Sincronización

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

Sincronización en memoria compartida

- La comunicación se realiza a través de memoria compartida.
  - Es necesario sincronizar accesos a variable compartida.

- **Alternativas**:
  - Comunicación 1-1.
  - Comunicación colectiva (1-N).

Comunicación 1 a 1

- Asegurar que la lectura (**recepción**) se produce después de la escritura (**emisión**).
- En caso de **reutilización** (bucles):
  - Asegurar que escritura (**emisión**) después de lectura (**recepción**) anterior.
- Necesidad de accesos con **exclusión mutua**.
  - Solamente uno de los procesos accede a una variable a la vez.
- **Sección crítica**:
  - Secuencia de instrucciones que acceden a una o varias variables con exclusión mutua.

Comunicación colectiva

- Necesidad de **coordinación** de **múltiples accesos** a variables.
  - Escrituras sin interferencias.
  - Lecturas **deben esperar** a que datos estén disponibles.
- Deben **garantizarse accesos** a variable en **exclusión mutua**.
- Debe **garantizarse** que **no se lee resultado** hasta que todos han ejecutado su sección crítica.

Suma de un vector

Sección crítica en bucle

```
void f(int max) {
  vector<double> v = get_vector(max);
  double sum = 0;

  auto do_sum = [&](int start, int n) {
    for (int i=start; i<n; ++i) {
      sum += v[i];
    }
  }

  thread t1{do_sum,0,max/2};
  thread t2{do_sum,max/2+1,max};
  t1.join();
  t2.join();
}
```

Sección crítica fuera bucle

```
void f(int max) {
  vector<double> v = get_vector(max);
  double sum = 0;

  auto do_sum = [&](int start, int n) {
    double local_sum = 0;
    for (int i=start; i<n; ++i) {
      local_sum += v[i];
    }
    sum += local_sum
  }

  thread t1{do_sum,0,max/2};
  thread t2{do_sum,max/2+1,max};
  t1.join();
  t2.join();
}
```

Soporte hardware

- Necesidad de fijar un orden global en las operaciones.
  - Modelo de consistencia puede ser **insuficiente y complejo**.
  - Se suele **complementar** con operaciones de **lectura-modificación-escritura**.
  - **Ejemplo en IA-32**:
    - Instrucciones con prefijo **LOCK**.
    - Acceso al bus de **forma exclusiva** si posición **no está en caché**.

Primitivas: Test and set

- Instrucción **Test and Set**:
  - **Secuencia atómica**:

    1. **Lee posición de memoria** en registro (se devolverá como resultado).
    2. **Escribe el valor 1** en la posición de memoria.

- **Usos**: IBM 370, Sparc V9

Primitivas: Intercambio

- Instrucción de **intercambio** (swap):
  - **Secuencia atómica**:
    1. **Intercambia** los contenidos de una **posición de memoria** y un **registro**.
    2. **Incluye** una **lectura de memoria** y una **escritura de memoria**.
  - **Más general que test-and-set**.
- **Instrucción IA-32**:
  - **XCHG reg**, **mem**
- **Usos**: Sparc V9, IA-32, Itanium

Primitivas: Captación y operación

- Instrucción de **captar y operar** (fetch-and-op):
  - Diversas operaciones: **fetch-add**, **fetch-or**, **fetch-inc**, ...
  - **Secuencia atómica**:
    1. Lee posición de memoria en registro (devuelve este valor).
    2. Escribe en posición de memoria el resultado de aplicar al valor original una operación.
  - **Instrucción IA-32**:
    - **LOCK XADD reg**, **mem**
  - **Usos**: IBM SP3, Origin 2000, IA-32, Itanium.

