

Tarea 4

Hernández Sánchez Juan René

2016630187

Sistemas de Sincronización de Procesos.

Al momento de tener procesos cooperativos nace la necesidad de tener comunicación entre esos procesos, pero al momento de hacer esto, esos procesos empiezan a luchar por los recursos y archivos y es necesario implementar un Sistema para poder administrar y Sincronizar esos procesos para evitar inconsistencias en los datos.

En el peor de los casos podría haber interbloqueos (**abrazos de la muerte**) y parar por completo el sistema.

Uno de los problemas son las condiciones de carrera, esto pasa cuando dos o más procesos están leyendo ó escribiendo datos compartidos y el resultado depende de quién se ejecuta y exactamente cuándo lo hace.

Para evitar las condiciones de carrera es buscar una manera de prohibir que más de un proceso lea y escriba datos compartidos al mismo tiempo. Es decir **exclusión mutua**, significa que mientras un proceso está bloccando, leyendo ó escribiendo ciertos recursos compartidos los demás procesos se excluyen, es decir, no ingresan y no interfieren hasta que el proceso termine de usar esa sección,

Esta parte del programa donde pueden surgir condiciones de carrera se le llama **Sección Crítica ó Región Crítica**.

Aunque evitar que más de un proceso ingrese a la Sección Crítica al mismo tiempo ayuda a evitar las condiciones de carrera, no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Se necesitan cumplir 4 condiciones para tener una buena solución:

Tarea 4

Hernández Durley Jair René

2016630187

- 1) No puede haber más de un proceso de manera simultánea dentro de sus regiones críticas. (mutua exclusión)
- 2) No puede hacerse suposiciones acerca de las velocidades o el número de CPUs.
- 3) Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos. (Proceso Simplificado)
- 4) Ningún proceso tiene que esperar para siempre para entrar a su región crítica. (Espera Limitada Simplificada)

Algunos autores marcan cierta estructura de programa para el uso y manejo de secciones críticas:

Repeat

 Sección de ingreso

 Sección Crítica

 Sección de egreso

 Sección Restante

Until false;

- La Sección de ingreso se encarga de solicitar permiso para ingresar a su sección crítica.
- La Sección de egreso pueden ser algunos protocolos para el correcto manejo del algoritmo.
- La Sección Restante es el resto del código que no interfiere con la Sección Crítica.

Con este modelo se agregan 2 condiciones más para la correcta implementación:

Progreso: Si ningún proceso se está ejecutando en su sección crítica y hay procesos que desean ingresar a sus secciones críticas, solo los que **No** se estén ejecutando en su sección Restante podrán participar en la decisión de cuál proceso será el siguiente en ingresar en su sección crítica.

Espera Limitada: Hay un **Límite** para el # de veces que se permite a otros procesos ingresar en sus secciones críticas **después** de que un proceso ha solicitado ingresar en su sección crítica y **antes** de que se le otorgue la autorización para hacerlo.

Tarea 4

400/150/91

Hernández Sánchez Juan René

2016630182

Existen varias formas de lograr la Exclusión mutua:

Exclusión mutua con espera ocupada.

Deshabilitando interrupciones: En sistemas con un solo procesador lo más simple es hacer que cada proceso deshabilite todas las interrupciones justo después de entrar a su región crítica y las rehabilite justo después de salir. Normalmente esta técnica es útil dentro del mismo Sistema Operativo, pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

VARIABLES DE CANDADO: Se tiene una sola variable compartida (de candado), que al principio es \emptyset . Cuando un proceso desea entrar a su región crítica primero evalúa el candado.

Si el candado es \emptyset , el proceso lo fija en 1 y entra a la región crítica. Si el candado ya es 1 sólo espera hasta que el candado se haga \emptyset . Aunque este mismo sistema genera problemas de incongruencia de datos y condiciones de carrera.

Alternancia estricta: Este método hay una variable que lleva la cuenta acerca de qué proceso le tocó entrar a su región crítica y examinar si actualizar la memoria compartida.

Espera ocupada → Acción de evaluar en forma continua una variable hasta que aparezca cierto valor.

Candado de giro → Candado que se utiliza para manejar la espera ocupada.

Este método hace que el CPU tenga momentos de ocio además que viola el principio 3, donde un proceso está bloqueado porque no quiere entrar a la región crítica.

Hay muchos otros algoritmos como: el algoritmo de Dekker, la solución de Peterson, la instrucción TSL, XCHG, bit de Donor y Despotor, entre otros.

Mecanismos de IPC y Sincronización en POSIX y System V

Estos mecanismos están implementados como una unidad y comparten características comunes, entre las que están:

- Cada mecanismo tiene una **tabla** cuyas entradas describen el uso que se hace del mismo.
- Cada entrada de la tabla tiene una **llave** numérica elegida por el Usuario, normalmente generada por alguna función estándar.
- Cada mecanismo dispone de una llamada **get** para crear una entrada nueva ó recuperar alguna existente.
- El núcleo busca dentro de la tabla alguna entrada que se ajuste a la llave suministrada.
- Si la llave toma el valor de **IPC_PRIVATE** el núcleo ocupa la primera entrada hasta que la liberemos.
- Si dentro de la máscara está activo el bit **IPC_CREATE**, el núcleo crea una nueva entrada en caso de que no haya ninguna que responda a la llave suministrada.
- Si además de **IPC_CREATE** está activo el bit **IPC_EXCL**, el núcleo devuelve un error en caso de que ya exista una entrada para la llave suministrada.
- Si todo funciona de manera correcta, el núcleo devuelve un **descriptor** que se podrá usar en otras llamadas.
- Para cada mecanismo **IPC**, el núcleo aplica la siguiente fórmula para calcular el índice de acceso a la tabla

$$\text{IndiceTabla} = \frac{\text{Número_descriptor}}{\text{Número de entradas en la tabla}} \times 100$$

- Cada entrada en la tabla tiene un registro de permisos que incluye:
 - ID de usuario y de grupo del proceso que ha reservado la entrada.
 - ID de usuario y grupo modificados por la llamada de control del mecanismo.
 - Conjunto de bits con los permisos de Lectura, escritura y ejecución.
 - El grupo y otros usuarios.
- Cada entrada contiene información de estado, en la que se incluye el identificador del último proceso que ha utilizado la entrada.
- Cada mecanismo de IPC tiene una llamada de control que permite leer y modificar el estado de una entrada reservada y también permite liberarla.

Semáforos :

Fueron propuestos por Edsger Dijkstra en 1965. Es una variable entera para contar el número de señales de despertar, guardadas para un uso futuro.

Puede tener un valor cero, indicando que no se guardaron señales de despertar ó algún valor positivo si estuvieran pendientes una o más señales de despertar.

Este mecanismo previene la colisión que se produce cuando dos o más procesos soliciten simultáneamente el uso de un recurso compartido.

Este objeto de tipo entero puede hacer **Dos operaciones atómicas estándar:**

espera (wait) P & señal (Signal) V
 down ó sleep up ó wakeup.

Su definición clásica es:

espera (S): while $S \leq 0$ do nada; Señal (S): $S := S + 1;$
 $S := S - 1;$

Las modificaciones del valor entero del Semáforo (S) en las operaciones **esperar** y **señal** se deben hacer de manera **Irreversible**. Esto significa que, ningún otro proceso puede modificar simultáneamente el valor de ese mismo semáforo. Además cada operación debe ejecutarse sin interrupciones.

En otras palabras:

espera (S): comprueba si el valor es mayor que cero, de ser así, disminuye el valor (utiliza una señal de despertar almacenada) y continúa.

Si el valor es cero, el proceso se pone a dormir sin completar la operación.

señal (S): incrementa el valor del semáforo almacenado. Si uno o más procesos estaban inactivos en ese semáforo, sin poder completar la operación anterior, el sistema selecciona uno al azar y permite que complete su operación.

Semáforos Binarios: Son semáforos inicializados en 1 y son utilizados por dos o más procesos para asegurar que sólo uno de ellos puede entrar a su región crítica. y sólo puede tomar dos valores.

Se pueden usar los semáforos para cumplir la exclusión mutua y para sincronizar los procesos.

Toda esta implementación que dice **Dijkstra** es la que está implementada en los semáforos de **System V de UNIX** de forma generalizada. También se pueden realizar operaciones de **P y V** que actualizan de forma átomo. Todos los semáforos asociados bajo un mismo identificador.

Semáforo UNIX System V

Se compone de los siguientes elementos:

- El valor del semáforo
- El identificador del último proceso que manipula el semáforo
- El número de procesos que hay esperando a que el valor del S. incremente,
- El número de procesos que hay esperando a que el semáforo tome valor 0.

Sus llamadas son:

Semget: para crear un semáforo o habilitar el acceso a uno ya existente.

Semctl: para realizar operaciones de control e inicialización.

Semop: para realizar operaciones P y V sobre el semáforo.

Petición de Semáforos (Semget)

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

Si se llama de manera correcta devuelve un identificador con el que podremos acceder a los semáforos en sucesivas llamadas. Si falla da -1.

Key - es la llave que indica a qué grupo de semáforos queremos acceder.

nsems - es el total de semáforos que van a estar agrupados bajo el ID devuelto por **semget**

semflg - es una máscara de bits que indican el modo de adquisición del ID. algunos de sus valores son:

0400 - Permiso de lectura para el usuario

0200 - Permiso de modificación para el usuario

0060 - Permiso de lectura y modificación para el grupo

0006 - Permiso de lectura y modificación para otros usuarios.

