Resumen

By

Alejandro Becerra Acevedo

para continuar con el padre. Las variables de condición, además de unLock también requieren un flag (bandera) y una señal.

Los Locks garantizan exclusión mutua en una sección crítica.

Similitudes

- Ambas primitivas se utilizan para coordinar la ejecución de múltiples hilos o procesos que comparten recursos, asegurando unacceso ordenado y seguro.
- Las variables de condición suelen usarse junto con locks para gestionar acceso controlado a recursos compartidos en escenariosmás complejos.

Semáforo

Es una estructura de sincronización que tiene un contador en ella. Permite representar tanto el comportamiento de un Lock como el de una condición devariable.

Mutex	Semáforo
Exclusión mutua: permite que un único hilo acceda a un recurso.	Sincronización y control del acceso a recursos múltiples.
Binario (bloqueado/desbloqueado).	Contador entero (puede ser mayor a 1).
Tiene un dueño: el hilo que lo bloquea debe liberarlo.	No tiene dueño: cualquier hilo puede realizar Signal.

Wait: Decrementa el valor del contador. Si el número resultante es negativo,el hilo se pone a dormir.

Post: Incrementa el valor del contador. Si hay hilos a la espera del semáforo,despierta alguno de ellos.

Deadlock (interbloqueo)

Se produce cuando 2 o más hilos se bloquean a la espera de un evento que va a suceder.

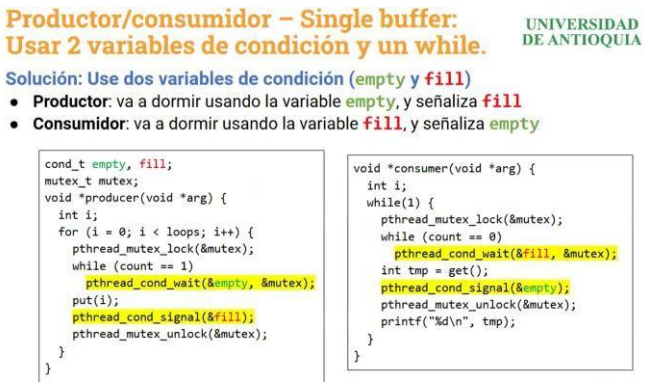
Starvation (inanición)

Se ocasiona cuando uno o más hilos esperan indefinidamente el acceso a unrecurso compartido, debido a que este aún no ha sido liberado.

Problemas clásicos de sincronización

1. **El problema del productor-consumidor (o del buffer limitado)**Un productor genera datos y los coloca en un buffer compartido, mientras que un consumidor extrae datos del mismo buffer. El desafío está en garantizar que:
   - El productor no intente agregar datos cuando el bufferesté lleno.
   - El consumidor no intente retirar datos cuando el bufferesté vacío.

Un productor despierta consumidores y un consumidor despiertaproductores para evitar **DEADLOCKS.**



Solución típica:

- Utilizar semaforos o monitores para sincronizar elacceso al buffer.
- Dos contadores: uno para rastrear elementos llenos yotro para rastrear espacios vacíos.
- Exclusión mutua para proteger las operaciones sobre elbuffer.

2. El problema de los lectores-escritores

Un recurso compartido (como una base de datos) puede ser leídopor múltiples procesos simultáneamente (lectores), pero solo unopuede escribir (escritor), y los escritores no pueden acceder al recurso mientras haya lectores activos.

Requisitos:

- Los lectores no deben interferir entre sí.
- Un escritor debe tener acceso exclusivo.
- Puede priorizarse a los lectores o a los escritoresdependiendo del diseño.

Solución típica:

- Usar **lectura-escritura locks** (rw-locks) que permitan accesocompartido para lectores y acceso exclusivo para escritores.
- Manejo cuidadoso de prioridades para evitar **inversión deprioridades** o **inanición**.

Solución típica:

- Evitar el interbloqueo introduciendo reglas como:
- Limitar el número de filósofos que pueden intentarcomer al mismo tiempo.
- Hacer que cada filósofo tome ambos tenedores oninguno.
- Numerar los tenedores y hacer que los filósofos los tomen en un orden específico (por ejemplo, primero eltenedor con el número más bajo).

