SCD Temas 1 y 2

Aurelia Nogueras

Contents

Tema 1: Introducción

Conceptos básicos

- Programa secuencial: se ejecuta en secuencia.
- Programa concurrente: se puede ejecutar en paralelo.
- Proceso: ejecución de un programa secuencial.
- Concurrencia: describe el potencial para la ejecución paralela.
- Programación concurrente: técnicas y notaciones para paralelismo potencial y resolver problemas de sincronización y comunicación.
- Programación paralela: acelera la resolución de problemas mediante el procesamiento en paralelo.
- Programación distribuida: varios componentes en diferentes localizaciones trabajando juntos.
- Programación de tiempo real: sistemas que están funcionando continuamente con restricciones estrictas en la respuesta temporal.

La programación concurrente nos permite mejorar la eficiencia y la calidad.

Modelo abstracto y consideraciones sobre el hardware

- Concurrencia en sistemas monoprocesador: paralelismo virtual. Múltiples procesos se reparten los ciclos de CPU (multiprogramación). Sincronicación y comunicación mediante variables compartidas.
- Concurrencia en multiprocesadores de memoria compartida: los procesadores comparten un espacio de direcciones (pueden o no compartir memoria). Pueden usar variables compartidas en dicho espacio de direcciones.
- Concurrencia en sistemas distribuidos: los procesadores interactúan a través de una red de interconexión (no hay memoria común).

Dentro del modelo abstracto de concurrenia, podemos distinguir sentencias atómicas y no atómicas.

- Sentencia atómica: se ejecuta de principio a fin sin verse afectada por otras sentencias. Por ejemplo: leer una celda de memoria y cargar su valor en un registro, incrementar el valor de un registro o escribir el valor de un registro en una celda. El estado final está determinado al no depender de otras instrucciones.
- Sentencias no atómicas: la mayoría de las sentencias son no atómicas. En ellas hay indeterminación del estado final. Las interfoliaciones son las posibles mezclas de las sentencias atómicas.
- El entrelazamiento preserva la consistencia.
- No se hacen suposiciones temporales, salvo que el tiempo es mayor que cero y el progreso es finito.
- Estado e historia de un programa concurrente.

- Sistemas estáticos y dinámicos: según puedan o no variar el número de procesos durante la ejecución.
- Grafo de sincronización: grafo dirigido acíclico. Muestra las restricciones.
- Definición estática de procesos: se usa la clave **process**. Las variables compartidas se inicializan antes de empezar ningún proceso. También están los vectores de procesos, que se diferencian en el valor de una constante.
- **Fork-Join**: fork indica que la rutina nombrada puede comenzar su ejecución al tiempo que comienza la sentencia siguiente (bifurcación). Por otro lado, join espera la terminación de la rutina nombrada antes de proseguir con la sentencia siguiente. Es práctica y potente, pero de difícil comprensión al no estar estructurada.
- **Cobegin-coend**: los procesos están estructurados. Las sentencias en un bloque cobegin-coend comienzan su ejecución todas a la vez. Es menos potente que fork-join, pero más comprensible.

Exclusión mutua y sincronización

Algunas de las interfoliaciones posibles en un conjunto de procesos no son válidas para el funcionamiento requerido de nuestro programa. Esto implica que hay una condición de sincronización o restricción sobre el orden de mezcla. Un caso particular de sincronización es la exclusión mutua, que son secuencias finitas de instrucciones que deben ejecutarse de principio a fin por un único proceso sin que otro proceso esté ejecutándolas al mismo tiempo.

- Exclusión mutua: a las instrucciones restringidas se les denomina sección crítica. La exclusión mutua tiene lugar cuando, en cada instante de tiempo, solo hay un proceso ejecutando cualquier instrucción de la sección crítica (de principio a fin).
- Condición de sincronización: no son correctas todas las posibles interfoliaciones. Por ejemplo, en el problema del productor-consumidor.

Propiedades de los sistemas concurrentes

Las propiedades son los atributos ciertos para todas las secuencias de interfoliación. Hay dos tipos: de seguridad y de vivacidad.

- Propiedades de seguridad (safety): deben cumplirse en cada instante, como la exlusión mutua, la ausencia de interbloqueo (los procesos esperan algo que nunca sucederá)...
- Propiedades de vivacidad (liveness): deben cumplirse eventualmente. Son dinámicas, como la ausencia de inanición (un proceso no puede ser indefinidamente pospuesto) o la equidad (justicia relativa respecto a los procesos).

Verificación de programas concurrentes

Para probar que un programa cumple una propiedad, podemos enfocarlo de distintas formas:

- Posibilidad: realizando ejecuciones. No obstante, solo consideramos un número limitado de historias.

- Enfoque operacional: análisis de todos los posibles casos. El problema es que puede haber un gran número de interfoliaciones.
- Enfoque axiomático: a partir de fórmulas lógicas se prueba la verificación deseada. El invariante global es el predicado que referencia variables globales siendo cierto en el estado inicial y manteniéndose ante cualquier asignación. En el productor-consumidor, sería: `consumidos <= producidos <= consumidos + 1`.

