



Universidad  
del Valle de México  
LAUREATE INTERNATIONAL UNIVERSITIES®

# Actividad 9

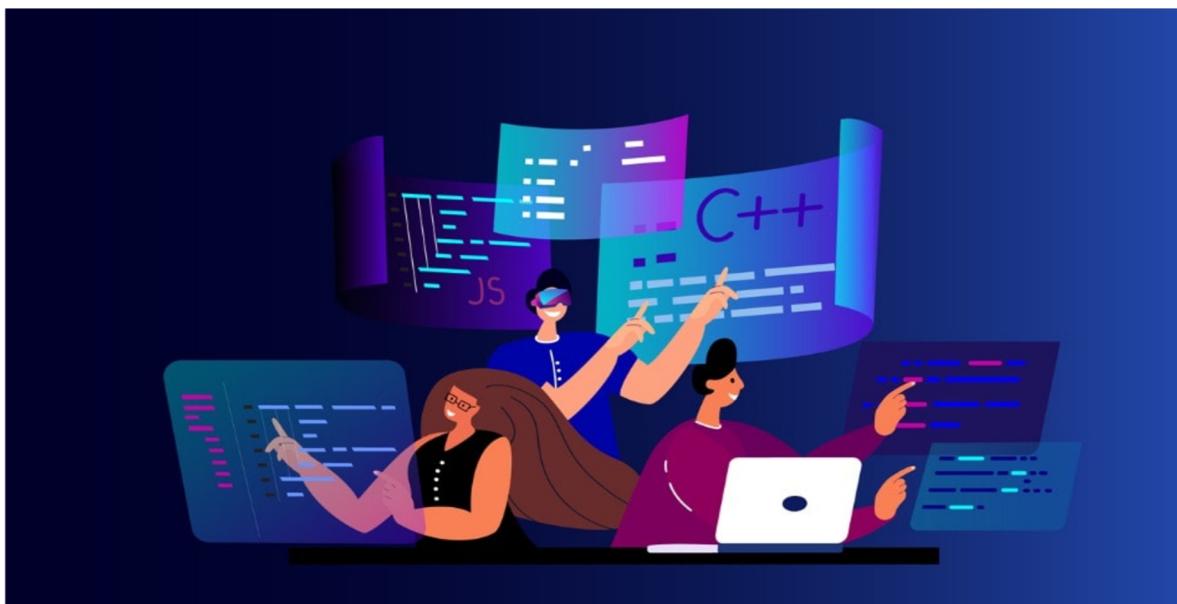
## Proyecto integrador etapa 3

### PROGRAMACIÓN CONCURRENTE

Docente. Cesar Garcia Martinez

José Emiliano Jauregui Guzmán

Por siempre responsable de lo que se ha cultivado



17 de febrero - 24 de febrero del 2025

## Introducción

El método de Jacobi es una técnica iterativa utilizada para resolver sistemas de ecuaciones lineales, especialmente aquellos de gran tamaño y complejidad. A través de aproximaciones sucesivas, este algoritmo permite obtener soluciones de las incógnitas de manera eficiente, utilizando un proceso que va refinando los resultados en cada iteración. En el contexto de la programación concurrente, se optimiza aún más la eficiencia mediante el uso de hilos y semáforos, lo que permite que las variables sean actualizadas simultáneamente, aprovechando los recursos de los sistemas modernos, como los procesadores multicore. Este enfoque mejora la rapidez del cálculo, sincronizando el acceso a los recursos compartidos y garantizando que el sistema funcione correctamente a través de mecanismos como los semáforos, los cuales controlan el acceso a secciones críticas del código para evitar conflictos y asegurar la coherencia de los datos. En este sentido, la programación concurrente no solo optimiza el rendimiento, sino que también contribuye al desarrollo de competencias técnicas clave, como la gestión de hilos, la exclusión mutua y la comunicación entre procesos, elementos esenciales para la implementación de soluciones eficientes en sistemas complejos. Esta actividad, por lo tanto, permite aplicar los conocimientos adquiridos en el curso, asegurando la transversalidad de los contenidos y fortaleciendo el desarrollo de competencias para alcanzar el objetivo de formación planteado.



## I. Identificación y análisis de la librería PVM

### 1.1 Identificación de las primitivas de PVM en C

- Identificación de la librería PVM como máquina virtual paralela

**PVM (Parallel Virtual Machine)** es una librería que permite la ejecución de aplicaciones paralelas en redes de computadoras distribuidas. Su objetivo principal es convertir una red heterogénea de máquinas en una "máquina virtual" que pueda ejecutar tareas en paralelo, facilitando la programación distribuida. A través de PVM, los desarrolladores pueden crear aplicaciones que aprovechen el procesamiento en paralelo, donde los nodos (máquinas) colaboran para completar una tarea común. Esto es posible gracias a un sistema de comunicación basado en paso de mensajes, que permite la sincronización y el intercambio de datos entre los diferentes procesos ejecutados en los nodos.

Las **variables de entorno** de PVM son esenciales para configurar y controlar su funcionamiento. Algunas de las más importantes incluyen **PVM\_ROOT** (que define el directorio de instalación de PVM), **PVM\_HOSTS** (que lista las máquinas disponibles para ejecutar las tareas), y **PVM\_ARCH** (que especifica la arquitectura de la máquina). Además, existen variables como **PVM\_LISTENER** para controlar las conexiones entre nodos, **PVM\_GAPTIME** para gestionar tiempos de espera entre operaciones, y **PVM\_DEBUG** para activar la depuración. Estas variables permiten personalizar la ejecución según las necesidades del programa y la infraestructura de hardware. La variable **PATH** define los directorios donde el sistema operativo busca los ejecutables de programas, y es crucial para que PVM pueda localizar sus binarios y ejecutables cuando se inicia un programa.

libpvm3.a.	Esta librería brinda una gran cantidad de rutinas escritas en lenguaje C. Esta rutina siempre es requerida.
libfpvm3.a.	librería adicional que se requiere en caso de que la aplicación escrita en PVM utilice código de Fortran.
libgpvm.a.	librería requerida en caso de usar grupos dinámicos.

Los **componentes clave** de PVM incluyen las tareas, que son los procesos que se distribuyen entre los nodos, los nodos mismos, que son las máquinas conectadas a la red distribuida y el sistema de comunicación que permite que los nodos se coordinen y comparten información. Además, PVM actúa como un controlador de entorno, gestionando las interacciones entre los nodos y las tareas a través de las variables de entorno mencionadas.



## Ventajas de PVM:

1. Escalabilidad: Permite agregar fácilmente más nodos a la red, lo que mejora la capacidad de procesamiento y permite manejar cargas de trabajo mayores.
2. Portabilidad: Funciona en diversas plataformas, lo que facilita su uso en redes heterogéneas de computadoras con diferentes sistemas operativos y arquitecturas.
3. Flexibilidad: La posibilidad de definir diferentes configuraciones a través de las variables de entorno hace que PVM se adapte a diversas necesidades y entornos de ejecución.
4. Optimización de recursos: Facilita la ejecución de tareas en paralelo, aprovechando los recursos distribuidos para mejorar el rendimiento de aplicaciones complejas.

## Desventajas de PVM:

1. Complejidad de configuración: Aunque PVM proporciona gran flexibilidad, configurar adecuadamente las variables de entorno y la infraestructura puede ser desafiante, especialmente en redes grandes o heterogéneas.
  2. Dependencia de la red: El rendimiento de las aplicaciones paralelas depende fuertemente de la calidad de la red. La latencia y los problemas de conexión pueden afectar la eficiencia.
  3. Curva de aprendizaje: El uso de PVM requiere conocimiento en programación paralela y en la gestión de redes distribuidas, lo que puede representar una barrera para desarrolladores sin experiencia en este tipo de entornos.
  4. Desactualización: Aunque PVM fue popular en su tiempo, ha sido reemplazado en gran medida por tecnologías más modernas, como MPI (Message Passing Interface), que ofrecen mejores características y una comunidad más activa.
- Identificación de las primitivas en C para lograr compilar y correr un programa en Java en un ambiente paralelo.

Para compilar y ejecutar un programa Java en un ambiente paralelo utilizando PVM desde un programa en C, se deben utilizar varias primitivas de PVM que permiten gestionar los procesos distribuidos y la comunicación entre nodos.

## Primitivas de PVM Utilizadas:

- `pvm_mytid()`: Obtiene el identificador único de tarea de la tarea actual.
- `pvm_spawn()`: Lanza procesos en paralelo en nodos disponibles. Permite ejecutar el programa Java (por ejemplo, `java MiPrograma`) en los nodos.
- `pvm_send()` y `pvm_recv()`: Permiten la comunicación entre las tareas. Se usan para enviar y recibir mensajes entre los procesos distribuidos.
- `pvm_initsend()` y `pvm_upkstr()`: Permiten empaquetar y desempaquetar datos para su envío entre nodos.
- `pvm_exit()`: Finaliza el entorno de PVM y detiene los procesos.
- `pvm_perror()`: Muestra errores si algo falla durante la ejecución.



## **Inicialización y Finalización de PVM**

- `pvm_init()`: Descripción: Inicializa el entorno PVM. Se debe llamar antes de cualquier otra función de PVM.
- `pvm_exit()`: Finaliza el entorno PVM y cierra la conexión con el sistema PVM. Una vez llamada esta función, no se pueden hacer más llamadas a PVM.
- `pvm_mytid()`: Devuelve el identificador de tarea (Task ID) del proceso actual. Este ID es único para cada proceso dentro de PVM y se usa para identificar las tareas en la red PVM.
- `pvm_version()`: Devuelve la versión de PVM que está en uso. Esto puede ser útil para verificar la compatibilidad.

## **Creación de Tareas (Procesos en Paralelo)**

- `pvm_spawn()`: Lanza un nuevo proceso en un nodo de la red. Este proceso puede ser un programa (como un archivo ejecutable o un programa Java). Se utiliza para crear tareas en paralelo.

