Práctica 6: El problema de la cena de los filósofos

Pablo Rodríguez Fernández

Mayo 2020

I. Contenido del documento

En este documento ,tras una breve introducción al problema de la cena de los filósofos (o problema de los filósofos), se explicará como se han realizado las soluciones a este utilizando diferentes primitivas de sincronización. Concretamente, se han realizado dos versiones utilizando semáforos (una utilando hilos y otra procesos);una utilizando mutexes y variables de condición, y una última utilizando paso de mensajes.

II. El problema de los filósofos: Una breve introducción

El problema de los filósofos es un problema clásico de sincronización, enunciado por Djisktra en los años 60. En este problema se supone que existe una mesa redonda en la cual están sentados filósofos listos para comer. Ahora bien, cada filósofos necesita dos tenedores para comer de su plato, y los tenedores se comparten 2 a 2, como se puede ver en la **Figura 1**. Por tanto, se debe buscar la solución que permita un mayor número de filósofos comiendo simultáneamente, minimizando los tiempos de espera.

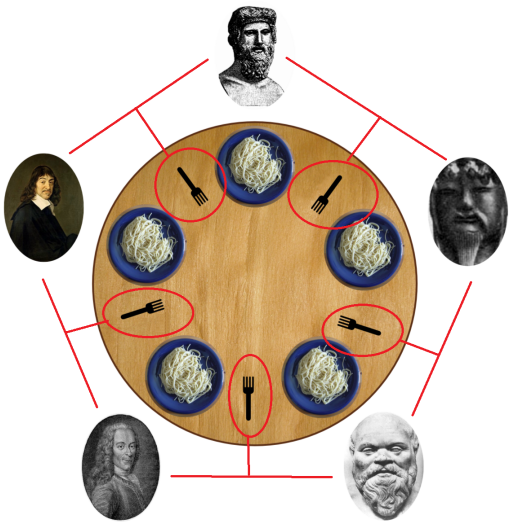

Figura 1: *Distribución de la mesa*

Existen una gran variedad de soluciones a este problema, explarándose en este caso dos:

- Una solución es la propuesta por **Tanenbaum**, y se basa en el empleo de la función **probar**, la cual realiza varias funciones, entre ellas comprobar si los tenedores están libres. Además, utiliza un array de **estados** para indicar si un filósofo está comiendo, hambriento (esperando para comer) o pensando (sin intención de comer).

- Otra solución basada en el empleo de un **camarero**, es decir, un hilo/proceso que regula cuando un filósofo puede tomar un tenedor, dejarlo... Esta solución está pensada para el uso de **paso de mensajes** como mecanismo de sincronización, debido a la gran diferencia que existe entre el empleo de esta herramienta con otras de sincronización.[1]

III. Implementaciones

En este apartado se comentarán los fragmentos más relevantes de las diferentes implementaciones, presentando un caso base de ejemplo (Semáforos) y mostrando luego las diferencias más destacables en los fragmentos mostrados. Para entender completamente el funcionamiento y la construcción de los ejercicios se recomienda ver los códigos (comentados) que acompañan a este informe.

A. Semáforos

La solución propuesta por Tanenbaum aparece resulta en su libro utilizando semáforos, siendo por tanto este apartado, a priori, el más sencillo.

En 1 pode observarse a declaración das variables globais compartidas por todos fíos, e que son unha parte fundamental da solución. Simplemente destacar el array *tenedores* simplemente almacena N letras, identificando cada letra a un tenedor, para facilitar la interpretación de la solución en el terminal

```
int N=0; //Numero de filosofos.
int *estado; //Array que lleva registro del estado de todos los filosofos
sem_t *mutex;//Toma los valores 0 o 1.Garantiza la exclusion mutua
//Un semáforo por filósofo, para bloquear a los filósofos hambrientos si sus tendores están ocupados
sem_t **semaforo;
char *tenedores;
```

Listing 1: *Declaración de variables globales*

En 2 puede observarse la función **filósofo**, que es ejecutada por cada uno de los hilos y realiza la simulación del comportamiento de un filósofo. [2]

```
int IZQUIERDO = (numFilosofo+N-1)%N;
int DERECHO = (numFilosofo+1)%N;

while(iteraciones--){
  pensar(numFilosofo);
  tomar_tenedores(numFilosofo);
  comer(numFilosofo);
  poner_tenedores(numFilosofo,IZQUIERDO,DERECHO);
}
```

Listing 2: *Declaración de variables globales*

Simplemente indicar que el cálculo de IZQUIERDO y DERECHO se realiza en el interior de esta función punto ya que, si bien Tanenbaum declara estas variables utilizando *define*, en este caso no es posible, por ser N definido por el usuario al inicio y, por tanto, tampoco se pueden realizar estos cálculos de manera global

[1] En el problema del productor-consumidor sucedía algo similar, ya que solución utilizando paso de mensajes era también diferente al resto.

