Instituto Politécnico Nacional

Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas

Sistemas Operativos

Unidad I:

Estructura de los sistemas operativos

Exclusión mutua

M. en C. Hermes Francisco Montes Casiano
hermes.upiicsa@gmail.com

Índice

Condiciones de competencia

- Los procesos que trabajan juntos comparten con frecuencia un espacio en común para almacenamiento (leer o escribir).
- El espacio compartido puede estar en memoria principal o puede ser un archivo compartido.
- A situaciones en la que mas de un proceso lee o escribe en ciertos datos compartidos y el resultado final depende de quién ejecute qué y en que momento se les denomina **condiciones de competencia**.

Índice

Introducción

- Se le denomina región critica al conjunto de instrucciones que trabajan (leer/escribir) con un recurso compartido, mismo que es motivo de una codición de competencia.
- Es necesario definir un mecanismo para prohibir que más de un proceso lea o escriba en los datos compartidos de forma concurrente.
- La **exclusión mutua** es una forma de garantizar que si un proceso utiliza una región de memoria o archivo compartido, los demás procesos no puedan utilizarlo.
- La elección de operaciones primitivas compartidas con el fin de obtener exclusión mutua es uno de los temas centrales en todo SO.

Exclusión mutua

- Si se logra que no ocurra que dos procesos se encuentren en su región crítica al mismo tiempo, se evitarían las condiciones de competencia.
- La solución debe cumplir con las siguientes condiciones:
  1. Dos procesos no deben encontrarse al mismo tiempo dentro de sus secciones críticas.
  2. No se deben hacer hipótesis sobre la velocidad o el número de CPU.
  3. Ninguno de los procesos que estén en ejecución fuera de su sección crítica puede bloquear a otros procesos.
  4. Ningún proceso debe esperar eternamente para entrar en su sección crítica.

Exclusión mutua

Espera ocupada

- La exclusión mutua con espera ocupada matiene a un proceso en espera (consumiendo tiempo de procesador) para entrar a la región crítica. Por ejemplo, un ciclo hasta que la región crítica sea liberada.
- Los enfoques más comunes que permiten la exclusiónmutua con espera ocupada son:
  - Desactivación de las interrupciones
  - Variables de cerradura
  - Alternancia estricta
  - Solución de Peterson

Desactivación de interrupciones

- La solución más simple es hacer que cada proceso desactivara todas sus interrupciones justo antes de ingresar a la región crítica y las activara de nuevo al salir de ella.
- No es correcto debido a que los procesos de usuario no deben tener la facultad de desactivar las interrupciones.

Problema!!!!

Si un proceso desactiva las interrupciones y nunca las vuelve a activar sería el fin del sistema, ya que nunca se volvería a ejecutar el planificador.

Desactivación de interrupciones

- La solución más simple es hacer que cada proceso desactivara todas sus interrupciones justo antes de ingresar a la región crítica y las activara de nuevo al salir de ella.
- No es correcto debido a que los procesos de usuario no deben tener la facultad de desactivar las interrupciones.

Facultad del núcleo

La desactivación de interrupciones es una tarea que resulta útil pero que sólo puede ser ejecutada por el núcleo.
Imaginemos el caso en el que se ejecuta el planificador y ocurre una interrupción de hardware, podrían surgir condiciones de competencia y la lista de procesos listos podría quedar en un estado inconsistente.

Exclusión mutua

Variables de cerradura

- Es una solución a nivel de software.
- Una variable de cerradura es una variable compartida (con valor unicial 0).
  - Si un proceso ingresa a la region crítica y la variable es 0, le asigna un 1 al candado.
  - Si el candado es 1, el proceso espera hasta que el candado vuelve a ser 0. Así, un 0 significa que ningún proceso está en su región crítica y un 1 significa que algún proceso está en su región crítica.

Exclusión mutua

Variables de cerradura

- Las variables de cerradura tienen el inconveniente del ejemplo del directorio *spooler*.
  1. Un proceso lee el candado y verifica que es 0.
  2. Antes de que el proceso pueda asignar 1 al candado, se planifica otro proceso, el cual se ejecuta y asigna uno al candado.
  3. Cuando el primer proceso continúa su ejecución, asignará 1 al candado y dos procesos estarán en su región crítica.