## **Comunicación entre Tareas**

- `pvm_initsend()`: Inicializa un buffer para el envío de datos. Debe ser llamado antes de empaquetar y enviar datos.
- `pvm_pkbyte()`: Empaquetá un buffer de bytes para enviarlo.
- `pvm_pkstr()`: Empaquetá una cadena de caracteres (string) para enviarla.
- `pvm_pkint()`: Empaquetá un arreglo de enteros.
- `pvm_send()`: Envía el mensaje empaquetado al proceso receptor.
- `pvm_recv()`: Recibe un mensaje de otro proceso. Debe ser llamado después de que el mensaje se haya enviado.
- `pvm_upkbyte()`: Desempaquetá un buffer de bytes que ha sido recibido.
- `pvm_upkstr()`: Desempaquetá una cadena de caracteres (string) que ha sido recibida.
- `pvm_upkint()`: Desempaquetá un arreglo de enteros que ha sido recibido.

## **Sincronización y Control de Procesos**

- `pvm_barrier()` (sin implementación nativa en PVM): Aunque PVM no incluye una barrera de sincronización nativa, se pueden implementar barreras utilizando comunicación entre procesos, como enviando señales entre tareas.
- `pvm_hup()`: Notifica que un proceso ha terminado inesperadamente o ha sido detenido de manera anómala. Esta función es útil para manejar fallos de tareas.
- `pvm_taskid()`: Devuelve el Task ID de un proceso dado su ID en la red PVM.

## **Obtención de Información sobre Tareas**

- `pvm_tasks()`: Obtiene información sobre las tareas en ejecución dentro de la red PVM.
- `pvm_tidtohost()`: Obtiene el nombre del host donde se ejecuta una tarea dada su Task ID.
- `pvm_hostbyname()`: Obtiene el Task ID de un proceso dado su nombre de host.

## **Flujo Básico del Programa:**

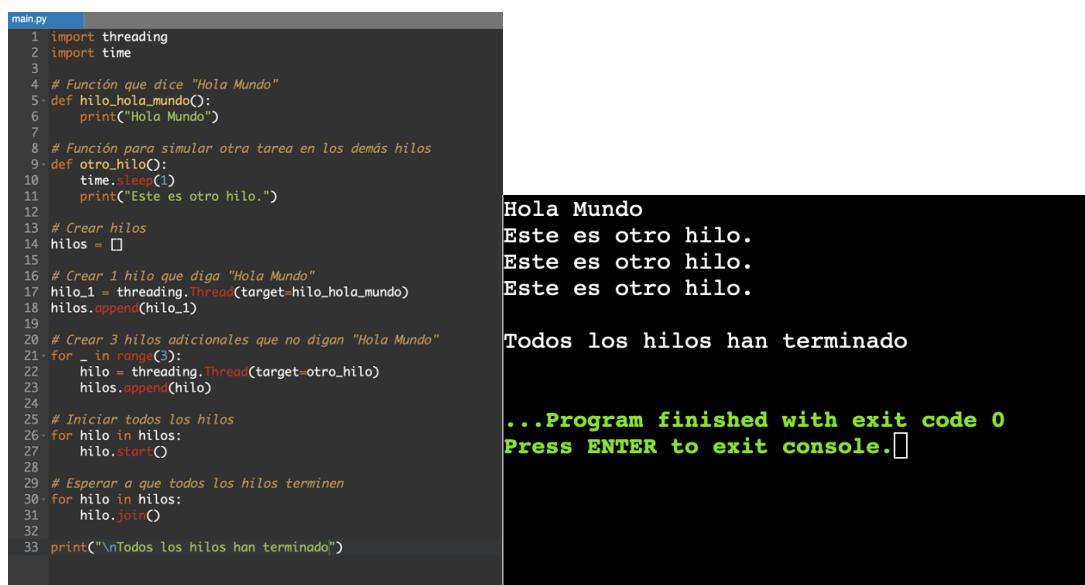
- Inicialización de PVM: Se inicializa el entorno de PVM con `pvm_init()`.



- Lanzamiento de procesos paralelos: Se crea el proceso Java con pvm\_spawn() en los nodos disponibles.
- Envío y recepción de mensajes: Se envían y reciben mensajes usando pvm\_send() y pvm\_recv().
- Finalización: Se finaliza con pvm\_exit().

### 1.2 Determinación de la funcionalidad de PVM en Java

Realizar un programa que verifique la comunicación entre el cliente y el servidor, por ejemplo, el más simple HolaMundo.c, para verificar la funcionalidad entre los nodos.



```

main.py
1 import threading
2 import time
3
4 # Función que dice "Hola Mundo"
5 def hilo_hola_mundo():
6     print("Hola Mundo")
7
8 # Función para simular otra tarea en los demás hilos
9 def otro_hilo():
10     time.sleep(1)
11     print("Este es otro hilo.")
12
13 # Crear hilos
14 hilos = []
15
16 # Crear 1 hilo que diga "Hola Mundo"
17 hilo_1 = threading.Thread(target=hilo_hola_mundo)
18 hilos.append(hilo_1)
19
20 # Crear 3 hilos adicionales que no digan "Hola Mundo"
21 for _ in range(3):
22     hilo = threading.Thread(target=otro_hilo)
23     hilos.append(hilo)
24
25 # Iniciar todos los hilos
26 for hilo in hilos:
27     hilo.start()
28
29 # Esperar a que todos los hilos terminen
30 for hilo in hilos:
31     hilo.join()
32
33 print("\nTodos los hilos han terminado")
  
```

## II. Identificar las funcionalidades de la programación concurrente

### 2.1 Elaborar un programa en C que sume los números dado un rango de números positivos dados mediante la línea de comandos:

- Primero. Realiza el programa usando un enfoque secuencial (sin paralelismo) observando su comportamiento
- Segundo. Realiza el programa en el lado del máster (master) con las primitivas para la comunicación en PVM y además tener como captura la línea de órdenes.
- Por último, se debe realizar el programa en el lado del esclavo (slave) con las primitivas propias de PVM, recibiendo los parámetros.
- Opcionalmente se puede usar el visor de PVM, “xpvm” para verificar el rendimiento (se sugiere usar números grandes de rango).
- Documentar las corridas, pruebas y funcionalidades del programa antes mencionado.



## Programa de suma de un rango de 1 al 10000 en secuencial.

```

main.c
1 #include <stdio.h>
2
3 // Función para sumar los números en un rango secuencial
4 long long suma_rango(int inicio, int fin) {
5     long long suma = 0;
6     // Mostrar cómo se va sumando cada número
7     for (int i = inicio; i <= fin; i++) {
8         suma += i;
9         printf("Sumando %d: Total acumulado = %lld\n", i, suma);
10    // Mostrar cada número sumado
11 }
12 return suma;
13}
14
15 int main() {
16     // Rango fijo de 1 a 10000
17     int rango_inicial = 1;
18     int rango_final = 10000;
19
20     // Calcular la suma usando la función secuencial
21     long long resultado = suma_rango(rango_inicial, rango_final);
22
23     // Imprimir el resultado final
24     printf("\nLa suma total de los números del %d al %d es: %lld\n", rango_inicial,
25           rango_final, resultado);
26
27     return 0; // Terminación exitosa
28 }
29

```

Sumando 9961: Total acumulado = 49615741  
 Sumando 9962: Total acumulado = 49625703  
 Sumando 9963: Total acumulado = 49635666  
 Sumando 9964: Total acumulado = 49645630  
 Sumando 9965: Total acumulado = 49655595  
 Sumando 9966: Total acumulado = 49665561  
 Sumando 9967: Total acumulado = 49675528  
 Sumando 9968: Total acumulado = 49685496  
 Sumando 9969: Total acumulado = 49695465  
 Sumando 9970: Total acumulado = 49705435  
 Sumando 9971: Total acumulado = 49715406  
 Sumando 9972: Total acumulado = 49725378  
 Sumando 9973: Total acumulado = 49735351  
 Sumando 9974: Total acumulado = 49745325  
 Sumando 9975: Total acumulado = 49755300  
 Sumando 9976: Total acumulado = 49765276  
 Sumando 9977: Total acumulado = 49775253  
 Sumando 9978: Total acumulado = 49785231  
 Sumando 9979: Total acumulado = 49795210  
 Sumando 9980: Total acumulado = 49805190  
 Sumando 9981: Total acumulado = 49815171  
 Sumando 9982: Total acumulado = 49825153  
 Sumando 9983: Total acumulado = 49835136  
 Sumando 9984: Total acumulado = 49845120  
 Sumando 9985: Total acumulado = 49855105  
 Sumando 9986: Total acumulado = 49865091  
 Sumando 9987: Total acumulado = 49875078  
 Sumando 9988: Total acumulado = 49885066  
 Sumando 9989: Total acumulado = 49895055  
 Sumando 9990: Total acumulado = 49905045  
 Sumando 9991: Total acumulado = 49915036  
 Sumando 9992: Total acumulado = 49925028  
 Sumando 9993: Total acumulado = 49935021  
 Sumando 9994: Total acumulado = 49945015  
 Sumando 9995: Total acumulado = 49955010  
 Sumando 9996: Total acumulado = 49965006  
 Sumando 9997: Total acumulado = 49975003  
 Sumando 9998: Total acumulado = 49985001  
 Sumando 9999: Total acumulado = 49995000  
 Sumando 10000: Total acumulado = 50005000

La suma total de los números del 1 al 10000 es: 50005000

...Program finished with exit code 0  
Press ENTER to exit console.

```
#include <stdio.h>
```

```

// Función para sumar los números en un rango secuencial
long long suma_rango(int inicio, int fin) {
    long long suma = 0;
    // Mostrar cómo se va sumando cada número
    for (int i = inicio; i <= fin; i++) {
        suma += i;
        printf("Sumando %d: Total acumulado = %lld\n", i, suma);
        // Mostrar cada número sumado
    }
    return suma;
}

int main() {
    // Rango fijo de 1 a 10000
    int rango_inicial = 1;
    int rango_final = 10000;

    // Calcular la suma usando la función secuencial
    long long resultado = suma_rango(rango_inicial, rango_final);

    // Imprimir el resultado final
    printf("\nLa suma total de los números del %d al %d es: %lld\n", rango_inicial,
          rango_final, resultado);

