

## Laboratorio 4 V2

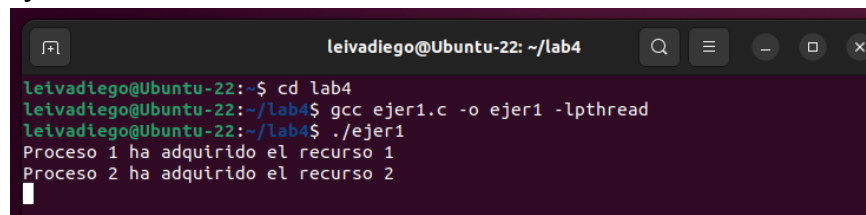
Diego Alberto Leiva Pérez  
Carné: 21752

### Ejercicio 1

#### Programa:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 pthread_mutex_t recurso1 = PTHREAD_MUTEX_INITIALIZER;
7 pthread_mutex_t recurso2 = PTHREAD_MUTEX_INITIALIZER;
8
9 void* proceso1(void* arg) {
10     while(1) {
11         pthread_mutex_lock(&recurso1);
12         printf("Proceso 1 ha adquirido el recurso 1\n");
13         sleep(1);
14         pthread_mutex_lock(&recurso2);
15         printf("Proceso 1 ha adquirido el recurso 2\n");
16         pthread_mutex_unlock(&recurso2);
17         pthread_mutex_unlock(&recurso1);
18         sleep(1);
19     }
20     return NULL;
21 }
22
23 void* proceso2(void* arg) {
24     while(1) {
25         pthread_mutex_lock(&recurso2);
26         printf("Proceso 2 ha adquirido el recurso 2\n");
27         sleep(1);
28         pthread_mutex_lock(&recurso1);
29         printf("Proceso 2 ha adquirido el recurso 1\n");
30         pthread_mutex_unlock(&recurso1);
31         pthread_mutex_unlock(&recurso2);
32         sleep(1);
33     }
34     return NULL;
35 }
36
37 int main() {
38     pthread_t t1, t2;
39     pthread_create(&t1, NULL, proceso1, NULL);
40     pthread_create(&t2, NULL, proceso2, NULL);
41     pthread_join(t1, NULL);
42     pthread_join(t2, NULL);
43     return 0;
44 }
```

#### Ejecución:



```
leivadiego@Ubuntu-22: ~/lab4
leivadiego@Ubuntu-22:~$ cd lab4
leivadiego@Ubuntu-22:~/lab4$ gcc ejer1.c -o ejer1 -lpthread
leivadiego@Ubuntu-22:~/lab4$ ./ejer1
Proceso 1 ha adquirido el recurso 1
Proceso 2 ha adquirido el recurso 2
```

El programa tiene 2 procesos que intentan acceder al mismo recurso, y por consiguiente se quedan en un deadlock, cada proceso esperando a que el recurso se libere.

### **Preguntas:**

- ***Describe cómo funciona un algoritmo de detección de deadlock y cómo se relaciona con la concurrencia en sistemas operativos.***

Un algoritmo de detección de deadlock identifica situaciones donde un conjunto de procesos está esperando por recursos que están siendo ocupados por otro proceso en el conjunto, sin posibilidad de liberación. Se relaciona con la concurrencia ya que los deadlocks típicamente ocurren en sistemas donde múltiples procesos compiten por recursos limitados.

- ***¿Qué estrategias podrían implementarse para prevenir deadlocks en sistemas concurrentes más complejos que el ejemplo proporcionado?***

Algunas estrategias incluyen:

- a. Prevención de Hold and Wait: Asegurarse de que los procesos soliciten todos los recursos al mismo tiempo, evitando que retengan unos mientras esperan por otros.
- b. Asignación de recursos en orden: Establecer un orden lineal de todos los recursos y requerir que cada proceso los solicite en este orden, evitando ciclos en la espera de recursos.
- c. Detección y recuperación: Permitir el deadlock pero detectarlo y recuperarse a través de la liberación de recursos o la finalización de procesos.

- ***Explica cómo se podría modificar el código para introducir una situación de interbloqueo más sutil que no sea tan evidente como la inversión de recursos.***

Se podría introducir un tercer proceso o más recursos, complicando la interdependencia. También, se podría hacer que el bloqueo sea menos obvio, quizás introduciendo condiciones adicionales para la adquisición de recursos o variando los tiempos de espera.

- ***¿Qué métodos de detección y resolución de deadlocks conoces y cómo se aplican en sistemas operativos modernos?***

Algunos métodos incluyen:

- a) Detección de ciclos en grafos de asignación de recursos: Si hay un ciclo en el grafo, existe un deadlock.
- b) Matriz de asignación y solicitud de recursos: Comparar las matrices para identificar si los procesos están en un estado de espera que no puede ser resuelto.
- c) Algoritmo del banquero: Es una estrategia preventiva que evalúa si los estados futuros después de una asignación de recursos conducirán a un estado seguro o no.

## Ejercicio 2

### Programa:

