

2012

Lectores y Escritores

Problema clásico de sincronización de Procesos

Informe de proyecto sobre el tema: Modelo de procesos secuenciales.
Procesos, Hilos y Planificación.

Autores:

- ✓ Jesse Daniel Cano Otero
- ✓ Marlon Jorge Remedios Gonzalez
- ✓ Antonio Membrides Espinosa

Posgrado de Software de Sistema,
UCI,
La Habana



Índice

Introducción	3
Modelo de Solución	3
Implementación	4
Conclusiones	7
Bibliografía	8

Introducción

Un sistema operativo multiprogramado es un caso particular de sistema concurrente donde los procesos compiten por el acceso a los recursos compartidos o cooperan dentro de una misma aplicación para comunicar información. Ambas situaciones son tratadas por el sistema operativo mediante mecanismos de sincronización que permiten el acceso exclusivo de forma coordinada a los recursos y a los elementos de comunicación compartidos.

La interacción entre procesos se plantea en una serie de situaciones clásicas de comunicación y sincronización. Entre estas se destacan la Cena de los Filósofos, Productores y Consumidores, Lectores y Escritores, Buffer Finito, etc, por tan solo citar algunos ejemplos. A continuación se procederá a describir particularmente el de Lectores y Escritores, en función de demostrar la necesidad de comunicar y sincronizar procesos.

En este tipo de problemas se evidencia una base de datos compartida por varios procesos, distinguiéndose entre ellos por dos tipos: los lectores que sólo necesitan leer el contenido de la base de datos sin necesidad de modificarla, y los escritores que requieren actualizar o adicionar determinada información. Obviamente, si dos o más lectores acceden a la base de forma simultánea no existirá ningún conflicto, sin embargo, si un escritor y algún otro escritor o lector acceden a la base de datos de manera simultánea, entonces si puede producirse una integración que generaría resultados inconsistentes o inesperados.

Teniendo en cuenta lo anteriormente planteado en las soluciones a dichas problemáticas se deben cumplir con las restricciones que se muestran a continuación:

- ✓ Sólo se permite que un escritor tenga acceso al objeto al mismo tiempo. Mientras un escritor esté accediendo al recurso compartido, ningún otro proceso lector o escritor podrá acceder al mismo.
- ✓ Se permite que múltiples lectores tengan acceso al objeto o recurso compartido, pues estos nunca modificarían el contenido del mismo.

Modelo de Solución

En este tipo de problemas es necesario disponer de servicios de sincronización que permitan a los procesos lectores y escritores sincronizarse adecuadamente en el acceso al recurso compartido. Después de un análisis se identificaron tres alternativas para dar solución a este tipo de problemas, tomándose como principio la asignación de prioridades:

1. Los lectores tienen prioridad sobre los escritores.
2. Los escritores toman prioridad sobre los lectores.
3. Tanto los escritores como los lectores poseen la misma prioridad.

A continuación se describirá a través de un pseudocódigo la primera, donde los lectores poseen prioridad sobre los escritores.

- Variables globales


```
rw_mutex: semaphore(1)
reader_mutex: semaphore(1)
readercount: integer(0)
```
- Hilo Escritor


```
while (true) {
    wait (rw_mutex);
    //... writing ...
    signal (rw_mutex);
}
```
- Hilo Lector


```
while (true) {
    wait(reader_mutex) ;
    readercount ++ ;
    if (readercount == 1) wait(rw_mutex);
    signal(reader_mutex)
    //... reading ...
    wait(reader_mutex);
    readercount --;
    if (readercount == 0) signal(rw_mutex);
    signal(reader_mutex);
}
```

En esta solución, el primer lector que obtiene el acceso a la base de datos realiza un *wait* sobre el semáforo denominado *rw_mutex*, garantizando de esta forma que queden en estado bloqueado aquellos procesos escritores que se apoderen del CPU. Los lectores siguientes sólo incrementan un contador denominado *readercount*. Al terminar, éstos decrementan dicha variable, de forma tal que el último en salir sea el responsable de realizar un *signal* sobre el semáforo *rw_mutex*, posibilitando entrar a un escritor bloqueado, si existe.