```



```

    return 0; // Terminación exitosa
}

```

## Simulador de un programa Master/Slave en c

```

main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #define BUFFER_SIZE 256
6
7 // Función para enviar un mensaje desde el proceso actual al otro proceso
8 void enviar_mensaje(int fd, const char *mensaje) {
9     write(fd, mensaje, strlen(mensaje) + 1); // Enviar mensaje (incluyendo el '\0')
10 }
11
12 // Función para recibir un mensaje desde el otro proceso
13 void recibir_mensaje(int fd, char *buffer) {
14     read(fd, buffer, BUFFER_SIZE); // Leer mensaje recibido
15 }
16
17 // Master envía un mensaje al Slave y espera la respuesta
18 void master(int fd_slave_write, int fd_master_read) {
19     char mensaje[BUFFER_SIZE] = "¡Hola, Slave! ¿Cómo estás?";
20     char respuesta[BUFFER_SIZE];
21
22     // Enviar mensaje al Slave
23     printf("[Master] Envio mensaje al Slave: %s\n", mensaje);
24     enviar_mensaje(fd_slave_write, mensaje);
25
26     // Esperar respuesta del Slave
27     recibir_mensaje(fd_master_read, respuesta);
28     printf("[Master] Recibido del Slave: %s\n", respuesta);
29 }
30
31 // Slave recibe un mensaje del Master y responde
32 void slave(int fd_master_write, int fd_slave_read) {
33     char mensaje[BUFFER_SIZE];
34     char respuesta[BUFFER_SIZE] = "¡Hola, Master! Estoy bien, ¿y tú?";
35
36     // Recibir mensaje del Master
37     recibir_mensaje(fd_slave_read, mensaje);
38     printf("[Slave] Recibido del Master: %s\n", mensaje);
39
40     // Enviar respuesta al Master
41     printf("[Slave] Envio mensaje al Master: %s\n", respuesta);
42     enviar_mensaje(fd_master_write, respuesta);
43 }
44
45 int main() {
46     int fd_master_to_slave[2]; // Pipe para la comunicación Master / Slave
47     int fd_slave_to_master[2]; // Pipe para la comunicación Slave / Master
48
49     // Crear los pipes
50     if (pipe(fd_master_to_slave) == -1 || pipe(fd_slave_to_master) == -1) {
51         perror("Error al crear pipes");
52         exit(1);
53     }
54
55     // Crear un proceso hijo (Slave)
56     pid_t pid = fork();
57
58     if (pid == -1) {
59         perror("Error al hacer fork");
60         exit(1);
61     }
62
63     if (pid > 0) { // Proceso padre (Master)
64         // Cerrar los extremos no necesarios del pipe
65         close(fd_master_to_slave[0]);
66         close(fd_slave_to_master[1]);
67
68         // Ejecutar la función de Master
69         master(fd_master_to_slave[1], fd_slave_to_master[0]);
70     }
}

```



```

70   // Esperar a que el proceso hijo termine
71   wait(NULL);
72
73 } else { // Proceso hijo (Slave)
74   // Cerrar los extremos no necesarios del pipe
75   close(fd_master_to_slave[1]);
76   close(fd_slave_to_master[0]);
77
78   // Ejecutar la función de Slave
79   slave(fd_slave_to_master[1], fd_master_to_slave[0]);
80
81   exit(0);
82 }
83
84 // Cerrar pipes al finalizar
85 close(fd_master_to_slave[0]);
86 close(fd_master_to_slave[1]);
87 close(fd_slave_to_master[0]);
88 close(fd_slave_to_master[1]);
89
90 return 0;
91 }
92 }

[Master] Enviando mensaje al Slave: ¡Hola, Slave! ¿Cómo estás?
[Slave] Recibido del Master: ¡Hola, Slave! ¿Cómo estás?
[Slave] Enviando mensaje al Master: ¡Hola, Master! Estoy bien, ¿y tú?
[Master] Recibido del Slave: ¡Hola, Master! Estoy bien, ¿y tú?

...Program finished with exit code 0
Press ENTER to exit console.

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFFER_SIZE 256

// Función para enviar un mensaje desde el proceso actual al otro proceso
void enviar_mensaje(int fd, const char *mensaje) {
    write(fd, mensaje, strlen(mensaje) + 1); // Enviar mensaje (incluyendo el
                                                // terminador '\0')
}

// Función para recibir un mensaje desde el otro proceso
void recibir_mensaje(int fd, char *buffer) {
    read(fd, buffer, BUFFER_SIZE); // Leer mensaje recibido
}

// Master envía un mensaje al Slave y espera la respuesta
void master(int fd_slave_write, int fd_master_read) {
    char mensaje[BUFFER_SIZE] = "¡Hola, Slave! ¿Cómo estás?";
    char respuesta[BUFFER_SIZE];

    // Enviar mensaje al Slave
    printf("[Master] Enviando mensaje al Slave: %s\n", mensaje);
    enviar_mensaje(fd_slave_write, mensaje);
}

```



```

// Esperar respuesta del Slave
recibir_mensaje(fd_master_read, respuesta);
printf("[Master] Recibido del Slave: %s\n", respuesta);
}

// Slave recibe un mensaje del Master y responde
void slave(int fd_master_write, int fd_slave_read) {
    char mensaje[BUFFER_SIZE];
    char respuesta[BUFFER_SIZE] = "¡Hola, Master! Estoy bien, ¿y tú?";

    // Recibir mensaje del Master
    recibir_mensaje(fd_slave_read, mensaje);
    printf("[Slave] Recibido del Master: %s\n", mensaje);

    // Enviar respuesta al Master
    printf("[Slave] Enviando mensaje al Master: %s\n", respuesta);
    enviar_mensaje(fd_master_write, respuesta);
}

int main() {
    int fd_master_to_slave[2]; // Pipe para la comunicación Master / Slave
    int fd_slave_to_master[2]; // Pipe para la comunicación Slave / Master

    // Crear los pipes
    if (pipe(fd_master_to_slave) == -1 || pipe(fd_slave_to_master) == -1) {
        perror("Error al crear pipes");
        exit(1);
    }

    // Crear un proceso hijo (Slave)
    pid_t pid = fork();

    if (pid == -1) {
        perror("Error al hacer fork");
        exit(1);
    }

    if (pid > 0) { // Proceso padre (Master)
        // Cerrar los extremos no necesarios del pipe
        close(fd_master_to_slave[0]);
        close(fd_slave_to_master[1]);

        // Ejecutar la función de Master
        master(fd_master_to_slave[1], fd_slave_to_master[0]);
    }

    // Esperar a que el proceso hijo termine
    wait(NULL);
}

```



```

} else { // Proceso hijo (Slave)
    // Cerrar los extremos no necesarios del pipe
    close(fd_master_to_slave[1]);
    close(fd_slave_to_master[0]);

    // Ejecutar la función de Slave
    slave(fd_slave_to_master[1], fd_master_to_slave[0]);

    exit(0);
}

// Cerrar pipes al finalizar
close(fd_master_to_slave[0]);
close(fd_master_to_slave[1]);
close(fd_slave_to_master[0]);
close(fd_slave_to_master[1]);

return 0;
}

```

### **III. identificar los conceptos y problemas en programación secuencial de sección crítica y exclusión mutua.**

#### *3.1 Identificar la exclusión mutua y la sección crítica*

- Identificación de una sección crítica como la memoria, un espacio de memoria, disco duro, entre otros ejemplos.

Una sección crítica puede ser cualquier sección de código en la que se accede a recursos compartidos, por lo general, consta de dos partes: la sección de entrada y la sección de salida. Estas se pueden identificar en las siguientes:

Memoria: Si dos o más procesos modifican al mismo tiempo la misma dirección de memoria, se pueden generar inconsistencias por lo que se vuelve una sección critica.

Espacio de memoria: Al ser un sistema multiproceso es normal que varios procesos estén compartiendo el espacio de memoria compartido para intercambiar información. Por lo que encontramos una sección critica que ya sin una implementación de seguridad podría dar una condición de carrera o alguna inconsistencia.

Disco duro: Este es otro ejemplo de un sistema de almacenamiento en el que hay recursos compartidos. Si varios procesos intentan leer o escribir en un mismo archivo al mismo tiempo pueden producir inconsistencias.



**Bases de Datos:** En los sistemas de base de datos múltiples procesos pueden acceder a la misma información o tablas que tengan pero de igual manera al mismo tiempo. Por lo que si no se sincroniza adecuadamente el acceso, pueden ocurrir problemas de consistencia.

Identificar una sección crítica varía dependiendo del recurso compartido y el acceso que tienen varios procesos. Estos recursos mencionados anteriormente nos ayudan a entender la sección crítica y de las diferentes circunstancias en las que podemos verla. Para poder evitar condiciones de carrera o inconsistencias de datos, es necesario poder implementar mecanismos de exclusión mutua que aseguren que solo un proceso pueda acceder a la sección crítica a la vez.

- Identificación de la exclusión mutua al respecto de los procesos, para comprender el funcionamiento de los sistemas en general.

La exclusión mutua es un objeto de programa que impide que varios usuarios accedan a la misma variable o datos compartidos al mismo tiempo. Con una sección crítica, una región de código en la que varios procesos o subprocesos acceden al mismo recurso compartido, esta idea se pone en práctica en la programación concurrente. Solo un subproceso puede usar un mutex a la vez, por lo tanto, al iniciar el programa, se crea un mutex con un nombre específico. Para evitar que otros subprocesos accedan a un recurso compartido al mismo tiempo, un subproceso que esté usando ese recurso debe bloquear el mutex. La exclusión mutua garantiza que, en cualquier momento, solo un proceso pueda acceder a una sección crítica de un programa.

Para esto nos podemos apoyar en la actividad de mecanismos de sincronización que elaboramos en esta materia en el cual se nos dio la tarea de describir a detalle cada mecanismo de exclusión mutua que podemos implementar.

**Candados (Mutex):** Es un bloqueo que permite que solo un proceso tenga acceso a un recurso en un momento determinado. Cuando un hilo entra, los demás tendrán que esperar a que el candado se libera antes de acceder a la misma sección crítica.