[2] Se omitió la declaración de los semáforos, creación de los hilos e demás acciones por ser similar a lo mostrado en informes anteriores

Se pasa ahora a comentar brevemente las funciones **tomar_tenedores** y **poner_tenedores**, que se pueden encontrar en el fragmento 3 y en el fragmento 4, respectivamente. En ambas funciones se realiza un down del mutex para acceder a la región crítica, en la cual se cambia el estado del filósofo (HAMBRIENTO si se está en tomar_tenedores, PENSANDO si se est á en poner_tenedores), y se invoca a la función *probar*. Esta invocación tiene diferentes intenciones en función de quién la invoque:

- Si la invocación la realiza tomar_tenedores, lo que se intenta es **obtener** los tenedores para el filósofo que ejecuta la función.
- Si la invocación la realiza poner_tenedores, se trata de lograr que nuevos filósofos puedan continuar la ejecución, otorgándole los tenedores a ellos si es posible. Por dicha razón en esta función se llama a probar dos veces, una para intentar que el filósofo IZQUIERDO avance, y otra para intentar que el filósofo DERECHO avance, utilizando el respectivo tenedor que deja libre el filósofo que pone los tenedores disponibles.

El código de la función probar puede encontrarse en 5. A mayores de lo comentado ya, decir que en esta función es necesario recalcular IZQUIERDO y DERECHO ya que puede ser invocado por un filósofo para que un filósofo vecino pruebe los tenedores, no él mismo (como sucede en tomar_tenedores)

```
  sem_wait(mutex);
  estado[nF]=HAMBRIENTO;
  printf("%s El filosofo %d esta hambriento\n",colors[nF%15],nF);
  probar(nF); //Intentamos obtener los tenedores
  sem_post(mutex);
  sem_wait(semaforo[nF]);
  sleep(1);
}
```

Listing 3: *Función tomar_tenedores*

```
void poner_tenedores(int nF,int IZQUIERDO,int DERECHO){
  sem_wait(mutex);//Entramos en la región crítica
  estado[nF]=PENSANDO;
  printf(" %s El filosofo %d esta pensando\n",colors[nF%15],nF);
  //Comprobamos si el filosofo izquierdo puede apovechar el tenedor izquierdo para ponerse a comer
  probar(IZQUIERDO);
   //Comprobamos si el filosofo derecho puede apovechar el tenedor derecho para ponerse a comer
  probar(DERECHO);
  sem_post(mutex);//Salimos de la región crítica
}
```

Listing 4: *Función poner_tenedores*

```
void probar(int nF){
  int IZQUIERDO = (nF+N-1)%N;
  int DERECHO = (nF+1)%N;
  if(estado[nF]==HAMBRIENTO && estado[IZQUIERDO]!=COMIENDO && estado[DERECHO]!=COMIENDO){
    estado[nF]=COMIENDO;
    printf("%s El filosofo %d esta comiendo\n",colors[nF%15],nF);
    sem_post(semaforo[nF]); //Avisamos al filosofo nF
  }
}
```

Listing 5: *Función probar*

A.1. Versión con procesos

A parte de la versión anterior, basada en el empleo de hilos, se pedía adaptar una de las versiones para un mecanismo de sincronización al uso de procesos. En este caso el paso de hilos a procesos es bastante inmediato, gracias al hecho de que los semáforos con un mismo son compartidos entre todos los procesos del sistema. Por tanto, basta crear los semáforos en un proceso padre y los hijos podrán acceder a ellos y ver los cambios que se realicen sin ningún problema. Simplemente detallar que, a mayores del cambio obvio que supone crear procesos en lugar de hilos, fue necesario utilizar una zona de memoria compartida, para que el array que almacena el estado de los distintos filósofos pudiese ser modificado por todos los procesos. Estas modificaciones pueden verse en el código anexo correspondiente.

B. Mutexes y variables de condición

El paso de semáforos a mutexes y variables de condición es bastante sencillo. Al estar la versión 'básica' de semáforos implementada utilizando hilos, basta con realizar los siguientes cambios:

- La variable mutex pasará a ser de tipo *pthread_mutex_t*, es decir, dejará de ser un mutex semáforo para ser un mutex 'mutex'. Por tanto, todas las llamadas up y down se transformarán en lock y unlock sobre el mutex.
- El array de semáforos asociados con cada filósofo se convierte en un array de variables de condición. Por tanto, en lugar de realizar un up de un semáforo, se enviará un cond_signal, y los mutex asociados a dicha variable de condición reaccionarán de la manera adecuada

