TEMA 3:

ARQUITECTURAS CON PARALELISMO A NIVEL DE THREAD (TLP)

Lección 4: SINCRONIZACIÓN.

La comunicación, uno-a-uno o colectiva, entre procesadores cuando estos comparten variables (memoria) se lleva a cabo a través de variables compartidas (memoria compartida). Para que la comunicación de datos entre flujos de instrucciones que se ejecutan de forma concurrente o en paralelo se lleve a cabo sin problemas es necesario añadir código que sincronice el flujo emisor y el receptor del dato para evitar que el receptor del dato acceda a la variable compartida antes de que el emisor lo haya calculado y colocado en dicha variable.

Comunicación en multiprocesadores y necesidad de usar código de sincronización.

Comunicación uno-a-uno.

- Se debe garantizar que el proceso que recibe lea la variable compartida cuando el proceso que envía haya escrito en la variable el dato a enviar.
- Si se reutiliza la variable para comunicación, se debe garantizar que no se envía un nuevo dato en la variable hasta que no se haya leído el anterior.

Comunicación colectiva.

- Ejemplo de comunicación colectiva: suma de n números:

Secuencial	**Paralela** (sum=0)

for (i=0 ; i<n ; i++) {
  sum = sum + a[i];
}
printf(sum);
	for (*i*=*ithread* ; *i*<n ; *i*=*i*+nthread) {
	  sump = *sump* + a[*i*];
	}
	sum = sum + *sump*; /* SC, sum compart. */
	if (*ithread*==0) printf(sum);

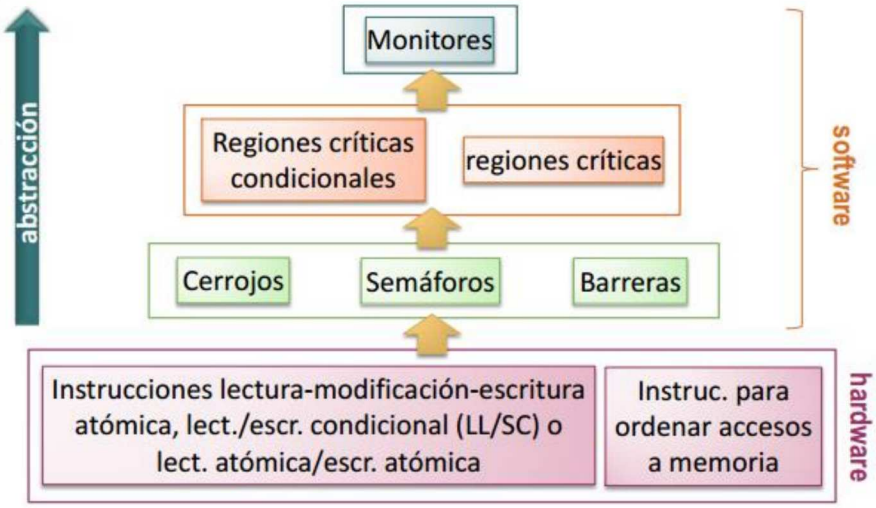
Se trata de un código secuencial para la suma de los componentes de un vector *a[ ]*. En la versión paralela, primero cada uno de los *nthread* flujo de instrucciones calcula en una variable local *sump* la suma de un subconjunto de los componentes de *a[ ]*. Después cada uno acumula en la variable compartida sum el resultado parcial que ha calculado en sumo. Para que la versión paralela imprima un resultado de suma correcto se debe añadir código de sincronización para asegurar que:

1. Los flujos de instrucciones acceden secuencialmente a modificar la variable compartida *sum*, es decir, que ejecuten uno detrás de otro en secuencia, no importa el orden, la sentencia *sum = sum + sump*.
2. El flujo *ithread=0* imprime el contenido de la variable *sum* cuando todos los flujos hayan acumulado en esta variable la suma parcial que han calculado en su variable privada *sump*.

A las secciones de código con acceso a variables compartidas a las que se requiere acceder de forma secuencial por parte de los flujos de instrucciones se las suele denominar **secciones críticas**.

El proceso 0 no debería imprimir hasta que no hayan acumulado *sump* en *sum* todos los procesos.

Soporte software y hardware de sincronización.



Cerrojos.

Un cerrojo, también denominado mutex, impide que varios procesos o hebras entren en la misma parte del código. Consta de dos funciones que se utilizan para sincronizar:

1. Función de adquisición lock( ) o cierre de cerrojo: Se ejecuta cuando se quiere adquirir el derecho a acceder a variables compartidas, lo hace cerrando/adquiriendo el cerrojo. Su implementación debe cumplir además los siguientes requisitos:
    - Si varios flujos intentan la adquisición/cierre a la vez, sólo uno de ellos lo debe conseguir, el resto debe pasar a una etapa de espera.
    - Todos los procesos que ejecuten lock( ) con el cerrojo cerrado/adquirido deben quedar en espera.
2. Función de liberación unlock( ) o apertura del cerrojo: Un flujo de instrucciones ejecuta un unlock( ) cuando ha terminado el acceso a las variables compartidas con el objetivo de liberar a uno de los flujos que está esperado el acceso a estas variables, el flujo liberado adquiere/cierra el cerrojo. Su implementación debe cumplir además el siguiente requisito:
    - Si no hay flujos de espera, permitirá que el siguiente flujo que ejecute la función lock( ) adquiera/cierra el cerrojo sin espera.