**Semáforo:** Un tipo variable entera que se utiliza para limitar el acceso a un recurso compartido por parte de varios subprocesos y evitar problemas críticos en secciones de un sistema concurrente, como un sistema operativo que puede gestionar varias tareas a la vez. Un semáforo utiliza dos operaciones, a saber, la wait y el signal para la sincronización del proceso.

**Monitores:** Es un tipo de datos abstractos que se utiliza para la sincronización de procesos de alto nivel. Ha sido desarrollado para superar los errores de tiempo que ocurren al usar el semáforo para la sincronización del proceso. Dado que el monitor es un tipo de datos abstractos, contiene las variables de datos compartidos. Estas variables de datos deben ser compartidas por todos los procesos. Por lo tanto, esto permite que los procesos se ejecuten en exclusión mutua.



Solo puede haber un proceso activo a la vez dentro de un monitor. Si cualquier otro proceso intenta acceder a la variable compartida en el monitor, se bloqueará y se alinearán en la cola para obtener acceso a los datos. Esto se hace mediante una variable condicional en el monitor. La variable condicional se utiliza para proporcionar un mecanismo de sincronización adicional.

Esto os enseña que la exclusión mutua es fundamental para gestionar el acceso concurrente a recursos compartidos y evitar problemas como condiciones de carrera, corrupción de datos, etc. Estos mecanismos que presente que solo fueron 3 candados, semáforos y monitores son importantes para que los procesos tengan una sincronización al momento de su acceso a la sección crítica a la vez, lo que nos asegura así una buena ejecución de programas multiproceso. Al poder implementar estos mecanismos en nuestros programas nos ahorraremos mucho defectos al momento de desarrollarlos.

### *3.2 Identificar los problemas que presentan la exclusión mutua y la sección crítica en programación secuencial*

- Identificar como la exclusión mutua involucra la determinación de las prioridades para el uso de una sección crítica al corrimiento del tiempo.

La exclusión mutua no solo garantiza que un proceso pueda acceder a una sección crítica a la vez, sino que también involucra la gestión de prioridades para asegurar que los procesos se ejecuten de manera eficiente y sin interferencias.

El uso de prioridades es importante cuando hay varios procesos que compiten por acceder a la misma sección crítica, ya que nos ayuda a la gestión y la asignación de recursos de forma más justa para evitar los interbloqueos.

En este caso en los sistemas de concurrencia los procesos no siempre tienen la misma importancia ya que algunos pueden necesitar acceso con mayor importancia que otros. Por eso, es necesario determinar una forma o mecanismo que nos permita gestionar las prioridades, para que nos ayude a que los procesos más importantes tengan acceso y no queden bloqueados indefinidamente.

Aquí hago una cita “El manejo de la exclusión mutua no solo trata de asegurar que solo un proceso acceda a la sección crítica, sino que también involucra la determinación de las prioridades para decidir qué proceso debe acceder primero, especialmente en sistemas con múltiples hilos de ejecución.”

**Stallings, W. (2018). Sistemas Operativos: Principios e Innovaciones**

Que a mi consideración hace una buena definición de las prioridades por lo que los mecanismos de exclusión mutua suelen estar diseñados para asignar prioridades a los procesos generalmente con mayor prioridad tienen acceso preferente a los recursos y en la asignación de prioridades puede ser dinámica, dependiendo de cómo se gestionen los recursos y cómo los procesos compitan. Por lo que se pueden reajustar las prioridades de los procesos en función del tiempo de espera o la urgencia de las tareas.



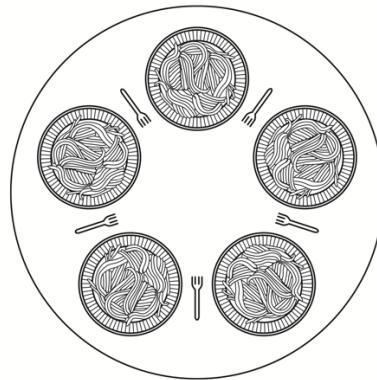
## IV. Aplicar la programación concurrente para solucionar problemas clásicos de sección crítica y exclusión mutua.

### 4.1 Aplicar la programación concurrente en los problemas: los filósofos comedores y lectores/escritores

- Exponer de manera general el problema de los filósofos comedores y lectores/escritores utilizando Java.

#### El problema de los filósofos comedores

En 1965, Dijkstra propuso y resolvió un problema de sincronización al que llamó el problema de los filósofos comedores. Desde ese momento, todos los que inventaban otra primitiva de sincronización se sentían obligados a demostrar qué tan maravillosa era esa nueva primitiva, al mostrar con qué elegancia resolvía el problema de los filósofos comedores. Este problema se puede enunciar simplemente de la siguiente manera. Cinco filósofos están sentados alrededor de una mesa circular. Cada filósofo tiene un plato de espagueti. El espagueti es tan resbaloso, que un filósofo necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor.



El problema es evitar que los filósofos se bloqueen entre sí y garantizar que todos puedan comer sin que se produzca un interbloqueo ni hambruna.



```
main.py | 1 import threading  
2 import time  
3  
4 # Número de filósofos  
5 N = 5  
6  
7 # Estados de los filósofos  
8 PENSANDO = 0  
9 HAMBRIENTO = 1  
10 COMIENDO = 2  
11  
12 # Arreglo de estados de los filósofos  
13 estado = [PENSANDO] * N  
14  
15 # Semáforo de exclusión mutua para la región crítica  
16 mutex = threading.Semaphore(1)  
17  
18 # Semáforos para cada filósofo  
19 s = [threading.Semaphore(0) for _ in range(N)]  
20  
21 # Definir la posición del vecino izquierdo y derecho  
22 def IZQUIERDO(i):  
23     return (i + N - 1) % N  
24  
25 def DERECHO(i):  
26     return (i + 1) % N  
27  
28 def pensar(i):  
    """Simula que el filósofo está pensando"""  
29     print(f"Filósofo {i} está pensando...")  
30     time.sleep(1)  
31  
32 def comer(i):  
    """Simula que el filósofo está comiendo"""  
33     print(f"Filósofo {i} quiere comer.")  
34     time.sleep(1)  
35  
36  
37     print(f"Filósofo {i} está comiendo...")  
38     time.sleep(1)  
39     print(f"Filósofo {i} ha terminado de comer.")  
40  
41 def probar(i):  
    """Función para probar si el filósofo puede comer"""  
42     izquierda = IZQUIERDO(i)  
43     derecha = DERECHO(i)  
44  
45     if estado[i] == HAMBRIENTO and estado[izquierda] != COMIENDO and estado[derecha] != COMIENDO:  
        estado[i] = COMIENDO  
46     elif estado[i] == COMIENDO:  
47         s[i].release() # Libera el semáforo para que el filósofo pueda comer  
48  
49 def tomar_tenedores(i):  
    """El filósofo trata de tomar los tenedores"""  
50     mutex.acquire() # entra a la región crítica  
51     estado[i] = HAMBRIENTO # El filósofo está hambriento  
52     probar(i) # Intenta tomar los tenedores  
53     mutex.release() # Sale de la región crítica  
54     s[i].acquire() # Se bloquea si no se pudo obtener los tenedores  
55  
56 def poner_tenedores(i):  
    """El filósofo pone los tenedores en la mesa"""  
57     mutex.acquire() # entra a la región crítica  
58     estado[i] = PENSANDO # El filósofo terminó de comer  
59     izquierda = IZQUIERDO(i)  
60     derecha = DERECHO(i)  
61     probar(izquierda) # Verifica si el vecino izquierdo puede comer  
62     probar(derecha) # Verifica si el vecino derecho puede comer  
63     mutex.release() # Sale de la región crítica  
64  
65 def filosofo(i):  
    """Simula el comportamiento del filósofo"""  
66     while True:  
67         pensar(i) # El filósofo está pensando  
68         tomar_tenedores(i) # El filósofo trata de tomar los tenedores  
69  
70  
71  
72         comer(i) # El filósofo está comiendo  
73         poner_tenedores(i) # El filósofo pone los tenedores  
74  
75 if __name__ == "__main__":  
76     threads = []  
77     for i in range(N):  
78         thread = threading.Thread(target=filosofo, args=(i,))  
79         threads.append(thread)  
80         thread.start()  
81  
82     for thread in threads:  
83         thread.join()  
84  
85  
86 Filósofo 0 está pensando...  
87 Filósofo 1 está pensando...  
88 Filósofo 2 está pensando...  
89 Filósofo 3 está pensando...  
90 Filósofo 4 está pensando...  
91 Filósofo 0 quiere comer.  
92 Filósofo 2 quiere comer.  
93 Filósofo 0 está comiendo...  
94 Filósofo 2 está comiendo...  
95 Filósofo 0 ha terminado de comer.  
96 Filósofo 0 está pensando...  
97 Filósofo 4 quiere comer.  
98 Filósofo 2 ha terminado de comer.  
99 Filósofo 2 está pensando...  
100 Filósofo 1 quiere comer.  
101 Filósofo 4 está comiendo...  
102 Filósofo 1 está comiendo...  
103 Filósofo 4 ha terminado de comer.  
104 Filósofo 4 está pensando...  
105 Filósofo 1 ha terminado de comer.  
106 Filósofo 3 quiere comer.  
107 Filósofo 1 está pensando...  
108 Filósofo 0 quiere comer.  
109 Filósofo 3 está comiendo...  
110 Filósofo 0 está comiendo...  
111 Filósofo 3 ha terminado de comer.  
112 Filósofo 3 está pensando...  
113 Filósofo 2 quiere comer.  
114 Filósofo 0 ha terminado de comer.
```



"El programa utiliza un arreglo de semáforos, uno por cada filósofo, de manera que los filósofos hambrientos puedan bloquearse si los tenedores que necesitan están ocupados. Observe que cada proceso ejecuta el procedimiento *filosofo* como su código principal, pero los demás procedimientos (*tomar\_tenedores*, *poner\_tenedores* y *probar*) son ordinarios y no procesos separados." Tanenbaum, A. S., & Bos, H. (2015). *Sistemas Operativos Modernos* (4<sup>a</sup> ed.). Pearson.

```

import threading
import time

# Número de filósofos
N = 5

# Estados de los filósofos
PENSANDO = 0
HAMBRIENTO = 1
COMIENDO = 2

# Arreglo de estados de los filósofos
estado = [PENSANDO] * N

# Semáforo de exclusión mutua para la región crítica
mutex = threading.Semaphore(1)

# Semáforos para cada filósofo
s = [threading.Semaphore(0) for _ in range(N)]

# Definir la posición del vecino izquierdo y derecho
def IZQUIERDO(i):
    return (i + N - 1) % N

def DERECHO(i):
    return (i + 1) % N

def pensar(i):
    """Simula que el filósofo está pensando"""
    print(f"Filósofo {i} está pensando...")
    time.sleep(1)

def comer(i):
    """Simula que el filósofo está comiendo"""
    print(f"Filósofo {i} quiere comer.")
    time.sleep(1)
    print(f"Filósofo {i} está comiendo...")
    time.sleep(1)
    print(f"Filósofo {i} ha terminado de comer.")
  