```
1 #include <pthread.h>
2 #include <semaphore.h>
3 #include <stdio.h>
4 #include <unistd.h>
5
6 #define NUM_FILOSOFOS 5
7 #define VECES_QUE_COMEN 10
8
9 sem_t tenedores[NUM_FILOSOFOS];
10 int contador_comidas[NUM_FILOSOFOS];
11
12 void* filosofo(void* num) {
13     int* i = (int*)num;
14     while (contador_comidas[*i] < VECES_QUE_COMEN) {
15         printf("Filósofo %d está pensando.\n", *i);
16         sleep(1);
17
18         sem_wait(&tenedores[*i]);
19         sem_wait(&tenedores[( *i + 1) % NUM_FILOSOFOS]);
20
21         contador_comidas[*i]++;
22         printf("Filósofo %d está comiendo por %dª vez.\n", *i, contador_comidas[*i]);
23         sleep(1);
24
25         sem_post(&tenedores[*i]);
26         sem_post(&tenedores[( *i + 1) % NUM_FILOSOFOS]);
27
28         sleep(1); // Añadido para dar tiempo de pensar
29     }
30     printf("Filósofo %d ha terminado de comer.\n", *i);
31     return NULL;
32 }
33
34 int main() {
35     int i;
36     pthread_t thread[NUM_FILOSOFOS];
37     int filosofos[NUM_FILOSOFOS];
38
39     for (i = 0; i < NUM_FILOSOFOS; i++) {
40         sem_init(&tenedores[i], 0, 1);
41         contador_comidas[i] = 0;
42     }
43
44     for (i = 0; i < NUM_FILOSOFOS; i++) {
45         filosofos[i] = i;
46         pthread_create(&thread[i], NULL, filosofo, (void*)&filosofos[i]);
47     }
48
49     for (i = 0; i < NUM_FILOSOFOS; i++) {
50         pthread_join(thread[i], NULL);
51     }
52
53     printf("Todos los filósofos han terminado de comer.\n");
54
55     return 0;
56 }
57
```

## Ejecución:

