



Universitat d'Alacant
Universidad de Alicante

GRADO EN INGENIERÍA INFORMÁTICA
CURSO ACADÉMICO 2025-2026

Sistemas Operativos

GRUPO 8 - Práctica 2



JULIÁN HINOJOSA GIL - DNI 48795869-N

Índice

1. Introducción	1
2. Objetivos	1
3. Desarrollo	1
3.1. Ejercicio 1: Comunicaciones en Red	1
3.1.1. Descripción del problema	1
3.1.2. Arquitectura del servidor	1
3.1.3. Arquitectura del cliente	2
3.2. Ejercicio 2: Problema del Productor-Consumidor	2
3.2.1. Descripción del problema	2
3.2.2. Mecanismos de sincronización	2
4. Código Fuente	2
4.1. Ejercicio 1: Comunicaciones en Red	2
4.1.1. Implementación del servidor	2
4.1.2. Implementación del cliente	4
4.2. Ejercicio 2: Problema del Productor-Consumidor	5
4.2.1. Declaraciones globales	5
4.2.2. Funciones auxiliares	5
4.2.3. Procesos concurrentes	6
4.2.4. Función principal	6
5. Conclusiones	7
6. Bibliografía	7

1. Introducción

Este documento presenta el desarrollo y los resultados de la práctica de Sistemas Operativos sobre comunicaciones en red y concurrencia. La práctica consta de dos ejercicios fundamentales que abordan conceptos clave en el desarrollo de sistemas distribuidos y la sincronización de procesos.

El primer ejercicio se centra en las comunicaciones en red mediante sockets, implementando una aplicación cliente-servidor que permite la transmisión de archivos. El servidor se ejecutará en el puerto 9999, esperando conexiones entrantes, mientras que el cliente se conectará para recibir el contenido de un archivo HTML (en este caso, una versión de la página de inicio de Google) y posteriormente lo abrirá en el navegador local.

El segundo ejercicio aborda el problema clásico de concurrencia del productor-consumidor con buffer limitado, implementándolo mediante semáforos en el simulador jBACI. Este problema ilustra los desafíos de la sincronización entre procesos que comparten recursos, aplicando mecanismos de exclusión mutua y señalización para evitar condiciones de carrera y garantizar la correcta operación del sistema.

2. Objetivos

Los objetivos de esta práctica son:

- Comprender y aplicar los conceptos fundamentales de comunicación en red mediante sockets TCP/IP
- Implementar correctamente una arquitectura cliente-servidor capaz de transferir archivos a través de la red
- Utilizar las llamadas al sistema relacionadas con comunicaciones en red en el lenguaje C
- Entender y aplicar los mecanismos de sincronización de procesos mediante semáforos
- Resolver el problema clásico del productor-consumidor con buffer limitado
- Dominar el uso correcto de semáforos para garantizar la exclusión mutua y evitar condiciones de carrera
- Desarrollar habilidades en la depuración y análisis de sistemas concurrentes

3. Desarrollo

3.1. Ejercicio 1: Comunicaciones en Red

3.1.1. Descripción del problema

En este ejercicio se desarrolla una aplicación cliente-servidor para transferir un archivo desde el servidor al cliente utilizando sockets en C. Se utiliza una versión de la página de inicio de Google como archivo de prueba para demostrar la transferencia de contenido HTML.

3.1.2. Arquitectura del servidor

El servidor se encarga de escuchar las conexiones entrantes en el puerto 9999. Cuando llega una nueva conexión, se crea un proceso hijo que gestiona la transferencia del archivo al cliente conectado. Esta arquitectura permite que el servidor continúe aceptando nuevas conexiones mientras se realizan las transferencias.

El archivo se envía en bloques de tamaño definido por la macro `BUFFER_SIZE`, lo que optimiza el uso de memoria y permite transferir archivos de cualquier tamaño. El servidor mantiene un máximo de 5 conexiones en cola esperando ser atendidas.