```



```

def probar(i):
    """Función para probar si el filósofo puede comer"""
    izquierda = IZQUIERDO(i)
    derecha = DERECHO(i)
    if estado[i] == HAMBRIENTO and estado[izquierda] != COMIENDO and
    estado[derecha] != COMIENDO:
        estado[i] = COMIENDO
        s[i].release() # Libera el semáforo para que el filósofo pueda comer

def tomar_tenedores(i):
    """El filósofo trata de tomar los tenedores"""
    mutex.acquire() # Entra a la región crítica
    estado[i] = HAMBRIENTO # El filósofo está hambriento
    probar(i) # Intenta tomar los tenedores
    mutex.release() # Sale de la región crítica
    s[i].acquire() # Se bloquea si no se pudo obtener los tenedores

def poner_tenedores(i):
    """El filósofo pone los tenedores en la mesa"""
    mutex.acquire() # Entra a la región crítica
    estado[i] = PENSANDO # El filósofo terminó de comer
    izquierda = IZQUIERDO(i)
    derecha = DERECHO(i)
    probar(izquierda) # Verifica si el vecino izquierdo puede comer
    probar(derecha) # Verifica si el vecino derecho puede comer
    mutex.release() # Sale de la región crítica

def filosofo(i):
    """Simula el comportamiento del filósofo"""
    while True:
        pensar(i) # El filósofo está pensando
        tomar_tenedores(i) # El filósofo trata de tomar los tenedores
        comer(i) # El filósofo está comiendo
        poner_tenedores(i) # El filósofo pone los tenedores

if __name__ == "__main__":
    threads = []
    for i in range(N):
        thread = threading.Thread(target=filosofo, args=(i,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
  
```



## El problema de los lectores y escritores

Otro problema famoso es el de los lectores y escritores (Courtois y colaboradores, 1971), que modela el acceso a una base de datos. Por ejemplo, imagine un sistema de reservación de aerolíneas, con muchos procesos en competencia que desean leer y escribir en él. Es aceptable tener varios procesos que lean la base de datos al mismo tiempo, pero si un proceso está actualizando (escribiendo) la base de datos, ningún otro proceso puede tener acceso a la base de datos, ni siquiera los lectores. El desafío radica en permitir que varios lectores accedan simultáneamente al recurso, pero asegurando que solo un escritor lo haga y que no haya lectura mientras se está escribiendo.

En esta solución, el primer lector en obtener acceso a la base de datos realiza una operación down en el semáforo *bd*. Los siguientes lectores simplemente incrementan un contador llamado *cl*. A medida que los lectores van saliendo, decrementan el contador y el último realiza una operación up en el semáforo, para permitir que un escritor bloqueado (si lo hay) entre.

```

main.py
1 import threading
2 import time
3 # Semáforos
4 mutex = threading.Lock() # Controla el acceso a 'cl'
5 bd = threading.Semaphore(1) # Controla el acceso a la base de datos
6 cl = 0 # Contador de lectores activos
7
8 # Simulación de leer base de datos
9 def leer_base_de_datos():
10    print("Leyendo base de datos...")
11    time.sleep(1) # Simula el tiempo de lectura
12
13 # Simulación de escribir en base de datos
14 def escribir_base_de_datos():
15    print("Escribiendo en base de datos...")
16    time.sleep(2) # Simula el tiempo de escritura
17
18 # Función del lector
19 def lector():
20    global cl
21    while True:
22        print("El lector está esperando para obtener acceso")
23        with mutex: # Obtiene acceso exclusivo a 'cl'
24            cl -= 1 # Ahora hay un lector más
25            print("El lector ha ingresado.")
26            if cl == 1: # Si es el primer lector, bloquea la base de datos para escritura
27                print("Es el primer lector, bloqueando la base de datos para el escritor")
28                bd.acquire()
29
30        # El lector puede leer ahora
31        print(f"El lector está leyendo {cl} lectores")
32        leer_base_de_datos()
33
34        with mutex: # Obtiene acceso exclusivo a 'cl'
35            cl += 1 # Ahora hay un lector menos
36            print(f"El lector ha terminado de leer. Número de lectores restantes: {cl}")
37
38        if cl == 0: # Si es el último lector, libera la base de datos
39            print("Es el último lector, liberando la base de datos para escritura...")
40            bd.release()
41
42            print("El lector ha terminado de usar los datos.")
43            time.sleep() # Simula el uso de los datos leídos
44
45 # Función del escritor
46 def escritor():
47    while True:
48        # Simula el pensamiento o procesamiento de datos previo
49        print("El escritor está pensando (región no crítica)")
50        time.sleep(2)
51
52        print("El escritor quiere escribir en la base de datos")
53        bd.acquire() # Obtiene acceso exclusivo a la base de datos
54
55        print("El escritor está actualizando los datos")
56        escribir_base_de_datos()
57
58        print("El escritor ha terminado de escribir y libera la base de datos.")
59        bd.release() # Libera el acceso exclusivo a la base de datos
60
61        # Crear y ejecutar los hilos de lectores y escritores
62        def crear_hilos():
63            num_lectores = 3
64            num_escritores = 2
65
66            # Crear hilos de lectores
67            hilos_lectores = [threading.Thread(target=lector) for _ in range(num_lectores)]
68
69            # Crear hilos de escritores
70            hilos_escritores = [threading.Thread(target=escritor) for _ in range(num_escritores)]
71
72            # Iniciar todos los hilos
73            for hilo in hilos_lectores + hilos_escritores:
74                hilo.daemon = True # Hacer los hilos daemon para que se cierren cuando termine el programa
75                hilo.start()
76
77            # Mantener el hilo principal activo para que los hilos continúen ejecutándose
78            for hilo in hilos_lectores + hilos_escritores:
79                hilo.join()
80
81        # Ejecutar el programa
82        if __name__ == "__main__":
83            crear_hilos()

```



```

El lector está esperando para obtener acceso
El lector ha ingresado.
El lector está esperando para obtener acceso
El lector está esperando para obtener acceso
Es el primer lector, bloqueando la base de datos para el escritor
El escritor está pensando (región no crítica)
El lector está leyendo (1 lectores)
Leyendo base de datos...
El escritor está pensando (región no crítica)
El lector ha ingresado.
El lector está leyendo (2 lectores)
El lector ha ingresado.
Leyendo base de datos...
El lector está leyendo (3 lectores)
Leyendo base de datos...
El lector ha terminado de leer. Número de lectores restantes: 2
El lector ha terminado de usar los datos.
El lector ha terminado de leer. Número de lectores restantes: 1
El lector ha terminado de usar los datos.
El lector ha terminado de leer. Número de lectores restantes: 0
Es el último lector, liberando la base de datos para escritura...
El lector ha terminado de usar los datos.
El escritor quiere escribir en la base de datos
El escritor está actualizando los datos
Escribiendo en base de datos...
El escritor quiere escribir en la base de datos
El lector está esperando para obtener acceso
El lector ha ingresado.
Es el primer lector, bloqueando la base de datos para el escritor
El lector está esperando para obtener acceso
El lector está esperando para obtener acceso
El escritor ha terminado de escribir y libera la base de datos.
El escritor está pensando (región no crítica)
El escritor está actualizando los datos
Escribiendo en base de datos...
El escritor quiere escribir en la base de datos
El escritor ha terminado de escribir y libera la base de datos.
El escritor está pensando (región no crítica)
  
```

"De esta forma, un escritor tiene que esperar a que terminen los lectores que estaban activos cuando llegó, pero no tiene que esperar a los lectores que llegaron después de él. La desventaja de esta solución es que logra una menor concurrencia y por ende, un menor rendimiento" Tanenbaum, A.S., & Bos, H. (2015). *Sistemas Operativos Modernos* (4<sup>a</sup> ed.). Pearson.

```

import threading
import time
# Semáforos
mutex = threading.Lock() # Controla el acceso a 'cl'
bd = threading.Semaphore(1) # Controla el acceso a la base de datos
cl = 0 # Contador de lectores activos

# Simulación de leer base de datos
def leer_base_de_datos():
    print("Leyendo base de datos...")
    time.sleep(1) # Simula el tiempo de lectura

# Simulación de escribir en base de datos
def escribir_base_de_datos():
    print("Escribiendo en base de datos...")
  
