

**Centro de Investigación en Computo**

**TAREA 2. DEMOSTRACIÓN DE  
ALGORITMO DE FILÓSOFOS  
COMENSALES**

*Materia: Sistemas Operativos*

22 de diciembre de 2025

## 1. Objetivo

Del programa implementado con pipes y semáforos de filósofos comensales, realizar la demostración de que estos no entran en *deadlock*, cada filósofo come al menos una vez y por último que dos filósofos diferentes no pueden utilizar el mismo recurso al mismo tiempo.

## 2. Filósofos comensales con *pipes*

A continuación, se presenta el código implementado con *pipes*

```
1  /*
2   * Programa que utiliza pipes para resolver el problema de
3   * los filosofos
4   * Forma de compilar: gcc filo_pipes.c -o filo_v1
5   * Ejecuci n: ./filo_v1
6   */
7
8 #define _GNU_SOURCE
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <unistd.h>
12 #include <fcntl.h>
13 #include <errno.h>
14
15 #define N 4
16 int fork_pipes[N][2];
17
18 void hijo(int id);
19 void pensar(int id);
20 void tomar(int id, int tenedor);
21 void comer(int id);
22 void soltar(int id, int tenedor, int tenedor2);
23
24 int main()
25 {
26     setbuf(stdout, NULL);
27
28     int i;
29     pid_t pid;
30     unsigned char disponible = 1;
31
32     for (i = 0; i < N; i++) {
33         if (pipe(fork_pipes[i]) == -1) {
34             perror("pipe");
35             exit(1);
36         }
37         write(fork_pipes[i][1], &disponible, 1);
```

```

38     }
39
40     for (i = 0; i < N; i++) {
41         pid = fork();
42         if (pid == 0) {
43             hijo(i);
44             exit(0);
45         }
46     }
47
48     while (1);
49 }
50
51 void hijo(int id)
{
52     int der = id;
53     int izq = (id + N - 1) % N;
54     int primero = izq < der ? izq : der;
55     int segundo = izq < der ? der : izq;
56
57     while (1) {
58         pensar(id);
59         tomar(id, primero);
60         tomar(id, segundo);
61         comer(id);
62         soltar(id, primero, segundo);
63     }
64 }
65
66
67 void pensar(int id)
{
68     printf("Filosofo %d: pensando...\n", id);
69     sleep(2);
70 }
71
72
73 void tomar(int id, int tenedor)
{
74     unsigned char disponible;
75     read(fork_pipes[tenedor][0], &disponible, 1);
76     printf("Filosofo %d tom el tenedor %d\n", id, tenedor)
77     ;
78 }
79
80 void comer(int id)
{
81     printf("Filosofo %d: COMIENDO\n", id);
82     sleep(2);
83 }
84
85 void soltar(int id, int tenedor, int tenedor2)

```

```

87 {
88     unsigned char disponible = 1;
89     write(fork_pipes[tenedor][1], &disponible, 1);
90     write(fork_pipes[tenedor2][1], &disponible, 1);
91     printf("Filosofo %d solt los tenedores %d y %d\n", id,
92           tenedor, tenedor2);
}

```

Código 1: Implementación del problema de los filósofos usando pipes

## 2.1. Demostración por contradicción de ausencia de inanición

A continuación, se demuestra la ausencia de inanición; es decir, que cada uno de los filósofos eventualmente logra comer.

**Suposición inicial** Suponemos que existe al menos un filósofo  $F_i$  que nunca logra comer durante la ejecución del programa.

**Desarrollo** Si  $F_i$  nunca come, entonces permanece bloqueado de manera indefinida esperando la obtención de al menos uno de los tenedores requeridos para comer (sección crítica). Por lo que:

- El filósofo  $F_i$  se encuentra bloqueado ejecutando una operación `read()` sobre el pipe que representa un tenedor.
- Dicho tenedor es liberado un número infinito de veces por otros filósofos, ya que estos entran y salen repetidamente de la sección crítica.

**Contradicción** Cada liberación del tenedor se realiza mediante la operación `write()`, la cual despierta a uno de los procesos bloqueados están esperando dicho recurso. Dado que los pipes mantienen un orden FIFO (*First in First Out*), el filósofo  $F_i$ , al permanecer bloqueado de forma continua, debe eventualmente ser seleccionado y despertado para adquirir el tenedor.

**Conclusión** Esto contradice la suposición inicial de que  $F_i$  nunca logra comer. Por lo que se demuestra que la suposición inicial es falsa. Y todos los filósofos logran comer.

## 2.2. Demostración por contradicción de ausencia de *dead-lock*

A continuación, se demuestra la ausencia de *deadlock*; es decir, que no puede alcanzarse un estado en el que todos los filósofos se encuentren bloqueados indefinidamente por la espera de recursos, en este caso, el tenedor.