Primitivas: Comparar e intercambiar

- Instrucción **Comparar-e-intercambiar** (compare-and-swap o compare-and-exchange):
  - Operación sobre **dos variables locales** (registros **a** y **b**) y una **posición de memoria** (variable **x**).
  - **Secuencia atómica**:
    1. Lee el valor de **x**.
    2. Si **x** es igual a registro **a** $\rightarrow$ intercambia **x** y registro **b**.
  - **Instrucción IA-32**:
    - **LOCK CMPXCHG mem**, **reg**
    - Usa implícitamente registro adicional **eax**.
  - **Usos**: IBM 370, Sparc V9, IA-32, Itanium.

Primitivas: Almacenamiento condicional

- Par de instrucciones **LL/SC** (Load Linked/Store Conditional).
  - **Funcionamiento**:
    - Si el contenido de una **variable leída** mediante **LL** se **modifica** antes de un **SC** el almacenamiento no se lleva a cabo.
    - Sin entre **LL** y **SC** ocurre **cambio de contexto**, **SC no se lleva a cabo**.
    - **SC** devuelve un **código de éxito/fracaso**.
  - **Ejemplo Power-PC**:
    - **LWARX**
    - **STWCX**
  - **Usos**: Origin 2000, Sparc V9, Power PC

Cerrojo

- Un **cerrojo** es un mecanismo que asegura la **exclusión mutua**.
- **Dos funciones de sincronización**:
  - **Lock(k)**:
    - Adquiere el cerrojo.
    - Si varios intentan adquirir el cerrojo, n-1 pasan a espera.
    - Si llegan más procesos, pasan a espera.
  - **Unlock(k)**:
    - Libera el cerrojo.
    - Permite que uno de los procesos en espera adquiera el cerrojo.

Mecanismos de espera

- **Dos alternativas**: **espera activa** y **bloqueo**.
- **Espera activa**:
  - El proceso queda en **bucle** que **constantemente** consulta valor de variable de control de espera.
  - **Spin-lock**.
- **Bloqueo**:
  - El proceso queda suspendido y cede el procesador a otro proceso.
  - Si un proceso ejecuta **unlock** y hay procesos **bloqueados** se libera a uno de ellos.
  - Requiere soporte de un planificador (típicamente SO o *runtime*).
- **La selección de la alternativa depende del coste**.

Componentes

- Tres **elementos de diseño** en un mecanismo de cerrojos: **adquisición**, **espera** y **liberación**.
- **Método de adquisición**:
  - Usado para intentar adquirir el cerrojo.
- **Método de espera**:
  - Mecanismo para esperar hasta que se pueda adquirir el cerrojo.
- **Método de liberación**:
  - Mecanismo para liberar un o varios procesos en espera.

Cerrojos simples

- Variable **compartida k** con dos valores.
  - **0** → **abierto**.
  - **1** → **cerrado**.

- **Lock(k)**
  - Si **k=1** → **Espera activa** mientras **k=1**.
  - Si **k=0** → **k=1**.
  - **No se debe permitir** que 2 procesos **adquieran cerrojo simultáneamente**.
    - Usar lectura-modificación-escritura para cerrar.

Implementaciones simples

Test and set

```
void lock(atomic_flag & k) {
  while (k.test_and_set())
    {}
}

void unlock(atomic_flag & k) {
  k.clear();
}
```

Captación y operación

```
void lock(atomic<int> & k) {
  while (k.fetch_or(1) == 1)
    {}
}

void unlock(atomic<int> & k) {
  k.store(0);
}
```

Implementaciones simples

Intercambio IA-32

```
do_lock:    mov eax, 1
repeat:     xchg eax, _k
            cmp eax, 1
            jz  repeat
```

Retardo exponencial

- **Objetivo**:
  - **Reducir** accesos a memoria.
  - **Limitar** consumo de energía.

Cerrojo con retardo exponencial

```
void lock(atomic_flag & k) {
  while (k.test_and_set())
  {
    perform_pause(delay);
    delay *= 2;
  }
}
```

- Se incrementa **exponencialmente** tiempo entre invocaciones a **text_and_set()**.