```



```

time.sleep(2) # Simula el tiempo de escritura

# Función del lector
def lector():
    global cl
    while True:
        print("El lector está esperando para obtener acceso")
        with mutex: # Obtiene acceso exclusivo a 'cl'
            cl += 1 # Ahora hay un lector más
            print(f"El lector ha ingresado.")
            if cl == 1: # Si es el primer lector, bloquea la base de datos para escritura
                print("Es el primer lector, bloqueando la base de datos para el escritor")
                bd.acquire()

        # El lector puede leer ahora
        print(f"El lector está leyendo ({cl} lectores")
        leer_base_de_datos()

        with mutex: # Obtiene acceso exclusivo a 'cl'
            cl -= 1 # Ahora hay un lector menos
            print(f"El lector ha terminado de leer. Número de lectores restantes: {cl}")
            if cl == 0: # Si es el último lector, libera la base de datos
                print("Es el último lector, liberando la base de datos para escritura...")
                bd.release()

        print("El lector ha terminado de usar los datos.")
        time.sleep(1) # Simula el uso de los datos leídos

# Función del escritor
def escritor():
    while True:
        # Simula el pensamiento o procesamiento de datos previo
        print("El escritor está pensando (región no crítica)")
        time.sleep(2)

        print("El escritor quiere escribir en la base de datos")
        bd.acquire() # Obtiene acceso exclusivo a la base de datos

        print("El escritor está actualizando los datos")
        escribir_base_de_datos()

        print("El escritor ha terminado de escribir y libera la base de datos")
        bd.release() # Libera el acceso exclusivo a la base de datos
  
```



```

# Crear y ejecutar los hilos de lectores y escritores
def crear_hilos():
    num_lectores = 3
    num_escritores = 2

    # Crear hilos de lectores
    hilos_lectores = [threading.Thread(target=lector) for _ in range(num_lectores)]

    # Crear hilos de escritores
    hilos_escritores = [threading.Thread(target=escritor) for _ in range(num_escritores)]

    # Iniciar todos los hilos
    for hilo in hilos_lectores + hilos_escritores:
        hilo.daemon = True # Hacer los hilos daemon para que se cierren cuando termine
    el programa
        hilo.start()

    # Mantener el hilo principal activo para que los hilos continúen ejecutándose
    for hilo in hilos_lectores + hilos_escritores:
        hilo.join()

# Ejecutar el programa
if __name__ == "__main__":
    crear_hilos()
  
```

## V. Describir de manera general el algoritmo de Jacobi para solución de sistemas de ecuaciones lineales.

### 5.1 Identificar los métodos de solución para sistemas de ecuaciones lineales.

- Investigar acerca de los sistemas de ecuaciones lineales, su definición y como se pueden solucionar
- Identificación de los métodos de solución directos e iterativos para sistemas de ecuaciones lineales

**Métodos directos** son aquellos que proporcionan una solución exacta en un número finito de pasos, siempre que el sistema tenga solución. Son apropiados para sistemas de tamaño moderado en ellos encontramos:

**Método de Gauss-Jordan:** es una extensión del método de eliminación de Gauss que no solo reduce la matriz A a triangular, sino que la lleva a la forma escalonada reducida, lo que permite obtener la solución directamente sin necesidad de sustitución hacia atrás. Se soluciona realizando operaciones elementales para reducir la matriz aumentada  $[A | b]$  a la forma escalonada reducida. Finalizado al leer la solución directamente desde la matriz resultante.



**Método de eliminación de Gauss:** es un procedimiento que transforma el sistema de ecuaciones en una matriz triangular superior a través de operaciones elementales sobre las filas de la matriz. Luego, se resuelve el sistema por sustitución hacia atrás. Se soluciona realizando operaciones elementales para transformar la matriz A en una forma triangular superior. Una vez obtenida la matriz triangular, resolver el sistema utilizando sustitución hacia atrás.

**Método de descomposición LU:** El método de descomposición LU factoriza la matriz A en el producto de una matriz triangular inferior L y una matriz triangular superior U. Esto permite resolver el sistema en dos pasos: primero se resuelve Ly=b, y luego Ux=y. Se soluciona al descomponer la matriz A en L y U mediante un proceso de factorización. Resolver Ly=b utilizando sustitución hacia adelante. Y resolver Ux=y utilizando sustitución hacia atrás.

**Método de descomposición QR:** Descompone la matriz A en el producto de una matriz ortogonal Q y una matriz triangular superior R, de forma que A=QR. Este método es útil especialmente cuando A no es cuadrada o está mal condicionada. Se soluciona al descomponer A en las matrices Q y R. Resolviendo el sistema Rx=Q<sup>T</sup>b.

**Métodos iterativos** son apropiados para sistemas grandes y dispersos, ya que no requieren que se almacenen grandes matrices. En lugar de obtener una solución exacta, proporcionan aproximaciones sucesivas que mejoran en cada iteración, en ellos encontramos:

**Método de jacobi:** cada incógnita se calcula de forma independiente en cada iteración, utilizando los valores de la iteración anterior. Es un método sencillo, pero con convergencia lenta. Su solución se basa en cada iteración, calcular una nueva aproximación para cada incógnita utilizando la fórmula:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^k}{a_{ii}}$$

Se repiten las iteraciones hasta que las soluciones coincidan.

**Método de Gauss-Seidel:** es una mejora del método de Jacobi. En cada iteración, las incógnitas ya actualizadas en la misma iteración se usan inmediatamente para calcular las siguientes incógnitas, lo que acelera la convergencia. Su solución se basa en cada iteración, calcular las incógnitas utilizando la fórmula:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k+1)}}{a_{ii}}$$

Se repiten las iteraciones hasta que las soluciones coincidan.

**Método de relajación (SOR):** es una variante del método de Gauss-Seidel que introduce un parámetro de relajación  $\omega$ . Este parámetro controla la rapidez con la que las soluciones se ajustan en cada iteración, acelerando la convergencia. Su solución se basa en actualizar las incógnitas utilizando la fórmula de Gauss-Seidel modificada por el parámetro  $w$ :



$$x_i^{(k+1)} = (1 - \omega)x_i^k + \omega\left(\frac{b_i - \sum_{j \neq i} a_{ij}x_j^{(k+1)}}{a_{ii}}\right)$$

**Método de gradiente conjugado:** es un algoritmo iterativo utilizado para resolver sistemas de ecuaciones lineales donde la matriz A es simétrica y definida positiva. Este método minimiza el error de la solución en cada paso. Solucionandolo al inicializar un vector de solución  $x_0$ . En cada iteración, calcular el siguiente vector de solución minimizando el error en la dirección del gradiente conjugado. Continuar iterando hasta que el error sea suficientemente pequeño.

5.2 Describir el método de Jacobi mediante el algoritmo que resuelve un sistema de ecuaciones lineales general.

- Exponer el algoritmo de Jacobi y describir su desarrollo
- Resolver un sistema de ecuaciones de tamaño 3x3 mediante el método de Jacobi
- Explicar las soluciones de Jacobi de manera manual

El método de Jacobi es un método iterativo que se utiliza para resolver sistemas de ecuaciones lineales de la forma general  $Ax=b$ . Su funcionamiento consiste en realizar iteraciones sucesivas en las que se calculan los valores aproximados de las incógnitas.

Su algoritmo se describe de la siguiente manera:

- Escribe el sistema de ecuaciones en su forma estándar. El sistema de ecuaciones debe estar expresado en forma de matriz:  $Ax=b$ , donde A es la matriz de coeficientes, x es el vector de incógnitas y b es el vector de términos constantes.
  - Despeja cada incógnita en términos de las otras incógnitas. Cada ecuación del sistema debe ser transformada para despejar una incógnita en función de las otras.
- La forma general de la ecuación será:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij}x_j^k \right)$$

$x_i^{(k+1)}$  es el valor de la incógnita  $x_i$  en la  $(k+1)$  iteración.

$x_j^k$  son los valores de las incógnitas en la iteración anterior k.

$a_{ii}$  es el coeficiente en la diagonal de la matriz A.

- Iterar hasta que la solución converja. Este proceso se repite para cada incógnita hasta que la diferencia entre los valores obtenidos en iteraciones sucesivas sea lo suficientemente pequeña. Se puede usar una tolerancia definida por el usuario o un número de iteraciones predefinido.

Sistema de ecuaciones de tamaño 3x3 mediante el método de Jacobi.

$$\begin{aligned} 4x_1 - x_2 + x_3 &= 3 \\ x_1 + 3x_2 + 2x_3 &= 7 \\ 2x_1 - x_2 + 3x_3 &= 5 \end{aligned}$$



Despejar las incógnitas

$$\begin{aligned}x_1 &= \frac{3 + x_2 - x_3}{4} \\x_2 &= \frac{7 - x_1 - 2x_3}{3} \\x_3 &= \frac{5 - 2x_1 + x_2}{3}\end{aligned}$$

Inicializar los valores

$$\begin{aligned}x_1^{(0)} &= 0 \\x_2^{(0)} &= 0 \\x_3^{(0)} &= 0\end{aligned}$$

Comenzamos con estos valores para comenzar a interar.

Iteración 1 ( $k = 1$ ):

$$\begin{aligned}x_1^{(1)} &= \frac{3 + 0 - 0}{4} = \frac{3}{4} = 0.75 \\x_2^{(1)} &= \frac{7 - 0 - 2(0)}{3} = \frac{7}{3} = 2.3333 \\x_3^{(1)} &= \frac{5 - 2(0) + 0}{3} = \frac{5}{3} = 1.6667\end{aligned}$$

Iteración 2 ( $k = 2$ ):

Usamos los valores obtenidos en la iteración anterior

$$\begin{aligned}x_1^{(2)} &= \frac{3 + 2.3333 - 1.6667}{4} = \frac{3.6666}{4} = 0.9167 \\x_2^{(2)} &= \frac{7 - 0.75 - 2(1.6667)}{3} = \frac{2.9166}{3} = 0.9722 \\x_3^{(2)} &= \frac{5 - 2(0.75) + 2.3333}{3} = \frac{5.8333}{3} = 1.9444\end{aligned}$$

Iteración 3 ( $k = 3$ ):

$$\begin{aligned}x_1^{(3)} &= \frac{3 + 0.9722 - 1.9444}{4} = \frac{2.0278}{4} = 0.5069 \\x_2^{(3)} &= \frac{7 - 0.9167 - 2(1.9444)}{3} = \frac{2.1945}{3} = 0.7315 \\x_3^{(3)} &= \frac{5 - 2(0.9167) + 0.9722}{3} = \frac{4.1388}{3} = 1.3796\end{aligned}$$

Iteración 4 ( $k = 4$ ):

$$\begin{aligned}x_1^{(4)} &= \frac{3 + 0.7315 - 1.3796}{4} = \frac{2.3519}{4} = 0.5879 \\x_2^{(4)} &= \frac{7 - 0.5069 - 2(1.3796)}{3} = \frac{3.7340}{3} = 1.2447 \\x_3^{(4)} &= \frac{5 - 2(0.5069) + 0.7315}{3} = \frac{4.7177}{3} = 1.5726\end{aligned}$$