A mayores de estos cambios, es necesario que la función tomar_tenedores incluya un bucle infinito en su interior, como se puede ver en 6. Esto es debido a que ,si no se recoge la señal enviada por una variable de condición, esta se pierde. Por tanto es necesario realizar una espera activa para evitar su pérdida.

```
pthread_mutex_lock(&mutex);
estado[nF]=HAMBRIENTO;
printf("%s El filosofo %d esta hambriento\n",colors[nF%15],nF);
probar(nF); //Intentamos obtener los tenedores
while(estado[nF]!=COMIENDO) pthread_cond_wait(&condicion[nF],&mutex);
pthread_mutex_unlock(&mutex);
```

Listing 6: *Bucle infinito de tomar_tenedor*

C. Paso de mensajes

Como se dijo anteriormente, la versión que utiliza paso de mensajes realiza una solución diferente a la de Tanenbaum. El primer cambio significativo es que la función probar desaparece para dejar paso a otra (de mayor complejidad) denominada **camarero**, que será ejecutada por un hilo a mayores, y regulará a los hilos filósofos.
A mayores de este cambio, destaca el hecho de que existirá una cola de mensajes diferente para cada filósofo (almacenadas en un vector, de manera similar a los semáforos en versiones anteriores), y otra para el camarero. Esta será la manera que se tendrá en este caso para interactuar entre los diferentes hilos, enviando los filósofos mensajes con peticiones a la cola del camarero, el cual responderá a cada uno en particular

Se mostrarán directamente a las funciones que simulan el comportamiento de los filósofos. Respecto a las funciones poner_tenedores y poner_tenedores, ambas realizan ahora la misma función (a vista de código), con la diferencia del estado que asignan al filósofo (que se mantiene

igual respecto a las otras versiones). Por tanto, se mostrará solo el código de una de ellas, en concreto de tomar_tenedores, que puede ser consultado en 7

```
void tomar_tenedores(int nF){
 char mensaje[2], vacio[0];
 sprintf(mensaje,"%d",nF); //Pasamos el numero a la cadena
  estado[nF]=HAMBRIENTO;
  printf("%s El filosofo %d esta hambriento\n",colors[nF%15],nF);
  //Enviamos el mensaje al camarero
  mq_send(almacenCamarero,mensaje,sizeof(mensaje),0);
  //Esperamos a que llegue un mensaje vacio
  mq_receive(almacenFilosofo[nF],vacio,sizeof(mensaje),0);
}
```

Listing 7: *tomar_tenedor con paso de mensajes*

Como se mencionó anteriormente, los filósofos envían mensajes a la cola del camarero, y esperan mensajes de este. El mensaje que envían al camarero contiene el número de filósofo que son, y el que reciben está vacío: simplemente se usa para indicar a un filósofo que puede continuar su ejecución, guardando cierto símil al envío de mensajes vacío por parte del consumidor al productor en el problema del productor-consumidor.

Finalmente, comentar las funcionalidades que realiza la función *camarero*. En esta función se utiliza una vector de tenedores libres, que indica que tenedores están libres en cada momento y cuales ocupados. Cuando llega un mensaje, se estudia el estado del filósofo que envía el mensaje:

- Si el filósofo está **HAMBRIENTO**, se comprueba si sus tenedores izquierdo y derecho están libres (usando el array de tenedores libres). En caso de ser así, se cambia se estado a COMIENDO ,se marcan los tenedores como ocupados y se le envía un mensaje para que continúe con al ejecución. Si por el contrario los tenedores están ocupados (o no están disponibles ambos), se almacena el número del filósofo en una cola, para que cuando queden tenedores libres se envía un mensaje a la cola de mensajes del filósofo para que pueda continuar su ejecución.
- En caso de que el estado del filósofo sea **COMIENDO**, significa que el mensaje se envía para liberar los tenedores pues el filósofo ha terminado de comer. En dicho caso, se envía un mensaje al filósofo para que continúe su ejecución y posteriormente se recorrer el array que almacena el número de los filósofos que están esperando por los tenedores. En caso de que debido a la entrega de nuevos tenedores puedan comer, se les notifica y se marcan los tenedores ocupados, como sucedía en el caso anterior. De no existir ningún filósofo que pueda continuar, se vuelve al principio de la función, estando el camarero esperando a que le lleguen nuevas peticiones.

Debido a la complejidad de esta función (en comparación con las otras mostradas en el documento), se recomienda ver su código, el cuál está comentado para facilitar su comprensión.

IV. Cierre del documento

Tras analizar las diferentes implementaciones, se ha visto como es posible implementar una solución satisfactoria para el problema de la cena de los filósofos con todos los mecanismos de sincronización estudiados durante el curso. Esto no es casualidad, ya que este problema fue usado durante mucho tiempo para demostrar que una nueva primitiva de sincronización era válida.