Implementación

Con el objetivo de implementar un sistema capaz de simular los procesos descritos en el pseudocódigo anterior, se decidió utilizar la librería *pthread* desarrollada para el lenguaje C++.

1. `#include <sys/time.h>`
2. `#include <stdio.h>`
3. `#include <pthread.h>`
4. `#include <errno.h>`

5. **pthread_mutex_t** rw_mutex = PTHREAD_MUTEX_INITIALIZER;
6. **pthread_mutex_t** reader_mutex = PTHREAD_MUTEX_INITIALIZER;
7. **int** readercount = 0;

De igual forma se requieren dos semáforos, uno que restrinja el acceso a la variable denominada *readercount*, la cual contiene la cantidad real de procesos lectores en ejecución y otro en cargado de controlar el estado de ejecución para aquellos procesos escritores que intente modificar los datos del recurso compartido, en este caso se definen como *reader_mutex* y *rw_mutex* respectivamente.

8. **void*** reader();
9. **void*** writer();

En función de diferenciar los tipos de procesos se definen dos funciones denominadas *reader* y *writer*. La primera encapsula el comportamiento de aquellos procesos que solo acceden al recurso compartido en modo de lectura y la segunda para aquellos que además son capaces de modificar dicho contenido de la base de datos.

10. **main()**
11. {
12. **pthread_t** reader_thread1;
13. **pthread_t** reader_thread2;
14. **pthread_t** reader_thread3;
15. **pthread_t** reader_thread4;
16. **pthread_t** writer_thread1;
17. **pthread_t** writer_thread2;
18. **pthread_create**(&reader_thread1, NULL, reader, NULL);
19. **pthread_create**(&reader_thread2, NULL, reader, NULL);
20. **pthread_create**(&reader_thread3, NULL, reader, NULL);
21. **pthread_create**(&reader_thread4, NULL, reader, NULL);
22. **pthread_create**(&writer_thread1, NULL, writer, NULL);
23. **pthread_create**(&writer_thread2, NULL, writer, NULL);
24. **pthread_join**(writer_thread2, NULL);
25. **pthread_join**(writer_thread1, NULL);
26. }

Para poner en práctica la efectividad de dicha implementación se decide crear cuatro procesos lectores y dos procesos escritores. En cualquiera de las creaciones de estos hilos se asume los atributos de configuración por defecto, por tanto el modo de ejecución sería PTHREAD_CREATE_JOINABLE, permitiendo al hilo principal esperar por la terminación de sus hijos. De igual forma para este ejemplo no es relevante los valores de entrada y retorno de estas subrutinas por tanto se les especifica valor nulo. Teniendo en cuenta que el orden de prioridad es de los lectores sobre los escritores se

establece que el hilo principal debe esperar por la terminación de los hilos escritores en función de observar el comportamiento de dicho programa.

```
27. void* reader()
28. {
29.     int i = 0;
30.     while (1)
31.     {
32.         pthread_mutex_lock(&reader_mutex);
33.         readercount++;
34.         if (readercount == 1) pthread_mutex_lock(&rw_mutex);
35.         pthread_mutex_unlock(&reader_mutex);
36.
37.         printf("start reading ...\n");
38.         sleep(1);
39.         printf("finish reading ...\n");
40.
41.         pthread_mutex_lock(&reader_mutex);
42.         readercount--;
43.         if (readercount == 0) pthread_mutex_unlock(&rw_mutex);
44.         pthread_mutex_unlock(&reader_mutex);
45.
46.         sleep(i%3);
47.         i++;
48.     }
49.     pthread_exit(NULL);
50. }
```

El comportamiento de la subrutina del lector es la misma que la descrita en el pseudocódigo, donde el primer lector bloquea el semáforo que restringe la ejecución de los procesos escritores y el último lo libera en función de que estos puedan acceder al recurso compartido. En este ejemplo en particular y para ambas subrutinas se utiliza el recurso *sleep* para dormir los procesos un tiempo determinado y dar cobertura al planificador para una simulación que se ajuste un poco más a lo que se desea demostrar.

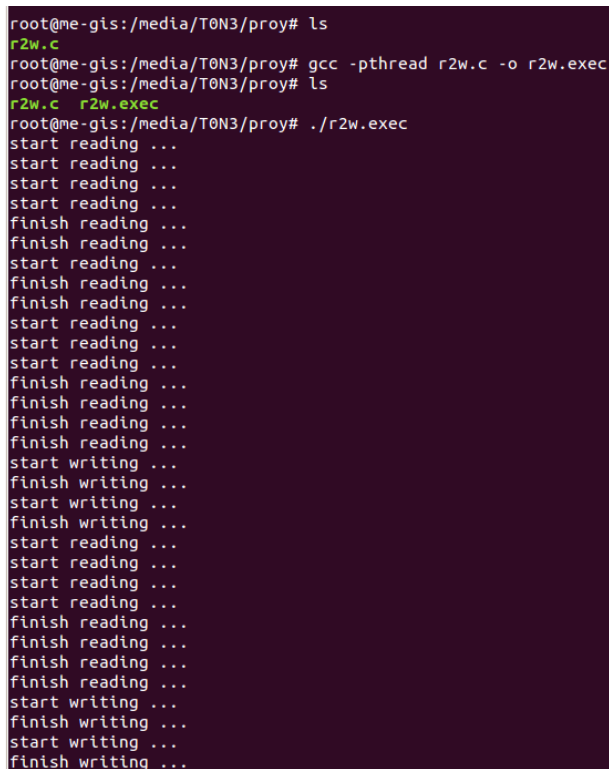
```
51. void* writer()
52. {
53.     int i;
54.     for (i=0; i<100; i++)
55.     {
56.         pthread_mutex_lock(&rw_mutex);
57.         printf("start writing ...\n");
58.         sleep(1);
59.     }
60. }
```

```

        printf("finish writing ...\n");
        pthread_mutex_unlock(&rw_mutex);
        sleep(i%2);
40.     }
41.     pthread_exit(NULL);
42. }

```

Para poner en práctica los elementos mencionados, es necesario la creación de un fichero que contenga el código fuente anteriormente descrito. Una vez confeccionado el mismo, se procederá a la generación del ejecutable a través de la herramienta *gcc* y por último se ejecutaría en una consola para visualizar su salida. Para mayor comprensión refiérase a la figura 1.



```

root@me-gis:/media/T0N3/proy# ls
r2w.c
root@me-gis:/media/T0N3/proy# gcc -pthread r2w.c -o r2w.exec
root@me-gis:/media/T0N3/proy# ls
r2w.c  r2w.exec
root@me-gis:/media/T0N3/proy# ./r2w.exec
start reading ...
start reading ...
start reading ...
start reading ...
finish reading ...
finish reading ...
start reading ...
finish reading ...
finish reading ...
start reading ...
start reading ...
start reading ...
finish reading ...
finish reading ...
finish reading ...
finish reading ...
start writing ...
finish writing ...
start writing ...
finish writing ...
start reading ...
start reading ...
start reading ...
finish reading ...
finish reading ...
finish reading ...
finish reading ...
start writing ...
finish writing ...
start writing ...
finish writing ...

```

Figura 1: Proceso de compilación y ejecución de la aplicación r2w.exec

Conclusiones

El acceso concurrente a datos compartidos puede crear inconsistencia en los datos. Mantener la consistencia de los datos requiere de mecanismos que aseguren la ejecución ordenada de procesos cooperativos.

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.

Evidentemente los procesos escritores se ven afectados en este sentido y puede darse el caso en que existan lectores requiriendo información que no haya sido insertada por algún escritor, teniendo en cuenta que este ha quedado bloqueado en espera de los restantes lectores. Por tanto una solución más equitativa podría ser aquella en la que ambas subrutinas tienen la misma prioridad, sin embargo esto depende del contexto en que se requiera aplicar, y perfectamente este modelo de solución puede ser factible para determinados entornos, en los que la actualización de los datos no sea tan relevante o frecuente.

Bibliografía

- ✓ Tanenbaum, A. S. (2003). *Sistemas Operativos Modernos*. Mexico: PEARSON EDUCACION.
- ✓ Downey, A. B. (2008). *The little Book of Semaphores*.