Iteración 5 ( $k = 5$ ):

$$x_1^{(5)} = \frac{3 + 1.2447 - 1.5726}{4} = \frac{2.6721}{4} = 0.6680$$



$$x_2^{(5)} = \frac{7 - 0.5879 - 2(1.5726)}{3} = \frac{3.2669}{3} = 1.0889$$

$$x_3^{(5)} = \frac{5 - 2(0.5879) + 1.2447}{3} = \frac{5.0689}{3} = 1.6896$$

Realice 5 iteraciones para ver un mejor resultado, por lo que podemos observar que la diferencia entre los valores calculados de las iteraciones está disminuyendo. Vamos a evaluar si el proceso ha convergido, calculamos la diferencia entre los valores de la iteración 5 y la iteración 4:

$$x_1^{(5)} - x_1^{(4)} = 0.6680 - 0.5879 = 0.0801$$

$$x_2^{(5)} - x_2^{(4)} = 1.0889 - 1.2447 = 0.1558$$

$$x_3^{(5)} - x_3^{(4)} = 1.6896 - 1.5726 = 0.117$$

Las diferencias entre los valores de las incógnitas en la iteración 5 y la iteración 4 son mayores que la tolerancia  $\epsilon=0.0001$ , lo que significa que no hemos alcanzado la convergencia aún. Por lo que necesitamos continuar con más iteraciones hasta que todas las diferencias sean menores que  $\epsilon=0.0001$ , lo que indicará que hemos encontrado una solución aproximada al sistema de ecuaciones.

## **VI. Implementar el algoritmo de Jacobi utilizando programación concurrente.**

*6.1 Aplicar la programación concurrente (hilos, procesos, semáforos, transacciones, etc.) para implementar el algoritmo de Jacobi en un sistema de ecuaciones lineales de gran tamaño.*

- Seleccionar algunos de los modelos de programación concurrente: hilos, procesos, semáforos, monitores, transacciones, etc.
- Retomar las clases antes mencionadas o implementar las clases de hilos, procesos, etc.
- Implementar la entrada de un sistema de tamaño mayor a 5x5, junto con la solución aproximada o inicial.
- Implementar la solución del algoritmo de Jacobi para el sistema planteado.
- Documentar el software, las pruebas y las corridas dada la aplicación instrumentada.



```

main.py
1 import threading
2 import numpy as np
3 import time
4
5 # Número de iteraciones y la tolerancia para la convergencia
6 MAX_ITER = 50
7 TOLERANCIA = 0.0001
8
9 # Semáforo para sincronizar el acceso a las variables compartidas
10 semaforo = threading.Semaphore(1)
11
12 # Tamaño del sistema (5x5)
13 n = 5
14
15 # Matriz A
16 A = np.array([[4, -1, 0, 0, 1],
17               [-1, 4, -1, 0, 0],
18               [0, -1, 4, -1, 0],
19               [1, 0, -1, 4, -1],
20               [0, 1, 0, -1, 3]], dtype=float)
21
22 # Vector b
23 b = np.array([15, 10, 13, 10, 11], dtype=float)
24
25 # Vector de soluciones θ
26 x = np.zeros(n, dtype=float)
27
28 # Vector para almacenar los valores calculados
29 x_new = np.zeros(n, dtype=float)
30
31 # Función para calcular una incógnita x_i en cada iteración
32 def calcular_x(i):
33     global x, x_new, A, b
34     suma = b[i]
35     for j in range(n):
36         if j != i:
37             suma -= A[i][j] * x[j]
38
39     # Calculamos el nuevo valor de x[i]
40     x_new[i] = suma / A[i][i]
41
42 # Función para realizar el método de Jacobi utilizando hilos
43 def jacobi():
44     global x, x_new
45     iteraciones = 0
46     while iteraciones < MAX_ITER:
47         print(f"\nIteración {iteraciones + 1}:")
48
49         # Iniciar hilos para cada incógnita
50         threads = []
51         for i in range(n):
52             thread = threading.Thread(target=calcular_x, args=(i,))
53             threads.append(thread)
54             thread.start()
55
56         # Sincronizar los hilos
57         for thread in threads:
58             thread.join()
59
60         # Mostrar los valores calculados de esta iteración
61         print(f"Soluciones actuales: {x_new}")
62
63         # Evaluar la diferencia entre las soluciones anteriores y las nuevas
64         diferencia = np.abs(x_new - x)
65         print(f"Diferencia entre iteraciones: {diferencia}")
66
67         # Verificar la convergencia
68         convergencia = True
69         for i in range(n):
70             if diferencia[i] > TOLERANCIA:
71                 convergencia = False
72                 break
73
74         # Si se cumple la convergencia, salimos
75         if convergencia:
76             print("\nConvergencia alcanzada.")
77             break
78
79         # Actualizamos los valores de x
80         semaforo.acquire()
81         x[:] = x_new[:]
82         semaforo.release()
83
84         iteraciones += 1
85
86     return x
87
88 # Probar la implementación
89 inicio = time.time()
90 solución = jacobi()
91 fin = time.time()
92
93 # Resultados
94 print("\nSolución aproximada:", solución)

```



```

Iteración 1:
Soluciones actuales: [3.75      2.5      3.25      2.5      3.66666667]
Diferencia entre iteraciones: [3.75      2.5      3.25      2.5      3.66666667]

Iteración 2:
Soluciones actuales: [3.45833333 4.25      4.5      3.29166667 3.66666667]
Diferencia entre iteraciones: [0.29166667 1.75      1.25      0.79166667 0.        ]

Iteración 3:
Soluciones actuales: [3.89583333 4.48958333 5.13541667 3.67708333 3.34722222]
Diferencia entre iteraciones: [0.4375      0.23958333 0.63541667 0.38541667 0.31944444]

Iteración 4:
Soluciones actuales: [4.03559028 4.7578125 5.29166667 3.64670139 3.39583333]
Diferencia entre iteraciones: [0.13975694 0.26822917 0.15625     0.03038194 0.04861111]

Iteración 5:
Soluciones actuales: [4.09049479 4.83181424 5.35112847 3.66297743 3.2962963 ]
Diferencia entre iteraciones: [0.05490451 0.07400174 0.05946181 0.01627604 0.09953704]

Iteración 6:
Soluciones actuales: [4.13387948 4.86040582 5.37369792 3.63923249 3.2770544 ]
Diferencia entre iteraciones: [0.04338469 0.02859158 0.02256944 0.02374494 0.0192419 ]

Iteración 7:
Soluciones actuales: [4.14583785 4.87689435 5.37490958 3.62921821 3.25960889]
Diferencia entre iteraciones: [0.01195837 0.01648853 0.00121166 0.01001429 0.01744551]

Iteración 8:
Soluciones actuales: [4.15432136 4.88018686 5.37652814 3.62217015 3.25077462]
Diferencia entre iteraciones: [0.00848351 0.00329251 0.00161856 0.00704805 0.00883427]

Iteración 9:
Soluciones actuales: [4.15735306 4.88271238 5.37558925 3.61824535 3.24732777]
Diferencia entre iteraciones: [0.0030317 0.00252552 0.00093889 0.00392481 0.00344685]

Iteración 10:
Soluciones actuales: [4.15884615 4.88323558 5.37523943 3.61639099 3.24517766]
Diferencia entre iteraciones: [0.00149309 0.0005232 0.00034982 0.00185436 0.00215011]

Iteración 11:
Soluciones actuales: [4.15951448 4.8835214 5.37490664 3.61539273 3.24438514]
Diferencia entre iteraciones: [0.00066833 0.00028582 0.00033279 0.00099826 0.00079252]

Iteración 12:
Soluciones actuales: [4.15978406 4.88360528 5.37472853 3.61494432 3.24395711]
Diferencia entre iteraciones: [2.69584531e-04 8.38845846e-05 1.78109471e-04 4.48409258e-04
 4.28024470e-04]

Iteración 13:
Soluciones actuales: [4.15991204 4.88362815 5.3746374 3.6147254 3.24377968]
Diferencia entre iteraciones: [1.27977264e-04 2.28687651e-05 9.11311685e-05 2.18929618e-04
 1.77431281e-04]

Iteración 14:
Soluciones actuales: [4.15996212 4.88363736 5.37458839 3.61462626 3.24369908]
Diferencia entre iteraciones: [5.00750115e-05 9.21152379e-06 4.90152133e-05 9.91349283e-05
 8.05994611e-05]

Convergencia alcanzada.

Solución aproximada: [4.15991204 4.88362815 5.3746374 3.6147254 3.24377968]

...Program finished with exit code 0
Press ENTER to exit console.

```



...Program finished with exit code 0  
Press ENTER to exit console.

```
print(f"Soluciones actuales: {x_new}")
```

En cada iteración del bucle while en la función jacobi(), se imprime el vector x\_new, que contiene los valores de las soluciones calculadas en esa iteración.

```
diferencia = np.abs(x_new - x)
```

```
print(f"Diferencia entre iteraciones: {diferencia}")
```

Para ver cómo cambia la solución en cada iteración, calculamos la diferencia absoluta entre las soluciones actuales (x\_new) y las soluciones de la iteración anterior (x).

```
convergencia = True
```

```
for i in range(n):
```

```
    if diferencia[i] > TOLERANCIA:
```

```
        convergencia = False
```

```
        break
```

```
if convergencia:
```

```
    print("\nConvergencia alcanzada.")
```

```
    break
```

Después de mostrar la diferencia, evaluamos si todas las diferencias son menores que la tolerancia. Si todas las diferencias son pequeñas, consideramos que el algoritmo ha convergido y rompemos el ciclo. Si no se ha alcanzado la convergencia, actualizamos el valor de x con los nuevos valores en x\_new para continuar en la siguiente iteración.