3.1.3. Arquitectura del cliente

El cliente se encarga de conectarse al servidor especificado mediante su dirección IP (pasada como argumento). Una vez establecida la conexión, recibe el archivo en bloques y lo escribe en un archivo local llamado `Google_recibido.html`. Al completar la recepción, el cliente automáticamente abre el archivo en el navegador predeterminado del sistema.

3.2. Ejercicio 2: Problema del Productor-Consumidor

3.2.1. Descripción del problema

El ejercicio 2 implementa el problema clásico del productor-consumidor con buffer limitado utilizando semáforos en el simulador jBACI. Este problema ilustra la sincronización entre procesos concurrentes que comparten un recurso común (el buffer).

3.2.2. Mecanismos de sincronización

La solución utiliza tres semáforos:

- **productores**: Semáforo contador que representa los espacios disponibles en el buffer (inicializado a 4).
- **consumidores**: Semáforo contador que representa los ítems disponibles para consumir (inicializado a 0).
- **accesoBuffer**: Semáforo binario que garantiza exclusión mutua en el acceso al buffer compartido (inicializado a 1).

Esta implementación reproduce el ejemplo de la transparencia vista en clase, adaptándolo al entorno de jBACI con comentarios explicativos para facilitar su comprensión.

4. Código Fuente

A continuación se presenta el código implementado para ambos ejercicios, junto con las explicaciones de los fragmentos más relevantes.

4.1. Ejercicio 1: Comunicaciones en Red

4.1.1. Implementación del servidor

El servidor comienza creando y configurando el socket que utilizará para aceptar conexiones. Este proceso incluye la creación del socket, su configuración y el enlace a una dirección y puerto específicos:

```
1 // Creación del socket, se gestiona el error si no se puede crear
2 sockfd = socket(AF_INET, SOCK_STREAM, 0);
3 if (sockfd == -1) {
4     fprintf(stderr, "Error al crear el socket");
5     exit(EXIT_FAILURE);
6 }
7 printf("Socket creado con éxito\n");
8
9 // Configuración de la estructura sockaddr_in
10 serverAddr.sin_port = htons(DEFAULT_PORT); // puerto del servidor
11 serverAddr.sin_family = AF_INET; // familia de direcciones IPv4
```

```

12 serverAddr.sin_addr.s_addr = INADDR_ANY; // aceptar conexiones en cualquier interfaz de
   red
13
14 // Enlazar el socket a la dirección y puerto especificados, con gestión de errores
15 if (bind(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
16     fprintf(stderr, "Error al enlazar el socket");
17     exit(EXIT_FAILURE);
18 }

```

Listing 1: Configuración del servidor

La estructura sockaddr_in se configura estableciendo:

- **sin_port**: Convertido al formato de red mediante `htons`
- **sin_family**: Establecido en `AF_INET` para indicar el uso de IPv4
- **sin_addr.s_addr**: Configurado con `INADDR_ANY` para aceptar conexiones en cualquier interfaz de red

Una vez configurado, el servidor entra en un bucle infinito para aceptar y gestionar conexiones entrantes:

```

1 // Poner el socket en modo escucha para aceptar conexiones entrantes, máximo 5
   conexiones en cola
2 listen(sockfd, 5);
3 printf("Esperando nuevas conexiones...\n");
4
5 // Bucle infinito para aceptar y manejar conexiones entrantes
6 while(1) {
7     // Aceptar una nueva conexión de cliente
8     size = sizeof(clientAddr);
9     clientSocketfd = accept(sockfd, (struct sockaddr *)&clientAddr, &size);
10    if (clientSocketfd == -1) {
11        fprintf(stderr, "Error aceptando la conexión\n");
12        continue;
13    }
14    printf("Conexión aceptada!!!\n");
15    ....
16 }

```

Listing 2: El servidor acepta conexiones

Tras aceptar la conexión, el servidor crea un proceso hijo mediante `fork()` para gestionar la transferencia. Este proceso hijo se encarga de leer el archivo y enviarlo al cliente en bloques:

```

1 //Leer y enviar el archivo por trozos en base al tamaño del buffer
2 while((bytesRead = read(filefd, buffer, sizeof(buffer))) > 0) {
3     bytesSent = write(clientSocketfd, buffer, bytesRead);
4     if (bytesSent == -1) {
5         fprintf(stderr, "Error en la transferencia del archivo\n");
6         break;
7     }
8 }

```

Listing 3: Envío del archivo en bloques

Una vez completada la transferencia, el proceso hijo cierra el descriptor de archivo y el socket del cliente, finalizando su ejecución. El proceso padre del servidor continúa en el bucle, esperando nuevas conexiones.

4.1.2. Implementación del cliente

El cliente comienza creando su socket y configurándolo para conectarse al servidor. La dirección IP del servidor se proporciona como argumento en la línea de comandos:

```

1 // Crear el socket para la conexión al servidor
2 socketfd = socket(AF_INET, SOCK_STREAM, 0);
3 if (socketfd == -1) {
4     fprintf(stderr, "Error al crear el socket:\n");
5     exit(EXIT_FAILURE);
6 }
7 fprintf(stdout, "Socket creado con éxito\n");
8
9 // Configurar la estructura sockaddr_in del servidor
10 serverAddr.sin_port = htons(DEFAULT_PORT); // puerto del servidor
11 serverAddr.sin_family = AF_INET; // familia de direcciones IPv4
12 serverAddr.sin_addr.s_addr = inet_addr(argv[1]); // dirección IP del servidor

```

Listing 4: Configuración del socket del cliente

La configuración es similar a la del servidor, con la diferencia de que `sin_addr.s_addr` se establece mediante `inet_addr()`, que convierte la dirección IP en formato de texto a su representación numérica.

A continuación, el cliente establece la conexión con el servidor:

```

1 // Conectar al servidor, con gestión de errores
2 if (connect(socketfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
3     fprintf(stderr, "Error al conectar\n");
4     close(socketfd);
5     exit(EXIT_FAILURE);
6 }

```

Listing 5: Conexión al servidor

Con la conexión establecida, el cliente procede a recibir el contenido del archivo. Se crea un archivo local donde se almacenará el contenido recibido, y luego se lee y escribe en bloques:

```

1 // Crear el archivo donde se guardará el contenido recibido
2 filefd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
3 if (filefd == -1) {
4     fprintf(stderr, "Error al crear el archivo de destino\n");
5     return -1;
6 }
7
8 printf("Recibiendo archivo...\n");
9
10 // Recibir el contenido del archivo y guardararlo
11 while((bytesReceived = read(socketfd, buffer, buffer_size)) > 0) {
12     if (write(filefd, buffer, bytesReceived) == -1) {
13         fprintf(stderr, "Error al escribir en el archivo\n");
14         close(filefd);
15         return -1;
}

```

```

16 }
17 }
```

Listing 6: Recepción del archivo en bloques

Finalmente, una vez completada la recepción del archivo, el cliente automáticamente lo abre en el navegador predeterminado. Se utilizan directivas del preprocesador para ejecutar el comando apropiado según el sistema operativo:

```

1 #ifdef __APPLE__
2     system("open Google_recibido.html");
3 #elif __linux__
4     system("xdg-open Google_recibido.html");
5 #endif
```

Listing 7: Abrir el archivo en el navegador por defecto

4.2. Ejercicio 2: Problema del Productor-Consumidor

4.2.1. Declaraciones globales

La implementación comienza declarando los semáforos y variables globales que serán compartidas entre los procesos productor y consumidor:

```

1 semaphore productores, consumidores;
2 binarysem accesoBuffer;
3
4 const int bufferSize = 4;
5 int buffer = 0;
```

Listing 8: Definición de semáforos y variables globales

Los semáforos declarados cumplen las siguientes funciones:

- **productores**: Semáforo contador que controla los espacios disponibles en el buffer
- **consumidores**: Semáforo contador que controla los ítems disponibles para consumir
- **accesoBuffer**: Semáforo binario para exclusión mutua en el acceso al buffer

