Exposicion De Sistemas Operativos

Helen L. Bernal, Joe F. Velez, Renán O. Pérez, Jonatán J. Guillén y Manuel A. Burgos

Facultad de Ciencias Matemáticas y Física, Universidad de Guayaquil

SOF-S-MA-3-1: Sistemas Operativos

Ing. Perez E. Charles

15 de septiembre del 2020

Problemas de Concurrencia

Concepto

En los sistemas de tiempo compartido (aquellos con varios usuarios, procesos, tareas, trabajos que reparten el uso de CPU entre estos) se presentan muchos problemas debido a que los procesos compiten por los recursos del sistema. Imagine que un proceso está escribiendo en la unidad de cinta y se le termina su turno de ejecución e inmediatamente después el proceso elegido para ejecutarse comienza a escribir sobre la misma cinta. El resultado es una cinta cuyo contenido es un desastre de datos mezclados. Así como la cinta, existen una multitud de recursos cuyo acceso debe ser controlado para evitar los problemas de la concurrencia.
El sistema operativo debe ofrecer mecanismos para sincronizar la ejecución de procesos: semáforos, envío de mensajes, pipes, etc. Los semáforos son rutinas de software (que en su nivel más interno se auxilian del hardware) para lograr exclusión mutua en el uso de recursos. Para entender este y otros mecanismos es importante entender los problemas generales de concurrencia, los cuales se describen enseguida.

Existe un conjunto de problemas clásicos de sincronización que se suelen emplear como modelo para explicar las herramientas estudiadas en este tema. La elección no es arbitraria: estos problemas presentan aspectos claves dentro del ámbito de concurrencia que se suelen dar en la mayor parte de los problemas de sincronización.

Lectores y Escritores

Historia (para ambos problemas, solo lo subrayado)

A menudo se asume que Dijkstra instigó el estudio de la programación concurrente en su ahora clásico artículo "Procesos secuenciales de cooperación", publicado en 1967.

Ciertamente, en este artículo vemos la introducción de algunos problemas ahora bien conocidos como "The Dining Philosophers "," The Sleeping Barber "y" The Dutch Flag Problem ", y quizás lo más importante, el problema de la sección crítica y su solución utilizando semáforos. Este artículo fue, de hecho, el primero en tener una visión de alto nivel de la programación concurrente. Como nota aparte, es interesante observar que el mismo artículo introduce la noción de interbloqueo y presenta un algoritmo que puede detectar la posible presencia de bloqueos. Sin embargo, a principios de la década de 1970 P.J. Curtois propuso solución al problema de lectores y escritores en una base de datos, muchos procesos van a intentar leer o escribir al mismo tiempo, lo cual no podemos permitir, ya que, nuestra información se podría perder, duplicar, corromper entre otros. Solo es posible que un proceso lea o escriba y que los demás esperen su turno.

Primer problema de los lectores y escritores (First Readers-Writers Problem)

En este problema de sincronización existen dos tipos de procesos: procesos lectores interesados en la lectura de una serie de variables compartidas y procesos escritores que modifican el contenido de esas variables compartidas. El acceso concurrente de varios procesos lectores debe permitirse, pero nunca debe permitirse el acceso concurrente de procesos lectores y escritores. El primer problema de los lectores escritores es aquel que da preferencia a los procesos lectores frente a los procesos escritores, es decir, ningún lector debería esperar a menos que un escritor ya haya obtenido permiso para escribir.

Segundo problema de los lectores escritores (Readers and Writers Second Problem)

El segundo problema de los lectores escritores es aquel que da preferencia a los procesos escritores frente a los procesos lectores, es decir, ningún nuevo lector debería entrar a leer desde el momento que un escritor esté esperando.

Puntos principales del problema

- Modela el acceso a una base de datos
- Podría haber varios lectores al mismo tiempo
- Pero si un escritor está en la base de datos, ningún otro proceso (escritor o lector) podrá ingresar.
- Los escritores requieren acceso exclusivo a la BD (base de datos).
- Si un lector está usando la BD (base de datos) y llega otro u otros lectores, estos podrían ingresar.
- Si un lector está en la BD (base de datos) y llega un escritor este se suspende hasta que salga el último lector. Si llega otro lector se formaría atrás del escritor. Esto merma la concurrencia.