Problema

El enfoque ya no cumple con el 1 críterio

Exclusión mutua

Alternancia estricta

- El proceso en espera mantiene una prueba continua del valor de una variable hasta que adquiere algún valor. A dicha espera se le denomida **espera activa**.
- Este enfoque es ineficiente ya qye desperdicia tiempo de CPU.
- La espera activa debe utilizarse cuando exista una espectativa razonable de que la espera será corta.
- Cuando el primer proceso sale de la región crítica, cambia el valor de la variable de turno para que el otro proceso pueda entrar a la región crítica.
- Éste enfoque no es recomendable cuando uno de los procesos es más rápido que el otro.

Exclusión mutua

Alternancia estricta

- Este enfoque es ineficiente ya qye desperdicia tiempo de CPU.
- La espera activa debe utilizarse cuando exista una espectativa razonable de que la espera será corta.
- Cuando el primer proceso sale de la región crítica, cambia el valor de la variable de turno para que el otro proceso pueda entrar a la región crítica.
- Éste enfoque no es recomendable cuando uno de los procesos es más rápido que el otro.

Problema

El enfoque ya no cumple con el 3 críterio

Solución de Peterson

- Peterson descubrio una forma sencilla de lograr exclusión mutua combinando candados y variables de advertencia.
- Cada proceso antes de utilizar las variables compartidas invoca a la función *enter_region* con su número de proceso.
- Despúes de haber utilizado las variables compartidas, el proceso debe invocar *leave_region* para indicar que ya termino y permitir que otro proceso entre si lo desea.
- El funcionamiento sería el siguiente:
  1. El proceso 0 invoca *enter_region*, en indica su interés asignando TRUE a su elemento del arreglo interested (*interested[0]=TRUE*)y asigna turn a 0.
  2. El proceso 1 invoca *enter_region* y permanecera en un ciclo hasta que se asigne FALSE a *interested[0]*, lo cual sucederá hasta que el proceso 0 invoque la función *leave_region*

Exaclusión mutua

Dormir y despertar

- La solución de Petersón(software) y TSL (hardaware) son correctas para garantizar la exclusión mutua, sin embargo tiene el inconveniente de la espera activa.
- La espera ocupada no solo desperdicia tiempo de CPU, sino que tambien puede tener resultados inesperados: **Problema de la Inversión de Prioridad**.
- El problema de la *Inversión de Prioridad* se presenta cuando un proceso con prioridad inferior impide la ejecución de un proceso con mayor prioridad.

Exaclusión mutua

Dormir y despertar

- El enfoque de *dormir y despertar* se basa en un conjunto de primitivas que bloquean a los procesos cuando no pueden ingresar a sus regiones críticas en lugar de desperdiciar tiempo de CPU.
- Las primitivas más comunes son **SLEEP** y **WAKEUP**:
  - SLEEP: es una llamada al sistema que hace que el proceso que la invoca se bloquee, es decir, se suspenda hasta que otro proceso lo despierte.
  - WAKEUP: es una llamada al sistema que despierta a un proceso que se encuentra en estado de bloqueo.
- Las llamadas **SLEEP** y **WAKEUP** pueden tener un parámetro, una dirección de memoria que permite enlazar los *SLEEP* con los *WAKEUP*.

El problema del productor consumidor

- Tambien conocido como el problema del *buffer limitado*.
- Dos procesos comparten un *buffer* de tamaño fijo. Uno de ellos, **el productor**, coloca información en el buffer, y el otro, el **el consumidor**, la saca.
- Surgen problemas cuando el productor quiere colocar un nuevo elemento y no puede por que el buffer está lleno. La solución es que el productor se duerma hasta **el consumidor haya sacado uno o mas elementos**.
- De forma similar, si el consumidor desea sacar un elemento del buffer y este se encuentra vacío, se duerme hasta que el **productor coloca uno o más elementos y lo despierta**.

