

1 COMUNICACIÓN ENTRE PROCESOS

Los procesos con frecuencia necesitan comunicarse con otros procesos. Por tanto es deseable tener mecanismos para esa comunicación en una forma bien estructurada y que no utilice interrupciones. En la literatura puede encontrarse el término **comunicación entre procesos** o **IPC** para referirse a estos mecanismos¹.

En la llamada IPC, en general se tienen tres problemas, el primero es cómo un proceso le pasa información a otro. El segundo problema tiene que ver con asegurarse de que dos o más procesos no se estorben mutuamente al efectuar actividades críticas. El tercero se relaciona con la secuencia correcta cuando existen dependencias: Si el proceso A produce datos y el proceso B los imprime, B tiene que esperar hasta que A haya producido algunos datos antes de comenzar a imprimir.

Programación Concurrente

Es el nombre dado a la notación y técnicas de programación utilizados para expresar paralelismo potencial y resolver los problemas de sincronización y comunicación resultantes. Véase [2].

Variables compartidas

Las variables compartidas son objetos a los que más de un proceso tienen acceso; la comunicación por lo tanto, puede proceder con cada proceso referenciando esas variables cuando sea apropiado.

Paso de mensajes

El paso de mensajes involucra intercambio de datos explícito entre dos procesos por medio de un mensaje por medio de un mensaje que pasa de un proceso a otro.

La elección entre variables compartidas y paso de mensajes recae en los diseñadores del lenguaje o del sistema operativo.

Las variables compartidas son fáciles de soportar si hay memoria compartida entre los procesos. Si no es así, aun se pueden usar si el hardware incorpora un medio de comunicación.

Similarmente, una primitiva de paso de mensajes puede ser soportada a través de memoria compartida o una red de paso de mensajes física.

1.1 Condiciones de Competencia

En un programa concurrente, dos procesos que están colaborando podrían compartir cierto almacenamiento común en el que ambos pueden leer y escribir. El almacenamiento compartido puede estar en la memoria principal o puede ser

¹Véase por ejemplo [4]

un archivo compartido; la ubicación de la memoria compartida no altera la naturaleza de la comunicación ni los problemas que surgen.

Aunque las variables compartidas aparecen como una forma directa de pasar información entre procesos, su uso sin restricción no es confiable y es inseguro debido a problemas de actualizaciones múltiples.

Considere dos procesos actualizando una variable compartida X , con la asignación

$$X = X + 1;$$

En la mayoría de los hardware esto no será ejecutado en una operación **indivisible** (atómica), sino que será implementada en tres instrucciones distintas:

- 1) Cargar el valor de X en algún registro (o en la cima del stack);
- 2) Incrementar en uno el valor en el registro; y
- 3) Almacenar el valor en el registro de regreso a X .

Como las tres operaciones no son indivisibles, dos procesos actualizando la variable simultáneamente podrían entrelazar sus acciones y producir un resultado incorrecto. Por ejemplo, si X era originalmente 5, los dos procesos podrían cargar en sus registros e incrementar y entonces almacenar 6.

Condiciones de competencia/carrera

A las situaciones en las que dos o más procesos leen o escriben datos compartidos y el resultado final depende de quién se ejecuta precisamente cuándo, se denominan **condiciones de competencia**. También se puede encontrar para estas situaciones, el término **condiciones de carrera**, véase por ejemplo [3].

Depurar programas que tienen condiciones de carrera no es algo sencillo. Los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.

¿Qué ocasiona las condiciones de carrera?

La dificultad mencionada en el ejemplo de la variable compartida X ocurrió debido a que un segundo proceso, digamos proceso B empezó a utilizar una variable compartida X antes de que el primer proceso, digamos el proceso A terminara de trabajar con ella.

El problema de evitar las condiciones de carrera se puede formular como sigue: parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no producen condiciones de carrera. Sin embargo, algunas veces un proceso tiene que acceder a la memoria compartida o a archivos compartidos, o hacer cosas críticas que pueden producir carreras. Esa parte del programa en la que se accede a la memoria compartida se conoce como región crítica o 'bf sección crítica. Si pudieramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria compartida, los archivos compartidos y cualquier otro recurso compartido es buscar alguna manera de evitar que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Dicho en otras palabras, lo que necesitamos es **exclusión mutua**, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo.

1.1.1 Exclusión mutua con espera activa

Existen varios enfoques para lograr la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso pueda entrar a su propia región crítica y ocasionar problemas.

Inhabilitación de interrupciones

La solución más sencilla es hacer que cada proceso inhabilite las interrupciones justo después de ingresar en su región crítica y vuelva a habilitarlas justo antes de salir de ella. Con las interrupciones inhabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de interrupciones de reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a ningún otro proceso. Así, una vez que un proceso ha inhabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor a que otro proceso intervenga.

Este enfoque casi nunca resulta atractivo, porque no es prudente conferir a los procesos de usuario la facultad de desactivar las interrupciones. Supongamos que uno de ellos lo hiciera, y nunca habilitara las interrupciones otra vez. Esto podría terminar con el funcionamiento del sistema. Además si el sistema es multiprocesador, con dos o más CPU, la inhabilitación de las interrupciones afectaría solo a la CPU que ejecutara la instrucción de inhabilitación; las demás seguirían con las interrupciones habilitadas y podrían acceder a la memoria compartida.

Por otro lado, en muchos casos es necesario que el kernel mismo inhabilite las interrupciones durante unas cuantas instrucciones mientras actualiza variables o listas. Si ocurriera una interrupción en un momento en que la lista de procesos listos, por ejemplo, está en un estado inconsistente, ocurrirían condiciones de competencia. La conclusión es: la inhabilitación de interrupciones suele ser una técnica útil dentro del sistema operativo mismo pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

En la búsqueda de soluciones a los problemas de exclusión mutua se han propuesto dos tipos de enfoque: soluciones no usan ayuda del hardware y soluciones que si usan ayuda del hardware. Las primeras pretenden ser más portables y se conocen como soluciones software, mientras que las segundas pretenden ser más simples. Exploraremos superficialmente primero el tipo de solución software y después el tipo de solución que si usa ayuda del hardware.

Variables de candado

Supongamos que tenemos una sola variable (de candado/lock) compartida cuyo valor inicial es 0. Cuando un proceso quiere entrar en su región crítica, lo primero que hace es probar el candado. Si el candado es 0, el proceso le asigna 1 y entra en su región crítica; si 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.

Desafortunadamente, esta idea contiene exactamente el mismo defecto fatal que vimos en el ejemplo de $X = X + 1$. Supongamos que un proceso lee el candado y ve que es 0. Antes de que este proceso pueda asignar 1 al candado, se planifica otro proceso, el cual se ejecuta y asigna 1 al candado. Cuando el primer proceso continúa su ejecución, asignará 1 al candado, y dos procesos estarán en su región crítica al mismo tiempo.

Podría pensarse que este problema puede superarse leyendo primero el valor del candado, y verificándolo otra vez justo antes de guardarle el 1 en él, pero esto no sirve de nada. La competencia ocurriría entonces si el segundo proceso modifica el candado justo después de que el primer proceso terminó su segunda verificación.

Alternancia estricta

Un algoritmo de alternancia estricta se muestra a continuación

<pre>while(TRUE){ while(turn!=0);/* esperar */ critical_region_0(); turn=1; noncritical_region_0(); }</pre>	<pre>while(TRUE){ while(turn!=1);/* esperar */ critical_region_1(); turn=0; noncritical_region_1(); }</pre>
(a)	(b)

La variable interna **turn**, que inicialmente es 0, indica a quien le toca entrar en la región crítica y examinar o actualizar la memoria compartida. En un principio, el proceso 0 inspecciona **turn**, ve que es 0, y entra en su región crítica. El proceso 1 también ve que **turn** es 0 y se mantiene en un ciclo corto probando **turn** continuamente para detectar el momento en que cambia a 1. Esta prueba continua de una variable hasta que adquiere algún valor se denomina **espera activa**,² y normalmente debe evitarse, ya que desperdicia tiempo de CPU. La espera activa solo debe usarse cuando exista una expectativa razonable de que la espera será corta.

Cuando el proceso 0 sale de la región crítica, asigna 1 a **turn**, a fin de que el proceso 1 pueda entrar en su región crítica. Supongamos que el proceso 1 termina su región crítica rápidamente, de modo que ambos procesos están en sus regiones no críticas, y **turn** vale 0. Ahora el proceso 0 ejecuta su ciclo completo rápidamente, regresando a su región no crítica y regresa al principio

²En el ámbito de la programación en lenguaje ensamblador se conoce con el nombre de polling.

de su ciclo después de haber asignado 1 a **turn**. Luego el proceso 0, termina su región no crítica y regresa al principio de su ciclo. Desafortunadamente, no puede entrar en su región crítica porque **turn** es 1 y el proceso 1 está ocupado en su región no crítica. Dicho de otro modo, la alternancia de turnos no es una buena idea cuando un proceso cuando un proceso es mucho más lento que el otro. Así pues, el proceso 0 está siendo bloqueado por un proceso que no está en su región crítica. De hecho, esta solución requiere que los dos procesos se alternen estrictamente en el ingreso a sus regiones críticas.

La instrucción TSL

Ahora examinaremos una propuesta que requiere un poco de ayuda del hardware. Muchas computadoras, sobre todo las diseñadas pensando en múltiples procesadores, tienen una instrucción **TEST AN SET LOCK** (TSL, probar y fijar Lock) que funciona como sigue. La instrucción lee el contenido de la palabra en memoria, lo coloca en un registro y luego almacena un valor distinto de cero en esa dirección de memoria. Se garantiza que las operaciones de leer la palabra y guardar el valor en ella son indivisibles; ningún otro procesador puede acceder a la palabra de memoria en tanto la instrucción no haya terminado. La CPU que ejecuta la instrucción TSL pone un candado al bus de memoria para que ninguna otra CPU pueda acceder a la memoria en tanto no termine.

Para usar la instrucción TSL creamos una variable compartida **lock** a fin de coordinar el acceso a la memoria compartida. Cuando **lock** es 0, cualquier proceso puede asignarle 1 usando la instrucción TSL y luego leer o escribir la memoria compartida. Cuando el proceso termina, asigna otra vez 0 a **lock** usando una instrucción **MOVE** ordinaria.

¿Cómo podemos usar esta instrucción para evitar que dos procesos entren simultáneamente en sus regiones críticas? La solución se da a continuación:

```
enter_region:
    TSL register,lock      |copiar lock en register y asignarle 1
    CPM register,#0        |?'era lock 0?
    JNE enter_region       |si no era cero, se asigno 1 a lock y se ejecuta ciclo
    ret                   |volver al invocador; se entro en la region critica

leave_region:
    MOVE lock,#0           |guardar un 0 en lock
    ret                   |volver al invocador
```

enter_region es una subrutina de cuatro instrucciones escrita en un lenguaje ensamblador ficticio (pero típico). La primera instrucción copia el valor antiguo de **lock** en el registro y luego asigna 1 a 'tt **lock**. Luego se compara el valor antiguo con 0. Si es distinto de 0, el candado ya estaba establecido, así que el programa simplemente vuelve al principio y lo prueba otra vez. Tarde o temprano el valor de **lock** será 0 (cuando el proceso que actualmente está en su región crítica termine lo que está haciendo dentro de dicha región) y la subrutina

regresará, con el candado establecido. Liberar el candado es sencillo, pues basta con almacenar 0 en `lock`. No se requieren instrucciones especiales.

Ya tenemos una solución al problema de la región crítica que es directa. Antes de entrar en su región crítica un proceso invoca `enter_region`, la cual realiza espera activa hasta que el candado está libre; luego adquiere el candado y regresa. Después de la región crítica el proceso invoca `leave_region`, que almacena un 0 en `lock`.

Implementación en xv6

En el archivo `x86.h` de xv6 encontramos la función

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) :
                 "cc");
    return result;
}
```

También en el archivo `spinlock.c` se tiene la implementación de spinlocks (locks de giro o candados de giro) para exclusión mutua.

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");

    // The xchg is atomic.
    // It also serializes, so that reads after acquire are not
    // reordered before it.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Record info about lock acquisition for debugging.
    lk->cpu = cpu;
```

```

    getcallerpcs(&lk, lk->pcs);
}

// Release the lock.
void
release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    // The xchg serializes, so that reads before release are
    // not reordered after it. The 1996 PentiumPro manual (Volume 3,
    // 7.2) says reads can be carried out speculatively and in
    // any order, which implies we need to serialize here.
    // But the 2007 Intel 64 Architecture Memory Ordering White
    // Paper says that Intel 64 and IA-32 will not move a load
    // after a store. So lock->locked = 0 would work here.
    // The xchg being asm volatile ensures gcc emits it after
    // the above assignments (and after the critical section).
    xchg(&lk->locked, 0);

    popcli();
}

```

Una nota sobre la instrucción **XCHG** de los microprocesadores Intel. Véase [1].