**Suposición inicial** Suponemos que el sistema alcanza un estado de *deadlock*. En dicho estado, cada filósofo mantiene un tenedor y permanece bloqueado, esperando adquirir el segundo.

**Desarrollo** Cada filósofo adquiere los tenedores siguiendo un orden creciente, por lo que todos los filósofos primero solicitan el tenedor con índice menor y posteriormente el tenedor con índice mayor. Por lo que:

- Cada filósofo posee un tenedor de índice menor.
- Cada filósofo espera un tenedor de índice mayor.

Esto induce una relación estrictamente creciente entre los índices de los tenedores retenidos y solicitados.

**Contradicción** Sea  $T_k$  el tenedor con mayor índice en el sistema. Por construcción del algoritmo, ningún filósofo puede estar esperando un tenedor con índice mayor que  $T_k$ . Sin embargo, en un estado de *deadlock*, el filósofo que posee  $T_k$  debería estar esperando otro tenedor con índice superior, lo cual es imposible.

**Conclusión** Esto contradice la suposición inicial de que el sistema entra en un estado de *deadlock*, comprobando que la adquisición ordenada de recursos rompe la condición de esperas circular.

### 2.3. Demostración por contradicción de exclusión mutua

A continuación, se demuestra la propiedad de exclusión mutua; es decir, que ningún tenedor puede ser utilizado simultáneamente por más de un filósofo.

**Suposición inicial** Suponemos que existen dos filósofos distintos  $P_i$  y  $P_j$  que poseen simultáneamente el mismo tenedor  $T_k$ .

**Desarrollo** Cada tenedor es un *pipe* de un *byte*. La presencia del byte representa que el tenedor se encuentra disponible, mientras que la ausencia del mismo indica que el tenedor está ocupado.

**Contradicción** Ambos filósofos tienen el tenedor  $T_k$ , por lo que tuvieron que haber ejecutado con éxito una operación `read()` sobre el pipe asociado a dicho tenedor.

Sin embargo, la operación `read()` es atómica y bloqueante. Dado que el *pipe* es de un *byte*, solo una de las operaciones `read()` puede consumir dicho byte. Una vez consumido, el pipe queda vacío y cualquier intento posterior de lectura se bloquea hasta que el recurso sea liberado mediante una operación `write()`.

Por lo tanto, es imposible que dos procesos distintos consuman simultáneamente el mismo byte del pipe.

**Conclusión** Esto contradice la suposición inicial , por lo que ningún tenedor puede ser poseído simultáneamente por más de un filósofo.

### 3. Filósofos comensales con semáforos

A continuación, se presenta el código implementado con semáforos.

```
1  /*
2   Programa que utiliza semaforos para resolver el problema
3   de los filosofos
4   Forma de compilar: gcc filo_semaforos.c -o filo_v2
5   Ejecucion: ./filo_v2
6 */
7
8 #define _GNU_SOURCE
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <unistd.h>
12 #include <pthread.h>
13 #include <semaphore.h>
14
15 #define N 4
16
17 sem_t tenedores[N];
18 pthread_t filosofos[N];
19
20 void *filosofo(void *arg);
21 void pensar(int id);
22 void tomar(int id, int tenedor);
23 void comer(int id);
24 void soltar(int id, int tenedor1, int tenedor2);
25
26 int main(void)
27 {
28     int i;
29
30     for (i = 0; i < N; i++) {
31         sem_init(&tenedores[i], 0, 1);
32     }
33
34     for (i = 0; i < N; i++) {
35         int *id = malloc(sizeof(int));
36         if (!id) {
37             perror("malloc");
38             exit(EXIT_FAILURE);
39         }
40         *id = i;
41         pthread_create(&filosofos[i], NULL, filosofo, id);
42     }
43 }
```

```

42
43     for (i = 0; i < N; i++) {
44         pthread_join(filosofos[i], NULL);
45     }
46
47     return 0;
48 }
49
50 void *filosofo(void *arg)
51 {
52     int id = *(int *)arg;
53     free(arg);
54
55     int der = id;
56     int izq = (id + (N - 1)) % N;
57
58     int primero = izq < der ? izq : der;
59     int segundo = izq < der ? der : izq;
60
61     while (1) {
62         pensar(id);
63         tomar(id, primero);
64         tomar(id, segundo);
65         comer(id);
66         soltar(id, primero, segundo);
67     }
68
69     return NULL;
70 }
71
72 void pensar(int id)
73 {
74     printf("Filosofo %d: pensando...\n", id);
75     sleep(2);
76 }
77
78 void tomar(int id, int tenedor)
79 {
80     sem_wait(&tenedores[tenedor]);
81     printf("Filosofo %d tomo el tenedor %d\n", id, tenedor);
82 }
83
84 void comer(int id)
85 {
86     printf("Filosofo %d: COMIENDO\n", id);
87     sleep(2);
88 }
89
90 void soltar(int id, int tenedor1, int tenedor2)
91 {

```

```

92     sem_post(&tenedores[tenedor1]);
93     sem_post(&tenedores[tenedor2]);
94     printf("Filosofo %d solto los tenedores %d y %d\n",
95            id, tenedor1, tenedor2);
96 }

```

Código 2: Implementación del problema de los filósofos comensales usando semáforos

### 3.1. Demostración por contradicción de ausencia de inanición

A continuación, se demuestra la ausencia de inanición; es decir, que cada uno de los filósofos eventualmente logra comer.

