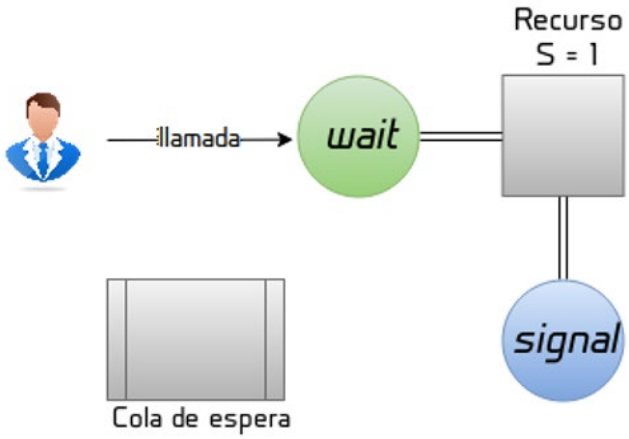
Semáforos

Los semáforos son un mecanismo de sincronización ampliamente utilizado en la programación concurrente y en sistemas operativos para gestionar el acceso de múltiples hilos o procesos a recursos compartidos. Fueron introducidos por el científico informático Edsger Dijkstra y se utilizan para evitar condiciones de carrera (race conditions) y asegurar que los recursos compartidos se utilicen de manera controlada.

Un semáforo es una variable o estructura que se utiliza para controlar el acceso de múltiples hilos o procesos a un recurso o sección crítica. Puede tomar distintos valores que indican si un recurso está disponible o no.



Existen dos tipos principales de semáforos:

- **Semáforo binario:** Toma solo dos valores, 0 y 1. Funciona como un interruptor para permitir o bloquear el acceso a un recurso (similar a un candado).
- **Semáforo contador:** Permite contar cuántos recursos están disponibles, permitiendo más de una entidad a la vez (p. ej., si tienes múltiples instancias de un recurso).

Los semáforos sirven para:

- **Controlar el acceso a recursos compartidos:** Cuando varios procesos o hilos intentan acceder a un recurso compartido (como una base de datos o una variable), los semáforos aseguran que no ocurra un acceso simultáneo no controlado.
- **Evitar condiciones de carrera:** Los semáforos evitan que dos procesos accedan a la vez a un recurso que pueda alterar su estado de forma inconsistente.
- **Coordinar procesos:** Pueden usarse para coordinar la ejecución de múltiples hilos o procesos, asegurando que se sigan ciertos pasos en un orden específico.

Ciclo de vida del semáforo

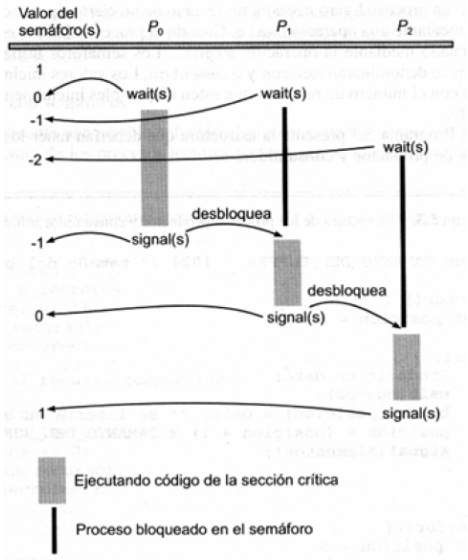
- **Inicialización:** Se crea un semáforo con un valor inicial, que puede ser 1 (en el caso de un semáforo binario) o cualquier número mayor en el caso de un semáforo contador.
- **Espera (Wait/P):** Cuando un hilo o proceso necesita acceder a un recurso protegido, realiza una operación de "wait" o "P" (operación de decremento). Si el valor del semáforo es mayor que 0, el hilo puede continuar y decrementar el valor del semáforo. Si es 0, el hilo se bloquea hasta que otro proceso libere el recurso.
- **Señalización (Signal/V):** Cuando un proceso termina de usar un recurso, realiza una operación de "signal" o "V" (operación de incremento), incrementando el valor del semáforo y despertando a cualquier proceso que esté esperando.
- **Destrucción:** Cuando el semáforo ya no es necesario, se libera la memoria o recursos asociados al mismo.

La sección crítica es una parte del código que accede a recursos compartidos (como un búfer, memoria compartida, archivos, etc.) y, por lo tanto, no debe ser ejecutada por más de un hilo o proceso a la vez. Es decir, es un bloque de código donde ocurre acceso y modificación a datos que son compartidos entre múltiples hilos o procesos.

Si más de un hilo o proceso ingresa en la sección crítica al mismo tiempo, podrían ocurrir condiciones de carrera o comportamientos inesperados. Para evitar esto, se usan mecanismos de sincronización como los semáforos o los mutex, que aseguran que solo un hilo o proceso acceda a la sección crítica en un momento dado.

Características de la sección crítica:

- **Acceso exclusivo:** Solo un hilo/proceso puede ejecutar el código de la sección crítica a la vez.
- **Protección:** Se debe proteger el acceso a la sección crítica usando mecanismos de sincronización como semáforos.
- **Eficiencia:** Se debe minimizar el tiempo dentro de la sección crítica para evitar bloqueos innecesarios.