Sincronización y modificación

- Se pueden **mejorar prestaciones** si se usa la **misma variable para sincronizar y comunicar**.
  - Se evita usar **variables compartidas** solamente para sincronizar.

Suma de un vector

```
double parcial = 0;
for (int i=iproc; i<max; i+=nproc) {
   parcial += v[i];
}
suma.fetch_add(parcial);
```

Cerrojos y orden de llegada

- **Problema**:
  - Implementaciones simples no fijan orden de adquisición de cerrojo.
  - Se podría dar inanición.

- **Solución**:
  - Hacer que el cerrojo se adquiera por **antigüedad** en la solicitud.
  - Garantizar orden FIFO.

Cerrojos con etiquetas

- **Dos contadores**:
  - **Contador de adquisición**: Número de procesos que han solicitado el cerrojo.
  - **Contador de liberación**: Número de veces que se ha liberado el cerrojo.

Lock:

- **Etiqueta** $\rightarrow$ Valor de contador de adquisición
- Se incrementa contador de adquisición.
- Proceso queda esperando hasta que contador de liberación coincida con etiqueta.

Unlock:

- Incrementa el contador de liberación.

Cerrojos basados en colas

- Mantener una **cola** con **procesos que esperan** para entrar en **sección crítica**.

- **Lock**:
  - Comprobar si cola está vacía.
  - Sin un proceso se une a la cola hace espera activa en una variable.
    - Cada proceso hace espera activa en una variable distinta.

- **Unlock**:
  - Eliminar proceso de cola.
  - Modificar variable de espera de proceso.

Barrera

- Una barrera permite **sincronizar** varios procesos en algún punto.
  - Garantiza que ningún proceso supera la barrera hasta que todos han llegado.
  - Usadas para sincronizar fases de un programa.

Barreras centralizadas

- **Contador** centralizado asociado a la **barrera**.
  - Cuenta el número de procesos que han llegado a la barrera.

- **Función barrera**:
  - Incrementa el contador.
  - Espera a que el contador llegue al número de procesos a sincronizar.

Barrera simple

Implementación simple

```
do_barrier(barrera, n) {
  lock(barrera.cerrojo);
  if (barrera.contador == 0) {
    barrera.flag=0;
  }
  contador_local = barrera.contador++;
  unlock(barrera.cerrojo);
  if (contador_local == np) {
    barrera.contador=0;
    barrera.flag=1;
  }
  else {
    while (barrera.flag==0) {}
  }
}
```

- Problema si se reusa la barrera en un bucle.

Barrera con inversión de sentido

Implementación simple

```
do_barrera(barrera, n) {
  flag_local = !flag_local;
  lock(barrera.cerrojo);
  contador_local = barrera.contador++;
  unlock(barrera.cerrojo);
  if (contador_local == np) {
    barrera.contador=0;
    barrera.flag=flag_local;
  }
  else {
    while (barrera.flag==flag_local) {}
  }
}
```

Barreras en árbol

- Una implementación simple de barreras no es **escalable**.
  - Contención en el acceso a variables compartidas.

- Estructuración de los procesos de llegada y liberación en forma de árbol.
  - Especialmente útil en redes distribuidas.

Resumen

- Necesidad de sincronización de accesos en memoria compartida:
  - Comunicación individual (1-1) y colectiva (1-N).
- Diversidad de primitivas hardware para sincronización.
- Cerrojos como mecanismo de exclusión mutua.
  - Espera activa frente a bloqueo.
  - Tres elementos de diseño: adquisición, espera y liberación.
- Los cerrojos pueden tener problemas al no fijar orden (inanición).
  - Soluciones basadas en etiquetas o colas.
- Las barreras ofrecen mecanismos para estructurar programas en fases.

Referencias

- **Computer Architecture. A Quantitative Approach.**
  5th Ed.
  Hennessy and Patterson.
  Secciones: 5.5

Sincronización

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid