Resumen Sincronización

Tomás Felipe Melli

Noviembre 2024

Índice

1 Sincronización de procesos

Una situación habitual en la programación **distribuida / paralela** es la **concurrencia o contención**. Dije muchas cosas ... vamos a definirlas :

- La **Programación Distribuida** es un enfoque de desarrollo de software donde los componentes de un sistema *están distribuidos* en varias computadoras o nodos de una red, en lugar de estar centralizados en un sóla máquina. Esta técnica permite que los sistemas manejen tareas más grandes y complejas al dividirlas entre diferentes recursos y ubicaciones.
- La **Programación Paralela** se refiere al *uso de múltiples procesadores o núcleos dentro de una sóla máquina* para ejecutar tareas *simultáneamente*. El objetivo es dividir una tarea grande en partes más pequeñas que puedan ser ejecutadas al mismo tiempo, acelerando el procesamiento.
- La **Concurrencia** se refiere a la *capacidad de un sistema para gestionar múltiples tareas* o procesos *que se superponen en el tiempo*. No necesariamente significa que las tareas se *ejecuten al mismo tiempo*, eso es paralelismo como vimos, sino que *se gestionan de manera que parece que están ejecutandose simultáneamente*. Normalmente, en sistemas concurrentes, las tareas *pueden ser intercaladas*, concepto llamado **interleaving**, es decir, partes de diferentes tareas pueden ser ejecutadas de manera alterna en el tiempo.
- La **Contención** ocurre cuando *múltiples* tareas o procesos *intentan acceder a un recurso compartido al mismo tiempo*, lo que puede llevar a conflictos y ralentizar el rendimiento. La contención se manifiesta cuando el acceso concurrente a un recurso provoca conflictos que deben resolverse.

2 Supongamos que tenemos dos procesos que ejecutan el mismo programa y comparten variables...

```
int ticket= 0;
int fondo= 0;

int donar(int donacion) {
  fondo+= donacion; // Actualiza el fondo
  ticket++;         // Incrementa el número de ticket
  return ticket;    // Devuelve el número de ticket
}
```

Con este scheduling posible ...

P_1	P_2	r_1	r_2	fondo	ticket	ret_1	ret_2
donar(10)	donar(20)			100	5		
load fondo		100		100	5		
add 10		110		100	5		
	load fondo	110	100	100	5		
	add 20	110	120	100	5		
store fondo		110	120	110	5		
	store fondo	110	120	!! 120	5		
	load ticket	110	5	120	5		
	add 1	110	6	120	5		
load ticket		5	6	120	5		
add 1		6	6	120	5		
store ticket		6	6	120	6		
	store ticket	6	6	120	!! 6		
return reg		6	6	120	6	6	
	return reg	6	6	120	6		6

Podemos ver que *P1* y *P2* comparten las variables **fondo** y **ticket**. EL problema es que **toda ejecución debería dar un resultado equivalente a alguna ejecución secuencial de los mismos procesos** y en este caso ocurre una **Race Condition**. Es decir, **el resultado depende del orden de ejecución**. Una potencial solución a este problema es la **Exclusión Mutua** mediante **Secciones Críticas**. Esto es,

un *sector de código donde sólo puede correrlo un proceso a la vez.* Todo proceso que esté esperando para entrar a *CRIT* va a entrar. Ningún proceso fuera de *CRIT* puede bloquear a otro.

3 Cómo se implementan las secciones críticas ?

Con **LOCKS**, es decir una variable *booleana compartida*. Si quiere entrar, le pongo 1 y al salir 0. Si está en 1, debo esperar a ver el 0. Si cuando veo un 0, se acaba el quantum y cuando me toca hay otro proceso, yo ni me enteré. Afortunadamente, el **Hardware** tiene una *instrucción* que permite estableces **atómicamente** el valor de una variable entera en 1. Es una operación **indivisible**, es decir que cuando se ejecuta, no se interrumpe, por tanto el **scheduler no lo puede desalojar en este punto**.