Componentes en un código para sincronización:

- Método de adquisición.
    - Método por el que un thread trata de adquirir el derecho a pasar a utilizar unas direcciones compartidas.
- Método de espera.
    - Método por el que un thread espera a adquirir el derecho a pasar a utilizar unas direcciones compartidas:
        * Espera ocupada.
        * Bloqueo.
- Método de liberación.
    - Método utilizado por un thread para liberar a uno (cerrojo) o varios (barrera) threads en espera.

Posibles implementaciones de cerrojos:

- Cerrojo simple: Se usa como cerrojo una variable compartida k que toma dos valores: 0 (abierto) y 1 (cerrado). El código de las funciones del cerrojo es el siguiente:
    1. Función de liberación unlock (k) o apertura del cerrojo: abre el cerrojo escribiendo un 0.
    2. Función de adquisición lock (k) o cierre de cerrojo:
        - Lee el cerrojo y lo cierra escribiendo un 1. Entre lectura y escritura no se debe realizar ningún otro acceso a k para evitar que más de un flujo encuentre el cerrojo abierto a la vez.
        - Si de la anterior lectura se obtiene:
            i. Un 1 (cerrojo cerrado), el flujo espera porque hay un flujo que tiene el cerrojo cerrado y estará accediendo a la variable compartida.
            ii. Un 0, el flujo adquiere el cerrojo y puede salir del lock (k).
        - Como resultado de la lectura:
            i. Si el cerrojo estaba cerrado el thread espera hasta que otro thread ejecute unlock(k).
            ii. Si estaba abierto adquiere el derecho a pasar a la sección crítica.

- Cerrojo con etiqueta: Se establece un orden FIFO en la adquisición del cerrojo, garantiza que los flujos de instrucciones adquieran el cerrojo en el orden en el que ejecutan la función de adquisición. El código de las funciones del cerrojo con etiqueta es el siguiente:

    1. Función de liberación unlock(contadores):

```
contadores.lib = (contadores.lib + 1)  mod  max_flujos_control;
```

    2. Función de adquisición lock(contadores):

```
contador_local_adq = contadores.adq;
contadores.adq = (contadores.adq + 1)  mod  max_flujos_control;
while (contador_local_adq <> contadores.lib) {};
```

        - Adquiere el cerrojo el flujo cuya etiqueta coincida con el contador de liberación. Se requiere una operación leer-incrementar-escribir indivisible; es decir, entre lectura y escritura no se debe realizar ningún otro acceso a memoria para evitar que más de un procesador pueda leer el mismo valor de contador.adq.
        - Si la etiqueta de un flujo no coincide con el valor del contador de liberación, el flujo espera porque hay un flujo que tiene el cerrojo cerrado.

Barreras.

Una barrera es un punto en el código en el que los flujos de instrucciones que colaboran en la ejecución del código que esperan entre sí. Cuando todos han llegado a la barrera, salen de la misma y continúan ejecutando el código que hay después de la barrera.

```
Barrera(id, num_threads) {
  if (bar[id].cont==0) bar[id].bandera=0;
  cont_local = ++bar[id].cont;
  if (cont_local ==num_threads) {
     bar[id].cont=0;
     bar[id].bandera=1;
  }
  else espera mientras bar[id].bandera=0;
}
```

```
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local)  //se complementa bandera local
  lock(bar[id].cerrojo);
     cont_local = ++bar[id].cont      //cont_local es privada
  unlock(bar[id].cerrojo);
  if (cont_local == num_procesos) {
     bar[id].cont = 0;                  //se hace 0 el cont. de la barrera
     bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else  while (bar[id].bandera != bandera_local) {};  //espera ocupada
}
```

La variable bar[ ] contiene una lista de barreras. Cada barrera tiene un contador bar[ ].cont para saber el número de flujos que han llegado a la barrera. Una variable de espera bar[ ].bandera para implementar la espera ocupada y una variable bar[ ].cerrojo, para implementar accesos secuenciales a variables compartidas.

Apoyo hardware a primitivas software.

Instrucciones de lectura-modificación-escritura atómicas.

Test&Set (x)

```
Test&Set (x) {
   temp = x ;
   x = 1 ;
   return (temp) ;
}
/* x compartida */
```

Fetch&Oper(x,a)

```
Fetch&Add(x,a) {
   temp = x ;
   x = x + a ;
   return (temp);
}/* x compartida,
a local */
```

Compare&Swap(a,b,x)

```
Compare&Swap(a,b,x){
   if (a==x) {
      temp=x ;
      x=b; b=temp ; }
}/* x compartida,
   a y b  locales */
```

Fetch&OR(x,a)

```
Fetch&OR(x,a){
        Tmp = x;
        x=x || a;
        Return
tmp;
}
```

Cerrojos simples con Test&Set y Fetch&Or.

```
lock(k) {
  while (test&set(k)==1) {} ;
}
/* k compartida */
```

```
lock(k) {
  while (fetch&or (k,1)==1)  {};
}
/* k compartida */
```

Cerrojos simples con Compare&Swap.

```
lock(k) {
  b=1
  do
    compare&swap(0,b,k) ;
  while (b==1);
}
/* k compartida, b local */
```