



**LaSalle**  
**Universidad**

**INGENIERÍA DE SOFTWARE**  
**SISTEMAS OPERATIVOS**

**Filósofos Comensales**  
**Elvis David Minaya Mamani**  
**10/05/2021**

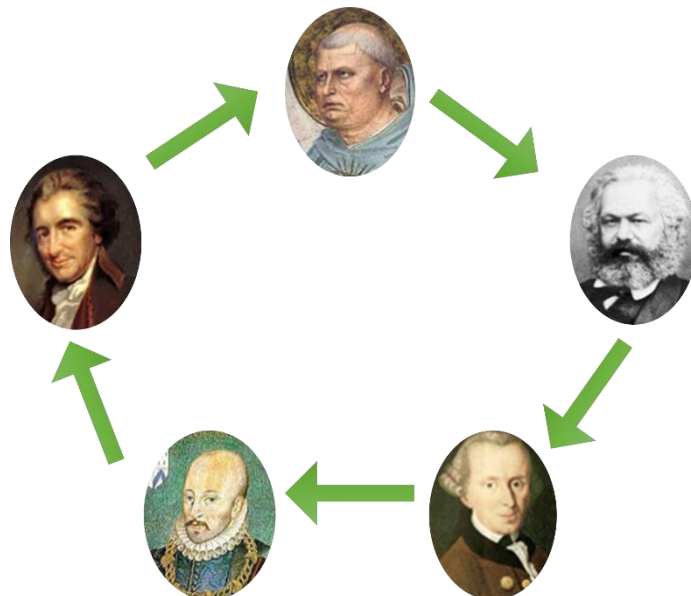
## FILÓSOFOS COMENSALES

### 1. Planteamiento del problema.

- Introducción: Alrededor de una mesa se sientan 5 filósofos, los cuales pasan su vida pensando y cenando. Para cada filósofo hay un plato de fideos, y un tenedor a su izquierda, para como los fideos es necesario ocupar dos tenedores y cada filósofo solo puede tomar el que está a su izquierda y derecha. Si cualquier filósofo toma un tenedor y el otro está ocupado este se quedara esperando con el tenedor en la mano, hasta que pueda coger el otro tenedor para poder empezar a comer.
- **Restricciones:**
  - a) Si dos filósofos juntos toman el mismo tenedor se produce una condición de carrera ya que ambos competirán por el mismo tenedor y uno de ellos tendrá que quedarse sin comer.
  - b) Si todos los filósofos toman el mismo tenedor al mismo tiempo, entonces todos se quedaran esperando eternamente, ya que nadie liberara el tenedor que les falta, se interpreta de tal forma que todos los filósofos, morirán de hambre al mismo tiempo se produce un interbloqueo.

### 2. Solución por turno cíclico.

- **Cíclico:** Proceso repetitivo, presentamos a los 5 filósofos y le asignamos un tenedor a cada uno, también cada uno tendrá un tiempo determinado para poder comer, cada vez que uno termine el siguiente continuara y así sucesivamente.



### 3. SOLUCION POR COLAS.

- **Colas:** Como podemos observar tenemos a los 5 filósofos en la mesa, el filósofo número 1 empieza a comer ya que tiene dos tenedores al igual que el filósofo número 3, el filósofo 5 se pone en turno de espera porque solo tiene un tenedor, y los filósofos restantes están pensando. Y así sucesivamente.



### 4. Solución y ejecución en lenguaje C.

- **Algoritmo 1**
  - **Estructura de filósofo.**

Tenemos la estructura filosofo que contiene un nombre, asignamos el \* para el mejor uso de memoria, cantidad de comida, por defecto pusimos 6 (modificable) y dos atributos de tipo tenedor.

```
struct filosofo{  
    char * nombre;  
    int cantComida;  
    struct tenedor * ten1;  
    struct tenedor * ten2;  
};
```

- **Estructura tenedor.**

La estructura tenedor contiene un estado, que será una bandera a él filósofo, cuando estado==1 libre, estado==2 ocupado.

```
struct tenedor{
    int estado;
};
```

- **Función comer.**

Creamos la función comer con el objetivo de repartir de manera equitativa los tenedores, de manera cíclica podemos asignar y rotar los tenedores entre todos los filósofos para que todos puedan comer satisfactoriamente.

```
void * comer( void * h1){
    struct filosofo * fil;
    fil = (struct filosofo*) h1;
    printf("%s %s \n", fil->nombre, "esta pensando");
    while(fil->cantComida > 0){
        if(fil->ten1->estado == 0 && fil->ten2->estado == 0){
            printf("%s %s \n", fil->nombre, "tiene hambre");
            fil->ten1->estado = fil->ten2->estado = 1;
            printf("%s %s \n", fil->nombre, "agarro los 2 tenedores");
            while(fil->cantComida > 0){
                fil->cantComida--;
                printf("%s %s \n", fil->nombre, "esta comiendo");
            }
        }
        else{
            printf("%s %s \n", fil->nombre, "no puede comer");
        }
    }
    fil->ten1->estado = fil->ten2->estado = 0;
    printf("%s %s \n", fil->nombre, "termino de comer");
}
```

- **CUERPO (MAIN).**

Creamos los n filósofos deseados y asignamos un tenedor cada uno (struct filósofo, struct tenedor) asignamos a todos los tenedores como estado=0 porque empiezan siendo libres.

Enviamos a nuestra función comer los n filósofos y ahí la magia del algoritmo.

```
int main() {
    pthread_t thread1, thread2, thread3, thread4, thread5;
    struct tenedor * ten1 = (struct tenedor *) malloc (sizeof(struct tenedor));
    struct tenedor * ten2 = (struct tenedor *) malloc (sizeof(struct tenedor));
    struct tenedor * ten3 = (struct tenedor *) malloc (sizeof(struct tenedor));
    struct tenedor * ten4 = (struct tenedor *) malloc (sizeof(struct tenedor));
    struct tenedor * ten5 = (struct tenedor *) malloc (sizeof(struct tenedor));
    struct filosofo * fil1 = (struct filosofo *) malloc (sizeof(struct filosofo));
    struct filosofo * fil2 = (struct filosofo *) malloc (sizeof(struct filosofo));
    struct filosofo * fil3 = (struct filosofo *) malloc (sizeof(struct filosofo));
    struct filosofo * fil4 = (struct filosofo *) malloc (sizeof(struct filosofo));
    struct filosofo * fil5 = (struct filosofo *) malloc (sizeof(struct filosofo));

    ten1->estado = ten2->estado = ten3->estado = ten4->estado = ten5->estado = 0;
    fil1->nombre = "Fujimori";
    fil1->cantComida = comida;
    fil1->ten1 = ten1;
    fil1->ten2 = ten2;
    fil2->nombre = "Castillo";
    fil2->cantComida = comida;
    fil2->ten1 = ten2;
    fil2->ten2 = ten3;
    fil3->nombre = "Acuna";
    fil3->cantComida = comida;
    fil3->ten1 = ten3;
    fil3->ten2 = ten4;
    fil4->nombre = "Ollanta";
    fil4->cantComida = comida;
    fil4->ten1 = ten4;
    fil4->ten2 = ten5;
    fil5->nombre = "Llica";
    fil5->cantComida = comida;
    fil5->ten1 = ten5;
    fil5->ten2 = ten1;
    int iret1, iret2, iret3, iret4, iret5;
    iret1 = pthread_create( &thread1, NULL, comer, (void*) fil1);
    iret2 = pthread_create( &thread2, NULL, comer, (void*) fil2);
    iret3 = pthread_create( &thread3, NULL, comer, (void*) fil3);
    iret4 = pthread_create( &thread4, NULL, comer, (void*) fil4);
    iret5 = pthread_create( &thread5, NULL, comer, (void*) fil5);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    pthread_join( thread3, NULL);
    pthread_join( thread4, NULL);
    pthread_join( thread5, NULL);
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    printf("Thread 3 returns: %d\n",iret3);
    printf("Thread 4 returns: %d\n",iret4);
    printf("Thread 5 returns: %d\n",iret5);
    return 0;
}
```

- **LIBRERIAS USADAS DE LENGUAJE C**

- 

`#include <stdio.h>` -> Standard input-output header" (cabecera estándar E/S). Flujo de de datos, consola.

`#include <stdlib.h>` -> Standard library o biblioteca estándar). Es el archivo de cabecera de la biblioteca estándar.