4 Qué hace TestAndSet() ?

```
1    bool TestAndSet(bool* destino){
2        bool res = *destino;
3        *destino = TRUE;
4        return res;
5    }
```

Básicamente esta función lo que hace es, logra leer un valor, lo setea *true*, pero el valor de retorno, es el valor que estaba antes ... confunde esto ? .. Retomemos el ejemplo del ticket, quiero asignar un número, si la variable "escritura" está en 0, puedo hacer *ticket++*, sino, esperamos **(busy waiting)** hasta que se pueda... Miremos esto :

```
// definimos la variable compartida
bool lock;

void main(){
    while (TRUE){
        while(TestAndSet(&lock)){
            // esto est  vac o a prop sito
        }
        // Por fuera hacemos lo que quer amos que estaba bloqueado
        // Al final de la secci n cr tica tenemos que reestablecer LOCK
        lock = FALSE;
    }
}
```

Cómo funciona esto ?
El proceso hace la lectura, le da TRUE, se mete en el ciclo y no sale (busy waiting). Sólo pasa este bucle en el momento en que el valor de lock es 0. Esto se debe a que, hace la lectura, lo setea TRUE y retorna FALSE, rompiendo el bucle y *pasa a la sección crítica.*
Cuál es el problema del Busy Waiting ?
El tema es que esta manera de sincronizar procesos es costosa en términos de consumos de CPU.

5 Qué pasa si usamos Sleep() - Wakeup() ? (Productor/Consumidor)

Supongamos el esquema productor/consumidor. Ambos comparten un buffer acotado. El productor agrega elementos y el consumidor los saca (concurrencia). Además, siel productor quiere poner algo cuando está lleno o el consumidor sacar cuando no hay ...

```
consumidor () {
    if (cantidad == 0) sleep();
}

productor (item, buffer) {
    cantidad++;
    wakeup();
}
```

Esto conlleva el famoso **Wake-Up Problem** :

Consumidor	Productor	Variables
		cant==0, buffer==[]
	agregar(i1, buffer)	cant==0, buffer==[i1]
¿ cant==0 ?		
	cant++ wakeup()	cant==1, buffer==[i1]
sleep()		

6 Semáforos

Afortunadamente, al cabezón Dijkstra, se le ocurrió definir un tipo de variable especial llamada **semáforo**. Una variable que :

- Puede ser utilizada con cualquier valor.
- Sólo se la puede manipular con dos operaciones **wait()** y **signal()** ambas sin interrupciones.

Existe un tipo especial de semáforo con *dominio binario* que se llama **mutex** por *mutual exclusion*.

7 Cómo podríamos usar el semáforo en el contexto de productor/consumidor ?

```
semáforo mutex = 1;
semáforo llenos = 0;
semáforo vacios = N; // Capacidad del buffer.

void productor() {
 while (true) {
      item = producir_item();
      wait(vacios);
      // Hay lugar. Ahora
      // necesito acceso
      // exclusivo.
      wait(mutex);
      agregar(item, buffer);
      cant++;
      // Listo, que sigan
      // los demás.
      signal(mutex);
      signal(llenos);
     }
}

void consumidor() {
 while (true) {
      wait(llenos);
      // Hay algo. Ahora
      // necesito acceso
      // exclusivo.
      wait(mutex);
      item = sacar(buffer);
      cant--;
      // Listo, que sigan
      // los demás.
      signal(mutex);
      signal(vacios);
      hacer_algo(item);
     }
}
```

Esto podría derivar en un problema si nos *olvidamos un signal o invertimos el orden.* **El sistema se traba porque A espera que suceda algo que sólo B puede provocar, pero B espera algo de A**. Esta situación se llama **DEADLOCK**.

8 Qué herramientas tenemos para solucionar deadlocks ?

En la actualidad contamos con alternativas para implementar secciones críticas como *bool atómico, int atómico, cola atómica.* También existe un *objeto* que implementa ciertas operaciones indivisibles a nivel HW llamado **TASLock**.