Tema 2: Sincronización en memoria compartida

Introducción a la sincronización en memoria compartida

Este tema aborda soluciones para exclusión mutua y sincronización en memoria compartida. Podemos distinguir dos categorías:

- Soluciones de bajo nivel con espera ocupada: basadas en programas con instrucciones de bajo nivel. Bucles indefinidos en los que se comprueba el estado continuamente (espera ocupada). Pueden ser software (operaciones sencillas de lectura y escritura) o hardware (cerrojos).
- Soluciones de alto nivel: capa software por encima de las anteriores. Bloqueo de procesos en lugar de espera ocupada. Algunas de estas soluciones son los semáforos, las regiones críticas condicionales y los monitores.

Semáforos para sincronización

Son un mecanismo que soluciona problemas de las soluciones de bajo nivel, y su ámbito es más amplio. Se bloquean los procesos en lugar de usar espera ocupada, resuelven fácilmente la exclusión mutua, pueden resolver problemas de sincronización, se implementan mediante instancias de una estructura de datos a las que se accede mediante subprogramas...

Un semáforo es una instancia de una estructura de datos que contiene un conjunto de procesos bloqueados y un valor natural (valor del semáforo). Reside en memoria compartida y se inicializa al principio de un programa.

Hay dos operaciones básicas:

- **sem_wait(s)**: si el valor de s es 0, bloquea el proceso. Se decrementa el valor del semáforo en una unidad.
- **sem_signal(s)**: incrementa el valor de s en una unidad. Si hay procesos esperando, reanuda uno de ellos.

El valor del semáforo nunca es negativo, porque se espera a que sea mayor que 1 para decrementarlo. Solo puede haber procesos esperando cuando el valor es 0.

El invariante de un semáforo sería:

`v_t = v_0 + n_s - n_w >= 0`,

donde v_t es el valor en ese instante, v_0 es el valor inicial, n_s es el número de llamadas a **sem_signal** y n_w es el número de llamadas a **sem_wait**.

Veremos tres problemas que pueden solucionar los semáforos: espera única (producir-consumir), exclusión mutua y problema del productor-consumidor.

Monitores como mecanismo de alto nivel

Hay algunos inconvenientes en el uso de semáforos, como que se basen en variables globales, su uso y función no explícitos y la dispersión de las operaciones. Por ello, recurrimos a los monitores.

Los monitores permiten definir objetos abstractos compartidos que contienen una colección de variables encapsuladas y un conjunto de procedimientos. Se garantiza la exclusión mutua de las variables encapsuladas y la sincronización requerida.

Una instancia del monitos se declara especificando las variables permanentes, los procedimientos del monitor y el código de inicialización (opcionalmente).

La exclusión mutua se basa en la cola del monitor. Dicha cola sigue una política FIFO. Para implementar la sincronización en monitores, disponemos de sentencias de bloqueo y activación y valores de las variables permanentes del monitor, que determinan si se cumple o no la condición.

Para las sentencias de bloqueo y activación, declaramos una variable permanente de tipo condition. A esta variable se asocia una cola de procesos en espera hasta que se haga cierta la condición. Puede invocar a wait (bloquea e introduce en la cola), a signal (libera un proceso de la cola) o a queue, que devuelve true si hay algún proceso en la cola.

Cuando un proceso llama a wait, se debe liberar la exclusión mutua del monitor. Más de un proceso puede estar dentro del monitor, pero solo uno ejecutándose.

Vamos a ver los patrones de solución para los tres problemas anteriores: espera única, exclusión mutua y productor-consumidor.

Colas de condición con prioridad: añadiendo parámetro al wait cuando espera la variable condición, que refleja la prioridad. Sería prioritario el de menor parámetro (menor tiempo).

La semántica de señales es la forma en la que se resuelve el conflicto tras hacerse un signal en una cola no vacía. Existen varias alternativas:

- Señalar y continuar (SC): el proceso señalador continúa su ejecución. El señalado espera bloqueado hasta adquirir de nuevo la exclusión mutua (en la cola del monitor). Se debe programar el wait en bucle para comprobar la condición constantemente.
- Señalar y salir (SS): el proceso señalado se reactiva y el señalador abandona el monitor. Obliga a que signal sea la última instrucción de los procedimientos que la usen.
- Señalar y esperar (SE): el proceso señalado se reactiva y el señalador espera bloqueado en la cola del monitor.
- Señalar y espera urgente (SU): el proceso señalado se reactiva y el señalador espera bloqueado en una cola específica para esto, con mayor prioridad que otros procesos.

La semántica SS condiciona el estilo de programación. Por otro lado, SE y SU son ineficientes si no hay código tras signal. Por último, SC es ineficiente al obligar a usar un bucle para cada signal. Para comparar las diferencias entre las semánticas, veremos la barrera parcial. Sería correcta solo con semántica SU.

Pueden implementarse los monitores con semáforos (para cola de monitores, espera urgente y variables condición).

Soluciones software con espera ocupada para E.M.

Estas soluciones deben cumplir la exclusión mutua, el progreso y la espera limitada. Además, también es deseable que cumplan la eficiencia y la equidad. Algunas soluciones software son:

- Refinamiento sucesivo de Dijkstra: se parte de una versión simple e incorrecta y se va mejorando para cumplir las propiedades. La versión final correcta es el algoritmo de Dekker.
- Algoritmo de Dekker:

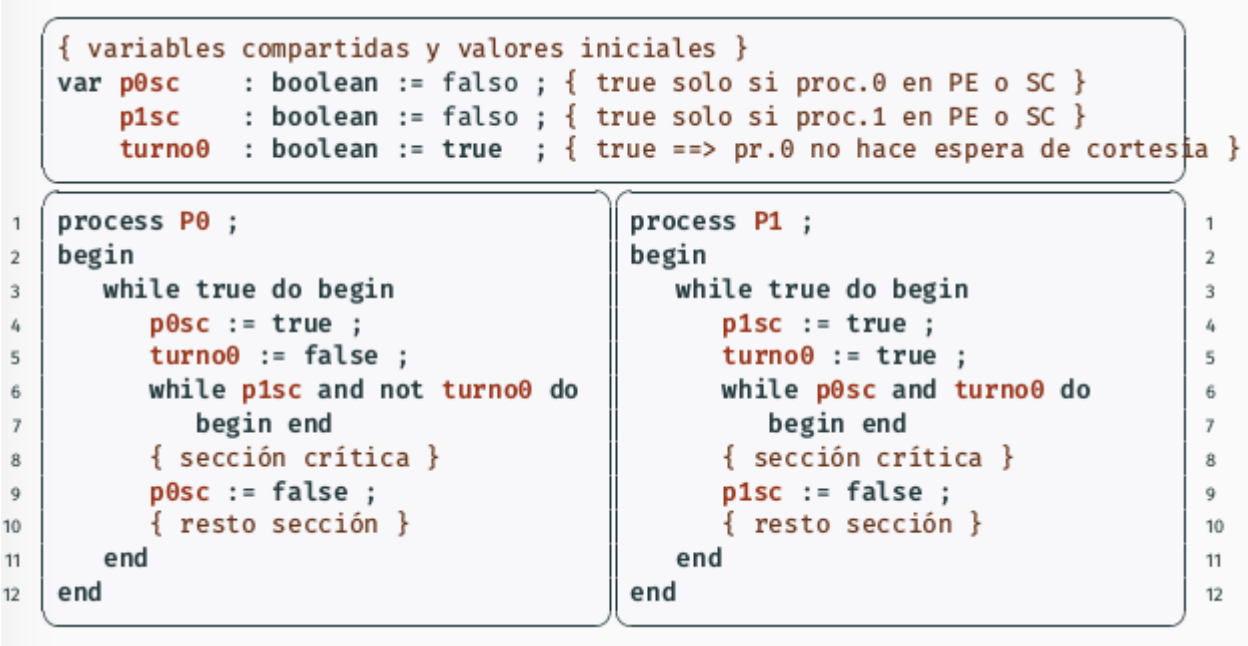
```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin
         if not turno0 then begin
            p0sc := false ;
            while not turno0 do
               begin end
            p0sc := true ;
         end
      end
      { sección crítica }
      turno0 := false ;
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin
         if turno0 then begin
            p1sc := false ;
            while turno0 do
               begin end
            p1sc := true ;
         end
      end
      { sección crítica }
      turno0 := true ;
      p1sc := false ;
      { resto sección }
   end
end
```

Figure 1: Dekker

- Algoritmo de Peterson: más simple que el de Dekker.



```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      turno0 := false ;
      while p1sc and not turno0 do
         begin end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      turno0 := true ;
      while p0sc and turno0 do
         begin end
      { sección crítica }
      p1sc := false ;
      { resto sección }
   end
end
```

Figure 2: Peterson

Soluciones hardware con espera ocupada (cerrojos) para E.M.

Vamos a ver una instrucción llamada TestAndSet. Se le pasa como argumento la dirección de memoria de la variable lógica que actúa como cerrojo. Se ejecuta de forma atómica.

Los cerrojos ocupan poca memoria y son eficientes en tiempo, pero las esperas ocupadas consumen CPU, no se cumple la equidad y los cerrojos son accesibles directamente. Por tanto, solo se usan desde componentes software del SO y con esperas ocupadas muy cortas.

Fuentes

Las fuentes de esta exposición son las siguientes:

Temario SCD: En PRADO, dentro de la parte teórica de la asignatura. Presentaciones de Carlos Ureña, José M. Mantas y Pedro Villar. Departamento de Lenguajes y Sistemas Informáticos (LSI). Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones. Universidad de Granada (UGR).