* Son iguales que los Bits de Open.

El ID devuelto por `semget` es heredado por los procesos hijos.

Control de las estructuras de semáforo (`semctl`)

#include "Los mismos que `semget()`"

int `semctl(int semid, int semnum, int cmd, arg)`

union `semun {`

int val;

struct `semid_ds *buf;`

ushort *array;

`} arg;`

Semid - es el ID devuelto por la llamada `semget`.

Semnum - indica cuál es el semáforo, de los que hay bajo `semid`, al que queremos acceder.

cmd - son las operaciones de control que se realizarán sobre el semáforo.

GETVAL - Leer el valor de un semáforo. ▲

SETVAL - Inicializa un semáforo con los valores dentro de `arg`

GETPID - Lee el último PID del último proceso que actuó sobre el semáforo. ▲

GETNCNT - Lee el número de procesos que hay esperando a que se incremente el valor del semáforo ▲

GETZCNT - Lee el número de procesos que hay esperando a que el semáforo tome el valor de cero. ▲

GETALL - Lee el valor de todos los semáforos asociados al `semid`. Estos valores se almacenan en `arg`.

SETALL - Inicializa el valor de todos los semáforos asociados al `semid`. Los valores deben de estar en `arg`.

IPC_STAT e **IPC_SET** - permiten leer y modificar la información administrativa asociada al `semid`.

IPC_RMID - Le indica al núcleo de borrar el conjunto de semáforos agrupados en `semid`. No tendrá efecto si hay algún proceso usando el semáforo.

Operaciones P y V (Semop)

#include "los mismos que semget y semctl"
int semop(int semid, struct sembuf *sops, int nsops)

Esta llamada realiza operaciones atómicas sobre los semáforos que hay asociados bajo **semid**.

Sops - es un puntero a un array de estructuras que indican las operaciones que llevarán a cabo sobre los semáforos.

nsops - es el número de elementos que tiene el array de operaciones.

struct sembuf {

 ushort sem_num; // número del semáforo
 short sem_op; // Operación; incrementar o decrementar
 short sem_flg; // Máscara de bits.

? i

Sem-num → número de semáforos, de 0 ... N-1 (como un array)

Sem-op → operación a realizar sobre el semáforo especificado en **Sem-num**
 si es < 0 decrementa (**P**), si es > 0 incrementa (**V**), si vale 0 no hace nada.

La operación **V** no hay problema, ya que un semáforo puede siempre tomar valores positivos.

La operación **P** no siempre se puede realizar ya que no puede tener valores negativos, en este caso, dependiendo del valor de **sem_flg** responderá de diferente forma.

Sem_flg → dependiendo de la máscara de bits el semáforo reaccionará de diferentes maneras a ciertos sucesos:

- **IPC-NOWAIT** - La llamada a semop devuelve el control en caso de que no se pueda satisfacer la operación especificada en **sem-op**.

- **IPC-WAIT** - Es la forma de trabajar por defecto del semáforo.

- **SEM_UNDO** - La operación se deshace cuando el proceso termina.

Este bit ↑ previene contra el bloqueo accidental de semáforos. Ya que algún proceso puede acabar de forma anormal, el núcleo se encarga de actualizar el valor del semáforo y deshace las operaciones realizadas sobre él cuando termina el proceso.

Semáforos **MUTEX**

Un semáforo **MUTEX** (Mutual exclusion) es una versión simplificada de un semáforo de Dijkstra (UNIX System V). Son buenas para administrar la exclusión mutua para cierto recurso compartido o pieza de código. Son especialmente útiles al usar paquetes de hilos (threads) que se implementan en espacio de usuario.

Ya hemos hablado de estos semáforos, pero con el nombre de **Semáforos binarios**. Un **MUTEX** es una variable que puede estar en Dos estados: **Abierto** o **Desbloqueado** o **Cerrado** o **Bloqueado**.

Solo se requiere de un solo bit para representarlo, pero normalmente se usa un entero, Si vale 0 está abierto, cualquier otro valor está cerrado. La implementación básica de un **MUTEX** es:

repeat

espera (mutex);

Sección Crítica

señal (mutex);

Sección restante.

until false;

Se utilizan dos procedimientos con los mutex.

Cuando un hilo o proceso necesita acceso a una sección crítica, llama a **mutex_lock**.

Si el mutex está abierto (o decir que la sección crítica está libre), la llamada tiene éxito y puede entrar a la sección crítica.

En caso contrario, el hilo o proceso se

queda bloqueado hasta que el proceso que estaba en la sección crítica termine y llame a **mutex_unlock**.

Si se bloquean varios procesos por el mutex, se selecciona uno al azar y se permite que adquiera el mutex.

La API para manejo de MUTEX con hilos es **pthreads**.

Pthreads proporciona varias funciones que pueden utilizarse para sincronizar los hilos. Algunas de sus llamadas a funciones son:

#include <pthread.h>

pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

Crea un mutex con las características de attr. por defecto attr=NULL.

pthread_mutex_destroy(pthread_mutex_t *mutex);

Destruye o más bien deja indefinido el mutex, (no inicializado).

pthread_mutex_lock(pthread_mutex_t *mutex);

Bloquea el mutex de llamada. Si ya estaba bloqueando se bloquea el proceso o hilo hasta que este disponible.

pthread_mutex_trylock(*mutex);

Funciona igual que el anterior pero si está bloqueando por otro hilo, entonces retorna el control al proceso, no se queda bloqueado.

pthread_mutex_unlock(*mutex);

Liberá el objeto mutex al que hace referencia.

para mas información están las especificaciones en internet.

Semáforos en POSIX.

La norma POSIX define de manera muy parecida a Dijkstra sobre el uso e implementación de semáforos.

La norma distingue entre **semáforos nominales o con nombre** y los **semáforos anónimos**.

Los semáforos **nominales (con nombre)** se utilizan para sincronizar procesos no relacionados y se les asocia un nombre en la jerarquía del sistema de ficheros, muy parecido a los sistemas FIFO.

Los semáforos **anónimos** permiten sincronizar procesos que forman parte de una misma jerarquía o hilos de un mismo proceso,

Semáforo NOMINAL

#include <semaphore.h>

```
sem_t *Sem_open(const char *name, int oflag, [mode_t mode,  
unsigned value]);
```

name - ruta que sirve como nombre del semáforo.

oflag - máscara de bits que indica el modo de apertura.

mode - es una máscara con los permisos de Lectura/escritura del semáforo.

value - es el valor de inicialización el cual es value <SEM_VALUE_MAX,

Si la llamada es satisfactoria devuelve el valor de Dirección del semáforo creado, si no, devuelve SEM_FAILED y escribe en errno.

#include <semaphore.h>

Semáforo ANÓNIMO

```
int sem_init(sem_t *sem, int pshared, unsigned value);
```

sem - referencia al semáforo

value - valor de inicialización

pshared - si es = φ el semáforo será compartido solo por hilos del proceso

si es ≠ φ sera compartido por procesos emparentados padre-hijo.

Una diferencia interesante es como gestionan los bloques y desbloques los semáforos **POSIX** a los de **UNIX System V**

Posix: ~~funciones de manejo de hilos~~

Bloqueo: Si el semáforo vale \emptyset , la operación produce el bloqueo del hilo que la llama y **lo añade a la cola de hilos en espera**; caso contrario decrementa el valor del semáforo. ~~función de manejo de hilos~~

Desbloqueo: Si hay algún hilo bloqueado esperando en cola, es extraído y desbloqueado; en caso contrario se incrementa el valor del semáforo.

La política de distribución puede ser **SCHED_FIFO, SCHED_RR** entre otras.

Una vez ya creado el semáforo, no importa si es **Nominal o Anónimo** se usan las siguientes funciones:

#include <Semaphore.h>

```
int sem_post(sem_t *sem); // operación P  
int sem_wait(sem_t *sem); // operación V  
int sem_trywait(sem_t *sem); // operación V sin bloqueo.
```

#include <Time.h>

```
int sem_timedwait(sem_t *sem const struct timespec *abs_timeout);
```

esta última intenta realizar el bloqueo hasta que lo consigue o hasta que el reloj alcanza la hora referenciada por `abs_timeout`.

Todas estas funciones devuelven cero si se ejecutan de manera satisfactoria y -1 si dan error.

int sem_getvalue(sem_t *sem, int *sval); // función para consultar el valor del semáforo y lo guarda en `sval`.

Si $>\emptyset$ desbloqueada, si $=\emptyset$ semáforo bloqueado si $<\emptyset$ el absoluto representa el número de hilos esperando a ser desbloqueados.

Para liberar los recursos del semáforo se utilizan las siguientes funciones.

#include <Semaphore.h>

```
int sem_close (sem_t *sem); // cerramos el semáforo creado previamente  
int sem_unlink (const char *name); // borra el nombre en la jerarquía  
int sem_destroy (sem_t *sem); // eliminamos semáforo ANONIMO
```

Estos son solo algunos de los métodos de sincronización de procesos, otros pueden ser colas de procesos, candados o ceraduras POSIX, entre otros.