8.1 TASLock / Spin Lock

```
private bool reg;

atomic bool get() { return reg; }

atomic void set(bool b) { reg = b; }

atomic bool getAndSet(bool b) {
  bool m = reg;
  reg = b;
  return m;
}

atomic bool testAndSet() {
  return getAndSet(true);
}
```

```
TASLock mtx;
int donar(int donacion) {
  int res;
  // Inicio de la sección crítica.
  mtx.lock();
  fondo += donacion;
  mtx.unlock();
  // Fin de la sección crítica.

  // Inicio de otra sección crítica.
  mtx.lock();
  res = ticket; ticket++;
  mtx.unlock();
  // Fin de la sección crítica.

  return res;
}
```

En base a esto construimos un **mutex** conocido como **spin lock** :

```
atomic<bool> reg;

void create() { reg.set(false); }

void lock() { while (reg.testAndSet()) {} }

void unlock() { reg.set(false); }
```

En este caso, *Lock() no es atómico.*
Qué inconvenientes puede traer esto ?
No debemos olvidar el **unlock()** ya que produciría *busy waiting*. Sin embargo, su *overhead* puede ser menor que usando semáforos.
Cómo podemos minimizar el impacto ?
Existe **TTASLock** que es un *TestAndSet* también llamada **Local Spinning**. La idea es hacer un *test* antes de mandarse con el **TAS()**. Es más eficiente esto pues el *while* hace *get()* en vez de *TestAndSet()*. Mientras sea *TRUE* lee memoria cache (cache hit) cuando un proceso hace *unlock()* hay cache miss.

9 Qué otros objetos atómicos existen ?

9.1 Registros Read-Modify-Write

```
atomic int getAndInc() {
  int res = reg;
  reg++;
  return res;
}

atomic int getAndAdd(int v) {
  int res = reg;
  reg = reg + v;
  return res;
}

atomic T compareAndSwap(T  u, T  v) {
  T res = reg;
  if (u == res) reg = v;
  return res;
}
```

9.2 Cola

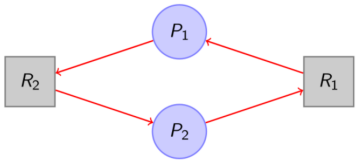
```
atomic enqueue(T item) {
  mtx.lock();
  q.push(item);
  mtx.unlock();
}

atomic bool dequeue(T *pitem) {
  bool success;
  mtx.lock();
  if (q.empty) {
    pitem = null;  success = false;
  }
  else {
    pitem = q.pop(); success = true;
  }
  mtx.unlock();
  return success;
}
```

10 Condiciones de Coffman

Las *Condiciones de Coffman* son condiciones para determinar la existencia de un Deadlock (tiene falencias).

- **Exclusión mutua** : un recurso no puede estar asignado a más de un proceso.
- **Hold and Wait** : los procesos que ya tienen algún recurso pueden solicitar otro.
- **No-Preemption** : no hay mecanismo compulsivo para quitarle los recursos a un proceso.
- **Espera Circular** : tiene que haber un ciclo $N \geq 2$ procesos tal que P_i espera un recurso que tiene P_{i+1}

Una forma de detectarlos es con un grafo bipartito donde los nodos son procesos P y recursos R. Los arcos se definen $P \to R$ si P le solicita a R y $R \to P$ si P adquirió R :



Si existe un ciclo, entonces, hay deadlock.