El problema del productor consumidor

- La variable *count* produce una condición de competencia.
  1. El buffer se encuentra vacío, y el consumidor verifica que *count es 0*, y antes de que se duerma ocurre un cambio de contexto.
  2. El productor agrega un elemento al *buffer* e incrementa *count*, así que asume que antes de agregarlo *count era 0*. Por lo tanto, el consumidor invoca *wakeup* para envíar la señal. Debido a que el proceso consumidor no se había dormido la señal se pierde.
  3. Tarde o temprano el proceso productor llenará el *buffer* y se dormira, por lo tanto ambos procesos dormirán eternamente.

Problema

La escencia del problema es que se perdió la señal enviada para despertar a un proceso que (todavía) no estaba dormido. Si no se perdiera, todo funcionaría!!!!!.

Índice

Introducción

- Dijstra (1965) sugirió usa una variable entera para contar el número de señales de despertar guardadas para su futuro.
- En respuesta se introdijo un tipo de variable llamado **semáfor**.
- Un *semáforo* puede tener el valor 0, indicando que no había señales, o algún valor positivo si había una o más señales de despertar pendientes.
- Dijstra propuso tener dos operaciones, DOWN y UP (generalizaciones de SLEEP y WAKEUP).
- La operacion DOWN(abajo) aplicada a un semáforo verifica:
  - Si el valor es mayor que 0, si es mayor decrementa en uno su valor.
  - Si es igual que 0, el proceso se pone a dormir sin completar la operación DOWN.

Semáforos

- La operación UP incrementa el valor del semáforo direccionado.
- Si hay uno o más procesos bloqueados en espera de ese semáforo, imposibilitados para completar una operación DOWN, el sistema elegirá uno de ellos y completará la operación DOWN.

Operaciones atómicas

Las operaciones DOWN y UP son operaciones indivisibles, lo cual garantiza que una vez que una operación de semáforo se ha iniciado, ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado o bloqueado.

Semáforos

- La operación UP incrementa el valor del semáforo direccionado.
- Si hay uno o más procesos bloqueados en espera de ese semáforo, imposibilitados para completar una operación DOWN, el sistema elegirá uno de ellos y completará la operación DOWN.

Anotación...

Originalmente, Dijstra, utilzó las letras **P** y **V** en lugar de **DOWN** y **UP** respectivamente, ya que el hablaba Holándes y éstas carecen de significado para quienes no hablan ese idioma.

Semáforos

Solución al problema del Productor - Consumidor

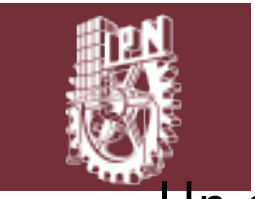
- El método normal consiste en implementar UP y DOWN como llamadas al sistema, para que el sistema operativo inhabilite brevemente todas las interrupciones.
- Las interrupciones sólo se deshabilitan mientras prueba el semáforo, lo actualiza y prone el proceso a dormir.
- Desahabilitar las interrupciones no tiene consecuencias, ya que sólo son unas cuantas instrucciones.
- La solución utiliza 3 semáforos:
  - *full:* utilizado para contar el número de ranuras que están llenas. Inicialmente vale 0.
  - *empty:* se utiliza para contar el número de ranuras que están vacías. Inicialmente es igual al tamaño del buffer.
  - *mutex:* se utiliza para asegurar que el productor y consumidor no acceden al buffer al mismo tiempo. Inicial mente es 1.

Semáforos

Solución al problema del Productor - Consumidor

Exclusión mutua

A los semáforos que inicialmente se les asigna 1 y son utilizados por 2 o más procesos se les denomina **semáforos binarios**. Si cada proceso ejecuta DOWN justo antes de entrar y UP justo despues de salir de su región crítica, la exlusión mutua está garantizada.



Implementación de semáforos

Un semáforo en UNIX se compone de los siguientes elementos:

- El valor del semáforo
- El identificador del último proceso que manipuló el semáforo
- El número de procesos que hay esperando a que el valor se incremente
- El número de procesos que esperando a que el semáforo tome el valor 0

Operaciones

Las llamadas relacionadas con los semáforos son: **semget**, para crear un semáforo o habilitar el acceso a uno ya existente; **semctl**, para realizar operaciones de control e inicialización; y **semop**, para realizar operaciones down y up.