Restricciones

- Solo se permite que un escritor tenga acceso al objeto al mismo tiempo. Mientras el escritor esté accediendo al objeto, ningún otro proceso lector ni escritor podrá acceder a él.
- Se permite que múltiples lectores tengan acceso al objeto, ya que ellos nunca van a modificar el contenido del mismo.
- Primer Problema: No se debe tener ningún lector en espera a menos que el escritor tenga el permiso del uso del objeto.
- Segundo Problema: Si un escritor está esperando acceder al objeto, ningún otro lector puede comenzar a leer.

Solución

Uso de semáforos

En esta solución, el primer lector que obtiene el acceso a la base de datos realiza un *wait* sobre el semáforo *bd*. Los lectores siguientes solo incrementan un contador, *nl*. Al salir los lectores, estos decrementa el contador, y el último en salir realiza un *signal* sobre el semáforo, lo que permite entrar a un escritor bloqueado, si existe.

Una hipótesis implícita en esta solución es que los lectores tienen prioridad sobre los escritores. Si surge un escritor mientras varios lectores se encuentran en la base de datos el escritor debe esperar. Pero si aparecen nuevos lectores, y queda al menos un lector accediendo a la base de datos, el escritor deberá esperar hasta que no haya más lectores interesados en la base de datos.

Definir prioridades a lectores y escritores.

Cuando un escritor está escribiendo los datos, todos los demás escritores o lectores se bloquearán hasta que el escritor termine de escribir.