11 Correctitud de programas

Miremos el siguiente ejemplo de programa en el que se ejecuta una serie de problemas en simultáneo P_i donde $i \in [0, ..., N-1]$

```
// Semáforos
semaphore sem[N+1];

// Inicialización
proc init() {
  for(i = 0; i<N+1; i++)
    sem[i] = 0;

  for (i = 0; i<N; i++)
    spawn P(i);

  sem[0].signal();
}
```

```
// Proceso i
proc P(i) {
 // Esperar turno
 sem[i].wait();
 // Ejecutar
 print("Soy el proc " + i);
 // Avisar al próximo
 sem[i+1].signal();
}
```

Nos preguntamos si *es correcto este programa ?*
En el caso de **Programas Paralelos**, la noción de *correcto* no es unívoca. Es un *conjunto de propiedades* planteadas sobre toda ejecución. La cuestión estará entonces en poder demostrar o argumentar que cumplan con las siguientes **propiedades**

- **Safety** : el sistema evita estados incorrectos o inconsistentes durante la ejecución concurrente de tareas.
  Ejemplo : se garantiza la exclusión mutua, no habrá deadlock, se garantiza la sincronización, etc.
- **Liveness** : se garantiza que los procesos *no se queden atascados indefinidamente* y que el sistema eventualmente haga sus progresos hacia el cumplimiento de sus objetivos.
  Ejemplo : no estancamiento, **progreso**, prevención de bloqueos
- **Fairness** : se garantiza que **todos los procesos o hilos tengan una oportunidad justa para acceder a recursos compartidos** y realizar sus tareas.

Para trabajar con estas propiedades se desarrollaron **lógicas temporales** que son herramientas para especificar y razonar sobre el comportamiento de sistemas a lo largo del tiempo. Algunos ejemplos son *LTL (Linear Temporal Logic), CTL(Computation Tree Logic),* entre otras.
Para demostrar estas propiedades hay que tener un modelo formal del comportamiento del sistema : se utilizan distintos tipos de **autómatas** para ello. Podemos usar herramientas como *Theorem, Provers, Model Checkers...*
No nos vayamos por las ramas, *es correcto el programa ?*

Tenemos una serie de procesos P_i con $i \in [0, .., N-1]$. Cada proceso P_i ejecuta una tarea S_i. Toda ejecución debe garantizar turnos. Las tareas S_i se ejecutan en orden : $S_0, S_1, ..., S_{N-1}$

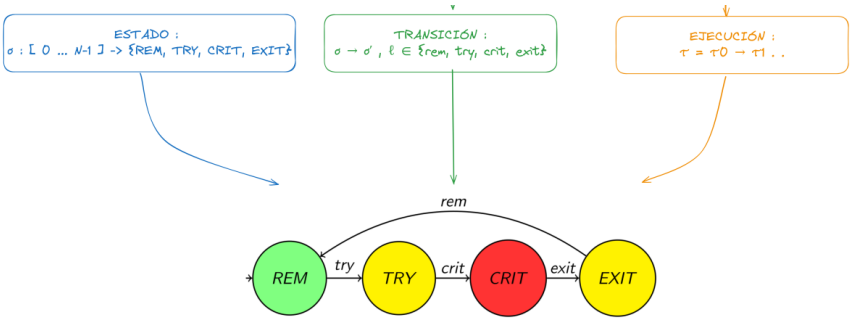
12 Rendez-Vous o Barrera

Es una **barrera de sincronización**. Cada P_i con $i \in [0, ..., N-1]$ tiene que ejecutar $a(i), b(i)$. La propiedad **barrera** garantiza que $b(j)$ se ejecuta **después de todos los** $a(i)$. No hay que imponer ningún orden entre los $a(i)$ ni los $b(i)$. Veamos el siguiente ejemplo :

```
atomic<int> cant = 0;  // Procs que terminaron a
semaphore barrera = 0; // Barrera baja

proc P(i) {
 a(i);
 // ¿Se puede ejecutar b?
 if (cant.getAndInc() < N-1)
    // No. Esperar.
    barrera.wait();
 else
    // Sí. Entrar y avisar
    barrera.signal();
 // Ejecutar b (sección crítica).
 b(i);
}
```

Es correcto ?
Con *safety* no alcanza, necesitamos dar **condiciones de progreso** : *todo proceso que intenta acceder a la sección crítica, en algún momento, lo logra, cada vez que intenta.*

12.1 Cómo se formaliza esto ? Modelo de Lynch



- **Wait Freedom** : todo proceso que intenta acceder a la sección crítica, en algún momento lo logra, cada vez que lo intenta. Esto demuestra el **progreso global absoluto**

Para toda ejecución τ, estado τ_k y todo proceso i,
si $\tau_k(i) = TRY$
entonces $\exists j > k$, tal que $\tau_j(i) = CRIT$.

$$WAIT\text{-}FREEDOM \equiv \forall i. \Box\ IN(i)$$

- **Starvation Freedom**, progreso global dependiente.