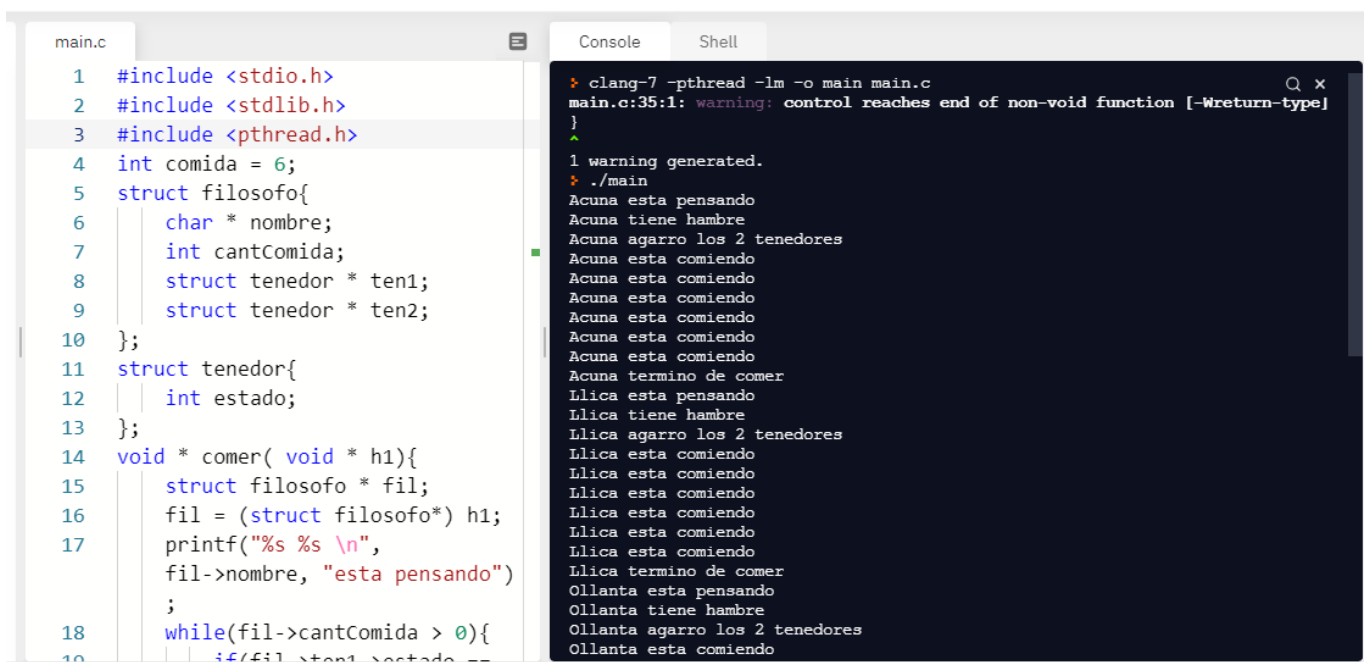
`#include <pthread.h>` -> Languages provide the POSIX thread(**pthread**) standard API(Application program Interface).

- **IMÁGENES DE COMPILACION.**

COMPILADOR REPLIT C

IDE DEV C++

SUBLIME TEXT WITH MINGW

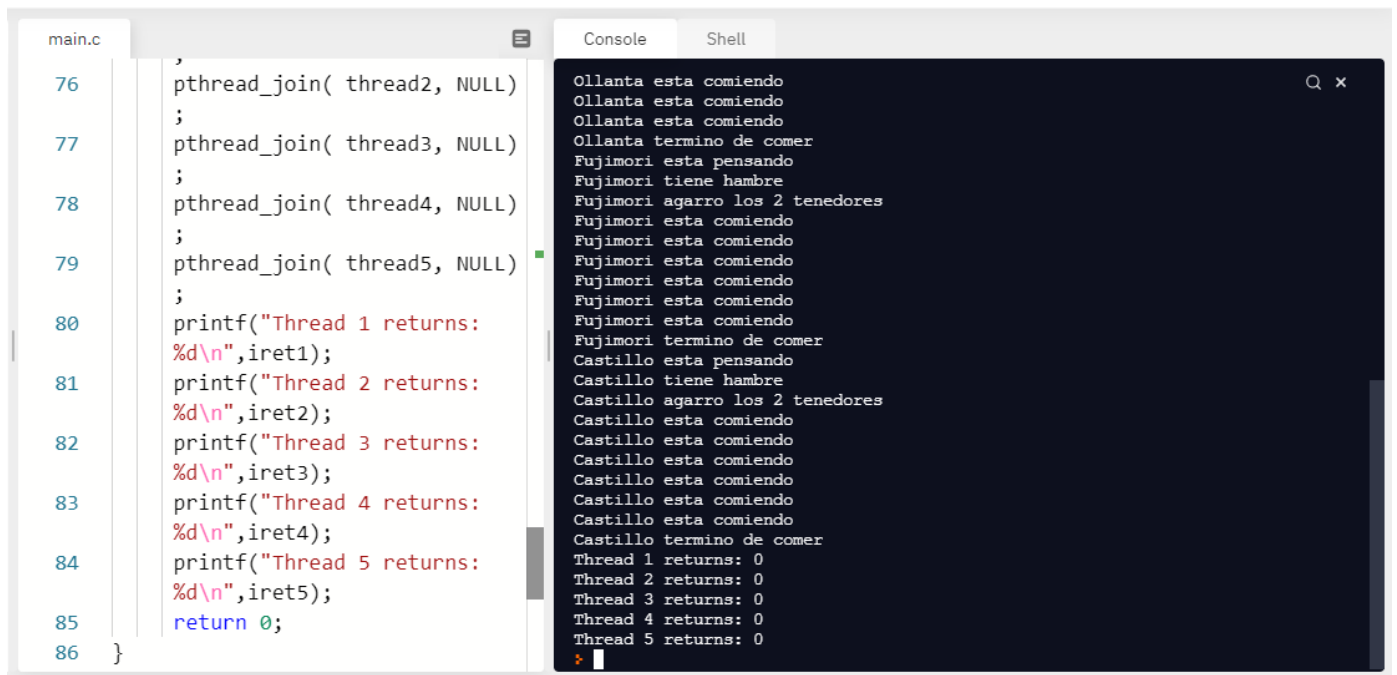


The screenshot shows a code editor with a file named `main.c`. The code defines a philosopher structure and a function to simulate a dining philosophers problem. The console output shows the compilation command `clang-7 -pthread -lm -o main main.c` and the execution output, which displays the actions of three philosophers: Acuna, Illica, and Ollanta, including thinking, getting forks, eating, and finishing.

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 int comida = 6;
5 struct filosofo{
6     char * nombre;
7     int cantComida;
8     struct tenedor * ten1;
9     struct tenedor * ten2;
10 };
11 struct tenedor{
12     int estado;
13 };
14 void * comer( void * h1){
15     struct filosofo * fil;
16     fil = (struct filosofo*) h1;
17     printf("%s %s \n",
18         fil->nombre, "esta pensando")
19     ;
20     while(fil->cantComida > 0){
21         if(fil->ten1->estado ==
```

```
clang-7 -pthread -lm -o main main.c
main.c:35:1: warning: control reaches end of non-void function [-Wreturn-type]
}
^
1 warning generated.
./main
Acuna esta pensando
Acuna tiene hambre
Acuna agarro los 2 tenedores
Acuna esta comiendo
Acuna esta comiendo
Acuna esta comiendo
Acuna esta comiendo
Acuna esta comiendo
Acuna esta comiendo
Acuna termino de comer
Illica esta pensando
Illica tiene hambre
Illica agarro los 2 tenedores
Illica esta comiendo
Illica esta comiendo
Illica esta comiendo
Illica esta comiendo
Illica esta comiendo
Illica termino de comer
Ollanta esta pensando
Ollanta tiene hambre
Ollanta agarro los 2 tenedores
Ollanta esta comiendo
```



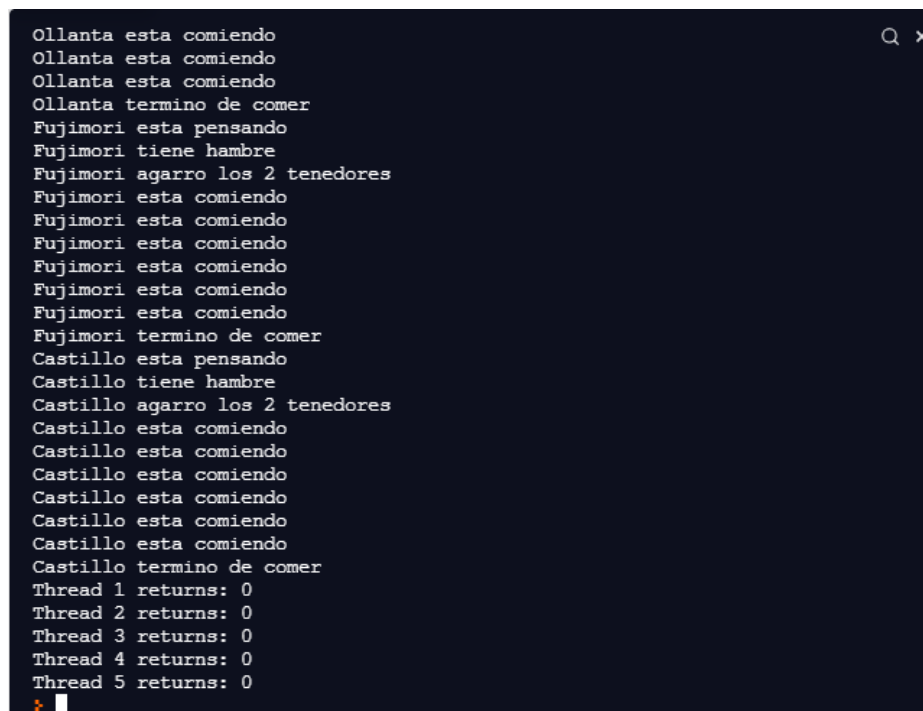


The image shows a code editor with a file named `main.c` and a console window. The code in `main.c` is as follows:

```
76     ,  
    pthread_join( thread2, NULL)  
77     ;  
    pthread_join( thread3, NULL)  
78     ;  
    pthread_join( thread4, NULL)  
79     ;  
    pthread_join( thread5, NULL)  
80     ;  
    printf("Thread 1 returns:  
81     %d\n",iret1);  
    printf("Thread 2 returns:  
82     %d\n",iret2);  
    printf("Thread 3 returns:  
83     %d\n",iret3);  
    printf("Thread 4 returns:  
84     %d\n",iret4);  
    printf("Thread 5 returns:  
85     %d\n",iret5);  
86     return 0;  
}
```

The console output is as follows:

```
Ollanta esta comiendo  
Ollanta esta comiendo  
Ollanta esta comiendo  
Ollanta termino de comer  
Fujimori esta pensando  
Fujimori tiene hambre  
Fujimori agarro los 2 tenedores  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori termino de comer  
Castillo esta pensando  
Castillo tiene hambre  
Castillo agarro los 2 tenedores  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo termino de comer  
Thread 1 returns: 0  
Thread 2 returns: 0  
Thread 3 returns: 0  
Thread 4 returns: 0  
Thread 5 returns: 0
```



The console output is as follows:

```
Ollanta esta comiendo  
Ollanta esta comiendo  
Ollanta esta comiendo  
Ollanta termino de comer  
Fujimori esta pensando  
Fujimori tiene hambre  
Fujimori agarro los 2 tenedores  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori esta comiendo  
Fujimori termino de comer  
Castillo esta pensando  
Castillo tiene hambre  
Castillo agarro los 2 tenedores  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo esta comiendo  
Castillo termino de comer  
Thread 1 returns: 0  
Thread 2 returns: 0  
Thread 3 returns: 0  
Thread 4 returns: 0  
Thread 5 returns: 0
```

## 5. PUNTO DE VISTA, PERSONAL.

- Después de haber investigado y analizado los diferentes algoritmos he llegado a la conclusión de que el tiempo va a depender mucho de la cantidad de datos a procesar y con cantidad me refiero a la  $n$  filósofos y  $n$  tenedores en juego, como también el volumen de comida.
- Como dato adicional acotar que este fue un ejercicio planteando en la RPC de agosto del 2012 planteándolo como un reto de programación actual (hoy) el cual se busca mejorar e crear nuevos algoritmos.
- Por mi parte podría acotar como juego o reto de programación el poder agregar una regla más la cual sería que un filósofo puede robar/coger un tenedor del filósofo de su derecha, siempre y cuando el filósofo de su izquierda no este comiendo. La idea se me ocurrió al recordar una de las pruebas de inteligencia que se aplicaban a una población totalmente pobre para medir sus capacidades de oportunidad, esta idea se me ocurrió en base a la serie 3% en netflix.

## 6. DEADLOCK

Tráfico solo en una dirección.

- Cada sección de un puente puede verse como un recurso.
- Si se produce un interbloqueo, se puede resolver si un automóvil retrocede (recursos y reversión).
- Es posible que sea necesario realizar una copia de seguridad de varios automóviles si se produce un punto muerto.

### Safety Algorithm

1. Initialize work = available  
Initialize finish[i] = false, for  $i = 1, 2, 3, \dots, n$
2. Find an  $i$  such that:  
finish[i] == false and need[i] <= work  
  
If no such  $i$  exists, go to step 4.
3. work = work + allocation[i]  
finish[i] = true  
goto step 2
4. if finish[i] == true for all  $i$ , then the system is in a safe state.



- **Evitar deadlock.**

- Prevención de interbloqueo

Los interbloques se pueden prevenir evitando al menos una de las cuatro condiciones requeridas:

- Exclusión mutua:

Los recursos compartidos, como los archivos de solo lectura, no dan lugar a interbloques.

Desafortunadamente, algunos recursos, como impresoras y unidades de cinta, requieren acceso exclusivo mediante un solo proceso.

