz gráfica. También se utiliza para realizar un seguimiento del objeto que se está produciendo o consumiendo.

Semáforos definidos:

semaphore mutex: Es un semáforo utilizado para garantizar la exclusión mutua. Asegura que solo un proceso (productor o consumidor) pueda acceder a la sección crítica (el buffer) a la vez.

semaphore empty: Este semáforo representa el número de espacios vacíos en el buffer. Inicialmente, se establece en el tamaño del buffer (sizeB) porque el buffer está vacío.

semaphore full: Este semáforo representa el número de elementos llenos en el buffer. Inicialmente, se establece en 0 porque el buffer está vacío.

Función encargada de la interfaz gráfica.

La función dibujar() crea y visualiza objetos en una interfaz gráfica. Inicializa las coordenadas pos1 y pos2, y luego utiliza un ciclo for para crear y hacer visibles sizeB objetos (donde sizeB es el tamaño del buffer), incrementando pos1 en cada iteración para espaciar los objetos horizontalmente. Después del ciclo, crea dos objetos adicionales con identificadores 100 y 300, y los posiciona en coordenadas específicas, haciendo que sean visibles.

Funciones del productor:

void producir(): Es la función encargada de la sección crítica para el productor. En este momento no realiza ninguna función específica ya que no se especifica una.

void mensajePro(): Imprime un mensaje en la consola indicando que el productor ha producido un elemento, cambia el color del objeto producido para reflejar en la interfaz gráfica, e incrementa el índice del objeto para preparar la producción del siguiente elemento.

void productor(): Implementa el comportamiento del productor en el problema clásico del productor-consumidor. Utiliza un ciclo infinito para producir elementos continuamente. Primero, espera a que haya espacio disponible en el buffer (esperando en el semáforo empty). Luego, adquiere el turno de exclusión mutua (esperando en el semáforo mutex). Una vez dentro de la sección crítica, produce un nuevo elemento y actualiza el estado del buffer. Después de producir el elemento, se incrementa el contador de elementos llenos (señalando el semáforo full) para desbloquear al consumidor, permitiéndole consumir el elemento producido.

Funciones del consumidor:

void consumir(): Es la función encargada de la sección crítica para el consumidor. En este momento no realiza ninguna función específica ya que no se especifica una.

void mensajeCon(): Imprime un mensaje en la consola indicando que el consumidor ha consumido un elemento, decrementa el índice del objeto para reflejar el consumo, y cambia el color del objeto consumido a 1 para actualizar la interfaz gráfica y mostrar visualmente que el elemento ha sido consumido.

void consumidor(): Implementa el comportamiento del consumidor en el problema clásico del productor-consumidor. Utiliza un ciclo infinito para consumir elementos continuamente. Primero, espera a que haya al menos un elemento disponible en el buffer (esperando en el semáforo full). Luego, adquiere el turno de exclusión mutua (esperando en el semáforo mutex). Una vez dentro de la sección crítica, consume un elemento y actualiza el estado del buffer. Después de consumir el elemento, decrementa el contador de elementos llenos (señalando el semáforo empty) para desbloquear al productor, permitiéndole producir un nuevo elemento.

Función principal:

Inicializa los semáforos necesarios para la sincronización entre el productor y el consumidor: empty se inicializa con el tamaño del buffer (permitiendo al productor producir

hasta que el buffer esté lleno), mutex se inicializa en 1 (permitiendo solo un proceso a la vez en la sección crítica), y full se inicializa en 0 (haciendo que el consumidor espere hasta que haya elementos disponibles). Luego, llama a la función dibujar() para configurar la interfaz gráfica. Finalmente, utiliza la construcción cobegin para iniciar tanto el productor como el consumidor, permitiendo que ambos procesos se ejecuten concurrentemente.

Capturas de pantalla del programa:

```
1 //-----DECLARACIÓN DE CONSTANTES
2 semaphore mutex, empty, full=0;
3
4 const int sizeB = 5;
5 const int figura = 1;
6 const int color = 2;
7 int objeto = 0;
8
9 //-----GRÁFICAS
10 void dibujar(){
11     int pos1 = 60, pos2 = 200;
12
13     for(objeto; objeto < sizeB; objeto++){
14         create(objeto, figura, color, pos1, pos2, 45, 45);
15         makevisible(objeto, 1);
16         pos1 = pos1 + 100;
17     }
18     objeto = 0;
19     create(100, 3, 5, pos1-40, pos2-100, 45, 45);
20     makevisible(100, 1);
21     create(300, 3, 5, pos1-40, pos2+100, 45, 45);
22     makevisible(300, 1);
23 }
24
25 //-----PRODUCTOR
26 void producir(){
27     // SECCIÓN CRÍTICA
28 }
29
30 void mensajePro(){
31     cout<<"Productor produce."<<endl;
32     changecolor(objeto, 5);
33     objeto++;
34 }
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
```

```


void productor(){
53
54     for(;;){
55         changecolor(100, 1); //Esperamos a que se agregue al buffer
56
57         wait(empty);
58
59         changecolor(100, 5);
60
61         wait(mutex);
62
63         producir();
64
65         mensajePro();
66
67         signal(mutex); //Esperamos a que se decremente el buffer
68
69         signal(full); //Desbloqueamos al consumidor
70     }
71 }
72
73 //-----CONSUMIDOR
74 void consumidor(){
75     // SECCIÓN CRÍTICA
76 }
77
78 void mensajeCon(){
79     cout<<"Consumidor consume"<<endl;
80
81     objeto--;
82
83     changecolor(objeto, 1); //Imprimimos el objeto en la interfaz gráfica como consumido
84 }
85
86 void consumidor(){
87     for(;;){
88         changecolor(300, 1);
89
90         wait(full); //Esperamos hasta que el productor acabe
91
92         changecolor(300, 5);
93
94         wait(mutex); //Exclusion mutua
95
96         consumir();
97
98         mensajeCon();
99
100        signal(mutex);
101
102        signal(empty);
103    }
104 }
105
106 //-----[MAIN]-----
107 void main(){
108
109     initialise(empty, sizeB); // El productor se bloquea hasta que el consumidor lo desbloquee
110     initialise(mutex, 1); // Solo un proceso a la vez
111     initialise(full, 0); // El consumidor se bloquea hasta que el productor le desbloquee
112     dibujar();
113
114     cobegin
115     {
116         productor();
117         consumidor();
118     }
119 }
120
121

```


NOTA:

Para ver el funcionamiento de mejor forma, junto con los programas se adjuntan videos sobre el funcionamiento de este programa.

En caso de que los mismos den algún problema, alternativamente se adjuntan los siguiente enlaces:

 Interfaz gráfica.mp4 :

https://drive.google.com/file/d/1KzPotxvP33qLYRyqv4W1H_vfwWEo8gv5/view?usp=sharing

 Funcionamiento del código.mp4

<https://drive.google.com/file/d/18EFy93dCpbCwA6arUrj1Oceabc6TwdYo/view?usp=sharing>