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                  /* repeat forever */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc + 1;                /* one reader more now */
        if (rc == 1) down(&db);     /* if this is the first reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        read_data_base();           /* access the data */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc - 1;                /* one reader fewer now */
        if (rc == 0) up(&db);       /* if this is the last reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        use_data_read();            /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data();            /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base();          /* update the data */
        up(&db);                    /* release exclusive access */
    }
}
```

Conclusión

Filósofos Comensales

Historia

Fue formulado originalmente en 1965 por Edsger Dijkstra como un ejercicio de examen para estudiantes, presentado en términos de computadoras que compiten por el acceso a periféricos de unidades de cinta . Poco después, Tony Hoare en 1985 le dio al problema su formulación actual.

Edsger Wybe Dijkstra nació en Rotterdam (Holanda) en 1930. Estudió Física Teórica, trabajó en el Centro Matemático de Amsterdam y finalmente se entregó al estudio de la

Programación (uno de los problemas con que se encontró es que ser programador no estaba oficialmente reconocido como profesión por aquel entonces).

A principios de los 70 aceptó un puesto como desarrollador en Estados Unidos, país en el que permaneció hasta su muerte en 2002. Fueron numerosas sus contribuciones al avance de la programación informática.

El problema de la cena de los filósofos. (Explicación)

Cinco filósofos se sientan en una mesa redonda. En el centro de la mesa hay un plato de arroz. El arroz es tan escurridizo que un filósofo necesita dos palillos para comerlo, pero cada filósofo sólo posee un palillo, luego existe el mismo número de filósofos que de palillos. La vida de un filósofo consta de períodos alternados de comer y pensar. Cuando un filósofo siente hambre, intenta coger el palillo de la izquierda y si lo consigue, lo intenta con el de la derecha. Si logra asir dos palillos toma unos bocados y después deja los palillos y sigue pensando.

El objetivo consiste en encontrar un recurso que permita que los filósofos nunca se mueran de hambre. Porque:

– Dos filósofos contiguos no pueden comer a la vez (**exclusión mutua**).

– Si un filósofo está comiendo, los contiguos no pueden hacerlo hasta que aquél termine (**sincronización**).

– El filósofo que termina de comer debe ceder los palillos para su posterior utilización (**interbloqueo**).

Algunas posibles soluciones:

Por turno cíclico

Se empieza por un filósofo, que si quiere puede comer y después pasa su turno al de la derecha. Cada filósofo sólo puede comer en su turno. Problema: si el número de filósofos es muy alto, uno puede morir de hambre antes de su turno.

Varios turnos

Se establecen varios turnos. Para hacerlo más claro supongamos que cada filósofo que puede comer (es su turno) tiene una ficha que después pasa a la derecha. Si por ejemplo hay 7 comensales podemos poner 3 fichas en posiciones alternas (entre dos de las fichas quedarían dos filósofos).

Se establecen turnos de tiempo fijo. Por ejemplo cada 5 minutos se pasan las fichas (y los turnos) a la derecha.

En base al tiempo que suelen tardar los filósofos en comer y en volver a tener hambre, el tiempo de turno establecido puede hacer que sea peor solución que la anterior. Si el tiempo de turno se aproxima al tiempo medio que tarda un filósofo en comer esta variante da muy buenos resultados. Si además el tiempo medio de comer es similar al tiempo medio en volver a tener hambre la solución se aproxima al óptimo.

Colas de palillos

Cuando un filósofo quiere comer se pone en la cola de los dos tenedores que necesita. Cuando un tenedor está libre lo toma. Cuando toma los dos tenedores, come y deja libre los tenedores.

Visto desde el otro lado, cada tenedor sólo puede tener dos filósofos en cola, siempre los mismos.

Esto crea el problema comentado de que si todos quieren comer a la vez y todos empiezan tomando el tenedor de su derecha se bloquea el sistema (deadlock).

Resolución de conflictos en colas de palillos

Cada vez que un filósofo tiene un tenedor espera un tiempo aleatorio para conseguir el segundo tenedor. Si en ese tiempo no queda libre el segundo tenedor, suelta el que tiene y vuelve a ponerse en cola para sus dos tenedores.

Si un filósofo A suelta un tenedor (porque ha comido o porque ha esperado demasiado tiempo con el tenedor en la mano) pero todavía desea comer, vuelve a ponerse en cola para ese tenedor. Si el filósofo adyacente B está ya en esa cola de tenedor (tiene hambre) lo toma y si no vuelve a cogerlo A.

Es importante que el tiempo de espera sea aleatorio o se mantendrá el bloqueo del sistema.

El portero del comedor

Se indica a los filósofos que abandonen la mesa cuando no tengan hambre y que no regresen a ella hasta que vuelvan a estar hambrientos (cada filósofo siempre se sienta en la misma silla). La misión del portero es controlar el número de filósofos en la sala, limitando su

número a n-1, pues si hay n-1 comensales seguro que al menos uno puede comer con los dos tenedores.

Conclusión

Este planteamiento resulta muy útil para los que estudian informática porque ayuda a pensar en los sistemas que tienen recursos limitados (en el ejemplo anterior los palillos son limitados) y en los clientes (programas y usuarios): hay que darles servicio (comida) a todos en un tiempo razonable.

Se trata de que los recursos sean utilizados de la manera más eficiente por todos los procesos implicados. Hay algoritmos para solucionarlo pero todos los métodos pasan por asignar prioridades y/o tiempos máximos de uso de los recursos.

La finalidad es demostrar que se presentarán problemas ante la falta de una apropiada sincronización y evitar la peligrosa condición de carrera.

Bibliografía

Lectores y Escritores

1. https://books.google.es/books?hl=es&lr=&id=fRK3lbTrNy4C&oi=fnd&pg=PP1&dq=lectores+y+escritores+sistemas+operativos+ejercicio+de+concurrencia&ots=0wziWrD76A&sig=jQ47lE835PgHPn6k9835LOqEIyM#v=onepage&q=lectores%20y%20escritores f=true
2. http://www.redtauros.com/Clases/Gestion_SO/02%20Sistemas%20Operativos%20-%20Administracion%20de%20procesos.pdf
3. https://fdocuments.ec/reader/full/so-sincronizacion-de-procesos

Filósofos Comensales

1. https://sistemasoperativo.wordpress.com/historia-de-los-sistemas-operativos-y-la-computadora/problemas-clasicos/
2. https://studylib.es/doc/662871/problema-de-los-filosofos-comensales#
3. https://docs.python.org/es/3/library/threading.html#semaphore-example
4. https://docplayer.es/50433340-Problemas-clasicos-el-problema-de-los-filosofos-comensales.html