Desarrollo:

Para la presente práctica se busca aplicar y demostrar el funcionamiento de los semáforos, para ello, se va a realizar un balanceador de carga, que va a permitir distribuir las solicitudes entre varios servidores, en este caso se realizará una simulación de un balanceador de carga, para ello se van a utilizar los semáforos para distribuir las solicitudes de conexión de distintos clientes.

Comenzaremos creando un archivo donde se utilizará lenguaje C, en mi caso lo nombre como balanceadorSemaforo.c:

```
#define PORT 9090
#define MAX_CLIENTS 10
#define NUM_SERVERS 3

sem_t semaphore;
```

Se definen las constantes clave del programa. PORT especifica el puerto en el que el servidor escuchará las conexiones entrantes. MAX_CLIENTS establece el número máximo de clientes que pueden estar en cola de espera mientras se procesan otras conexiones, y NUM_SERVERS indica el número máximo de servidores que procesarán solicitudes simultáneamente. El semáforo semaphore se declara aquí para controlar el acceso a los servidores.

```
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);
}
```

Esta función maneja las solicitudes de los clientes. Cuando se recibe una conexión, el socket del cliente se pasa como argumento. sem_wait se utiliza para controlar cuántos servidores pueden manejar solicitudes simultáneamente, garantizando que no se supere el número máximo definido por

NUM_SERVERS. Luego, el servidor recibe los datos enviados por el cliente usando recv, los procesa y envía una respuesta de confirmación con send. Finalmente, cierra la conexión y libera el semáforo con sem_post, lo que permite que otro servidor procese una nueva solicitud.

```
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);
```

En la función main, se inicializan las variables del socket del servidor y el cliente. Se configura la estructura sockaddr_in para especificar la dirección IP y el puerto en el que el servidor escuchará conexiones. El semáforo se inicializa con sem_init, indicando que se permitirá que NUM_SERVERS servidores manejen solicitudes simultáneamente.

```
    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");
```

Se configura la estructura server_addr con la dirección IP y el puerto, y se asocia el socket con esta dirección utilizando bind(). El servidor comienza a escuchar las conexiones entrantes con listen(), esperando hasta MAX_CLIENTS conexiones simultáneamente en la cola.

```
    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;
```

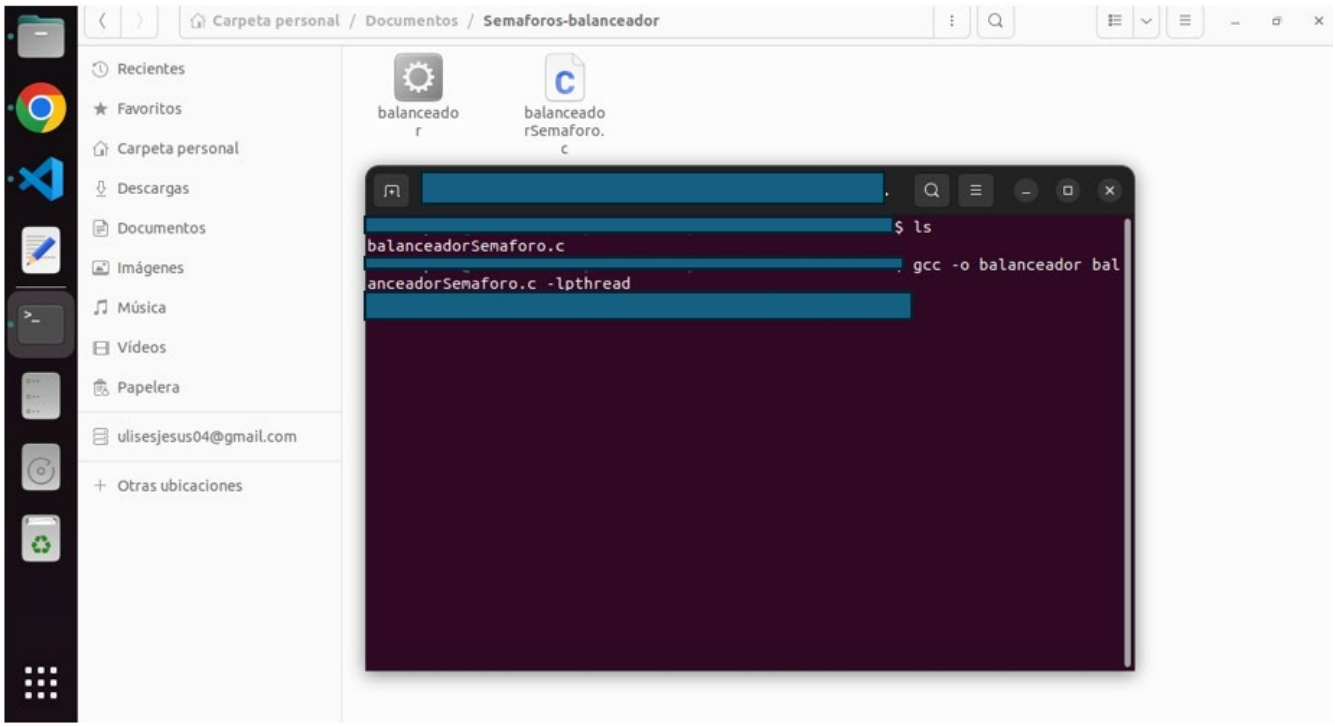
```
        if (pthread_create(&server_thread, NULL, handle_server, (void*)new_sock)
< 0) {
            perror("Error al crear el hilo");
            return 1;
        }

        pthread_detach(server_thread);
    }

    if (client_socket < 0) {
        perror("Error en accept");
        return 1;
    }
```