Como estamos usando hilos para calcular cada incógnita de forma concurrente, utilizamos un semáforo (semaforo.acquire() y semaforo.release()) para garantizar que la actualización del vector x no cause condiciones de carrera.

```
import threading
```

```
import numpy as np
```

```
import time
```

```
# Número de iteraciones y la tolerancia para la convergencia
```

```
MAX_ITER = 50
```

```
TOLERANCIA = 0.0001
```

```
# Semáforo para sincronizar el acceso a las variables compartidas
```

```
semaforo = threading.Semaphore(1)
```

```
# Tamaño del sistema (5x5)
```

```
n = 5
```

```
# Matriz A
```

```
A = np.array([[4, -1, 0, 0, 1],
```

```
[-1, 4, -1, 0, 0],
```

```
[0, -1, 4, -1, 0],
```



```

[1, 0, -1, 4, -1],
[0, 1, 0, -1, 3]], dtype=float)

# Vector b
b = np.array([15, 10, 13, 10, 11], dtype=float)

# Vector de soluciones 0
x = np.zeros(n, dtype=float)

# Vector para almacenar los valores calculados
x_new = np.zeros(n, dtype=float)

# Función para calcular una incógnita x_i en cada iteración
def calcular_x(i):
    global x, x_new, A, b
    suma = b[i]
    for j in range(n):
        if j != i:
            suma -= A[i][j] * x[j]

    # Calculamos el nuevo valor de x[i]
    x_new[i] = suma / A[i][i]

# Función para realizar el método de Jacobi utilizando hilos
def jacobi():
    global x, x_new
    iteraciones = 0
    while iteraciones < MAX_ITER:
        print(f"\nIteración {iteraciones + 1}:")

        # Iniciar hilos para cada incógnita
        threads = []
        for i in range(n):
            thread = threading.Thread(target=calcular_x, args=(i,))
            threads.append(thread)
            thread.start()

        # Sincronizar los hilos
        for thread in threads:
            thread.join()

        # Mostrar los valores calculados de esta iteración
        print(f"Soluciones actuales: {x_new}")

        # Evaluar la diferencia entre las soluciones anteriores y las nuevas
        diferencia = np.abs(x_new - x)
        print(f"Diferencia entre iteraciones: {diferencia}")
  
```



```

# Verificar la convergencia
convergencia = True
for i in range(n):
    if diferencia[i] > TOLERANCIA:
        convergencia = False
        break

# Si se cumple la convergencia, salimos
if convergencia:
    print("\nConvergencia alcanzada.")
    break

# Actualizamos los valores de x
semaforo.acquire()
x[:] = x_new[:]
semaforo.release()

iteraciones += 1

return x

# Probar la implementación
inicio = time.time()
solucion = jacobi()
fin = time.time()

# Resultados
print("\nSolución aproximada:", solucion)

```

## Conclusión

Para concluir esta ultima etapa del proyecto integrador, rescate mucha información que me ayudo al reforzamiento de los conocimientos adquiridos de la programación concurrente, como el método de Jacobi, al ser una técnica iterativa, ofrece una solución eficaz para sistemas de ecuaciones lineales de gran tamaño, permitiendo la aproximación de las incógnitas a través de iteraciones sucesivas. Al integrar la programación concurrente, se logra optimizar el rendimiento, aprovechando la capacidad de los procesadores modernos para realizar múltiples cálculos simultáneamente, ya que la actividad nos solicito realizarlo de manera manual lo cual me hizo ver todo los cálculos que el algoritmo hace en unos segundos. El uso de hilos y semáforos no solo mejora la eficiencia de la solución, sino que también asegura la integridad de los datos mediante la sincronización de los procesos, evitando conflictos o corrupción. Estos conceptos, junto con los mecanismos de exclusión mutua y la gestión de hilos, fortalecen el aprendizaje de competencias clave en la programación, permitiendo aplicar conocimientos teóricos a problemas



prácticos de gran escala. Esta actividad, al combinar los aspectos de la programación concurrente y la resolución de sistemas lineales, contribuye al desarrollo de habilidades fundamentales para abordar desafíos complejos en el campo de la computación.

## Referencia

(S/f). Ibiblio.org. Recuperado el 30 de enero de 2025, de <https://www.ibiblio.org/pub/linux/docs/LuCaS/Manuales-LuCAS/doc-cluster-computadoras/doc-cluster-computadoras-html/node44.html>

(S/f-b). Redalyc.org. Recuperado el 30 de enero de 2025, de <https://www.redalyc.org/pdf/944/94403110.pdf>

CAPITULO 4. (s/f). Angelfire.com. Recuperado el 30 de enero de 2025, de <https://www.angelfire.com/nb/iamtheevil/capitulo4.htm>

P.O.P.V.F.A. a. A. (s/f). A.2. *Obtención de PVM*.Uva.es. Recuperado el 30 de enero de 2025, de <https://www.infor.uva.es/~bastida/Arquitecturas%20Avanzadas/pvm.pdf>

Jorba, E. y Suppi, R. (2004). *Administración Avanzada GNU Linux*. Recuperado el 30 de enero de 2025, de <https://libros.metabiblioteca.org/bitstream/001/425/1/871.pdf>

(S/f-c). Wikipedia.org. Recuperado el 30 de enero de 2025, de [https://es.wikipedia.org/wiki/Máquina\\_Virtual\\_Paralela](https://es.wikipedia.org/wiki/Máquina_Virtual_Paralela)

Academia Usero Estepona Videos Educativos (Productor). (06 de Marzo de 2015). *Lectores y escritores con locks Concurrencia Java*. Recuperado el 9 de febrero de 2025, de <https://www.youtube.com/watch?v=Pq7w9gxcuWA>

Sáenz, J. (Productor). (18 de Octubre de 2015). *Monitores y Semáforos en java*. Recuperado el 9 de febrero de 2025, de <https://www.youtube.com/watch?v=IU3Vfjpt8A>

Tanenbaum, A. (2009). *Sistemas Operativos Modernos*. Recuperado el 9 de febrero de 2025, de <https://gc.scalahed.com/recursos/files/r161r/w21040w/Sistemas%20operativos%20modernos.pdf>

Pavón, J. (2015). *Programación Concurrente con Java. Diseño de Sistemas Operativos*. Recuperado el 9 de febrero de 2025, de <http://grasia.fdi.ucm.es/jpavon/docencia/dso/programacionconcurrentejava.pdf>



Improve, K. F. (2010, diciembre 25). *Critical section in synchronization*. GeeksforGeeks. Recuperado el 9 de febrero de 2025, [https://www-geeksforgeeks.org.translate.goog/g-fact-70/?x\\_tr\\_sl=en&x\\_tr\\_tl=es&x\\_tr\\_hl=es&x\\_tr\\_pto=rq](https://www-geeksforgeeks.org.translate.goog/g-fact-70/?x_tr_sl=en&x_tr_tl=es&x_tr_hl=es&x_tr_pto=rq)

*Mutual exclusion.* (2023, enero 24). NordVPN. Recuperado el 9 de febrero de 2025, [https://nordvpn.com.translate.goog/cybersecurity/glossary/mutual-exclusion/?x\\_tr\\_sl=en&x\\_tr\\_tl=es&x\\_tr\\_hl=es&x\\_tr\\_pto=sqe](https://nordvpn.com.translate.goog/cybersecurity/glossary/mutual-exclusion/?x_tr_sl=en&x_tr_tl=es&x_tr_hl=es&x_tr_pto=sqe)

Silberschatz Galvin (S/f). *Sistemas Operativos* Edu.sv. Recuperado el 9 de febrero de 2025, de [http://www.fmu.es.edu.sv/files/Sistemas\\_Operativos-Silberschatz\\_Galvin.pdf](http://www.fmu.es.edu.sv/files/Sistemas_Operativos-Silberschatz_Galvin.pdf)

William Stalling (S/f-b). *Sistemas Operativos* Edu.bo. Recuperado el 9 de febrero de 2025, de <http://cotana.informatica.edu.bo/downloads/Sistemas%20Operativos.pdf>

Generalidades, 1. (s/f). METODOS NUMERICOS (IC343). Wordpress.com. Recuperado el 22 de febrero de 2025, de <https://cristiancastrop.wordpress.com/wp-content/uploads/2010/09/silabo-metodos-numericos-ic343-2015.pdf>

(S/f). Academia.edu. Recuperado el 22 de febrero de 2025, de [https://www.academia.edu/34219302/Guía\\_de\\_Cálculo\\_Numérico](https://www.academia.edu/34219302/Guía_de_Cálculo_Numérico)

(S/f). Edu.co. Recuperado el 22 de febrero de 2025, de <http://artemisa.unicauc.edu.co/~cardila/Chapra.pdf>

UNJu Virtual - Plataforma de Educación a Distancia. (s/f). Edu.ar. Recuperado el 22 de febrero de 2025, de <https://virtual.unju.edu.ar/mod/resource/view.php?id=20629&forceview=1>

Análisis Numérico - Richard L. Burden & J. Douglas Faires (7ma Edición) PDF. (s/f). Scribd. Recuperado el 22 de febrero de 2025, de <https://es.scribd.com/document/353338955/Analisis-Numerico-Richard-L-Burden-J-Douglas-Faires-7ma-Edicion-pdf>

(S/f). Libretexts.org. Recuperado el 22 de febrero de 2025, de [https://espanol.libretexts.org/Matemáticas/Algebra\\_lineal/Álgebra\\_Matricial\\_con\\_APLICACIONES\\_Computacionales\\_\(Colbry\)/06%3A\\_03\\_Asignación\\_en\\_Clase\\_-Resolver\\_Sistemas\\_Lineales\\_de\\_Ecuaciones/6.2%3A\\_Método\\_Jacobi\\_para\\_resolver\\_ecuaciones\\_lineales](https://espanol.libretexts.org/Matemáticas/Algebra_lineal/Álgebra_Matricial_con_APLICACIONES_Computacionales_(Colbry)/06%3A_03_Asignación_en_Clase_-Resolver_Sistemas_Lineales_de_Ecuaciones/6.2%3A_Método_Jacobi_para_resolver_ecuaciones_lineales)

(S/f-b). Edu.ec. Recuperado el 22 de febrero de 2025, de [http://cimqsys.esepoch.edu.ec/direccion-publicaciones/public/docs/books/2023-04-20-144505-metodos\\_numericos.pdf](http://cimqsys.esepoch.edu.ec/direccion-publicaciones/public/docs/books/2023-04-20-144505-metodos_numericos.pdf)