```
leivadiego@Ubuntu-22: ~/lab4
leivadiego@Ubuntu-22:~$ cd lab4
leivadiego@Ubuntu-22:~/lab4$ gcc ejer2.c -o ejer2 -lpthread
leivadiego@Ubuntu-22:~/lab4$ ./ejer2
Filósofo 0 está pensando.
Filósofo 4 está pensando.
Filósofo 1 está pensando.
Filósofo 2 está pensando.
Filósofo 3 está pensando.
Filósofo 2 está comiendo por 1ª vez.
Filósofo 1 está comiendo por 1ª vez.
Filósofo 2 está pensando.
Filósofo 0 está comiendo por 1ª vez.
Filósofo 4 está comiendo por 1ª vez.
Filósofo 1 está pensando.
Filósofo 0 está pensando.
Filósofo 3 está comiendo por 1ª vez.
Filósofo 4 está pensando.
Filósofo 2 está comiendo por 2ª vez.
Filósofo 1 está comiendo por 2ª vez.
Filósofo 2 está comiendo por 10ª vez.
Filósofo 3 está pensando.
Filósofo 1 está comiendo por 10ª vez.
Filósofo 0 está comiendo por 10ª vez.
Filósofo 2 ha terminado de comer.
Filósofo 1 ha terminado de comer.
Filósofo 4 está comiendo por 10ª vez.
Filósofo 0 ha terminado de comer.
Filósofo 3 está comiendo por 10ª vez.
Filósofo 4 ha terminado de comer.
Filósofo 3 ha terminado de comer.
Todos los filósofos han terminado de comer.
leivadiego@Ubuntu-22:~/lab4$
```

El programa hace uso de semáforos para evitar race conditions como un deadlock, permitiendo que todos los filósofos coman.

(NOTA: se definió que los filósofos comieran hasta 10 veces para evitar un programa infinito)

**Preguntas:**

- ***¿Cuáles son las limitaciones del enfoque de solución con semáforos en el problema de los filósofos cenando? ¿Se pueden mejorar estas soluciones?***

Los semáforos pueden ser difíciles de manejar correctamente y propensos a errores como deadlocks y starvation. Además, no proporcionan un alto nivel de abstracción, lo que puede llevar a soluciones más complejas y difíciles de entender.

- ***Describe un escenario en el que la implementación actual podría conducir a un deadlock y propón una solución alternativa para evitarlo.***

Un escenario de deadlock ocurre cuando cada filósofo toma el tenedor a su izquierda y espera por el tenedor a su derecha, creando un ciclo de espera. Una solución es implementar el algoritmo del filósofo zurdo, donde todos los filósofos, excepto uno, toman primero el tenedor a su izquierda y luego el de su derecha, y el filósofo restante hace lo contrario.

- ***¿Qué diferencias clave existen entre la solución con semáforos y otras técnicas de concurrencia, como monitores o variables de condición?***

Los monitores y las variables de condición ofrecen un nivel de abstracción más alto que los semáforos, permitiendo una sincronización más estructurada y fácil de entender. Los monitores encapsulan los recursos compartidos y las variables de condición permiten que los hilos esperen por ciertas condiciones.

- ***¿Cómo se puede garantizar la equidad en la asignación de recursos en el problema de los filósofos cenando?***

Para asegurar la equidad, se puede implementar un algoritmo que siga una política de turno, asegurando que cada filósofo tenga la oportunidad de comer en un orden específico, o utilizar una cola para controlar el acceso a los tenedores.

### Ejercicio 3

Programa:

```
1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NUM_PROCESOS 5
6 #define NUM_RECURSOS 3
7
8 int recursos_disponibles[NUM_RECURSOS];
9 int max[NUM_PROCESOS][NUM_RECURSOS];
10 int asignacion[NUM_PROCESOS][NUM_RECURSOS];
11 int necesidad[NUM_PROCESOS][NUM_RECURSOS];
12
13 bool verificarEstadoSeguro() {
14     int trabajo[NUM_RECURSOS];
15     bool terminado[NUM_PROCESOS];
16     for (int i = 0; i < NUM_RECURSOS; i++) {
17         trabajo[i] = recursos_disponibles[i];
18     }
19     for (int i = 0; i < NUM_PROCESOS; i++) {
20         terminado[i] = false;
21     }
22     for (int k = 0; k < NUM_PROCESOS; k++) {
23         for (int i = 0; i < NUM_PROCESOS; i++) {
24             if (!terminado[i]) {
25                 bool puede_proceder = true;
26                 for (int j = 0; j < NUM_RECURSOS; j++) {
27                     if (necesidad[i][j] > trabajo[j]) {
28                         puede_proceder = false;
29                         break;
30                     }
31                 }
32                 if (puede_proceder) {
33                     for (int j = 0; j < NUM_RECURSOS; j++) {
34                         trabajo[j] += asignacion[i][j];
35                     }
36                     terminado[i] = true;
37                 }
38             }
39         }
40     }
41     for (int i = 0; i < NUM_PROCESOS; i++) {
42         if (!terminado[i]) {
43             return false;
44         }
45     }
46     return true;
47 }
48
```

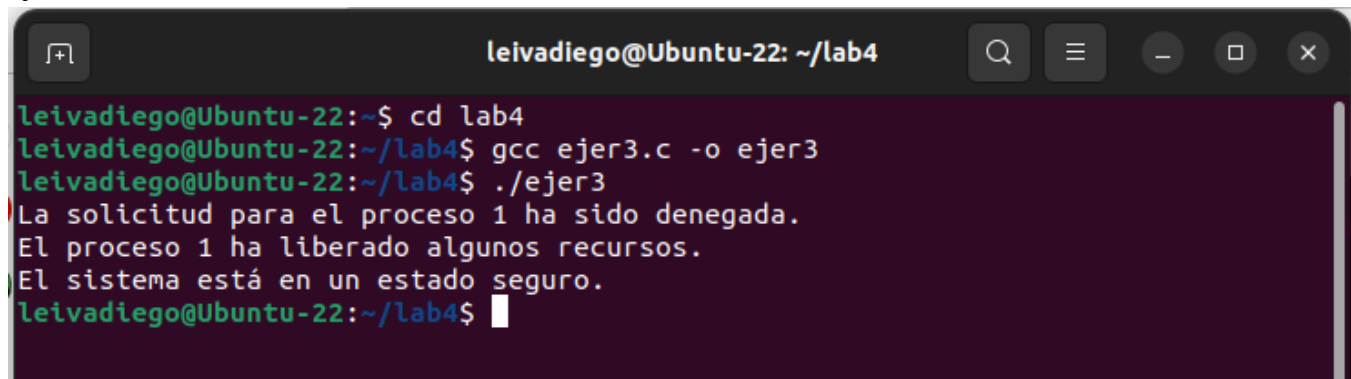
---

```

49 bool solicitarRecursos(int idProceso, int solicitud[]) {
50     for (int i = 0; i < NUM_RECURSOS; i++) {
51         if (solicitud[i] > necesidad[idProceso][i] || solicitud[i] > recursos_disponibles[i]) {
52             return false;
53         }
54     }
55     for (int i = 0; i < NUM_RECURSOS; i++) {
56         recursos_disponibles[i] -= solicitud[i];
57         asignacion[idProceso][i] += solicitud[i];
58         necesidad[idProceso][i] -= solicitud[i];
59     }
60     if (verificarEstadoSeguro()) {
61         return true;
62     } else {
63         for (int i = 0; i < NUM_RECURSOS; i++) {
64             recursos_disponibles[i] += solicitud[i];
65             asignacion[idProceso][i] -= solicitud[i];
66             necesidad[idProceso][i] += solicitud[i];
67         }
68         return false;
69     }
70 }
71
72 void liberarRecursos(int idProceso, int liberacion[]) {
73     for (int i = 0; i < NUM_RECURSOS; i++) {
74         recursos_disponibles[i] += liberacion[i];
75         asignacion[idProceso][i] -= liberacion[i];
76         necesidad[idProceso][i] += liberacion[i];
77     }
78 }
79
80 int main() {
81     for (int i = 0; i < NUM_PROCESOS; i++) {
82         for (int j = 0; j < NUM_RECURSOS; j++) {
83             necesidad[i][j] = max[i][j] - asignacion[i][j];
84         }
85     }
86     int solicitud[NUM_RECURSOS] = {1, 0, 2};
87     if (solicitarRecursos(1, solicitud)) {
88         printf("La solicitud para el proceso 1 ha sido concedida.\n");
89     } else {
90         printf("La solicitud para el proceso 1 ha sido denegada.\n");
91     }
92     int liberacion[NUM_RECURSOS] = {0, 1, 1};
93     liberarRecursos(1, liberacion);
94     printf("El proceso 1 ha liberado algunos recursos.\n");
95     if (verificarEstadoSeguro()) {
96         printf("El sistema está en un estado seguro.\n");
97     } else {
98         printf("El sistema no está en un estado seguro.\n");
99     }
100     return 0;
101 }

```

### Ejecución:

A terminal window titled 'leivadiego@Ubuntu-22: ~/lab4' with standard window controls. The terminal shows the following commands and output:

```
leivadiego@Ubuntu-22:~$ cd lab4
leivadiego@Ubuntu-22:~/lab4$ gcc ejer3.c -o ejer3
leivadiego@Ubuntu-22:~/lab4$ ./ejer3
La solicitud para el proceso 1 ha sido denegada.
El proceso 1 ha liberado algunos recursos.
El sistema está en un estado seguro.
leivadiego@Ubuntu-22:~/lab4$
```

El programa implementa una versión simple del algoritmo del banquero para evitar deadlocks en la asignación de recursos. En este caso se intenta conceder una solicitud de recursos al proceso 1. En este caso fue denegada, mostrando que esos recursos podrían llevar a un estado inseguro, por lo que después de ser denegado, este proceso 1 libera recursos y se vuelve a verificar el estado del sistema.

### Preguntas:

• ***¿Qué requisitos deben cumplirse para que un sistema esté en un estado seguro según el algoritmo del banquero? ¿Por qué es importante?***

- Disponibilidad de recursos: Debe haber suficientes recursos disponibles para satisfacer la demanda de al menos uno de los procesos pendientes.
- Orden de asignación: Debe existir un orden secuencial de procesos tal que cada proceso pueda obtener los recursos necesarios para completar su tarea y, después de terminar, liberar todos sus recursos, uno por uno, para el siguiente proceso en la secuencia.
- Necesidades no excedentes: Un proceso solo puede reclamar una cantidad de recursos que no exceda la cantidad declarada como máxima en el principio.
- Recursos asignables: En cualquier estado dado, los recursos que un proceso espera recibir, sumados a los que ya posee, no deben exceder el número total de recursos del sistema.

Es importante porque garantiza que siempre habrá un camino para que todos los procesos se completen sin caer en un deadlock, lo que significa que el sistema puede continuar operando eficientemente sin interrupciones.



• ***Explica cómo se pueden detectar ciclos de espera en un grafo de asignación de recursos y cómo esto se relaciona con la posibilidad de un deadlock.***

- Los ciclos de espera en un grafo de asignación de recursos ocurren cuando hay una cadena cerrada de procesos donde cada proceso posee al menos un recurso que el siguiente proceso en la cadena está esperando.
- Para detectar ciclos de espera, se puede utilizar un algoritmo de búsqueda en grafos como el de búsqueda en profundidad (DFS) o búsqueda en anchura (BFS). Si durante la búsqueda se vuelve a un nodo ya visitado, se ha encontrado un ciclo.
- La presencia de un ciclo en el grafo de asignación de recursos es una condición necesaria para un deadlock. Significa que hay un conjunto de procesos que están esperando indefinidamente por recursos que nunca se liberarán.

• ***¿Cómo afectaría una implementación incorrecta del algoritmo del banquero al sistema? Proporciona ejemplos concretos.***

- Deadlocks: Una implementación incorrecta podría no detectar adecuadamente un estado inseguro, lo que podría llevar a deadlocks donde los procesos se quedan esperando por recursos que nunca se liberan.
- Starvation: Si el algoritmo no se aplica de manera justa, algunos procesos podrían sufrir de starvation, donde nunca reciben los recursos necesarios para ejecutarse.
- Rendimiento: Podría haber un impacto negativo en el rendimiento si el algoritmo es demasiado conservador y limita la concurrencia más de lo necesario, o si es demasiado permisivo y asigna recursos de una manera que aumenta el riesgo de deadlock.
- Ejemplo: Si el sistema no retrocede cuando un estado inseguro es detectado tras una solicitud de recursos, podrían comenzar a suceder deadlocks.

• ***¿Cuál es la complejidad computacional del algoritmo del banquero y cómo podría impactar en sistemas con un gran número de procesos y recursos?***

- La complejidad computacional del algoritmo del banquero es  $O(n^2 * m)$  para cada solicitud de recursos, donde  $n$  es el número de procesos y  $m$  es el número de tipos de recursos.
- En sistemas con un gran número de procesos y recursos, esta complejidad puede llevar a que el algoritmo sea bastante lento, lo que podría no ser práctico en sistemas de tiempo real o en situaciones donde las solicitudes de recursos y las liberaciones ocurren con mucha frecuencia.
- Esto podría conducir a un cuello de botella donde la gestión de recursos se convierte en el limitante principal de la performance del sistema.

**Repositorio de GITHUB:**

<https://github.com/LeivaDiego/Lab4-Sistos.git>