4.2.2. Funciones auxiliares

Se definen funciones auxiliares para simular las operaciones de producción y consumo, así como las operaciones sobre el buffer compartido:

```

1 void producir() {cout << "se produce" << endl;}
2 void consumir() {cout << "se consume" << endl;}
3
4 void anadir_buffer() {
5     cout << "se incrementa el buffer" << endl;
6     buffer++;
7 }
```

```

9 void coger_buffer() {
10    cout << "se saca el buffer" << endl;
11    buffer--;
12 }

```

Listing 9: Funciones para el productor y consumidor

Estas funciones se dividen en dos categorías:

- **Simulación:** `producir()` y `consumir()` simulan las operaciones externas de producción y consumo
- **Gestión del buffer:** `anadir_buffer()` y `coger_buffer()` manejan las operaciones sobre el buffer compartido, incrementando o decrementando el contador

4.2.3. Procesos concurrentes

Los procesos productor y consumidor utilizan los semáforos para sincronizar el acceso al buffer compartido y evitar condiciones de carrera:

```

1 void productor(){
2     while(1) {
3         producir();
4         wait(productores); // P(buffer)
5         wait(accesoBuffer); // P(accesoBuffer)
6         anadir_buffer();
7         signal(accesoBuffer); // V(accesoBuffer)
8         signal(consumidores); // V(consumer)
9     }
10 }
11
12 void consumidor() {
13     while(1) {
14         wait(consumidores); // P(consumer)
15         wait(accesoBuffer); // P(accesoBuffer)
16         coger_buffer();
17         signal(accesoBuffer); // V(accesoBuffer)
18         signal(productores); // V(buffer)
19         consumir();
20     }
21 }

```

Listing 10: Procesos de productor y consumidor

El patrón de sincronización implementado garantiza:

- Exclusión mutua en el acceso al buffer mediante `accesoBuffer`
- Prevención de desbordamiento del buffer mediante el semáforo `productores`
- Prevención de acceso a buffer vacío mediante el semáforo `consumidores`

4.2.4. Función principal

La función principal inicializa los semáforos con los valores apropiados y lanza los procesos concurrentes:

```
1 void main() {
2     initialsem(productores, bufferSize);
3     initialsem(accesoBuffer, 1);
4     initialsem(consumidores, 0);
5
6     cobegin{
7         productor();
8         consumidor();
9     }
10 }
```

Listing 11: Inicialización y creación de procesos

5. Conclusiones

Tras la realización de esta práctica, se pueden extraer las siguientes conclusiones:

- La programación con sockets TCP/IP permite implementar sistemas distribuidos robustos y eficientes. La correcta configuración de las estructuras `sockaddr_in` y el uso adecuado de las llamadas al sistema son fundamentales para establecer comunicaciones fiables entre procesos en diferentes máquinas.
- El modelo cliente-servidor implementado demuestra la importancia de una correcta gestión de errores y del uso de procesos hijo para atender múltiples conexiones simultáneas, permitiendo que el servidor continúe aceptando nuevas peticiones mientras se transfieren archivos.
- Los semáforos son herramientas esenciales para la sincronización de procesos concurrentes. En el problema del productor-consumidor, el uso correcto de semáforos contadores y binarios garantiza la exclusión mutua en el acceso al buffer compartido y evita situaciones de interbloqueo o inanición.
- La sincronización mediante semáforos requiere un diseño cuidadoso del orden de las operaciones `wait` y `signal` para evitar condiciones de carrera. El patrón implementado (esperar por recursos, adquirir el mutex, operar, liberar el mutex, señalizar disponibilidad) es fundamental en sistemas concurrentes.
- La práctica refuerza la comprensión de conceptos fundamentales de sistemas operativos como la comunicación entre procesos (IPC), la gestión de recursos compartidos y los mecanismos de sincronización, habilidades esenciales para el desarrollo de sistemas complejos y distribuidos.

6. Bibliografía

- Stallings, W. (2005). *Sistemas Operativos: Aspectos internos y principios de diseño* (5^a ed.). Madrid: Pearson Educación.
- Transparencias del Tema 3: Conurrencia - Sincronización y Comunicación de Procesos. Material docente de la asignatura de Sistemas Operativos.