Solución con semáforos

- Solución:
  - Cambiar el orden en el que los semáforos son adquiridos

```
1   void getforks() {
2       if (p == 4) {
3               sem_wait(forks[right(p)]);
4               sem_wait(forks[left(p)]);
5       } else {
6               sem_wait(forks[left(p)]);
7               sem_wait(forks[right(p)]);
8       }
9   }
```

29

```
6   void putforks() {
7       sem_post(forks[left(p)]);
8       sem_post(forks[right(p)]);
9   }
```

```
void* productor(void* arg) {
   for (i = 0; i < num; i++) {
      sem_wait(&empty);
      sem_wait(&mutex);
      put(i);
      sem_post(&mutex); // LÍNEA-A
      sem_post(&full);  // LÍNEA-B
   }
}
```

```
void* consumidor(void* arg) {
   while (!done) {
      sem_wait(&full);  // LÍNEA-C
      sem_wait(&mutex); // LÍNEA-D
      int tmp = get(i);
      sem_post(&mutex);
      sem_post(&empty);
      // hacer algo con tmp ...
   }
}
```

El valor inicial de empty debería ser del tamaño del buffer

El valor inicial de full debería ser 0, ya que indica la cantidad de elementos disponibles.

El valor inicial de mutex debería ser 1, ya que asegura que el primer hilo en llegar pueda adquirir el lonk

Se intercambia el orden de la LÍNEA-C con la LÍNEA-D

Resultado: La aplicación dejaría de ser funcional. Si el consumidor adquiere elmutex después de verificar full, varios consumidores podrían entrar a la sección crítica simultáneamente, lo que rompería la exclusión mutua. Esto podría llevar a condiciones de carrera al acceder al búfer compartido.

Se elimina la LÍNEA-A

Resultado: La aplicación dejaría de ser funcional. Sin liberar el mutex despuésde que el productor realiza su tarea, otros hilos (productores o consumidores) no podrán acceder al búfer, lo que resultará en un deadlock.

Se elimina la LÍNEA-B

Resultado: La aplicación dejaría de ser funcional. Al no incrementar el semáforo full, el consumidor nunca será notificado de que hay elementos disponibles en el búfer, lo que provocará que quede bloqueado indefinidamente en sem_wait(&full).

Se intercambia el orden de la LÍNEA-A con la LÍNEA-B

Resultado: La aplicación dejaría de ser funcional. Incrementar primero el semáforo full podría permitir que el consumidor acceda al búfer antes de que el productor haya terminado su operación (no ha liberado el mutex). Esto podría provocar una condición de carrera o el consumo de datos incorrectos.

Spinlocks

Los spinlocks son mecanismos de sincronización que funcionan haciendoque un hilo "gire" en un bucle mientras espera a que un recurso compartido se libere. Esto implica que el hilo permanece activo, realizando constantemente comprobaciones, en lugar de ceder el controlde la CPU.

En un sistema monoprocesador, solo un hilo puede ejecutarse a la vez. Si un hilo adquiere el spinlock y otro hilo intenta adquirirlo, este último entrará en un bucle infinito (spinning), pero nunca tendrá la oportunidadde ejecutarse porque el primer hilo no cede el procesador hasta que termine. Esto resulta en una pérdida total de tiempo de CPU.

Desabilitar interrupciones

Deshabilitar interrupciones es un mecanismo que puede utilizarse en sistemas de un solo procesador para garantizar que una sección crítica seejecute sin ser interrumpida. Sin embargo, no es adecuado para implementar primitivas de sincronización en sistemas multiprocesador por varias razones fundamentales:

1. No detiene a otros procesadores:

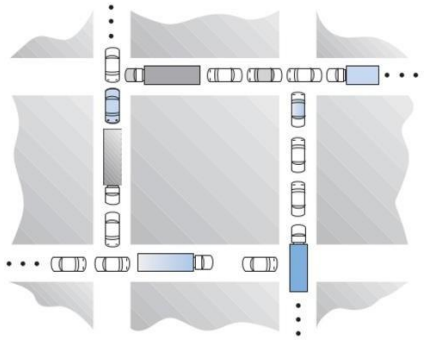
En un sistema multiprocesador, cada procesador tiene su propia unidad de ejecución y puede estar ejecutando hilos independientes en paralelo. Sideshabilitas las interrupciones en un procesador, los otros procesadores continúan ejecutándose normalmente, lo que significa que el acceso a los recursos compartidos entre procesadores no se sincroniza. Deshabilitar interrupciones en un solo procesador no previene que otros procesadoresaccedan a las secciones críticas, lo que puede llevar a condiciones de carrera y resultados impredecibles.

2. Riesgo de interbloqueo (deadlock):

Si deshabilitas las interrupciones, los procesadores pueden quedar bloqueados esperando que otro procesador libere un recurso, lo que podría generar un interbloqueo. Esto es particularmente problemático en sistemas multiprocesador, ya que se pierde la capacidad de gestionar adecuadamente las situaciones en las que dos o más procesadores esperanpor un recurso bloqueado por otro. Sin la capacidad de interrumpir el

proceso y reprogramar el acceso al recurso, los procesadores puedenquedarse bloqueados indefinidamente.