Para toda ejecución τ,
si para todo estado τ_k y proceso i tal que
$\tau_k(i) = CRIT$,
  $\exists j > k$, tal que $\tau_j(i) = REM$
entonces para todo estado $\tau_{k'}$ y todo proceso i',
  si $\tau_{k'}(i') = TRY$
  entonces $\exists j' > k'$, tal que $\tau_{j'}(i') = CRIT$.

$$STARVATION\text{-}FREEDOM \equiv \forall i.\Box\ OUT(i) \implies \forall i.\Box\ IN(i)$$

- **Lock Freedom**, progreso del sistema

Para toda ejecución τ y estado τ_k,
si en τ_k hay un proceso i en TRY y ningún i' en $CRIT$
entonces $\exists j > k$, t. q. en el estado τ_j algún proceso i' está en $CRIT$.

$$LOCK\text{-}FREEDOM \equiv \Box(\#TRY \geq 1 \wedge \#CRIT = 0 \implies \Diamond \#CRIT > 0)$$

- **Exclusión Mutua**

Para toda ejecución τ y estado τ_k, no puede haber más de un proceso i tal que $\tau_k(i) = CRIT$.

$$EXCL \equiv \Box\ \#CRIT \leq 1$$

- **Fairness** o ecuanimidad

Para toda ejecución τ y todo proceso i,
si i puede hacer una transición ℓ_i en una cantidad infinita de estados de τ
entonces existe un k tal que $\tau_k \xrightarrow{\ell_i} \tau_{k+1}$.

13 Livelock

Decimos que un conjunto de proceso entán en **livelock** si **cambian su estado en respuesta a los cambios de otros pero ninguno de ellos hace progreso hacia la consecución de sus objetivos**.
Veamos un ejemplo : queda poco espacio en disco. Proceso A detecta la situación y notifica al proceso B (bitácora del sistema). B registra el evento en disco, disminuyendo el espacio libre, lo que hace que A detecte la situación y ...

14 Propiedad SWMR (Single-Writer Multiple-Readers)

En bases de datos sucede mucho que hay variables compartidas en donde los escritores necesitan acceso EXCLUSIVO pero los lectores pueden ser muchos simultáneamente. Esto es una propiedad llamada SWMR (SINGLE-WRITER / MULTIPLE-READERS)

$$\forall \tau.\ \forall k.\ \exists i.\ \text{writer}(i) \wedge \tau k(i) = \text{CRIT} \Longrightarrow \forall j \neq i.\ \tau k(j) \neq \text{CRIT}$$
$$\forall \tau.\ \forall k.\ \exists i.\ \text{reader}(i) \wedge \tau k(i) = \text{CRIT} \Longrightarrow \forall j \neq i.\ \tau k(j) = \text{CRIT} \Longrightarrow \text{reader}(j)$$

```
semaphore wr = 1;
semaphore rd = 1;
int readers = 0;

proc writer(i) {
  // TRY
  wr.wait();
  // CRIT
  write();
  // EXIT
  wr.signal();
}
```

```
proc reader(i) {
  // TRY
  rd.wait();
  readers++;
  if (readers == 1)
    wr.wait();
  rd.signal();
  // CRIT
  read();
  // EXIT
  rd.wait();
  readers--;
  if (readers == 0)
    wr.signal();
  rd.signal();
}
```

Puede haber inanición de escritores, ya que puede haber siempre al menos un leto. Se viola la propiedad de progreso global dependiente (Starvatio Freedom).