**Suposición inicial** Suponemos que existe al menos un filósofo  $F_i$  que nunca logra comer durante la ejecución del programa.

**Desarrollo** Cada tenedor es un semáforo binario inicializado en uno con las operaciones de `sem_wait()` la cual, es bloqueante y la operación `sem_post()` la cual libera el recurso y despierta a un hilo bloqueado. Por lo que:

- El filósofo  $F_i$  nunca come, permanece bloqueado indefinidamente con la operación `sem_wait()` sobre alguno de los tenedores.
- Dichos tenedores son liberados un número infinito de veces por otros filósofos, ya que estos entran y salen repetidamente de la sección crítica.

**Contradicción** Cada liberación de un tenedor se realiza mediante la operación `sem_post()`, la cual incrementa el valor del semáforo y despierta a uno de los hilos bloqueados esperando dicho recurso. Dado que los tenedores se liberan de forma finita y los filósofos adquieren los recursos siguiendo un orden total determinado por sus índices, el filósofo  $F_i$ , al permanecer bloqueado de forma continua, debe eventualmente adquirir el recurso y entrar a la sección crítica.

**Conclusion** Esto contradice la suposición inicial de que el filósofo  $F_i$  nunca logra comer. Por lo tanto, se concluye que la suposición inicial es falsa y que todos los filósofos eventualmente logran comer.

### 3.2. Demostración por contradicción de ausencia de *dead-lock*

A continuación, se demuestra la ausencia de *deadlock*; es decir, que no puede alcanzarse un estado en el que todos los filósofos se encuentren bloqueados indefinidamente por la espera de recursos, en este caso, el tenedor.

**Suposición inicial** Suponemos que el sistema alcanza un estado de *deadlock*. En dicho estado, cada filósofo mantiene un tenedor y permanece bloqueado, esperando adquirir el segundo.

**Desarrollo** Cada filósofo necesita obtener dos tenedores(semáforos) para poder comer, esto se realiza tomando el tenedor con el menor índice y posteriormente el tenedor con el mayor índice.

Si existiera un *deadlock*, entonces debería existir una espera circular entre los filósofos, donde cada filósofo espera un tenedor que está siendo retenido por otro filósofo en el ciclo.

**Contradicción** La existencia de una espera circular implicaría que algún filósofo intentara adquirir primero un tenedor de mayor índice y luego uno de menor índice. Sin embargo, esto es imposible, ya que el algoritmo impone estrictamente que todos los filósofos adquieran primero el tenedor de menor índice y luego el de mayor.

**Conclusión** Esto contradice la suposición inicial de que el sistema entra en un estado de *deadlock*, comprobando que la adquisición ordenada de recursos rompe la condición de espera circular.

### 3.3. Demostración por contradicción de la propiedad de exclusión mutua

A continuación, se demuestra la propiedad de exclusión mutua; es decir, que ningún tenedor puede ser utilizado simultáneamente por más de un filósofo.

**Suposición inicial** Suponemos que existen dos filósofos distintos  $P_i$  y  $P_j$  que poseen simultáneamente el mismo tenedor  $T_k$ .

**Desarrollo** Cada tenedor está representado por un semáforo binario inicializado en uno. Para que un filósofo pueda tomar un tenedor, debe ejecutar la operación `sem_wait()` sobre el semáforo correspondiente.

La operación de `sem_wait()` garantiza que, si el valor del semáforo es cero, el proceso que invoca la operación queda bloqueado hasta que otro proceso libere el recurso mediante `sem_post()`.

**Contradicción** Para que dos filósofos  $P_i$  y  $P_j$  utilicen simultáneamente el mismo tenedor  $T_k$ , ambos tendrían que haber ejecutado exitosamente `sem_wait()` sobre el mismo semáforo de  $T_k$  sin que se haya ejecutado una operación `sem_post()` intermedia. Sin embargo, esto es imposible, ya que el semáforo binario solo permite que un único filósofo cambie su valor de uno a cero, bloqueando a cualquier otro filósofo que intente acceder al mismo recurso.

**Conclusión** Esto contradice la suposición inicial , por lo que ningún tenedor puede ser poseído simultáneamente por más de un filósofo.