Interbloqueo



Indique el nombre de cada una de las cuatro condiciones necesarias para que exista un interbloqueo. Muestre que cada condición se presenta en este ejemplo.

Mutua exclusión.

Los recursos (en este caso, las intersecciones de tráfico) solo pueden ser ocupados por un vehículo a la vez. Cada vehículo ocupa una sección específica de la intersección, y ningún otro puede entrar mientras esta esté ocupada.

Retención y espera.

Un proceso (vehículo) retiene un recurso (una sección de la intersección)
mientras espera por otro recurso (otra sección). Cada vehículo está reteniendosu posición actual mientras espera a que la sección delante de él se desocupe.

No expropiación (no apropiatividad).

Los recursos no pueden ser liberados por la fuerza; deben ser liberadosvoluntariamente por el proceso que los posee. Ningún vehículo puede retroceder o liberar su posición actual hasta que pueda avanzar.

Espera circular.

Existe una cadena cerrada de procesos donde cada proceso está esperando un recurso que está siendo ocupado por el siguiente proceso en la cadena. Los vehículos están bloqueados en un ciclo: cada uno espera que el vehículo delante de él se mueva, pero ninguno puede avanzar.

¿Cómo evitar el interbloqueo?

Regla: No entrar a la intersección hasta que haya espacio suficiente para cruzarla completamente.

Técnica aplicada: Implementar una política de entrada condicional basada en disponibilidad. Esto se conoce como el protocolo de "no bloqueo anticipado". Si los vehículos no ingresan a la intersección hasta que puedan atravesarla completamente, se elimina la posibilidad de formar una cadena de espera circular, ya que los recursos (secciones) no se bloquean innecesariamente.

Variables de condición como semáforos

Lock como semáforo

Código de implementación del lock empleando semáforos

```
sem_init(sem_t *s, int initval) {
  s->value = initval
}

sem_wait(sem_t *s) {
  while (s->value <= 0)
    put_self_to_sleep();
  s->value--;
}

sem_post(sem_t *s) {
  s->value++;
  wake_one_waiting_thread();
}
```

```
typedef struct __lock_t {
  // whatever data structs you need goes here
  sem_t value;
} lock_t;

void init(lock_t *lock) {
  // init code goes here
  sem_init(&lock->value, 0, 1);
}

void acquire(lock_t *lock) {
  // lock acquire code goes here
  sem_wait(&lock->value);
}

void release(lock_t *lock) {
  // lock release code goes here
  sem_post(&lock->value);
}
```

Problemas de concurrencia

Violación de atomicidad

- Hay una violación en la **serialización** deseada para los **accesos a memoria**.
  - Ejemplo en MySQL
    - Dos hilos diferentes acceden al campo `proc_info` en la estructura `struct thd`.

```
1   Thread1::
2   if(thd->proc_info){
3     …
4     fputs(thd->proc_info , …);
5     …
6   }
7
8   Thread2::
9   thd->proc_info = NULL;
```

7

Violación de Orden

- El **orden de ejecución** deseado para dos accesos a memoria **se invierte!**
  - **A** debería ejecutarse siempre antes de **B**, pero **el orden no se asegura** en tiempo de ejecución.
  - Ejemplo:

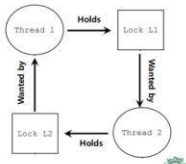
```
1   Thread1::
2   void init(){
3       mThread = PR_CreateThread(mMain, …);
4   }
5
6   Thread2::
7   void mMain(…){
8       mState = mThread->State
9   }
```

10

Problemas con Interbloqueos (*deadlocks*)

```
Thread 1:       Thread 2:
lock(L1);       lock(L2);
lock(L2);       lock(L1);
```

- Presencia de un ciclo:
  - `Thread1` adquiere el *lock* L1 y espera por el *lock* L2
  - `Thread2` adquiere el *lock* L2 y espera por el *lock* L1



13

```
int done = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
  printf("child\n");
  mutex_lock(&m);
  done = 1;
  cond_signal(&c);
  mutex_unlock(&m);
}
int main(int argc, char *argv[]) {
  pthread_t c;
  printf("parent: begin\n");
  pthread_create(c, NULL, child, NULL);
  mutex_lock(&m);
  while(done == 0)
      cond_wait(&c, &m);
  mutex_unlock(&m);
  printf("parent: end\n");
}
```

Implementación con variables de Condición (CV)

```
sem_t s;
void *child(void *arg) {
  printf("child\n");
  sem_post(&s); // Increment
}
int main(int argc, char *argv[]) {
  sem_init(&s, 0, 0);
  pthread_t c;
  printf("parent: begin\n");
  pthread_create(c, NULL, child, NULL);
  sem_wait(&s); // decrement, wait for counter == 0
  printf("parent: end\n");
}
```

Implementación con semáforos