XCHG

La instrucción de intercambio (**XCHG**) intercambia el contenido de un registro con el contenido de cualquier otro registro o una localidad de memoria. La instrucción **XCHG** no se puede ejecutar en registros de segmento ni con datos de memoria a memoria. Los intercambios son de tamaño byte, palabra o doble palabra (solo en 80386/80486 o superiores). En la siguiente tabla aparecen las formas de la instrucción **XCHG**

Simbólica	Funciones
XCHG reg,reg	Intercambia registros de byte, palabra y doble palabra
XCHG reg,mem	Intercambia datos en la memoria byte, palabra o doble palabra, con datos del registro.

Dormir y despertar

Tanto las soluciones software como las que usan TSL son correctas pero ambas tienen el defecto de requerir espesra activa. En esencia, lo que estas soluciones hacen es lo siguiente: cuando un proceso desea entrar en su región crítica verifica

si está permitida la entrada; si no, el proceso simplemente repite un ciclo corto esperando a hasta que lo esté.

Este enfoque no solo desperdicia tiempo de CPU, sino que también puede tener efectos inesperados. Consideremos una computadora con dos procesos, H de alta prioridad, y L de baja prioridad. Las reglas de planificación son tales que H se ejecuta siempre que está en el estado listo. En un momento dado, con L en su región crítica, H queda listo para ejecutarse (p.ej. se completó una operación de E/S). H inicia ahora la espera activa, pero dado que L nunca se planifica mientras H se está ejecutando, L nunca tiene oportunidad de salir de su región crítica, y H permanece en un ciclo infinito. Esta situación se conoce como **problema de inversión de prioridad**. Véase [4] pag. 64.

Examinemos ahora algunas primitivas de comunicación entre procesos que se bloquean (se duermen) en lugar de desperdiciar tiempo de CPU cuando no se les permite entrar en sus regiones críticas. Una de las más sencillas es el par **SLEEP** y **WAKEUP**. **SLEEP** (dormir) es una llamada al sistema que hace que el invocador se bloquee, es decir, se suspenda hasta que otro proceso lo despierte. La llamada **WAKEUP** (despertar) tiene un parámetro, el proceso que se debe despertar. Como alternativa, tanto **SLEEP** como **WAKEUP** pueden tener un parámetro cada uno, una dirección de memoria que sirve para enlazar los **SLEEP** con los **WAKEUP**.

El problema productor-consumidor

Como ejemplo de uso de estas primitivas, consideremos el problema de **productor-consumidor** (también conocido como problema de **buffer limitado**). Dos procesos comparten un mismo *buffer* de tamaño fijo. Uno de ellos, el productor, coloca información en el *buffer*, y el otro, el consumidor, la saca.

Surgen problemas cuando el productor quiere colocar un nuevo elemento en el *buffer*, pero este ya está lleno. La solución es que el productor se duerma y sea despertado cuando el consumidor haya retirado uno o más elementos. De forma similar, si el consumidor desea sacar un elemento del *buffer* y ve que está vacío, se duerme hasta que el productor pone algo en el *buffer* y lo despierta.

Este enfoque parece muy sencillo, pero da lugar a los mismos tipos de condiciones de carrera que se han mencionado antes. Para seguir la pista al número de elementos contenidos en el *buffer*, necesitaremos una variable, *count*. Si el número máximo de elementos que el *buffer* puede contener es *N*, el código del productor primero verificará si *count* es igual a *N*. Si es así, el productor se dormirá, si no, el productor agregará un elemento e incrementará *count*.

El código del consumidor es similar: primero se prueba *count* para ver si es 0. Si es así, el consumidor se duerme; si no, el consumidor saca un elemento y decrementa *count*. Cada uno de estos procesos verifica también si el otro debería estar durmiendo, y si no es así, lo despierta. El código del productor y del consumidor se muestra en la siguiente figura

```
#define N 100                /* numero de ranuras en el buffer */
int count=0;                 /* numero de elementos en el buffer */

void producer(void){
    while(TRUE){
```



```

        produce_item();          /* generar el siguiente elemento */
        if(count==N)sleep(); /* si el buffer esta lleno, dormir */
        enter_item();           /* colocar elemento en el buffer */
        count=count+1;          /* incrementar la cuenta de elementos */

        if(count==1)wakeup(consumer);
    }
}

void consumer(void){
    while(TRUE){
        if(count==0)sleep(); /* si el buffer esta vacio, dormir */
        remove_item();       /* remover elemento del buffer */
        count=count-1;       /* decrementar la cuenta de elementos */

        if(count==N-1)wakeup(producer);/* estaba lleno el buffer? */
        consume_item();       /* imprimir elemento */
    }
}

```

Volvamos ahora a la condición de carrera. Ésta puede ocurrir porque el acceso a *count* es irrestricto, y podría presentarse la siguiente situación. El 'it buffer está vacío y el consumidor acaba de leer *count* para ver si es 0. En ese instante, el planificador decide dejar de ejecutar el consumidor temporalmente y comenzar a ejecutar el productor. Éste coloca un elemento en el *buffer*, incrementa *count*, y observa que ahora vale 1. Esto implica que antes *count* valía 0, y por ende que el consumidor está durmiendo, así que el productor invoca *wakeup* para despertar al consumidor.

Desafortunadamente, el consumidor todavía no estaba dormido lógicamente, de modo que la señal de despertar se pierde. Cuando el consumidor reanuda su ejecución, prueba el valor de *count* que había leído previamente, ve que es 0 y se duerme. Tarde o temprano el productor llenará el *buffer* y se dormirá. Ambos seguirán durmiendo eternamente.

La esencia del problema aquí es que se perdió una llamada enviada para despertar a un proceso que (todavía) no estaba dormido. Si no se perdiera, todo funcionaría. Una compostura rápida consiste en modificar las reglas y agregar un **bit de espera de despertar** a la escena. Cuando se envía una llamada de despertar a un proceso que está despierto, se enciende este bit. Después, cuando el proceso trata de dormirse, si el bit de espera de despertar está encendido, se apagará pero el proceso seguirá despierto. El bit de espera de despertar actúa como una alcancia de señales de despertar.

Lamentablemente, si existen tres o más procesos, un bit de espera de despertar es insuficiente. Se podría crear otro “parche” y agregar un segundo bit de espera de despertar, o quizá 8 o 32, pero en principio el problema sigue ahí.

Semáforos

En 1965, E. W. Dijkstra introdujo un nuevo tipo de variable llamada **semáforo**.

Un semáforo podría tener el valor 0, indicando que no había señales de despertar guardadas, o algún valor positivo si había una o más señales de despertar pendientes.

Dijkstra propuso tener dos operaciones **DOWN** y **UP** (generalizaciones de **sleep** y **wakeup**, respectivamente). La operación **DOWN** aplicada a un semáforo verifica si el valor es mayor que 0; de ser así, decrementa el valor (esto es, gasta una señal de despertar almacenada) y continúa. Si el valor es 0, el proceso se pone a dormir sin completar la operación **DOWN** por el momento. La verificación del valor, su modificación y la acción de dormirse, si es necesaria, se realizan como una sola acción atómica indivisible. Se 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. Esta atomicidad es absolutamente indispensable para resolver los problemas de sincronización y evitar las condiciones de carrera.

La operación **UP** incrementa el valor del semáforo direccionado. Si uno o más procesos están durmiendo en espera de ese semáforo, imposibilitados de completar una operación **DOWN** previa, el sistema escoge uno de ellos (p.ej. al azar) y le permite completar su **DOWN**. Así, después de un **UP** con un semáforo que tiene procesos durmiendo esperando, el semáforo seguirá siendo 0, pero habrá un proceso menos que se halle en fase de durmiendo esperando. La operación de incrementar el semáforo y despertar un proceso también es indivisible. Ningún proceso se bloquea durante un **UP**.

References

- [1] Barry B. Brey, “Los Microprocesadores Intel, Arquitectura, programación e interfases.” Prentice-Hall 3^a Edición, 1995.
- [2] Alan Burns, Andy Wellings, “Real-Time Systems and Programming Languages”, Addison-Wesley, 3^a Edición, Pearson Education Limited 2001.
- [3] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, “Drivers en Linux, Técnicas y soluciones para el desarrollo de Controladores”, O’Reilly Anaya Multimedia, 2005.
- [4] Andrew S. Tanenbaum, “Sistemas Operativos, Diseño e implementación,” Editorial Pearson, 2002.