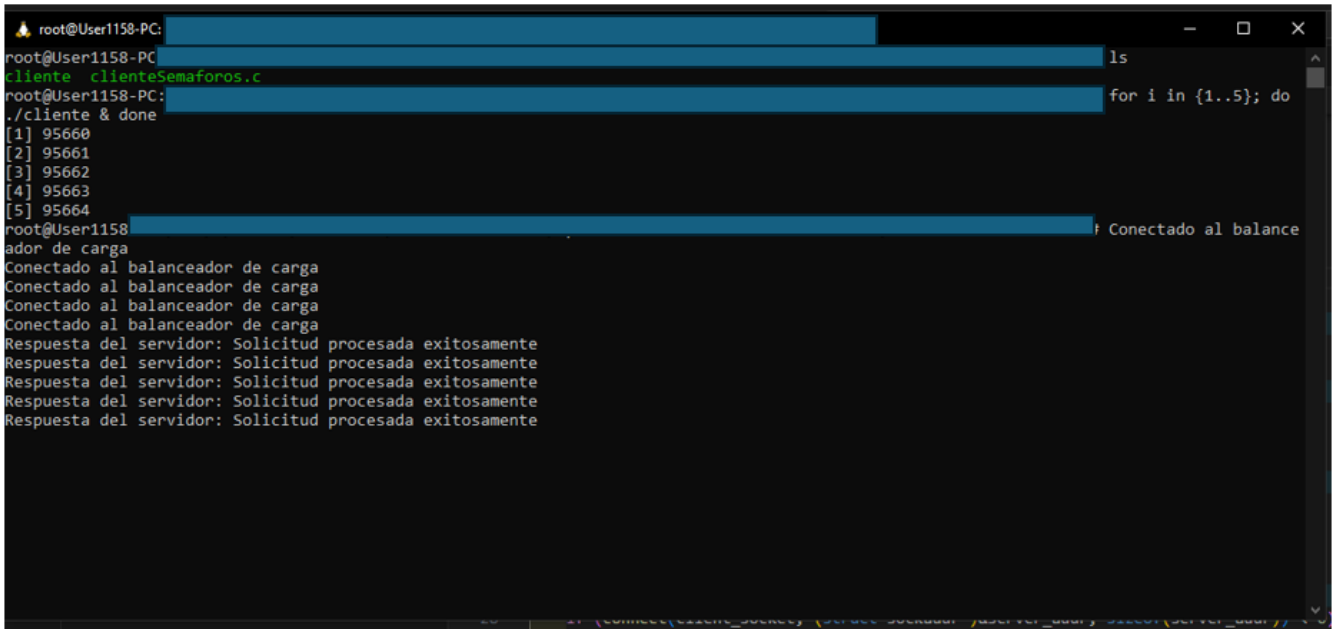
El servidor entra en un bucle que acepta conexiones entrantes con accept(). Por cada cliente que se conecta, se imprime un mensaje y se crea un nuevo hilo con pthread_create() para manejar la solicitud. Se pasa el socket del cliente al hilo, lo que permite procesar múltiples conexiones simultáneamente. Los hilos son "detached", lo que significa que no es necesario unirlos después de su finalización.

```
    sem_destroy(&semaphore);
    return 0;
}
```

Se destruye el semáforo con sem_destroy() para liberar los recursos del sistema asociados. Esto es importante para asegurar que no haya fugas de memoria o recursos cuando el balanceador de carga deje de funcionar.

Ahora se procede a compilar y ejecutar el cliente y servidor desarrollados:





```
root@User1158-PC:                                                    gcc -o cliente cli
enteSemaforos.c
root@User1158-PC:                                                    for i in {1..5}; d
o ./cliente & done
[1] 95060
[2] 95061
[3] 95062
[4] 95063
[5] 95064
root@User1158-P                                                      Conectado al balan
ceador de carga
Conectado al balanceador de carga
Conectado al balanceador de carga
Conectado al balanceador de carga
Conectado al balanceador de carga
Respuesta del servidor: Solicitud procesada exitosamente
Respuesta del servidor: Solicitud procesada exitosamente
Respuesta del servidor: Solicitud procesada exitosamente
Respuesta del servidor: Solicitud procesada exitosamente
Respuesta del servidor: Solicitud procesada exitosamente
root@User1158-PC:
```

Del lado del servidor podemos observar como muestra mensajes de distribución y conexión exitosa: