UMAG
Universidad de Magallanes

Departamento de Ingeniería en Computación

Sincronizacion de Procesos: El Problema de Los Filosofos Comensales

Práctica No° 4

Estudiantes:

Milton Hernández - milherna@umag.cl

Ayrton Morrison - ayrmorris@umag.cl

Fecha y hora: 22 de Noviembre del 2024 - 8:00 AM

Profesor: Eduardo Peña J.

Asignatura: Introducción a Sistemas Operativos

Status del documento

Práctica N°4 **Sincronizacion de Procesos: El Problema de Los Filosofos Comensales**			
Número de referencia del documento			4
Versión	**Revisión**	**Fecha**	**Razones del cambio**
1	1	22 de Noviembre del 2024	Primera revisión del documento

Registro de cambios del documento		**RCD N°:**	1
		Fecha:	22 de Noviembre del 2024
		Otiginado por:	Milton Hernández M. y Ayrton Morrison R.
		Aprobado por:	Milton Hernández M. y Ayrton Morrison R.
Titulo del documento:		Práctica N°4 Sincronizacion de Procesos: El Problema de Los Filosofos Comensales	
Número de referencia del documento:			004
Versión del documento	1	**Número de revisión:**	1
Página	**Párrafo**	**Razones del cambio**	

Índice

Resumen

La comunicación y sincronización entre procesos es una de las tareas más delicadas a la hora de trabajar con Sistemas Operativos, esto debido a que una mala gestión de este ámbito podría ocasionar pérdidas de información, procesos bloqueados permanentemente e incluso la proliferación de malware en el equipo. Por esta razón, desde los inicios de la computación, se han desarrollado diversas técnicas para dar un correcto manejo a la comunicación y sincronización de procesos y la comunicación de los mismos algunas de las cuales serán puestas a prueba en la realización del presente informe.

1. Introducción

"Cinco filósofos se sientan en una mesa redonda. Cada uno cuenta con un plato de espagueti y un tenedor a su lado. El espagueti es tan escurridizo que un filósofo necesita de DOS tenedores para comerlo. Entre cada dos platos existe un tenedor, por lo que existe un número de filósofos igual al de tenedores. Cada filósofo cuenta con dos períodos de tiempo: uno en el que come y otro en el que piensa. Al sentir hambre, cada filósofo intenta tomar los cubiertos a sus lados, lo intenta con la izquierda y lo intenta con la derecha, toma un bocado del espagueti y lo come. Al terminar, deja los tenedores en su lugar y vuelve a pensar." El problema propuesto anteriormente es el llamado: "*Problema de los filósofos comensales*", este problema fue propuesto por **Edsger W. Dijkstra**, e ilustra un problema de **sincronización de procesos**. Con esto en mente, el informe actual se encargará de abordar el problema, basándose principalmente en el pseudocódigo [1] propuesto por el académico (estrácto de código 4) e implementar una solución en lenguaje C, analizando el comportamiento de la solución propuesta.

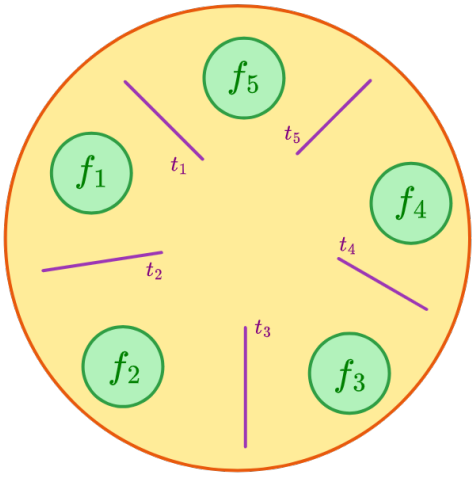

Figura 1: Problema de los filósofos comensales

[1] Un pseudocódigo es una manera de describir la lógica de un algoritmo o programa en lenguaje natural, imitando la estructura de un código sin seguir las reglas de un lenguaje de programación.

2. Objetivo Principal

Implementar mediante el uso del Lenguaje C, una implementación del problema de los filósofos comensales, haciendo uso de diversas herramientas del sistema operativo tales como los semáforos o los hilos.

3. Marco teórico

Los conceptos teóricos necesarios para el desarrollo de este trabajo son los siguientes:

3.1. Comunicación entre procesos

Por lo general, los proceso necesitan comunicarse para un correcto funcionamiento, un ejemplo util extraído del libro escrito por: Tanenbaum y Woodhull (1997) es el siguiente: "*...en un conducto de shell, la salida del primer proceso debe pasarse al segundo proceso, y así sucesiva mente. Por tanto, es necesaria la comunicación entre procesos...*"

La comunicación entre procesos o **IPC** revela tres problemas principales:

- Como compartir información entre procesos.
- Como sincronizar el trabajo de varios procesos.
- Como manejar la concurrencia de procesos.

El presente informe busca comprender una de las técnicas para la solución de este tipo de situaciones presentadas entre procesos la cuál es el uso de **semáforos**.

3.2. Hilos: Memoria compartida

La comunicación entre procesos en muchas ocasiones requiere que los procesos compartan información lo cuál no siempre es sencillo o eficiente.

Es por esta razón que existen los procesos livianos, también conocidos como hilos, o hebras que son partes de un programa que se ejecutan de forma independiente y que tienen la posibilidad de compartir memoria entre sí además de que el cambio de contexto entre ellos es más liviano para el sistema operativo en comparación a aquellos cambios de contexto de procesos pesados.

Las hebras no son la única manera de compartir memoria entre procesos, sin embargo será aquella que se usará durante el presente informe.

3.3. Condiciones de carrera y concurrencia entre procesos

Se conoce como condición de carrera o de concurso a aquella situación situación en la que uno o más procesos leen o escriben algunos datos compartidos y el resultado final depende de cuál se ejecuta en un momento preciso.

En particular se le llama **Sección Crítica** a aquella parte de un programa en que un proceso accesa a la memoria compartida con otro, momento en el cuál es probable encontrar una situación de carrera.

3.4. Semáforos

Existen diferentes técnicas que han sido creadas con el fin de superar las condiciones de carrera que se presentan en los sistemas operativos, algunas de ellas pueden ser:

- Deshabilitación de Interrupciones
- Variables de cierre
- Alternancia estricta
- La Solución Peterson o Dekker
- Los TSL
- Sleep & Wakeup
- Los semáforos
- Los monitores

En particular en el presente informe y por sugerencia del académico será abordada con más profundidad la utilización se semáforos.

Un semáforo es un tipo de variable especial que se utiliza para controlar el acceso a un recurso compartido. Un semáforo puede tener un valor entero no negativo. En el caso de que el valor sea mayor a 0, el semáforo cuenta con señales de acceso (podrían entenderse como los cupos que tiene el semáforo para dejar procesos entrar a una sección crítica), en caso contrario, el semáforo cuenta con señales de espera. Por lo que si un proceso intenta pasar por este, se bloqueará hasta que el semáforo cuente con señales de acceso.

Existen dos operaciones para los semáforos, UP y DOWN (también se pueden conocer como Wait y Post). La operación DOWN verifica si el valor del semáforo es mayor que 0, en caso de serlo, decrementa el valor del semáforo y permite el acceso al recurso compartido. Si el valor del semáforo es 0, el proceso se bloquea. La operación UP incrementa el valor del semáforo, si hay procesos bloqueados, a la hora de hacer un incremento uno de ellos será desbloqueado y podrá acceder al recurso compartido.

4. Procedimiento experimental y Análisis

Para la realización de este experimento se utilizó como base el pseudocódigo entregado por el profesor, el cual se puede encontrar en 4. Con base en este, se realizó un programa en C, el cual utiliza las bibliotecas:

- `semaphore.h`, encargada de proporcionar las funciones y estructuras necesarias para manejar semáforos.
- `pthread.h`, encargada de proporcionar las funciones y estructuras necesarias para manejar hilos.
- Demás bibliotecas estándar en el lenguaje C para entrada y salida, manejo de números aleatorios, etc···

El programa realizado se encarga de inicializar a los "Filósofos" mediante el uso de un hilo para cada uno, convirtiendo a cada uno de estos en un proceso liviano encargado de realizar las acciones de pensar, comer y tomar los tenedores. Para la coordinación entre estos, se utilizan semáforos los cuales controlan el acceso a los tenedores, de tal manera que los filósofos no puedan tomar los tenedores al mismo tiempo.

La ejecución del programa se realizó mediante impresiones por pantalla, las cuales van representando y detallando el estado en el cual se encuentra cada filósofo [THINKING, EATING, HUNGRY]. Además, se utilizó la función `sleep()` para simular el tiempo que tarda un filósofo en realizar cada acción (entre 1 y 10 segundos).

4.1. Afrontando el problema

Los tenedores han desaparecido

Para la solución del anterior problema no se tuvo en cuenta como tal la existencia de los tenedores, sino que se planteó el problema de la siguiente manera: "Un filósofo no puede empezar a comer si alguno de los filósofos adyacentes está comiendo", este enunciado, es equivalente al anteriormente mencionado pero permite una solución enfocada de otra manera.

En particular esto se puede observar en la función `test()` del programa, la cual verifica si un filósofo está intentando comer, y si aquellos adyacentes están comiendo, en caso de que no lo estén el filósofo cambiara su estado a EATING y puede proceder a comer.

Listing 1: Función para revisar si filosofo i tiene la posibilidad de comer

```
void test(int ID){
    // En caso de que tenga hambre compruebo si puedo comer.
    if(ProgramInfo.state[ID] == HUNGRY){
        if(ProgramInfo.state[LEFT] != EATING &&
            ProgramInfo.state[RIGHT] != EATING
        ){
            // En caso de poder entonces indico que estoy comiendo
```

```
            ProgramInfo.state[ID] = EATING;
            // En caso de que estuviera bloqueado por esperar a que los demas terminen de comer entonces me desbloqueo
            // Si no estoy bloqueado esta senhal evita que me bloquee por accion de la funcion "take\_forks()"
            sem_post(&ProgramInfo.s[ID]);
        }
        else{
            printf("Filosofo %d" ANSI_COLOR_RED " ESPERANDO" ANSI_COLOR_RESET "\n", ID);
        }
    }
}
```

Manejo de los semáforos

El semáforo más utilizado en este programa es el `mutex`, que evita que una hebra cambie el estado del filósofo correspondiente otra está en este mismo asunto, lo que evita que "los filósofos se confundan" o "realicen acciones simultáneas" lo que podría llevar a errores en el código.

Adicionalmente cada filósofo tiene su propio semáforo que permite controlar si está o no esperando a que el resto de los filósofos terminen de comer. Cuando un filósofo decide comer primero usa la función `test()` que, en caso de ser exitosa, llama a la función `sem_post()` (UP), sin embargo en **cualquier caso** luego de hacer el test se llama a la función `sem_wait()` (DOWN) lo que tendrá una de dos consecuencias:

1. Si el filósofo logró "tomar los tenedores" en la función test(), entonces su semáforo tendrá el valor de 1 por lo que el `sem_wait()` **no bloqueará** el proceso, sino que este se seguirá ejecutando (procediendo a comer).

2. Si el filósofo no logró "tomar los tenedores", entonces su semáforo se mantendrá en 0 y el `sem_wait()` bloqueará el proceso hasta que ambos filósofos adyacentes no estén comiendo (Esto se maneja ya que cuando un filósofo termina de comer manda una señal a los adyacentes para usar la función `test()`, que provocará el desbloqueo **en caso de que sea necesario**).

Listing 2: Función para que un filosofo tome sus tenedores

```
void take_forks(int ID){
    sem_wait(&ProgramInfo.mutex);       // Entro en la seccion critica
    ProgramInfo.state[ID] = HUNGRY;     // Indico que tengo hambre
    test(ID);                           // Compruebo si puedo o no comer
    sem_post(&ProgramInfo.mutex);       // Salgo de la seccion critica
    sem_wait(&ProgramInfo.s[ID]);       // Me bloqueo en caso que no haya podido tomar los tenedores
}
```

5. Datos obtenidos

Al hacer una ejecución del programa realizado se obtiene la siguiente salida:

Listing 3: Ejemplo de salida programa de los filosofos comensales

```
Filosofo 0 PENSANDO
Filosofo 1 PENSANDO
Filosofo 2 PENSANDO
Filosofo 3 PENSANDO
Filosofo 4 PENSANDO
Filosofo 2 COMIENDO
Filosofo 0 COMIENDO
Filosofo 3 ESPERANDO
Filosofo 4 ESPERANDO
Fin 2
        Filosofo 2 prueba a 1
        Filosofo 2 prueba a 3
Filosofo 2 PENSANDO
Filosofo 3 COMIENDO
Filosofo 1 ESPERANDO
Fin 0
        Filosofo 0 prueba a 4
        Filosofo 0 prueba a 1
Filosofo 0 PENSANDO
Filosofo 4 ESPERANDO
Filosofo 1 COMIENDO
Filosofo 2 ESPERANDO
Filosofo 0 ESPERANDO
Fin 3
        Filosofo 3 prueba a 2
        Filosofo 3 prueba a 4
Filosofo 3 PENSANDO
Filosofo 2 ESPERANDO
Filosofo 4 COMIENDO
Fin 4
        Filosofo 4 prueba a 3
        Filosofo 4 prueba a 0
Filosofo 4 PENSANDO
Filosofo 0 ESPERANDO
Fin 1
        Filosofo 1 prueba a 0
        Filosofo 1 prueba a 2
Filosofo 1 PENSANDO
Filosofo 0 COMIENDO
Filosofo 2 COMIENDO
```

```
41 Filosofo 1 ESPERANDO
42 Filosofo 3 ESPERANDO
43 Filosofo 4 ESPERANDO
44 Fin 0
45         Filosofo 0 prueba a 4
46         Filosofo 0 prueba a 1
47 Filosofo 0 PENSANDO
48 Filosofo 1 ESPERANDO
49 Filosofo 4 COMIENDO
50 Fin 2
51         Filosofo 2 prueba a 1
52         Filosofo 2 prueba a 3
53 Filosofo 2 PENSANDO
54 Filosofo 3 ESPERANDO
55 Filosofo 1 COMIENDO
```

A continuación se analizará el código 3:

- De la línea 1 a la 5 todos los filósofos son inicializados y están pensando.
- En la línea 6 el filósofo 2 empieza a comer.
- En la línea 7 el filósofo 0 empieza a comer.
- En la línea 8 el filósofo 3 quiere comer, pero queda en espera porque el filósofo 2 está comiendo (línea 6).
- En la línea 9 el filósofo 4 quiere comer, pero queda en espera porque el filósofo 0 está comiendo (línea 7).
- En la línea 10 el filósofo 2 termina de comer dado la posibilidad a 1 y a 3 de comer. Ante esta iniciativa el filósofo 3 empieza a comer pero el 1 queda en espera porque el filósofo 0 está comiendo (línea 7). En la línea 16 el filósodo 0 termina de comer dado la posibilidad a 4 y a 1 de comer. Ante esta iniciativa el filósofo 4 queda en espera porque el filósofo 3 está comiendo (línea 14) sin embargo el filósofo 1 empieza a comer.
- En la línea 22 el filósofo 2 quiere comer, pero queda en espera porque el filósofo 1 está comiendo (línea 21) y además el filósofo 3 está comiendo (línea 14).
- En la línea 23 el filósofo 0 quiere comer, pero queda en espera porque el filósofo 1 está comiendo.
- En la línea 24 el filósofo 3 termina de comer dado la posibilidad a 2 y a 4 de comer. Ante esta iniciativa el filósofo 2 queda en espera porque el filósofo 1 sigue comiendo (línea 21), pero el filósofo 4 puede empezar a comer.
- En la línea 30 el filósofo 4 termina de comer dado la posibilidad a 3 y a 0 de comer. Ante esta iniciativa el filósofo 3 no reacciona porque está pensando (línea 27), por otro lado el filósofo 0 queda esperando porque el filósofo 1 está comiendo (línea 21).

- En la línea 35 el filósofo 1 termina de comer dado la posibilidad a 0 y a 2 de comer. Ante esta iniciativa ambos filósofos 0 y 2 empiezan a comer ya que esta era la restricción que tenían para ello.
- En la línea 41 el filosofo 1 quiere comer, pero queda esperando porque el filósofo 0 está comiendo (línea 39) y el filósofo 2 está comiendo (línea 40).
- En la línea 42 el filosofo 3 quiere comer, pero queda esperando porque el filósofo 2 está comiendo (línea 40).
- En la línea 43 el filosofo 4 quiere comer, pero queda esperando porque el filósofo 0 está comiendo (línea 39).
- En la línea 44 el filosofo 0 termina de comer dando la posibilidad a 4 y a 1 de comer. Ante esta iniciativa el filósofo 1 queda esperando porque el filósofo 2 está comiendo (línea 40), pero el filósofo 4 puede empezar a comer.
- En la línea 50 el filosofo 2 termina de comer dando la posibilidad a 1 y a 3 de comer. Ante esta iniciativa el filósofo 3 queda esperando porque el filósofo 4 está comiendo (línea 49), pero el filósofo 1 puede empezar a comer.

Este programa continúa su ejecución de manera “infinita”, logrando una correcta sincronización de los filósofos.

6. Conclusiones

Como se observó a lo largo del escrito, el problema de los filósofos es un reto que puede ser abarcado desde distintas aristas, algunas más eficientes o completas que otras, pero todas con un mismo resultado, la correcta sincronización de los filósofos (de los procesos).

En este caso, se presentó una única solución, la cual se puede considerar como completa, debido a que se abarca el problema de manera general y se resuelven los problemas presentados en el enunciado entregado en la sección 1.

Es destacable mencionar el cómo este problema y su solución se pueden extrapolar en la modelación aplicaciones reales, como los son los problemas de E/S en los sistemas operativos.

Finalmente, se puede decir con seguridad que el objetivo planteado en la sección 2 fue cumplido a cabalidad habiendo comprendido el problema de los filósofos y una solución eficiente mediante el uso de hebras y semáforos.

7. Anexos

Listing 4: Pseudocodigo del problema de los filósofos comensales

```
// Definiciones iniciales

#define N 5 /* Numero de filosofos */
#define LEFT (i-1)%N /* Vecino a la izquierda de i */
#define RIGHT (i+1)%N /* Vecino a la derecha de i */
#define THINKING 0 /* Filosofando */
#define HUNGRY 1 /* Intentando comer */
#define EATING 2 /* Comiendo */

// Variables a utilizar
typedef int semaphore; /* Semaforos como int */
int state[N]; /* Estado de los filosofos */
semaphore mutex = 1; /* Exclusion mutua */
semaphore s[N]; /* Un semaforo por filosofo */

// Funcion que ejecutaria el i-esimo filosofo
philosopher(i)
int i;
{
    while (TRUE) {
    think();
    take_forks(i); /* Toma dos tenedores o bloquea */
    eat();
    put_forks(i); /* Deja los tenedores */
    }
}

// Funcion para que un filosofo tome sus tenedores
take_forks(i)
int i;
{
    down(mutex);
    state[i] = HUNGRY;
    test(i);
    up(mutex);
    down(s[i]);
}

// Funcion para revisar si filosofo i tiene la posibilidad de comer
test(i)
int i;
{
```

```
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(s[i]);
    }
}

// Funcion para devolver los tenedores del filosofo i a su lugar
put_forks(i)
int i;
{
    down(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(mutex);
}
```

Referencias

ipc.h. (2024). IBM. Consultado el 1 de noviembre de 2024, desde https://www.ibm.com/docs/es/aix/7.2?topic=files-ipch-file

Tanenbaum, A. S., & Woodhull, A. S. (1997). *Sistemas Operativos: Diseño e implementación* (R. Escalona, Trad.; Segunda edición). Prentice Hall.