Petición de semáforos

Con **semget** podemos crear o acceder a un conjunto de semáforos unidos bajo un mismo identificador común:

Cabeceras

#include< *sys*/*types.h* >
#include< *sys*/*ipc.h* >
#include< *sys*/*sem.h* >
int semget(key_t key, int nsems, int semflg)
Donde:

- *key*: llave que indica el grupo de semáforos a acceder.
- *nsems*: número de semáforos que estan agrupados bajo el mismo identificador devuelto.
- *semflg*: es una máscara de bits que indican el modo de adquisición del identificador.

Petición de semáforos

- Si la llamada funciona correctamente, devolverá un identificador con el que podremos acceder a los semáforos en sucesivas llamadas.
- Si algo falla, **semget** devuelve el valor -1 y en errno estará el código del error producido.

Comandos útiles

ipcs : lista los recursos compartidos existentes en un momento dado, listado de memorias compartidas (semáforos, colas de mensajes, etc).

ipcrm : permite eliminar alguno de esos recursos

Control de las estructuras de un semáforo

Con **semctl** podremos acceder a la información administrativa y de control que dispone el núcleo sobre un semáforo:

Cabeceras

#include< *sys*/*types.h* >
#include< *sys*/*ipc.h* >
#include< *sys*/*sem.h* >
int semctl(int semid, int semnum, int cmd, arg)

Control de las estructuras de un semáforo

- La llamada actua sobre un conjunto de semáforos que responden al identificador semid devuelto por una llamada precia a semget. El parámetro semnum indica cuál es el semáforo, de los que hay bajo semid, al que queremos acceder.
- La operación de control que se realizará sobre el semáforo viene indicada en cmd(Ver descripción de funciones).
- Si la llamada a semctl se realiza satisfactoriamente, la función devuelve un número cuyo significado dependerá del valor de cmd. Los valores significativos son: GETVAL,GETNCNT,GETZCNT o GETPID. Para cualquier otro valor ed cmd, la función devuelve 0 cuando se ejecuta satisfactoriamente o -1 en caso de error.

Operaciones *Down* y *Up*

Permiterelizar las operaciones Down y Up con los semáforos del UNIX System V, tenemos que usar la llamada semop:

Prototipo de la función

```
#include< sys/types.h >
#include< sys/ipc.h >
#include< sys/sem.h >
int semop(int semid, struct sembuf *sops, int nsops);
```

Operaciones *Down* y *Up*

- Esta llamada realiza operaciones atómicas sobre los semáforos que hay asociados bajo el identificador *semid*.
- *sops* es un puntero a un arrar de estructuras que indican las operaciones que se llevarán sobre los semáforos y *nops* es el total de elementos que tiene el array de operaciones.
- Cada elemento del array es una estructura del tipo **sembuf** que se define como sigue:

```
/* semop system call takes an array of these */
struct sembuf {
        ushort  sem_num;        /* semaphore index in array */
        short   sem_op;         /* semaphore operation */
        short   sem_flg;        /* operation flags */
};
```

Índice

Memoria Compartida

- La forma más rápida de comunicar dos procesos es hacer que compartan una zona de memoria.
- Para enviar datos de un proceso a otro sólo basta con escribir en la memoria compartida y esos datos estarán disponibles para que los lea cualquier proceso.
- La memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtuales es local a ese proceso y cualquier intento de direccionar a esa memoria desde otro proceso provocará una violación de segmento.

Zonas de memoria compartida

Para solucionar éste problema, los sistemas UNIX brindan la posibilidad de crear zonas de memoria con la característica de poder ser direccionadas por varios procesos simultáneamente.

Implementación de memoria compartida

- El espacio de direcciones de ésta memoria es virtual por lo que sus direcciones físicas asociadas podría variar con el tiempo.
- El sistema operativo es el encargado de realizar la traducción de direcciones físicar a virtuales y viceversa.
- Las llamadas para manipular la memoria compartida son:
  - *shmget*, para crear una zona de memoria compartida o habilitar el acceso a una ya creada.
  - *shmctl*, para acceder y modificar la información administrativa y de control que el núcleo le asocia a cada zona de memoria compartida.
  - *shmat*, para unir una zona de memoria compartida a un proceso
  - *shmt*, para separar una zona previamente unida.