- Mantener y esperar

Para evitar esta condición, se debe evitar que los procesos retengan uno o más recursos mientras esperan simultáneamente uno o más. Hay varias posibilidades para esto:

Exija que todos los procesos soliciten todos los recursos a la vez. Esto puede ser un desperdicio de recursos del sistema si un proceso necesita un recurso al principio de su ejecución y no necesita ningún otro recurso hasta mucho más tarde.

Exija que los procesos que contienen recursos deben liberarlos antes de solicitar nuevos recursos y luego volver a adquirir los recursos liberados junto con los nuevos en una única solicitud nueva. Esto puede ser un problema si un proceso ha completado parcialmente una operación utilizando un recurso y luego no logra reasignarlo después de liberarlo.

Cualquiera de los métodos descritos anteriormente puede conducir a la inanición si un proceso requiere uno o más recursos populares.

- Sin preferencia

La preferencia sobre las asignaciones de recursos del proceso puede evitar esta condición de interbloques, cuando sea posible.

Un enfoque es que si un proceso se ve obligado a esperar cuando solicita un nuevo recurso, entonces todos los demás recursos previamente retenidos por este proceso se liberan implícitamente (se apropian), lo que obliga a este proceso a volver a adquirir los recursos antiguos junto con los nuevos recursos en una sola solicitud, similar a la discusión anterior.

Otro enfoque es que cuando se solicita un recurso y no está disponible, entonces el sistema busca ver qué otros procesos tienen actualmente esos recursos y ellos mismos están bloqueados esperando algún otro recurso. Si se encuentra un proceso de este tipo, es posible que algunos de sus recursos se sustituyan y se agreguen a la lista de recursos que está esperando el proceso.

Cualquiera de estos enfoques puede ser aplicable a recursos cuyos estados se guardan y restauran fácilmente, como registros y memoria, pero generalmente no son aplicables a otros dispositivos como impresoras y unidades de cinta.

- Espera circular

Una forma de evitar la espera circular es numerar todos los recursos y exigir que los procesos soliciten recursos solo en orden estrictamente creciente (o decreciente).

En otras palabras, para solicitar el recurso  $R_j$ , un proceso debe liberar primero todo  $R_i$  tal que  $i > j$ .

Un gran desafío en este esquema es determinar el orden relativo de los diferentes recursos

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	0 1 0	7 5 3	3 3 2	7 4 3
$P_1$	2 0 0	3 2 2		1 2 2
$P_2$	3 0 2	9 0 2		6 0 0
$P_3$	2 1 1	2 2 2		0 1 1
$P_4$	0 0 2	4 3 3		4 3 1

- **BIBLIOGRAFIA.**

- <http://darksystem79.blogspot.com/2013/11/filosofos-comensales.html#:~:text=El%20problema%20de%20los%20fil%C3%B3sofos,procesos%20en%20un%20sistema%20operativo.>
- [https://es.wikipedia.org/wiki/Problema\\_de\\_la\\_cena\\_de\\_los\\_fil%C3%B3sofos](https://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_fil%C3%B3sofos)
- <https://www2.infor.uva.es/~cllamas/concurr/pract97/immartin/index.html>
- <https://www.studocu.com/pe/document/universidad-nacional-del-callao/sistemas-neumaticos-electroneumaticosoleohidraulicos-electrohidraulicos/informe/monografia-de-filosofos-comensales/5223748/view>
- <https://books.google.com.pe/books?id=dYZ8DwAAQBAJ&pg=PA297&dq=pthread.h+c%2B%2B&hl=es-419&sa=X&ved=2ahUKEwjspanW7MDwAhUiK7kGHfaRAIAQ6AEwAHoECAAQAg#v=onepage&q=pthread.h%20c%2B%2B&f=false>
- <https://web.cs.wpi.edu/~cs3013/c07/lectures/Section07-Deadlocks.pdf>
- [cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7\\_Deadlocks.html](http://cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html)